UNIVERSITAT DE BARCELONA

Trabajo de final de grado

GRADO DE INGENIERÍA INFORMÁTICA

Facultad de Matemáticas
Universidad de Barcelona

SHADING ANALYSIS ON UNITY3D

Autor: Raúl Pérez

Directora:	Dra. Anna Puig
Realizado en:	Departamento de Matemáticas e Informática

Barcelona, 22 de Junio de 2017

Abstract

The broad majority of videogames nowadays have visual features that make them easily recognizable to the common people. This project seeks to visually change the game *Fracsland* in a way that appeals more to young users. To reach this goal, some methods and techniques that can improve the sthetic of the game are to be analyzed using the game engine Unity3D. This project also makes a first approach to new technologies like virtual reality.

Resumen

En la actualidad la mayoría de videojuegos disponen de ciertas características visuales que les definen y que les hacen fácilmente reconocibles para la gente. Este proyecto busca modificar la visualización del juego *Fracsland* de manera que resulte más atractivo para un público infantil. Con el fin de conseguir este objetivo se van a analizar algunos métodos y técnicas que sirvan para mejorar la estética del juego utilizando el motor de desarrollo Unity3D. Este proyecto contempla también una primera aproximación a nuevas tecnologías como los dispositivos de realidad virtual.

Agradecimientos

Quiero agradecer a todas las personas sin las cuales este proyecto no hubiese sido posible, en especial a Anna, por su constancia como tutora y a Cristian por haber creado el juego que me ha permitido realizar este proyecto.

Índex

1 Introducción **1**

1.1 Ámbito y contexto 1

1.2 Objetivos del proyecto 4

1.3 Tareas 4

1.4 Planificación 6

1.5 Estructura de la memoria 7

2 Métodos de interacción de la luz con superfícies **8**

2.1 Tipos de luz 8

2.2 La iluminación en Unity 9

2.3 Reflexión Difusa 11

2.4 Reflexión Especular 12

2.4.1 Especularidad de Phong 13

2.4.2 Especularidad de *Blinn-Phong* 15

2.5 *Rim Lighting* 15

3 Técnicas de simulación de rugosidad **18**

3.1 *Bump Mapping* 18

3.1.1 Implementación en Unity 21

3.1.2 Integración en la escena 22

3.2 *Parallax mapping* 24

3.2.1 Mejoras respecto del *parallax mapping* convencional 27

4 Transparencias **32**

4.1 *Render Queues* 32

4.2 Materiales transparentes 32

4.3 Materiales semitransparentes 34

4.4 *Alpha cutoff* 36

5 Superfícies reflectivas y refractivas **39**

5.1 Reflexiones 39

5.1.1 Implementación en Unity 44

5.2 Materiales metálicos y dieléctricos 47

5.2.1 Implementación de materiales metálicos y dieléctricos utilizando reflexiones . . . 47
5.3 Refracciones . . . 50
5.3.1 Implementación de superfícies refractivas en Unity . . . 54

6 Visualización de sombras en algoritmos proyectivos 57
6.1 Visualización de sombras en la GPU . . . 57
6.2 Visualización de sombras en Unity . . . 58
6.2.1 Sombras con luces direccionales . . . 59
6.2.2 Sombras con luces puntuales . . . 62
6.2.3 Sombras con *Spot Lights* . . . 64
6.2.4 Sombras semitransparentes . . . 66
6.2.5 Problemas y soluciones . . . 69

7 Visualización de terrenos 74
7.1 Creación de hierba . . . 74
7.1.1 Hierba con *Geometry Shader* . . . 74
7.1.2 Hierba con textura . . . 78
7.2 Creación de agua . . . 86
7.2.1 Implementación del agua en Unity . . . 91
7.3 Aplicación de niebla . . . 96

8 Visualización no realista 102
8.1 Toon shading . . . 102
8.1.1 *Steep Toon Shading* . . . 102
8.1.2 *Flat Toon Shading* . . . 103
8.1.3 *Toon Shading* con texturas . . . 103
8.1.4 Implementación de *Toon Shading* en Unity . . . 104
8.2 Contornos . . . 107

9 Realidad Virtual en Unity 112
9.1 Funcionamiento de la realidad virtual en Unity . . . 113
9.2 Integración de realidad virtual en el proyecto . . . 113
9.2.1 Adaptación de la interfície de usuario a la realidad virtual . . . 115
9.2.2 Resultados . . . 118

10 Conclusiones y trabajo futuro 121

11 Apéndice: Manual técnico **122**

11.1 Guía de instalación para desarrolladores 122

11.1.1 Requisitos mínimos 122

11.1.2 Instalación del proyecto 122

1 Introducción

La mayoría de niños de entre 6 y 9 años presentan dificultades a la hora de comprender el concepto de la división. *Fracsland*[1] es un juego creado por Cristian Muriel Ordoñez mediante el motor gráfico Unity3D[1], que tiene como objetivo que los más pequeños aprendan a realizar divisiones de una forma eficaz y entretenida, así como de entregar al profesor herramientas útiles para personalizar el aprendizaje de cada alumno.

Este proyecto busca mejorar la experiencia de juego a través de la programación de programas de sombreado o *shaders* que cambien la estética de *Fracsland*. En concreto, el proyecto se ha centrado en una escena del juego: el poblado, que es la escena a la que el jugador vuelve cada vez que termina una misión. El poblado está formado por un terreno rodeado por agua, unas cuantas casas, algunos habitantes, sol y farolas como fuentes de luz y otros objetos como barriles o vallas. En la figura 1 se pueden observar algunas imágenes del poblado vistas desde diferentes perspectivas.

Con el fin de conseguir una visualización más desenfadada, en este proyecto se pretenden desarrollar e implementar diferentes métodos de *shading* para modificar la visualización de las superfícies de los objetos de la escena. Después de aplicar estos cambios, el poblado adquirirá una estética más infantil que se adhiere más al tipo de público que busca el juego. Los resultados de aplicar estos posibles cambios sobre la escena se observan en la figura 2.

Además de estos cambios realizados sobre la escena, este proyecto contempla la introducción de elementos de realidad virtual. La figura 3 muestra el auge en las compras de dispositivos relacionados con la realidad virtual en los últimos años, además de las predicciones para los próximos años. Como se observa en el gráfico, se prevé que este tipo de dispositivos terminen por consolidarse en el mercado del mundo de los videojuegos en los próximos tres años. Teniendo en cuenta estas predicciones se ha decidido adaptar este proyecto para poder ser utilizado por dispositivos de realidad virtual como el *Occulus Rift*.

Una vez adaptado, el jugador podrá relacionarse con la escena en primera persona. En la figura 4 se observan los resultados obtenidos utilizando el dispositivo *Occulus Rift* dentro del poblado.

El principal beneficio que esta tecnología aporta al usuario es la inmersión en el juego. Utilizando este tipo de tecnologías el usuario puede introducirse mejor en la escena convirtiéndose él mismo en el personaje principal, de forma que la función educativa del juego pase a un plano menos obvio.

1.1 Ámbito y contexto

Terminada la primera fase de desarrollo, *Fracsland* era un juego listo para poder ser utilizado en aulas, sin embargo, la estética visual que utilizaba no se correspondía

[1]Enlace de descarga: https://gitlab.com/muriel-maths/Fracsland.git

(a) Vista de una casa del poblado.

(b) Vista del terreno del poblado.

(c) Vista de un personaje del poblado.

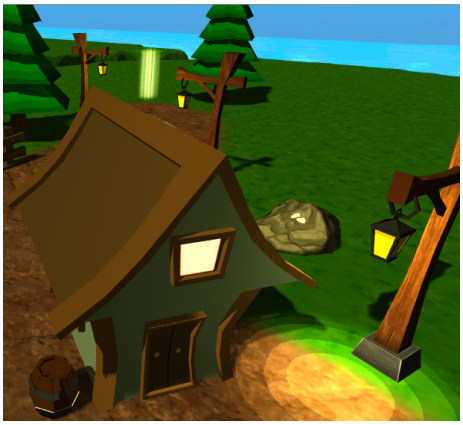
(d) Vista desde arriba del poblado.

Figura 1: Diferentes imágenes de la escena antes de ser modificada.

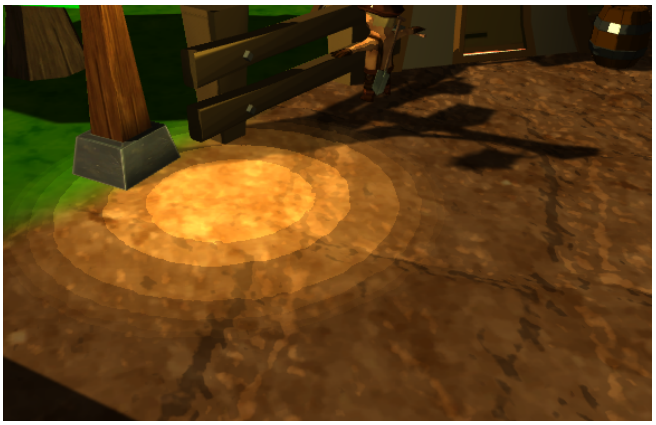
con la imagen que se quería dar al juego, por lo que se hizo necesario cambiarle la visualización. Para conseguir este objetivo era necesario cambiar los materiales que aparecían en las diferentes escenas por otros que dieran un aspecto menos realista, con el fin de que los jugadores se sumerjan más fácilmente en el juego. De esta forma, el jugador consigue dejar de lado más facilmente el pensamiento de estar realizando operaciones matemáticas para así poder centrarse en disfrutar de la experiencia que el juego ofrece.

Por otra parte, visto el auge de las tecnologías de realidad virtual como el *Occulus Rift* en los últimos años y la necesidad de introducir un nuevo nivel de inmersión en el juego, se ha decidido integrar como opción el poder utilizar realidad virtual dentro del proyecto. Esto no es casual, pues tratándose este proyecto de intentar mejorar la experiencia del usuario de manera que los jugadores eviten pensar directamente en las operaciones que están realizando, la introducción de un elemento de realidad virtual que haga que las aventuras del personaje principal sean también sus propias aventuras se hace necesario.

En relación con el grado de Ingeniería Informática de la Universidad de Barce-

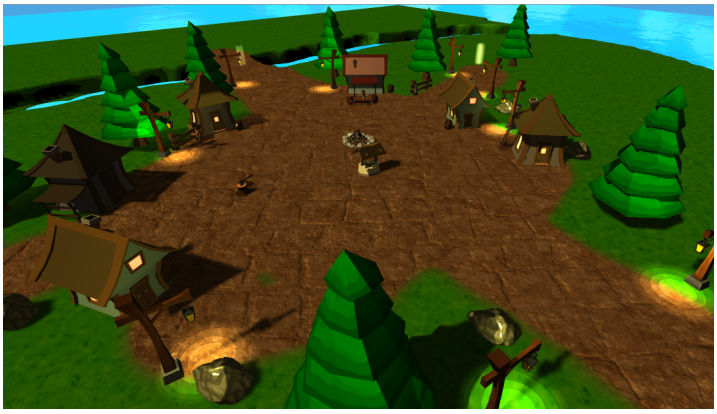


(a) Vista de una casa del poblado.



(b) Vista del terreno del poblado.

(c) Vista de un personaje del poblado.



(d) Vista desde arriba del poblado.

Figura 2: Diferentes imágenes de la escena antes de ser modificada.

lona, este proyecto se sitúa dentro del campo del estudio de gráficos, en concreto dentro de la asignatura *Gràfics i Visualització de dades*. A pesar de que en esta asignatura se centra en la implementación de una aplicación utilizando la API *OpenGL*, dos de los conceptos en los que se hacía hincapié eran por una parte la importancia del *pipeline* gráfico y por otra la utilidad de los programas de sombreado o *shaders* a la hora de simular materiales.

Este proyecto se apoya en esos dos conceptos. El principal y más importante es el de implementar *shaders* que permitan simular materiales no realistas mediante un lenguaje similar al utilizado en la asignatura, pero por otra parte, la comprensión del *pipeline* gráfico se vuelve un elemento indispensable, ya que sin ese elemento este proyecto no sería posible.

Además de los conceptos aprendidos en la asignatura *Gràfics i Visualització de dades*, para comprender algunas de las operaciones que se realizan son elementales nociones generales de álgebra y cálculo, en concreto subrayando los apartados de operaciones con matrices y cambios de base en el primero y trigonometría en el segundo. También son necesarios conceptos de física relacionados con las reflexiones y refracciones de la luz.

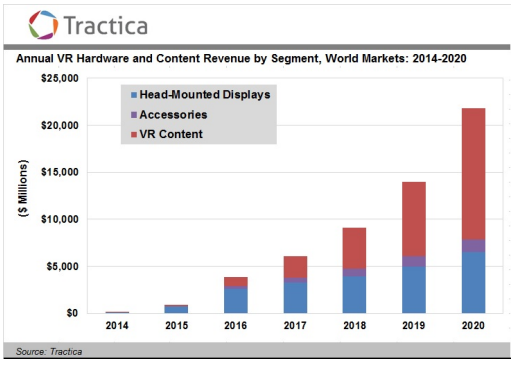


Figura 3: Gráfico de la venta de dispositivos de realidad virtual entre los años 2014 y 2016, junto con las predicciones hasta el año 2020.[2]

1.2 Objetivos del proyecto

Los objetivos de este proyecto se pueden dividir en dos apartados: profesionales y personales.

Los objetivos profesionales son, en primer lugar, poder conseguir que el juego *Fracsland* tenga una estética más atractiva que permita a los jugadores disfrutar más de la experiencia visual. Otro de los objetivos que se espera poder cumplir es la adaptación del juego a sistemas de realidad virtual, tratando de evitar problemas recurrentes con este tipo de tecnologías como son los mareos. Además de esto, también se espera que los conceptos aprendidos sobre las distintas técnicas que se aprendan queden plasmados en esta memoria como guía para futuros desarrolladores de *shaders* en el entorno de Unity3D.

Los objetivos personales no son simplemente recordar y establecer firmemente aquellos conceptos aprendidos en la asignatura *Gràfics i Visualització de dades*, sino ir más allá y aprender técnicas que no se hayan podido enseñar, además de entender bastante profundidad el funcionamiento interno del motor gráfico Unity3D.

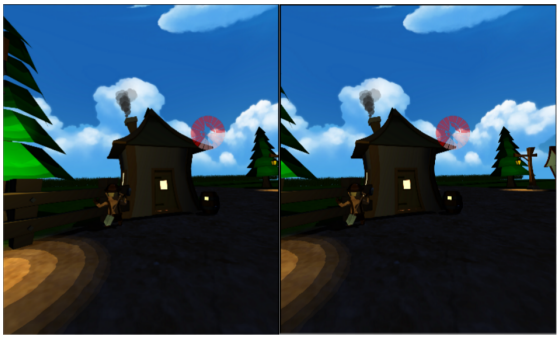
1.3 Tareas

La lista de tareas que se han planificado en este proyecto es la siguiente:

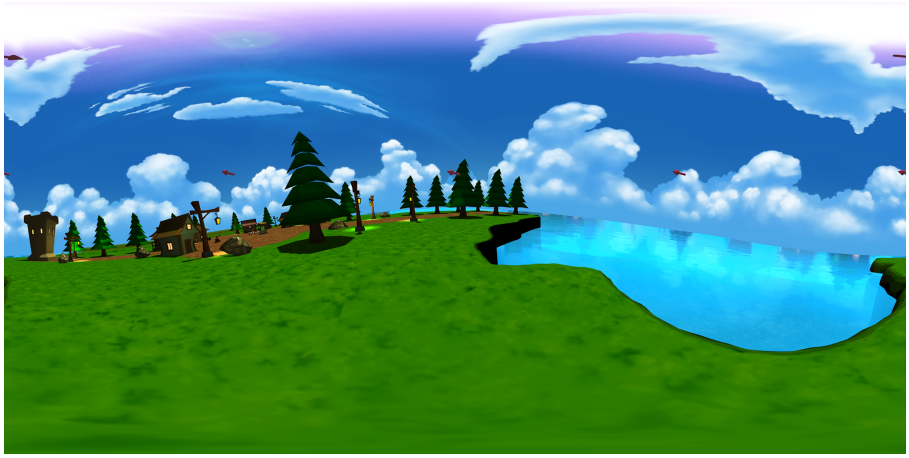
1. **Planificación:** Proceso de decisión y ordenación de las tareas que se van a realizar.

2. **Iluminación de la escena:** Generación de programas de sombreado básicos y adaptación de estos programas a los diferentes tipos de luz que les afecte. Técnicas de iluminación no realista.

3. **Integración de *bump mapping*:** Aplicación de la técnica *bump mapping* de generación de rugosidad en una superfície a través de una textura en los programas de sombreado.



(a) Vista estereoscópica de una casa del poblado.



(b) Vista en 360° del poblado.

Figura 4: Vistas del poblado con realidad virtual.

4. **Integración de *parallax mapping*:** Aplicación de la técnica *parallax mapping* de generación de rugosidad en una superfície a través de una textura en los programas de sombreado.

5. **Creación de materiales con transparencias:** Generación de programas de sombreado capaces de simular superfícies transparentes.

6. **Creación de materiales con semitransparencias:** Generación de programas de sombreado capaces de simular superfícies semitransparentes.

7. **Creación de materiles con superfícies reflectivas:** Creación de programas de sombreado que reflejen el entorno de un objeto.

8. **Creación de materiales con superfícies refractivas:** Creación de programas de sombreado capaces de simular una superfície refractiva.

9. **Integración de sombras:** Introducción de sombras sobre los objetos que utilice los programas de sombreado anteriores.

10. **Creación de hierba:** Creación de hierba mediante programas de sombreado mediante dos métodos diferentes. Integración en la escena de la mejor aproximación.

11. **Creación de niebla:** Adaptación de los programas de sombreado anteriores para introducir niebla en el poblado.

12. **Creación de agua:** Sustitución del agua actual del poblado mediante la creación de programas de sombreado a través de los cuales se genere agua con un estilo más acorde con el resto de la escena.

13. **Introducción de Realidad Virtual:** Adaptación de la escena para poder ser jugada a través de dispositivos de realidad virtual.

1.4 Planificación

El proyecto se ha comenzado el día 1 de Febrero y finaliza el día 22 de Junio. Teniendo en cuenta estos límites se ha diseñado una predicción sobre el tiempo que se deberá emplear en cada una de las tareas que se han definido. Esta predicción se muestra en el diagrama de Gantt de la figura 5.

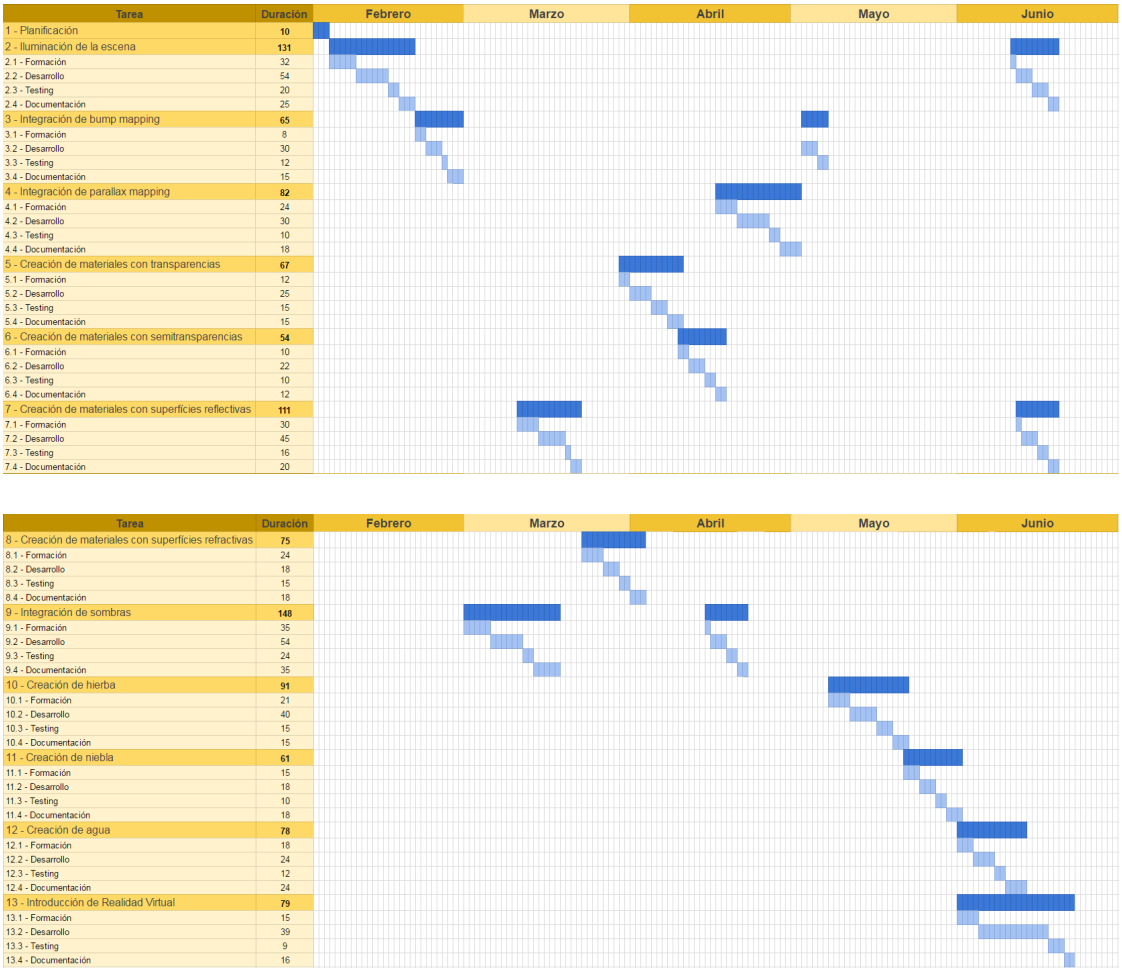


Figura 5: Diagrama de Gantt que muestra la predicción de las tareas planteadas del proyecto.

La predicción de este diagrama muestra que para realizar el proyecto será necesarias aproximadamente 1052 horas. Esto implica realizar 7 horas o 9 horas y media teniendo en cuenta los fines de semana entre el día de inicio y el de entrega.

En el diagrama se puede observar cómo algunas de las tareas vuelven con el tiempo. Esto principalmente se debe principalmente a que existen dependencias entre tareas, como es el caso de las sombras, que dependen de los materiales semitransparentes para realizar sombras semitransparentes.

1.5 Estructura de la memoria

Esta memoria se ha dividido en diversas secciones en las que se explican técnicas o grupos de técnicas con las que llevar a cabo las tareas del apartado 1.4. En concreto éstas secciones explican en primer lugar la teoría que se esconde detrás de las técnicas que se implementan, de manera que el lector pueda comprender la base con más claridad. Una vez explicada la teoría, dentro de un apartado de la sección, se procede a desglosar cómo ha sido implementada la técnica en Unity. En los casos en que el método que se analice haya jugado un papel relevante en la visualización, también se muestra cómo la integración del material en el poblado.

El orden en que se ha establecido a estas técnicas dentro de la memoria está relacionado con la dificultad de cada una, comenzando por aquellas con menor dificultad y finalizando por las que son consideradas más complejas.

2 Métodos de interacción de la luz con superfícies

Los *shaders* son programas compilados ejecutados en la GPU que permiten al programador entre otras utilidades determinar el modo en el que la luz de una escena interactúa con los objetos de una escena. En Unity cada objeto o *GameObject* tiene asociado un material que se construye mediante los cálculos realizados dentro del shader.

2.1 Tipos de luz

Antes de entrar en la interacción entre el objeto y las luces de una escena es necesario conocer los tipos de luz que existen en Unity. Se distinguen tres tipos:

1 Luces direccionales

Este tipo de luz es el más común y se utiliza para simular una luz distante como por ejemplo el Sol o la Luna. En la figura 6 se muestra un esquema básico de una luz direccional incidiendo sobre un objeto. Este tipo de luces están formadas por un vector de dirección y una intensidad. En la figura 7 se observa cómo a medida que aumenta la distancia entre la luz y el objeto los rayos de luz que impactan con el objeto tienen ángulos cada vez menores. Esto permite que en luces situadas a una distancia muy lejana, se pueda simplificar el modelo de interacción suponiendo que el ángulo entre dos vectores que tengan como origen la posición de la luz y como destino dos puntos diferentes de la superfície del objeto iluminado es siempre cero, o lo que es lo mismo, son rayos paralelos con una única dirección.

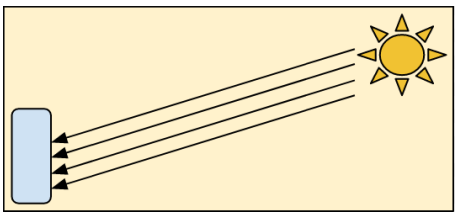


Figura 6: Esquema de una luz direccional.

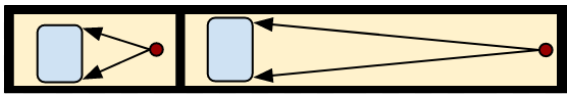


Figura 7: Modificación del ángulo de incidencia en función de la distancia.

2 Luces puntuales

Las luces puntuales son de todos los tipos de luz las más realistas ya que dispersan luz en todas direcciones como haría por ejemplo, la bombilla representada en la figura 8. Está formada por una posición, un rango de alcance y una intensidad. Se

diferencian en las luces direccionales prácticamente en todo, ya que no tienen una dirección única sino que tienen una dirección diferente para cada punto sobre el que incide.

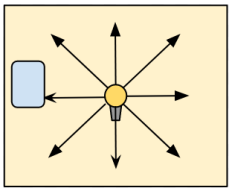


Figura 8: Esquema de una luz puntual.

3 *Spotlights*

Las *spotlights* son luces similares a linternas que iluminan un área. Están formadas por una posición, una distancia máxima, un ángulo de apertura y una intensidad. Un ejemplo de ellas está en la figura 9, donde se puede observar una *spotlight* enfocando un objeto dentro de un area con una apertura que describe un ángulo.

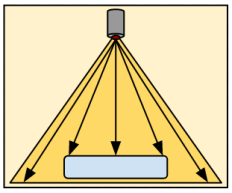


Figura 9: Esquema de un spotlight.

2.2 La iluminación en Unity

La iluminación sobre los objetos de una escena en Unity se realiza dentro de un *shader*, que contiene un programa *Vertex Shader* por el que pasan todos los vértices del modelo 3D que se está iluminando, y un *Fragment Shader* por el que más tarde pasan todos los *fragments*, que son píxeles potenciales con información de su posición en la escena, vectores o su profundidad respecto a la cámara entre otras informaciones. Estos dos programas, el *Vertex Shader* y el *Fragment Shader*, constituyen un paso o *Pass*. En Unity es posible que un mismo shader contenga más de un paso, es más, lo normal es que exista un mínimo de dos: el paso *ForwardBase* y el *ForwardAdd*.

En la página web de documentación de Unity [5] se explica cómo el paso Forward-Base contiene un programa *Vertex Shader*y un programa *Fragment Shader* en los que se calcula la iluminación respecto de la luz direccional más intensa, mientras que en paso ForwardAdd se calcula la iluminación del resto de luces. El color que da como resultado en cada paso la salida del *Fragment Shader* se mezcla entonces, produciendo el resultado final.

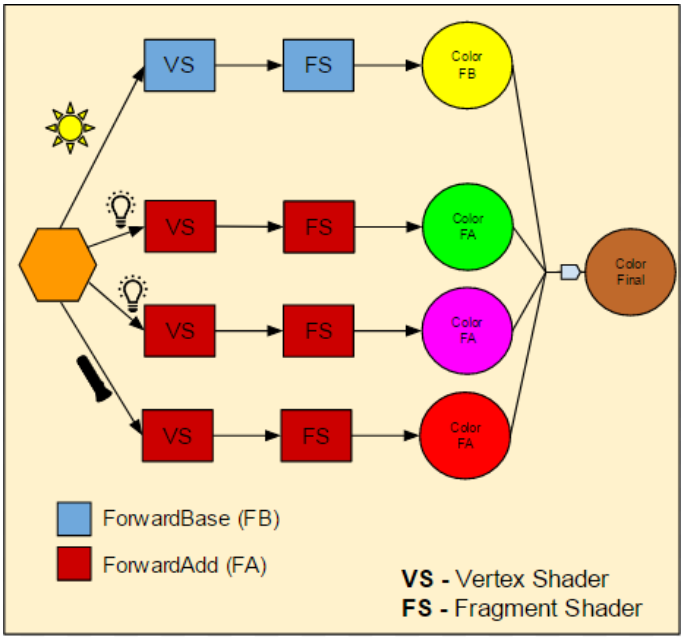


Figura 10: Esquema del funcionamiento en Unity de los pasos en *Forward Rendering*.

En la Figura 10 se observa cómo al *Vertex Shader* de cada uno de los pasos llega información proveniente del objeto, como puede ser el vector normal (perpendicular a la superfície), la coordenada de la textura en caso que haya o la posición, todo esto en coordenadas de objeto, es decir, en coordenadas normalmente expresadas respecto al centro del modelo tridimensional. Dentro del *Vertex Shader* se procesa esta información para cada vértice del objeto, generalmente pasándola a coordenadas de ventana a través de un cambio de base que se consigue multiplicando los vectores o posiciones por la matriz de cambio de base llamada Modelo-Vista-Proyección o MVP a partir de ahora. Una vez procesada esta información se empaqueta en una estructura y pasa al *Fragment Shader*, atravesando antes por un interpolador que, basándose en la información de los vértices que conforman las primitivas con las que está estructurado el objeto, interpolarán esos valores a los fragments o píxeles que haya dentro de éstas. Un ejemplo de esto se observa en la figura 11, donde se ve cómo en un plano formado por dos primitivas, en este caso triángulos, donde los colores de los vértices se interpolan a lo largo del plano. En este caso el color se ha calculado dentro del *Vertex Shader* y se ha pasado al *Fragment Shader*, donde se devuelve el color que en este caso ya le llega interpolado. Este método, conocido como sombreado de *Gouraud*[7] o *Gouraud shading*, implica realizar los cálculos de vectores, colores o puntos dentro de el *Vertex Shader* para que llegue así, interpolado al *Fragment Shader*. En la figura 11 izquierda se observa esta interpolación.

Existe, sin embargo, otra forma de realizar este proceso descrito por primera vez en [8], conocido como *Phong shading* que aunque más costoso da sin lugar a dudas mejores resultados. Se trata de calcular el color dentro del propio *Fragment Shader* utilizando información interpolada como por ejemplo vectores. La diferencia entre este método y el *Gouraud shading* es que en éste se interpola el color mientras que en

el *Phong shading* se interpola la información necesaria, como por ejemplo el vector normal, para realizar los cálculos con los que se determine el color. Los cálculos de esta forma se tienen que realizar por cada fragment del objeto y, por lo tanto, al haber siempre más fragments que vértices, la velocidad de procesamiento baja. Existen optimizaciones como los *Deferred Shaders* que focalizan en utilizar menos operaciones por pixel, aunque estas estrategias han quedado fuera del ámbito de este proyecto.

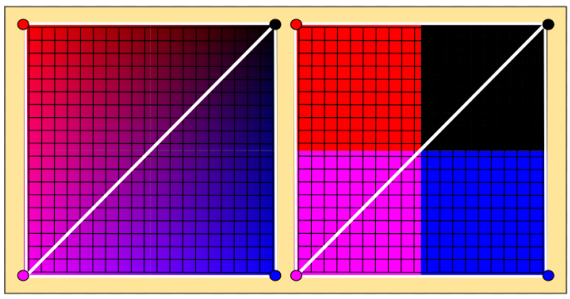


Figura 11: A la izquierda un plano de 4 vértices con el color calculado en el *Vertex Shader*. A la derecha el mismo plano con el color calculado en el *Fragment Shader*.

En este proyecto se ha considerado que los resultados visuales obtenidos realizando los cálculos de esta manera son mucho mejores y por lo tanto la mayoría de los cálculos se realizan en el *Fragment Shader*.

2.3 Reflexión Difusa

La reflexión difusa es la interacción entre un foco de luz y la superfície de un objeto, siendo esta interacción la que más información visual aporta al observador. El esquema básico que representa la interacción difusa entre un rayo de luz y una superfície se muestra en la figura 12, donde como se puede observar la luz se refleja en todas direcciones de manera que un observador desde cualquier punto verá el mismo color. La figura 14 representa un cyborg sin información de texturas simplemente mostrando la reflexión difusa sobre un cuerpo con color o albedo blanco con una luz direccional que apunta desde la derecha de la pantalla a la izquierda. Se puede observar cómo cuando la superfície está en perpendicular a la dirección de la luz el color es blanco mientras que en partes donde la superfície está girada respecto de esta dirección el color toma un color ennegrecido o directamente negro, lo cual indica que la reflexión difusa es más baja.

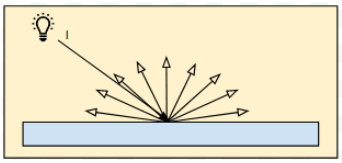


Figura 12: Representación de una interacción difusa.

En la figura 13 se muestran los vectores implicados en la reflexión difusa, que son el vector perpendicular a la superfície $\vec{n}$ o vector normal y el vector de incidencia

de la luz sobre el objeto que en este caso está representado por $\vec{l}$ de *light* (luz en inglés). El cálculo del valor de reflexión difusa se realiza mediante la ley de Lambert [9] que establece que la intensidad de la reflexión es directamente proporcional al coseno entre el vector $\vec{l}$ y $\vec{n}$. Es decir,

$$\vec{l} \cdot \vec{n} = ||\vec{l}|| \cdot ||\vec{n}|| \cdot \cos\theta$$
$$I = \hat{l} \cdot \hat{n} = \cos\theta.$$

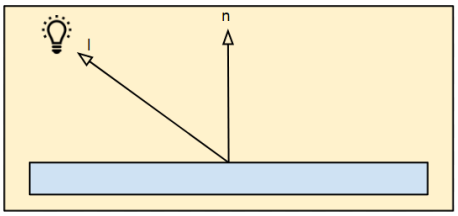


Figura 13: Esquema de la reflexión difusa.

Con esto se consigue,

$$\cos 0 = 1 \qquad \vec{l} = \vec{n}$$
$$\cos\frac{\pi}{2} = 0 \qquad \vec{l} \perp \vec{n}$$
$$\cos\pi = -1 \qquad \vec{l} = -\vec{n}$$
$$\cos\frac{3\pi}{2} = 0 \qquad \vec{l} \perp \vec{n}$$

Esto significa que si se mira el objeto desde la posición de la luz en la misma dirección la parte visible del modelo tiene una reflexión elevada salvo donde la superfície se empieza a curvar, ya que esto representa un cambio en el ángulo entre $\vec{l}$ y $\vec{n}$.

2.4 Reflexión Especular

Si la reflexión difusa representa la interacción básica entre una luz y un objeto, la reflexión especular reproduce el brillo provocado por una luz sobre una superfície pulida. En la figura 15 se observa la principal diferencia física entre este tipo de reflexiones y las reflexiones difusas, que es el de la distribución de los rayos reflejados. Ya se ha visto en el apartado 2.3 cómo las reflexiones difusas reflejan la luz recibida en todas direcciones, pues bien, las reflexiones especulares se comportan de forma opuesta, sólo reflejan en una sola dirección, de manera que un observador, dependiendo de dónde se encuentre verá o no el rayo reflejado. La reflexión especular

Figura 14: Representación de la reflexión difusa con albedo blanco.

depende de tres variables: la dirección de la luz, la orientación de la superfície y la posición del observador. Sin embargo, existen dos maneras de calcular el brillo especular de un objeto, que son el método usado por Phong, y la mejora que realizó James F. Blinn sobre este método más tarde, conocida como método de *Blinn-Phong* [10].

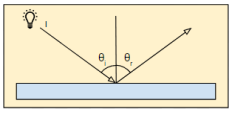


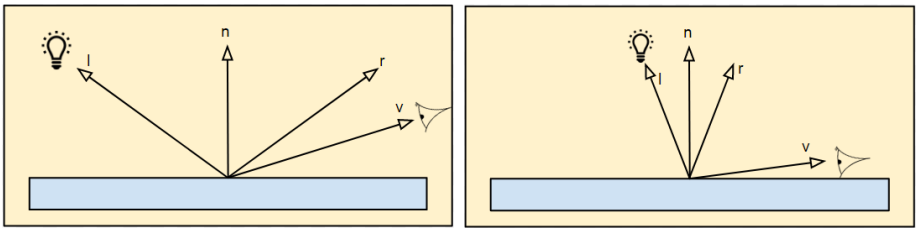
Figura 15: Representación de una interacción especular.

Figura 16: Representación de la reflexión especular sobre un albedo negro.

2.4.1 Especularidad de Phong

El método de Phong no se reduce simplemente a definir qué valores se van a interpolar si no que especifica tanto un método de reflexión difusa como de reflexión especular. En el caso de la reflexión difusa, Phong supone una superfície lambertiana, con lo que utiliza el método explicado en el apartado 2.3. Por otra parte,

Phong asume una superfície perfectamente pulida, es decir, que no tenga ningún tipo de rugosidad. A las micro rugosidades que puede presentar una superfície no ideal también se les llama *microfacet*. En una superfície ideal, según Phong, la intensidad de la especularidad depende del ángulo que forme la luz reflejada con la dirección en la que mira el observador. Los vectores que participan en el cálculo de la especularidad de un punto de una superfície, siguiendo el modelo de Phong, están representados en las figuras 17a y 17b.



(a) Posición A. (b) Posición B.

Figura 17: Esquema de funcionamiento de luz especular utilizando el vector $\vec{r}$.

En 17a $\vec{r}$ y $\vec{v}$ forman un ángulo menor que en 17b y por lo tanto, la intensidad será mayor en la primera.

Según Phong, la fórmula para conseguir la intensidad de la especularidad o k_{spec},

$$k_{spec} = \cos^n \theta_{\vec{r}\vec{v}} = (\hat{r} \cdot \hat{v})^n$$

donde n es un valor que modela la especularidad de un material.

(a) n = 20. (b) n = 60. (c) n = 200.

Figura 18: Muestras de la representación de la especularidad mediante Phong con diferentes valores de n.

2.4.2 Especularidad de *Blinn-Phong*

El modelo de reflexión especular de *Blinn-Phong* es muy similar al de Phong, sin embargo, introduce un nuevo vector que interviene en el proceso de cálculo de k_{spec}. El vector *half* o $\vec{h}$. Este vector se define como,

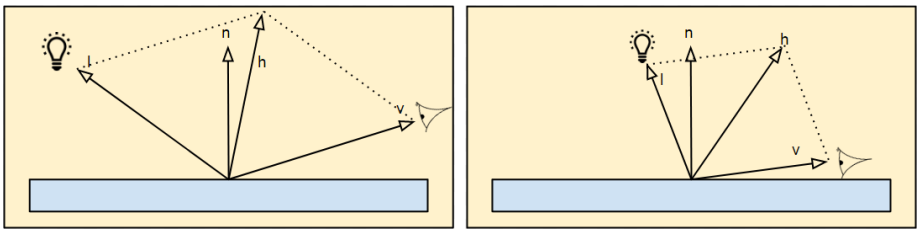
$$\hat{h} = \frac{\hat{l} + \hat{v}}{||\hat{l} + \hat{v}||} \tag{2.1}$$

es decir, se trata del vector que separa por la mitad a los vectores $\vec{l}$ y $\vec{v}$, como ya se observa en las figuras 19a y 19b. Por lo tanto, se puede afirmar que en el caso que $\vec{v}$ sea igual que el vector reflejado $\vec{r}$, $\vec{h}$ será igual que $\vec{n}$ y en ese caso interesa que k_{spec} sea máxima.

Según el modelo de *Blinn-Phong* el coeficiente de especularidad k_{spec} se calcula mediante la siguiente fórmula,

$$k_{spec} = \cos^m \theta_{\vec{n}\vec{h}} = (\hat{n} \cdot \hat{h})^m \tag{2.2}$$

donde m es un valor que modela la especularidad del material.



(a) Posición A. (b) Posición B.

Figura 19: Esquema de funcionamiento de luz especular según *Blinn-Phong*.

Este modelo está considerado más realista que el propuesto por *Phong* ya que entre otros motivos, el ángulo entre $\vec{n}$ y $\vec{h}$ nunca excede los 90°, no así los vectores $\vec{v}$ y $\vec{r}$ del modelo de *Phong*, con lo cual se consigue que la luz se comporte de forma más fiel a la realidad cuando se utilizan valores de m bajos donde el brillo ocupa más espacio en superficies grandes y planas. Además el modelo *Blinn-Phong* no requiere un gasto computacional extra demasiado elevado. Es por ello que es un modelo bastante aceptado para representar la interacción entre una luz y los objetos de una escena y por lo tanto es el modelo escogido en este proyecto.

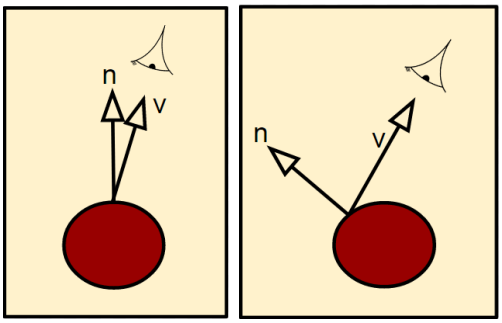
2.5 *Rim Lighting*

En algunos juegos a veces se observa cómo ciertos personajes u objetos tienen en un aura que les envuelve siempre sea cual sea la dirección en que se mire y les hace

(a) m = 40. (b) m = 120. (c) m = 400.

Figura 20: Muestras de la representación de la especularidad mediante Blinn-Phong con diferentes valores de m.

destacar, como se muestra en las figuras 22a, 22by 22c. Este efecto se conoce como *rim lighting* y consiste en resaltar aquellas zonas del modelo cuya normal forma un ángulo de entre aproximadamente 90° y 270° con el vector $\vec{v}$.



(a) Posición A. (b) Posición B.

Figura 21: Esquema de funcionamiento de luz especular según *Blinn-Phong*.

En la figura 21a aparece un observador que mira directamente a un punto de la superfície del objeto rojo. En este caso como el observador está mirando casi de frente al punto, es decir, el ángulo entre $\vec{n}$ y $\vec{r}$ es pequeño, el valor del *rim* tiene que ser bajo o directamente 0. En la figura 21b sin embargo, el vector normal del punto que se está analizando forma un ángulo de prácticamente 90° con $\vec{v}$ y por lo tanto, se sabe que ese punto está situado cerca del límite visual del observador en el objeto.

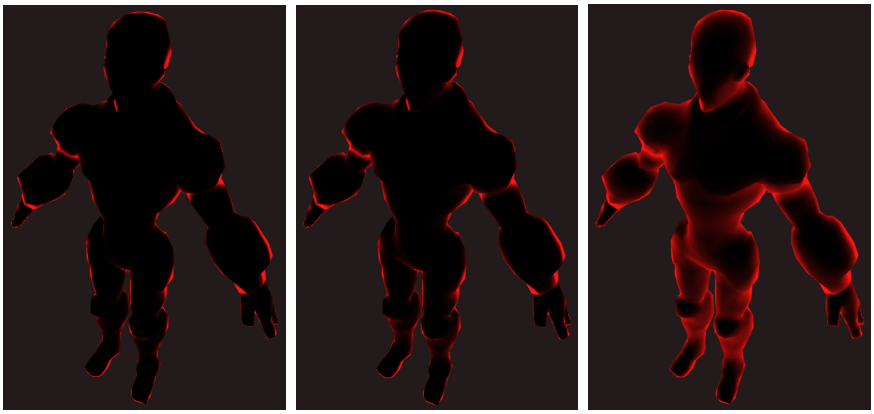
La fórmula utilizada es por tanto la siguiente,

$$k_{rim} = (1 - \cos\theta_{\vec{n}\vec{v}})^m = (1 - \hat{n} \cdot \hat{v})^m$$

donde m es un valor utilizado para enfatizar el efecto de *rim*.

En este caso no interesa simplemente el coseno entre $\vec{n}$ y $\vec{v}$, ya que esto implicaría que k_{rim} sería 1 cuando $\vec{n}$ y $\vec{v}$ fuesen iguales, es decir, cuando se mira un punto de frente. Es justo lo contrario lo que se busca, y por lo tanto, como el producto escalar entre $\vec{n}$ y $\vec{v}$ oscila entre 1 y -1 y los valores negativos son indiferentes para el observador ya que se encuentran más allá de su campo de visión en el objeto es posible modificar la función de la siguiente forma:

$$k_{rim} = (1 - max(0, \cos\theta_{\vec{n}\vec{v}}))^m = (1 - max(0, \hat{n} \cdot \hat{v}))^m$$



(a) m = 9. (b) m = 6. (c) m = 2.

Figura 22: Representación de un modelo con albedo negro y diferentes valores de m.

3 Técnicas de simulación de rugosidad

3.1 *Bump Mapping*

El *bump mapping* [11] es una técnica de simulación de rugosidad muy utilizada que permite modificar las normales de un modelo empleando para ello una textura que contenga información empaquetada sobre las normales, como la que se muestra en la figura 23, de manera que un modelo pueda aparentar ser más complejo de lo que en realidad es sin aumentar el número de triángulos que lo componen. Esta textura se llama *normal map* o *bump map*. En la figura 24 izquierda podemos observar una reflexión normal sobre una superficie plana, en el centro qué es lo que pasa cuando existe una alteración sobre esa superficie, y finalmente en la derecha cómo se consigue imitar esa alteración sin que ésta realmente exista, sólo modificando el vector normal $\vec{n}$ por el modificado $\vec{n'}$.

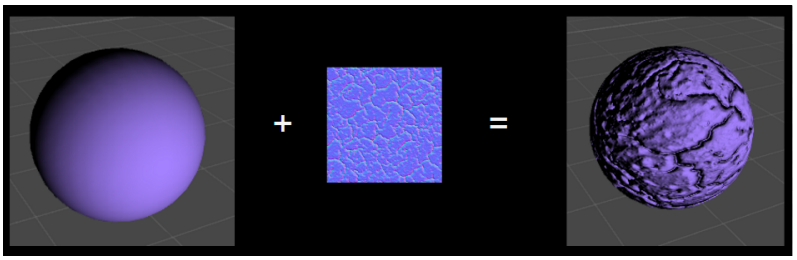


Figura 23: Muestra del resultado de aplicar una textura de normal sobre una superficie.

Para realizar esta técnica es necesario, en primer lugar, tener un *normal map* del que poder extraer la información sobre las normales. Estas normales descritas dentro de la textura tienen dos limitaciones importantes. La primera de ellas es que los valores de las componentes están comprendidos en el rango $[0, 1]$ pues es una textura RGBA, mientras que en un vector normal el rango está en $[-1, 1]$ debido a que estos vectores pueden apuntar en cualquier dirección y están normalizados, con lo que es necesario cambiar el rango. Esto se consigue multiplicando los componentes de la textura por 2 para pasar de [0,1] a [0,2] y luego restar uno para acabar con el rango deseado.

$$\vec{n'} = 2\vec{n'} - 1 \tag{3.1}$$

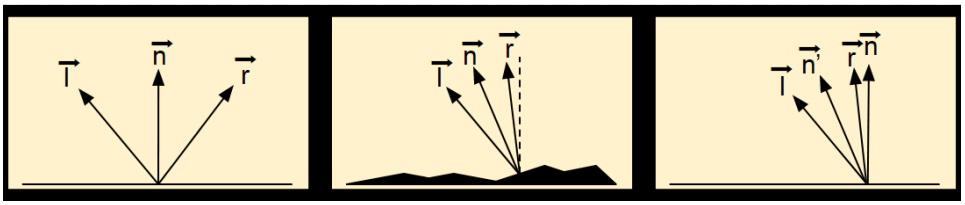


Figura 24: Representación del resultado de aplicar bump mapping sobre una superficie plana.

La segunda limitación es que los *normal maps* sólo tienen en cuenta dos componentes, que son los desplazamientos de la normal empaquetada en las direcciones de los vectores $\vec{u}$ y $\vec{v}$, que son los vectores que forman la textura bidimensional, por tanto es necesario explicar que el paso anterior sólo se aplica, considerando a $\vec{n'}$ un vector tridimensional, a las primeras dos componentes. Se necesita por tanto la componente de profundidad que en este caso corresponde a $\vec{n'_z}$. Esta componente se puede calcular una vez se tienen los componentes x e y. Debido a que el vector final $\vec{n'}$ tiene que estar normalizado, se sabe que su magnitud total tiene que ser uno. Por lo tanto se puede asumir,

$$n'^2_x + n'^2_y + n'^2_z = 1 \tag{3.2}$$

$$n'^2_z = \sqrt{1.0 - n'^2_x + n'^2_y} \tag{3.3}$$

Para tratar de ahorrar tiempo de cálculo también se puede realizar la aproximación,

$$n'_z = 1.0 - 0.5 \cdot (n'^2_x + n'^2_y) \tag{3.4}$$

Una vez superadas estas dos limitaciones, $\hat{n'}$ representa un vector tridimensional definido dentro del espacio de coordenadas formado por la base $\{(1,0,0),(0,1,0),(0,0,1)\}$, al que se llamará espacio local. Es este espacio en el que se define $\hat{n'}$, por lo tanto, uno diferente al que se va a utilizar para realizar los cálculos de la iluminación más tarde y por consiguiente debe ser transformado. El primer paso es por tanto la creación de una matriz de cambio de base M con la que poder realizar la siguiente operación,

$$\vec{n'_{mundo}} = M \cdot \hat{n'} \tag{3.5}$$

Donde $\vec{n'_{mundo}}$ es el vector normal en coordenadas de mundo y $\hat{n'}$ es la normal codificada en la textura con la tercera componente calculada según 3.4.

Esta matriz se construye a partir de los vectores $\hat{t}$, $\hat{b}$ y $\hat{n}$ en coordenadas de mundo, los cuales son, respectivamente, el vector tangente, el vector binormal y el vector normal de la superfície real, todos ellos forman un sistema de coordenadas llamado espacio tangencial, representado en la figura 25. Como estos tres vectores son ortonormales entre sí, sólo es necesario conocer dos de los tres vectores, ya que el tercero se puede conseguir mediante un producto vectorial, en este caso,

$$\hat{b} = ||\hat{t} \times \hat{n}|| \tag{3.6}$$

en que $\hat{t}$ es el vector tangente a la superfície y $\hat{n}$ el vector normal.

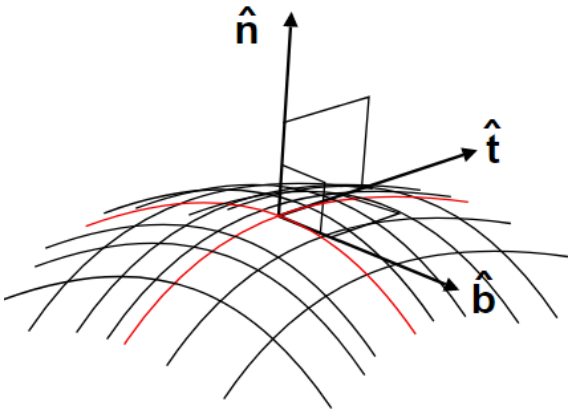


Figura 25: Representación del espacio tangencial.

Esta base es la que define el sistema de coordenadas del espacio tangente, ya que el plano formado por los vectores $\hat{t}$ y $\hat{b}$ corresponde al plano tangente a la superfície en el punto que se evalúa.

La matriz M se escribe,

$$M = \begin{pmatrix} t_x & b_x & n_x \\ t_y & b_y & n_y \\ t_z & b_z & n_z \end{pmatrix} \tag{3.7}$$

Sin embargo, aunque esto sea válido, se puede mejorar teniendo en cuenta la implementación que se hará más tarde. Utilizando la propiedad de las matrices,

$$(AB)^t = B^t A^t \tag{3.8}$$

es posible generar la matriz de cambio de base de una forma más eficiente, es decir, en vez de crear la matriz M de la forma,

$$M = \begin{pmatrix} t_x & t_y & t_z \\ b_x & b_y & b_z \\ n_x & n_y & n_z \end{pmatrix}^T \tag{3.9}$$

se puede conseguir el vector $\vec{n'_{mundo}}$ de la siguiente forma:

$$M^t = \begin{pmatrix} t_x & t_y & t_z \\ b_x & b_y & b_z \\ n_x & n_y & n_z \end{pmatrix} \tag{3.10}$$

$$\vec{v_{mundo}}^t = \vec{v_{local}}^t \cdot M^t \tag{3.11}$$

siendo de esta forma más eficiente ya que en el código no es necesario realizar la traspuesta de la matriz formada por los vectores.

3.1.1 Implementación en Unity

En primer lugar se tiene que conseguir una textura de normal e importarla a Unity con el campo *"Texture Type"* que aparece en el inspector cuando se importa una textura marcado a *"Normal Map"*. En segundo lugar, en el *shader* se tiene que añadir una propiedad nueva con la que se pueda introducir una textura con la línea

$$\_NombreVariable(\text{"}NombreEnInspector\text{"}, 2D) = \text{"}bump\text{"}\{\}$$

en el campo de propiedades para más tarde declarar esta misma textura dentro del paso, que contiene tanto el *Vertex Shader* como el *Fragment Shader*. Para conseguir los valores de las normales contenidos en el *normal map* éste se tiene que muestrear como se hace normalmente utilizando la función *tex2D* del lenguaje de *shaders CG*. que devuelve el valor de la textura dependiendo de los valores uv (parámetro s) que se pasan.

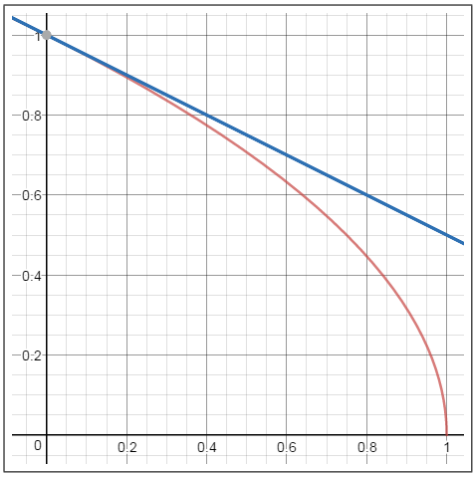
A la hora de muestrear la textura es necesario tener en cuenta que Unity utiliza un formato de compresión diferente para las texturas marcadas como *normal map* que se tiene que tener en cuenta en el código. Éste tipo de formato se llama *DXT5* y consiste en utilizar los canales g y a, que son canales con más precisión (6 y 8 bits respectivamente), para almacenar los valores que representan la normal ya que de esta manera la compresión afecta menos a la calidad. Por lo tanto, el valor perteneciente al vector u, normalmente asociado al canal r, pasa al canal a, mientras que el valor que corresponde al vector v se queda en el canal g.

Una vez cogida de la textura la muestra de los valores de la normal, se procede a realizar el *unpack* de la normal, que es el nombre con el que se conoce al procedimiento por el que el vector bidimensional comprendido por valores entre [0,1] pasa a ser un vector tridimensional con valores entre [-1,1], siguiendo los procesos explicados en las fórmulas 3.1 y 3.2. Respecto a las fórmulas 3.2 y 3.4, la figura 26b muestra las diferencias entre utilizar la fórmula real (izquierda) y la versión optimizada (derecha).

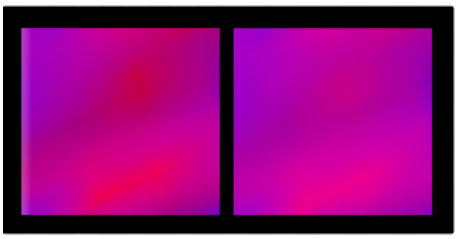
Existe también, además de estas, una tercera vía que consiste en crear una nueva variable en el inspector del material con la que se controle la profundidad manualmente. Este valor se tendrá que multiplicar por los valores muestreados del *normal map*. Esta es la opción que se ha escogido para este proyecto ya que permite al usuario final tener más control sobre esta técnica.

Una vez pasado el proceso de desempaquetado de las normales de la textura, se tiene que conseguir la matriz M^t explicada en 3.10, pero para ello es necesario antes contar con los vectores $\hat{t}$, $\hat{b}$ y $\hat{n}$.

Para conseguir estos vectores en Unity hay que añadir en la estructura de entrada al *Vertex Shader* dos campos: uno con la palabra clave o *semantic NORMAL* y otro con *TANGENT*. Estos dos campos devuelven dentro dentro de la estructura de



(a) En rojo la fórmula original (3.2).En azul la fórmula aproximada (3.4).



(b) Comparación del resultado aplicando la fórmula exacta (izquierda) y la aproximación (derecha).

entrada al *Vertex Shader*, gracias a haber indicado estas palabras, respectivamente el vector normal y el vector tangente en coordenadas de objeto.

La transformación de coordenadas de tangente al punto del objeto se calculará dentro del *Vertex Shader*, es decir, por cada vértice se calculará $\hat{t}$, $\hat{b}$ y $\hat{n}$ en coordenadas de mundo. En el caso de los vectores tangente y normal la transformación se consigue multiplicando por la matriz *unity_ObjectToWorld* y *unity_WorldToObject* respectivamente, siendo la normal diferente debido a que ésta es ortogonal a la superfície. En vector binormal, como ya se ha explicado, se consigue una vez $\hat{t}$, y $\hat{n}$ ya se encuentran en coordenadas de mundo, a través del producto vectorial entre estos dos, calculado con la función *cross*.

En la estructura de salida del *Vertex Shader* al *Fragment Shader* se añadirán tres campos utilizando tres *semantics* *TEXCOORDN*, donde N es un número entero de entre 0 y 16, que servirán para introducir los tres vectores necesarios para crear el sistema en coordenadas del espacio tangente. La palabra clave *TEXCOORD* es la que se utiliza para cualquier tipo de información que se quiera interpolar.

Dentro del *Fragment Shader* es necesario volver a normalizar estos tres vectores para evitar problemas con la interpolación que tiene lugar entre los dos programas.

El código que se utiliza para este método se encuentra dentro del fichero *"includes/BumpCommon"* en la función *GetBumpMappedNormals*.

3.1.2 Integración en la escena

A la hora de introducir *bump mapping* en la escena se han tenido que generar texturas de normal utilizando el programa *Crazybump* [2]. En el caso de este *shader* se ha decidido que la profundidad del *bump mapping* manualmente a través de una variable que oscila internamente entre 0 y 2. Contendrá una propiedad de tipo *Color* para teñir el modelo, dos propiedades para introducir texturas, una que hará de albedo y otra que servirá para la textura de normales. Además se ha creado un

campo de tipo *Float* para escoger el nivel de especularidad en el objeto.

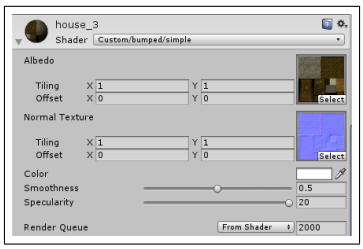


Figura 27: Interfaz del material al que se ha aplicado el *shader*.

Figura 28: Bump mapping con diferentes valores de pulidez (smoothness) sin texturas.

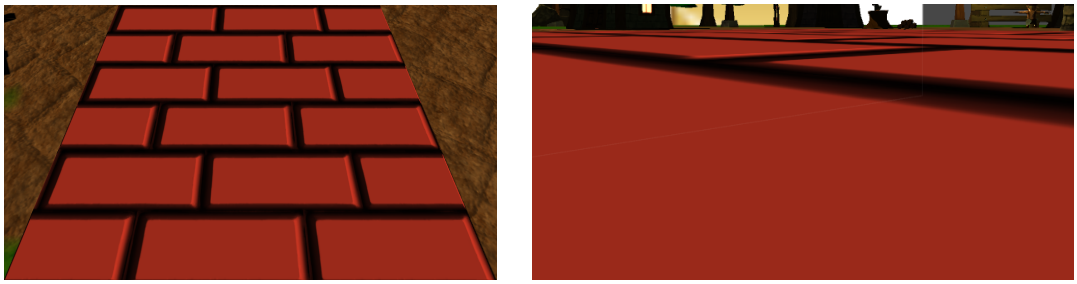
Figura 29: Bump mapping con diferentes valores de pulidez (smoothness) con textura.

Una vez creado el *shader* se crean materiales para cada objeto sobre el que se vaya a utilizar ésta técnica.

En las figuras 28 y 29 se puede visualizar el resultado del *shader* creado sobre el material de una casa de la escena.

3.2 *Parallax mapping*

El *bump mapping* es una técnica muy utilizada ya que es una forma poco costosa computacionalmente de mostrar rugosidad, sin embargo, presenta algunos problemas como por ejemplo cuando se mira un objeto o una superfície con un ángulo cercano a 0°, como es el caso de la figura 30b. Esto se debe a que el *bump mapping* no tiene en cuenta el vector $\vec{v}$, la dirección en la que mira el observador, en ningún momento y por lo tanto al mirar de cerca una superfície que implementa esta técnica se observa cómo la superfície no contiene en realidad ningún tipo de rugosidad.



(a) Plano con ladrillos y *bump mapping* visto desde arriba.

(b) Plano con ladrillos y *bump mapping* visto desde abajo.

Figura 30: Muestra de un plano con *bump mapping* desde diferentes ángulos.

La técnica *parallax mapping* [12] intenta evitar este problema alterando las coordenadas de una textura en función de la dirección en la que mire el observador. En la figura 31 se muestra cómo un observador, al intentar mirar el punto A debería ver el punto B, que a su vez representa el punto B' cuando se proyecta en la textura. Por lo tanto, lo que se persigue con esta técnica no es otra cosa que desplazar esta textura siguiendo el vector $\vec{O}$ representado, que se calcula de la siguiente forma,

$$O_x = \tan\theta_x \cdot height(B') \tag{3.12}$$
$$O_y = \tan\theta_y \cdot height(B') \tag{3.13}$$

Sin embargo, para conocer la altura en el punto B' de la textura, es necesario conocer previamente dónde se cruza el vector $\vec{v}$ con la nueva superfície representada en la figura por el color marrón. Para conseguir la altura en B' se necesita conocer o bien el vector $\vec{O}$ o bien el punto B, y como no se puede conseguir ninguno de estos de una forma simple es necesario hallar una aproximación.

Dado que las distancias entre los puntos colindantes que se evalúan de una textura son muy pequeñas, se puede suponer que la variación de las alturas almacenadas en esos puntos no es demasiado exagerada. Por consiguiente, una de las opciones más viables es la de suponer que la longitud de $\vec{P}$ es igual a la altura del valor en el punto A del *height map*. De manera que las ecuaciones 3.14 pasan a ser:

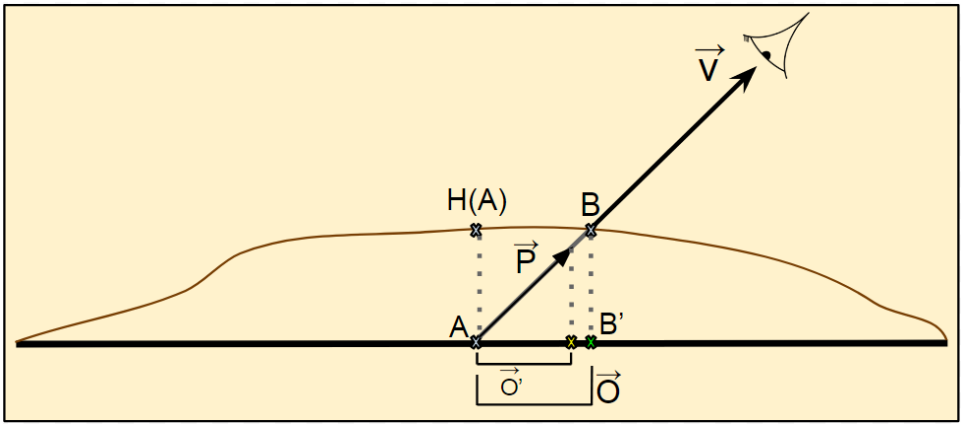


Figura 31: Esquema del funcionamiento del *parallax mapping*.

$$\vec{P} = height(A) \cdot \hat{v} \tag{3.14}$$

$$O'_{xy} = \vec{P}_{xy} \tag{3.15}$$

Es por esta razón que en la figura 31 el vector $\vec{P}$ no va del punto A al punto B, sino que va del punto A a un punto cercano a B debido a que el vector $\vec{P}$ se ha construido con la dirección de $\vec{v}$ y la altura almacenada en el punto A de la textura. Por este motivo el *parallax mapping* no tolera cambios de altura muy drásticos, ya que cuando esto sucede, la distancia real y la aproximada de $\vec{P}$ varian demasiado. Esto representa un problema en el resultado final que se intenta corregir siguiendo algunas aproximaciones que se explicarán más adelante.

Otro de los problemas es el que representa la propia rotación de la superfície del objeto y, por lo tanto, es necesario evitar problemas relacionados con la curvatura de ésta. La solución pasa por realizar los cálculos dentro de un sistema de coordenadas que, a diferencia del sistema de coordenadas de objeto o de mundo, tenga en cuenta una posible rotación diferente para cada punto de la superfície. Este sistema es el espacio tangencial, explicado en el apartado del *bump mapping*. Como ya se ha explicado, los vectores que forman la base son $\{\hat{t}, \hat{b}, \hat{n}\}$, que corresponden respectivamente a los ejes $\vec{x}$, $\vec{y}$ y $\vec{z}$ tal y como se muestra en la figura 25. La matriz que transporta la información a espacio tangencial tiene que estar, por tanto, formada por los vectores que forman la base $T = \{\hat{t}, \hat{b}, \hat{n}\}$ en la superfície. La transformación se realiza de mediante las siguientes operaciones:

$$M = \begin{pmatrix} t_x & t_y & t_z \\ b_x & b_y & b_z \\ n_x & n_y & n_z \end{pmatrix} \tag{3.16}$$

$$\vec{v_{tangent}} = M \cdot \vec{v_{obj}} \tag{3.17}$$

De esta manera se realiza un producto escalar entre $\hat{t}$, $\hat{b}$, $\hat{n}$, los tres expresados

en coordenadas de objeto, y $\vec{v_{obj}}$, que da como resultado el vector $\vec{v_{tangent}}$. Este proceso es más sencillo de comprender si se observa la figura 32.

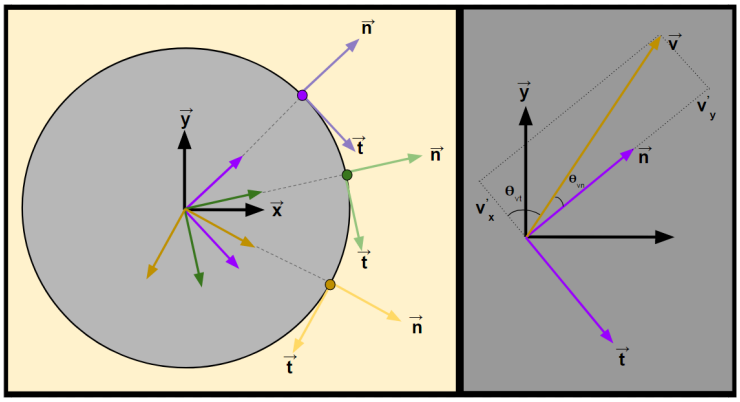


Figura 32: Esquema del paso a espacio tangencial en el *parallax mapping*.

En la imagen de la izquierda se observa cómo el espacio tangencial tiene en cuenta la curvatura de la superfície ya que va rotando en cada punto. En la imagen de la derecha están representadas dos bases: $S = \{\hat{x}, \hat{y}\}$ y $T = \{\hat{t}, \hat{n}\}$, en una simplificación en dos dimensiones. La base S corresponde con la base formada en coordenadas de objeto, mientras que la base T se corresponde con la base en la superfície a partir de los vectores $\hat{t}$ y $\hat{n}$. Además de estas bases está representado el vector $\vec{v}$. Teniendo en cuenta que los vectores que forman las bases están normalizados,

$$v'_x = ||\vec{v}|| \cdot cos(\theta_{vt}) \tag{3.18}$$
$$v'_y = ||\vec{v}|| \cdot cos(\theta_{vn}) \tag{3.19}$$

El vector $\vec{v'}$ es el vector $\vec{v}$ representado en espacio tangencial.

Para realizar esta técnica en primer lugar es necesaria una textura especial, llamada *height map*, que representa en escala de grises la altura de cada punto de la textura, como es por ejemplo la que se muestra en la figura 33, donde el color blanco o valor 1 representa la máxima altura y el color negro o valor 0, el mínimo. También es necesario una propiedad con la que indicar el factor de la altura, sin embargo en este proyecto, como se utilizan las dos técnicas explicadas en esta sección, simplemente se utiliza la misma propiedad *_Bumpiness* utilizada para el *bump mapping*.

Dentro del *Vertex Shader* se debe calcular el vector $\vec{v}$ en coordenadas tangenciales y para ello es necesario formar antes la matriz M. Como ya se ha explicado anteriormente, los vectores que forman esta matriz son $\hat{t}$, $\hat{b}$ y $\hat{n}$ en coordenadas de objeto y por lo tanto el vector $\vec{v}$ también debe estarlo. Dentro del *Fragment Shader* se utilizará este vector, así que es necesario que pase por el interpolador dentro de la estructura que sale del *Vertex Shader*.

La función que calcula las nuevas coordenadas de la textura según el método expuesto arriba es *CalculateBareParallaxTexcoords* y se encuentra en el archivo

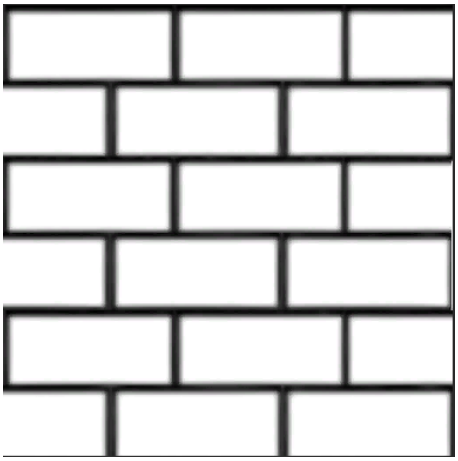
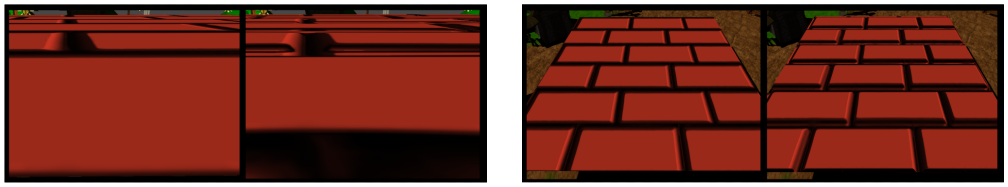


Figura 33: Textura utilizada para realizar *parallax mapping*.

'includes/UtilsBump.cginc'. Dentro de esta función lo que se hace es mapear la altura de la nueva superfície en el height map, para luego multiplicarla como se muestra en 3.14 por las tangentes de las componentes x e y respectivamente, que se calculan de la siguiente forma,

$$\tan\theta_{xy} = \frac{\vec{v}_{xy}}{\vec{v}_z}$$

Si se modifican las coordenadas de las texturas con el vector obtenido se completa el efecto del parallax. Los resultados obtenidos mediante este método en el terreno del proyecto se muestran en la figura 35.



(a) (b)

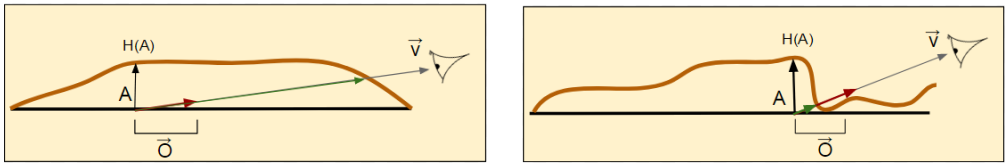
Figura 34: Plano sólo con *bump mapping* (izquierda). Plano con *bump mapping* y *parallax mapping* (derecha). Visto desde diferentes ángulos.

3.2.1 Mejoras respecto del *parallax mapping* convencional

Las mejoras al proceso convencional de *parallax mapping* pasan, como ya se ha explicado en el apartado anterior, por intentar aproximar la longitud del vector $\vec{P}$, que es el vector que va desde el punto de la textura A al que el observador está realmente mirando, que es el punto en la nueva superfície que el observador debería ver, es decir, B. Como ya se ha explicado, algunos de los problemas aparecen

cuando el observador mira la superfície desde un ángulo cercano a 0° o cuando la superfície que se intenta simular contiene cambios de altitud muy bruscos.

Los dos problemas están relacionados con que el vector $\vec{P}$ no sea más que una aproximación. El primero de los problemas está representado en el esquema de la figura 35a. Se puede intuir que cuánto menor es el ángulo entre la superfície y el observador, más fácil es que la aproximación del vector $\vec{P}$ tenga una magnitud demasiado corta. El segundo problema, producido cuando la transición de altura es demasiado rápida, está representado en el esquema de la figura 35b y se puede evitar utilizando *height maps* que tengan transiciones largas entre los colores negro y blanco.



(a) Observador demasiado cerca de la superfície.

(b) Transición de altura demasiado rápida.

Figura 35: Esquemas en los que se muestran los errores inherentes a la aproximación de $\vec{P}$.

Todas las soluciones a estos dos problemas implementadas pasan por intentar aproximar lo mejor posible la magnitud del vector $\vec{O}$, que representa el desplazamiento en la textura.

La primera aproximación lleva por nombre *Steep Parallax Mapping*[13]. Consiste en dividir la altura del mapa en n secciones de altura dh, como muestra la figura 36, para más tarde y empezando desde la sección de altura 1.0 comprobar entre qué secciones intersectan el vector $\vec{v}$ y la superfície almacenada dentro de la textura.

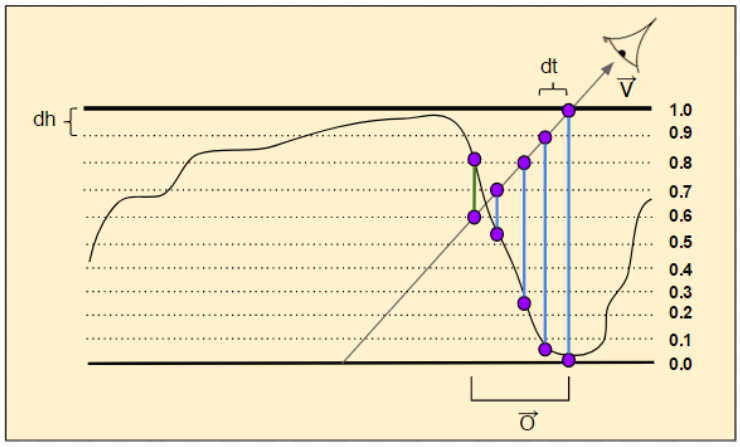


Figura 36: Esquema del *steep parallax mapping*.

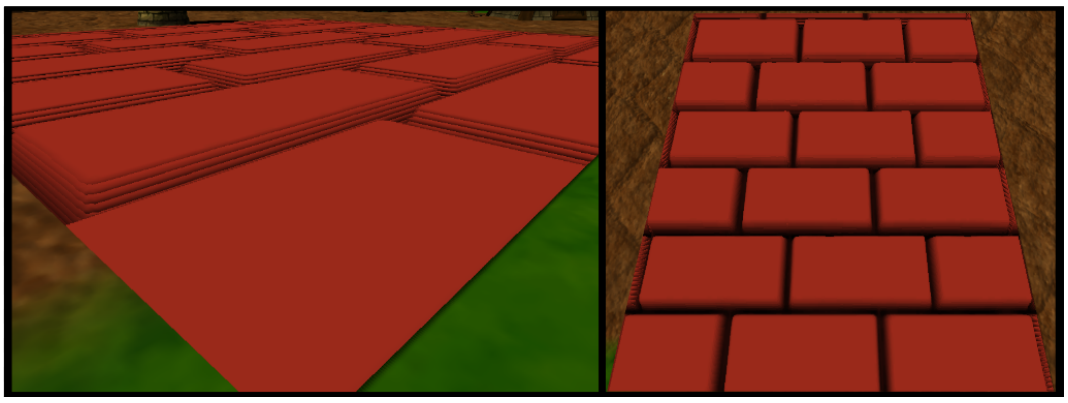
La superfície real en este caso es el plano representado por la sección de altura 1.0. A diferencia del *parallax mapping* básico la superfície simulada no está visualmente por encima de la superfície sino que la altura contenida en la textura representa

la profundidad. Para realizar *steep parallax mapping* se parte de las coordenadas de la textura en el punto que se analiza de la superfície real. A partir de ahí se tienen que conseguir los valores dh y dt, los cuales se van a utilizar para pasar de una sección a la siguiente en el caso de dh y para ir poco a poco incrementando las coordenadas de la textura en el caso de dt.

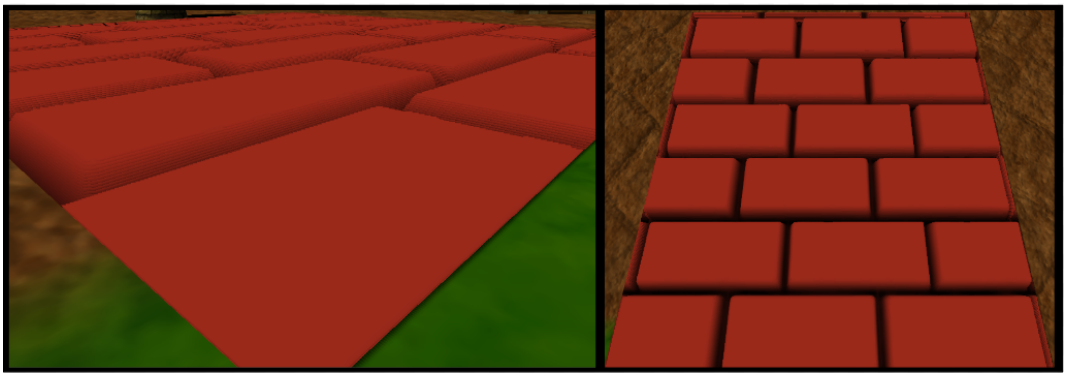
Una vez se tienen estos valores se inicia un bucle en el cual a cada iteración se avanza una sección y un paso dt en las coordenadas de la textura. Este bucle va actualizando tanto la altura de la sección como la altura de la superfície simulada y termina cuando la superfície simulada se encuentra por encima de la altura de la sección, ya que eso significa que la superfície simulada se ha encontrado con el vector $\vec{v}$ entre la sección anterior y la actual.

El resultado de este método son las coordenadas de la textura que salen del bucle, que son el resultado de sumar m veces el valor dt a las coordenadas de la textura originales, donde m es el número de secciones por las que se ha pasado antes de que finalizase el bucle.

Los resultados de aplicar *steep parallax mapping* se muestran sobre el plano anterior en la figura 37.



(a) *Steep parallax mapping* con $n = 12$.



(b) *Steep parallax mapping* con $n = 32$.

Figura 37: Resultados de aplicar *steep parallax mapping* con diferentes valores de n.

Se puede observar cómo el efecto mejora cuanto más alta sea n. Esto se debe a

que cuanto más alta sea n más pequeña será la sección y por tanto más pequeño el error. Por supuesto aumentar n implica un coste computacional más elevado por lo que por lo general lo adecuado es mantenerlo entre 10 y 20. También se infiere de los resultados que cuando el ángulo entre $\vec{v}$ y la superfície es de entre 45° y 90°, como es el caso de las imágenes a la derecha en 37a y 37b, las diferencias entre 12 secciones y 32 secciones prácticamente no se perciben, mientras que en las imágenes de la izquierda, con un ángulo cercano a los 0°, las diferencias entre los dos valores de n son notables. En conclusión, el ángulo crítico, donde es más sencillo que existan perturbaciones visibles, se encuentra alrededor de los 0°.

Con el fin de solventar estos errores, se ha implementado otro método para tratar de aproximar de una forma más certera el valor real del vector $\vec{O}$. Este método lleva por nombre *occlussion parallax mapping*[14] y consiste en una extensión del algoritmo utilizado por el *steep parallax mapping*.

Consiste en lo siguiente. Una vez se conocen las secciones entre las cuales se encuentra el punto de la superfície simulada que intersecciona con el vector $\vec{v}$ a partir del algoritmo del *steep parallax mapping*, se intenta aproximar a partir de una interpolación lineal el punto exacto en que se encuentra la intersección. Esta interpolación lineal se realiza utilizando como valor de peso la diferencia existente entre la altura de las secciones y su correspondiente altura en la superfície representada por el height map, tal y como muestra la figura 38.

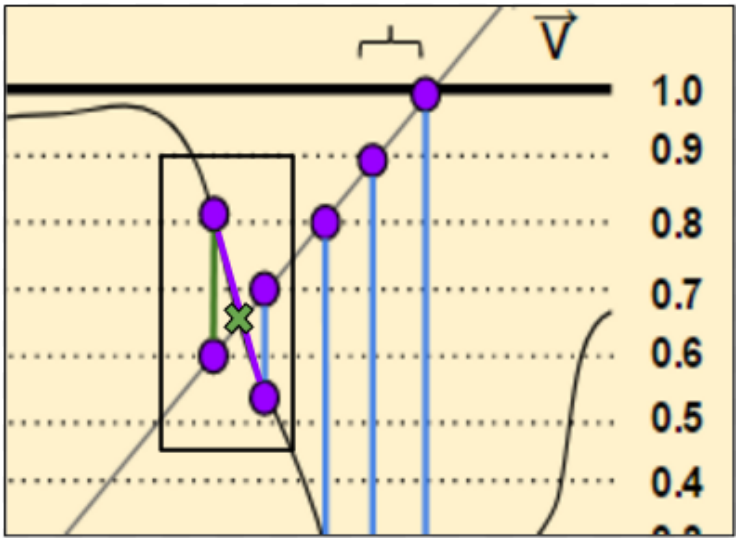
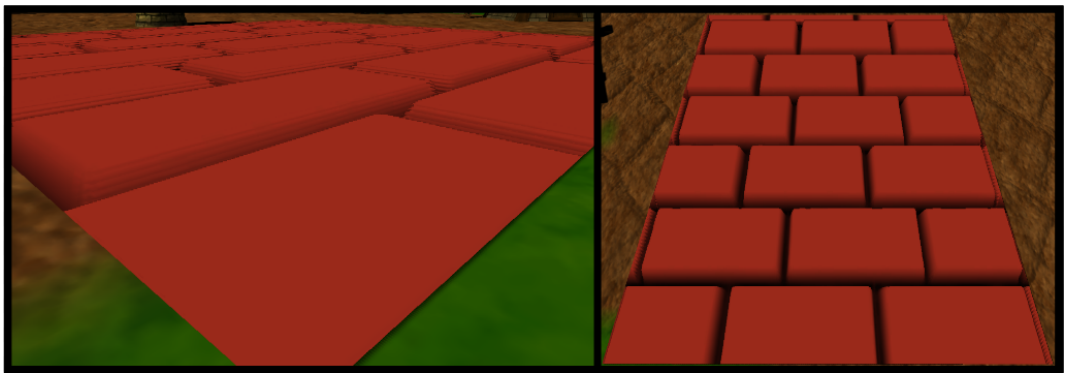
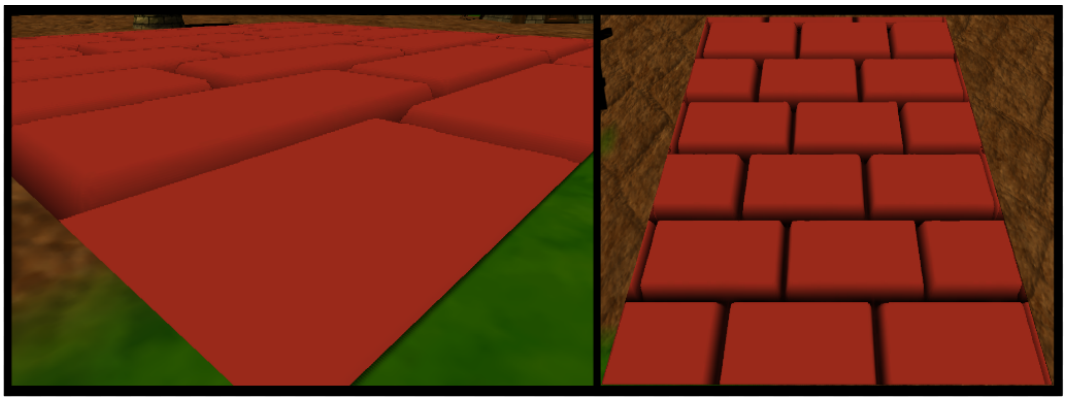


Figura 38: Esquema del *occlussion parallax mapping*.

En las siguientes comparaciones se observa cómo con el mismo valor de n las capas no se perciben tanto como en el caso del *steep parallax mapping* en los ángulos críticos.



(a) *Oclussion parallax mapping* con $n = 12$.



(b) *Occlussion parallax mapping* con $n = 32$.

Figura 39: Resultados de aplicar *occlussion parallax mapping* con diferentes valores de n.

4 Transparencias

Otra propiedad perteneciente a algunos materiales y que Unity permite implementar son las transparencias. Son transparentes aquellos materiales que permiten ver otros objetos situados detrás de su material, como por ejemplo un vaso de vidrio o una botella de plástico.

4.1 *Render Queues*

Una *Render Queue* es una cola en la cual están contenidos los *GameObjects* que esperan para ser pintados o renderizados. Unity dispone de varias *Render Queue* donde cada una tiene asociada un número entero que se encuentra incluido en el rango $[0, 5000]$. El número asociado a la Render Queue indica el orden en que va a ser renderizada ésta, siendo los primeros los numeros más bajos y los últimos los más altos. La figura 40 muestra las diferentes Render Queues (bolsas) de las que Unity dispone con sus valores correspondientes. Cada Render Queue contiene los materiales que se han asociado con ella por lo que, como se puede intuir, los *GameObjects* pertenecientes a la cola *Background* se pintarán antes en la escena que el resto, y los últimos serán los pertenecientes a la cola *Overlay*.

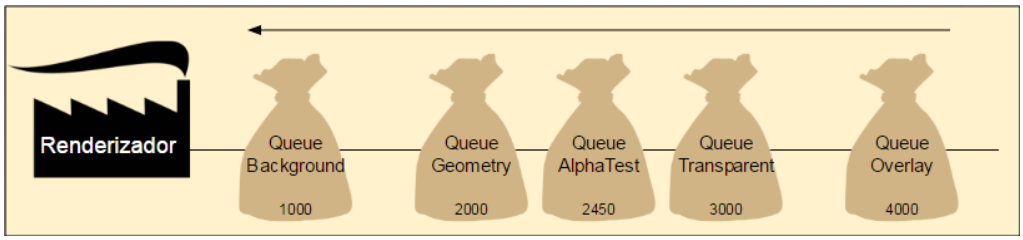


Figura 40: Esquema de la ordenación de las Render Queues en el proceso de renderización.

Dentro de un *shader* de Unity se puede indicar dentro de qué *Render Queue* debe estar incluido el material mediante la etiqueta *"Queue"= "RenderQueueName"*. Además es posible añadir números a las colas predefinidas. Por ejemplo, se puede crear un shader que tenga como número el valor de la cola *Transparent* más n, de manera que se pase al renderizador después de que el resto de materiales con *Transparent* ya estén pintados en la escena. Esto se consigue modificando levemente la etiqueta anterior, que quedaría de la siguiente forma: *"Queue"= "Transparent+5"* para $n = 5$.

4.2 Materiales transparentes

Los materiales puramente transparentes son aquellos que, o bien implementan transparencias totales o no las implementan. Es decir, o dejan ver por completo lo que hay detrás o son completamente opacos.

Para que un material pueda tener transparencias es necesario que pase por el renderizador después de que todos los objetos opacos ya estén en la escena. De esta manera lo único que tienen que hacer los shaders de los materiales transparentes es descartar fragments dentro del *Fragment Shader* mediante la instrucción *discard*. Con esta instrucción, el fragment no pasa por el renderizador, y por lo tanto se ve a través el objeto que haya detrás.

Sin embargo es necesario tener en cuenta a la hora de valorar el rendimiento de un *shader* que utilizar la instrucción *discard* en el *Fragment Shader* hace que algunas optimizaciones no se puedan realizar y por lo tanto el rendimiento puede bajar. Una optimización que se realiza es la del *Early Fragment Test*. Este test es un proceso entre el *Vertex Shader* y el *Fragment Shader* en el cual se descartan *fragments* en función de la profundidad respecto a la cámara, para evitar así tener que procesar un *fragment* que va a estar oculto por otro más cercano. Si se utiliza la instrucción *discard* ya no es posible asumir que un *fragment* pueda estar oculto por otro con menos profundidad ya que éste puede ser descartado más tarde en el *Fragment Shader*.

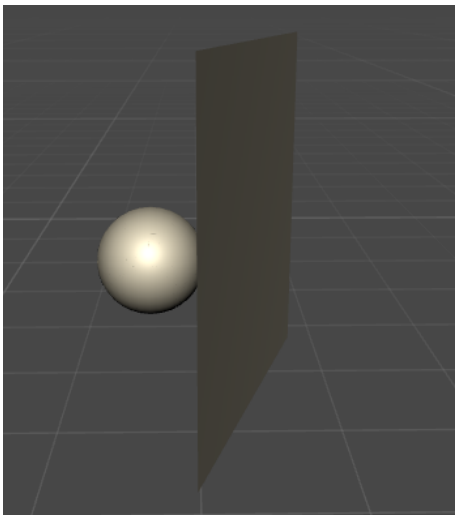
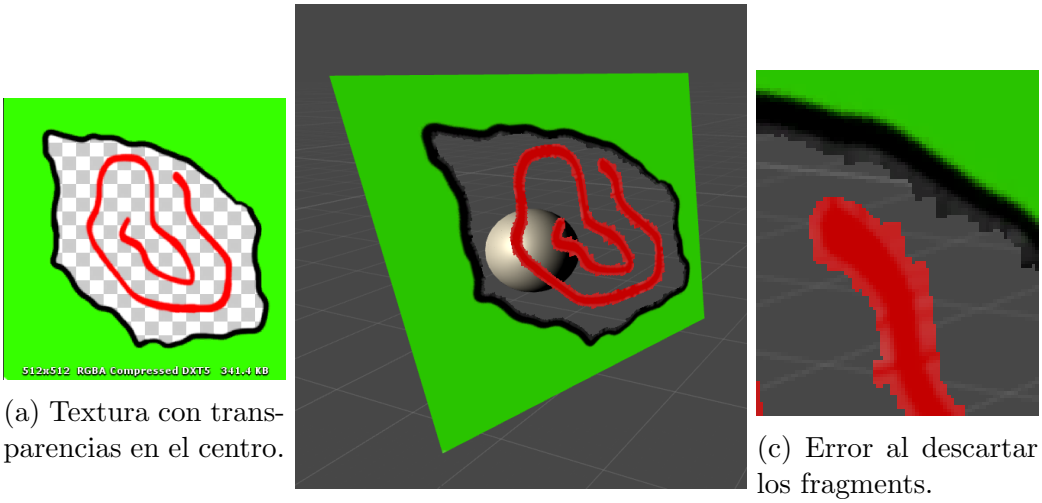


Figura 41: Escena con un plano y una esfera detrás.

Dada la escena representada en 41, se crea un *shader* que descarte aquellos *fragments* que coincidan con un valor de la textura en el canal alpha igual a 0. El resultado se observa en la figura 42

En 42c se puede ver cómo quedan pequeños rastros de fragments que no son transparentes y que sin embargo en la textura sí lo son. Esto se debe a que no todos los fragments que se observan transparentes en la textura tienen un valor exactamente igual a 0, sino que algunos tienen un valor cercano pero diferente. Esos pequeños errores corresponden a esos fragments, ya que al compararlos con 0 no van a ser descartados. Existe una solución a este problema, que pasa por utilizar la instrucción *AlphaToMask On*. Esta instrucción activa la utilización de una máscara de 0's y 1's, de forma que más tarde esta máscara se aplica contra los valores muestreados utilizando un operador *AND* y por lo tanto se descartan aquellas muestras a las que les toque un 0. En principio, esta máscara se construye



(a) Textura con transparencias en el centro.

(b) Plano con la textura aplicada.

(c) Error al descartar los fragments.

Figura 42: Resultado de aplicación de textura sobre *shader* con transparencias.

mediante un método llamado *Alpha-To-Coverage* que consiste en realizar varios muestreos dentro de la misma zona del fragment, *multisampling*, pero convirtiendo el valor original de la transparencia en una máscara de valores 0 y 1 de forma proporcional al valor original. Es decir, si el valor de la transparencia fuese 0, la máscara estaría únicamente compuesta por ceros, mientras que si, por otra parte, la transparencia es 1, toda la máscara estaría formada por 1's.

4.3 Materiales semitransparentes

En el apartado anterior se han explicado los materiales transparentes y cómo funcionan en Unity. En este apartado se van a explicar los materiales semitransparentes, que a diferencia de los puramente transparentes, que permiten que atraviese toda la luz, éstos absorven una parte, haciendo así que un objeto pueda mostrarse a sí mismo de la misma forma que a los objetos situados detrás de él.

Los materiales semitransparentes utilizan para conseguir ese efecto un método llamado *Alpha-Blending*, que consiste en realizar una mezcla entre el color del material semitransparente y el color que ya existiese en el mismo píxel, como muestra la figura 43.

Al igual que en el caso de los materiales transparentes, los semitransparentes también tienen que pasar por el renderizador cuando ya se han pintado en la escena todos los objetos opacos de las colas de números bajos. Sin embargo, éstos se tienen que añadir a la Render Queue *Transparent* ya que así pasarán al renderizador una vez ya hayan pasado los objetos con transparencias puras. Esto es importante por el siguiente motivo. Supongamos dos objetos donde uno de ellos, el objeto A, tapa al otro, B. Supongamos también que A tiene un material semitransparente mientras que B es transparente. Si A pasase antes que B por el renderizador, B aún no estaría en la escena y por lo tanto A, que implementa *alpha-blending*, no podría mezclar los colores de sus fragments con los de los fragments de las partes opacas de B. Un

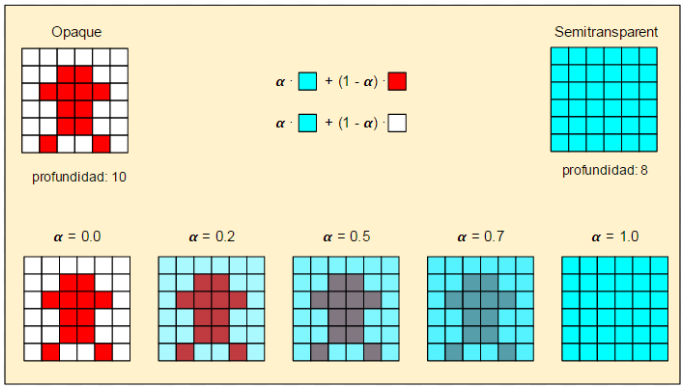
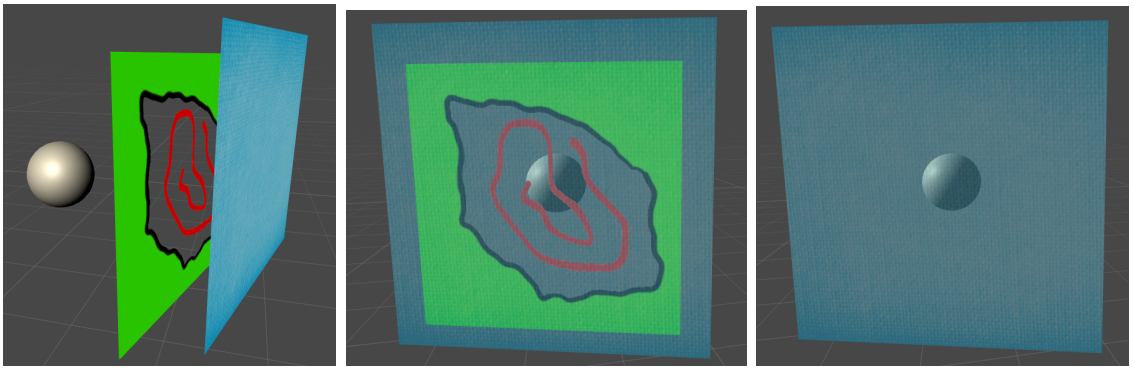


Figura 43: Explicación del proceso de *alpha-blending*.

ejemplo de este error se muestra en la escena representada en la figura 44a. En la figura 44b, el plano semitransparente azul pasa por el renderizador una vez ya ha pasado previamente el plano que tiene la textura verde transparente debido a que este último tiene un valor de *Render Queue* inferior. En la figura 44c, en cambio, estos valores se intercambian y por lo tanto es el plano semitransparente el que pasa antes por el renderizador. Al pasar antes el plano semitransparente, éste no puede realizar la mezcla con el plano transparente ya que aún no se ha pintado y por lo tanto, al estar al frente, el resultado es un plano semitransparente con un color único que anula completamente el plano que tiene detrás.



(a) Escena formada por una esfera (fondo), un plano transparente (centro) y un plano semitransparente (frente).

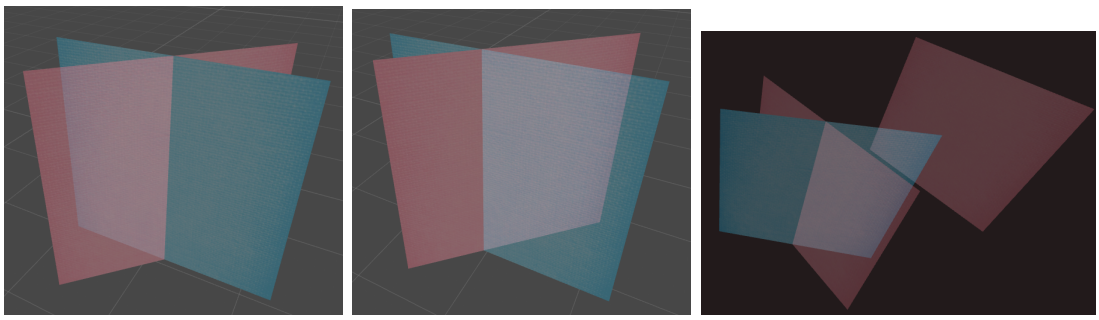
(b) Plano semitransparente azul con $\alpha = 0.5$ y en la cola *Transparent*(3000). Plano transparente en cola *AlphaTest*(2450).

(c) Plano semitransparente azul asociado a la cola *Geometry*(2000).

Figura 44: Muestra del resultado de que un objeto semitransparente esté asociado a una cola menor que otro objeto transparente.

Otro error que puede suceder cuando se utilizan materiales semitransparentes en Unity es que al cruzarse dos o más objetos que implementen semitransparencias, uno de los dos deje de transparentar al otro, como se puede ver en la figura 45. Cuando se trata de pintar dos objetos semitransparentes en la misma escena Unity ordena estos objetos basándose en la profundidad de su centro de manera que pasen

por el renderizador primero aquellos objetos más alejados y por lo tanto el orden es de atrás al frente. En este caso, los dos objetos tienen el mismo número de cola y el mismo nivel de transparencia, además de tener su centro en el mismo punto. Al estar cruzados, el objeto que se haya pintado antes (en este caso el azul) no tendrá en cuenta al siguiente que se pinte (el rosa) porque aún no existe en la escena y por lo tanto sólo aquél que se haya pintado segundo podrá dejar ver al otro plano. Esta falta de semitransparencia sólo afecta a uno de los planos respecto del otro, ya que como se muestra en la figura 45c en caso que apareciese otro plano semitransparente con un centro situado más allá de los planos cruzados el plano afectado sí que transparentaría al situado detrás.



(a) Escena formada por dos planos semitransparentes cruzados con el mismo valor de Render Queue.

(b) Muestra de cómo el centro de los objetos afecta la ordenación dentro de la Render Queue.

(c) El plano azul sigue siendo semitransparente con otros planos.

Figura 45: Errores al cruzar dos planos semitransparentes.

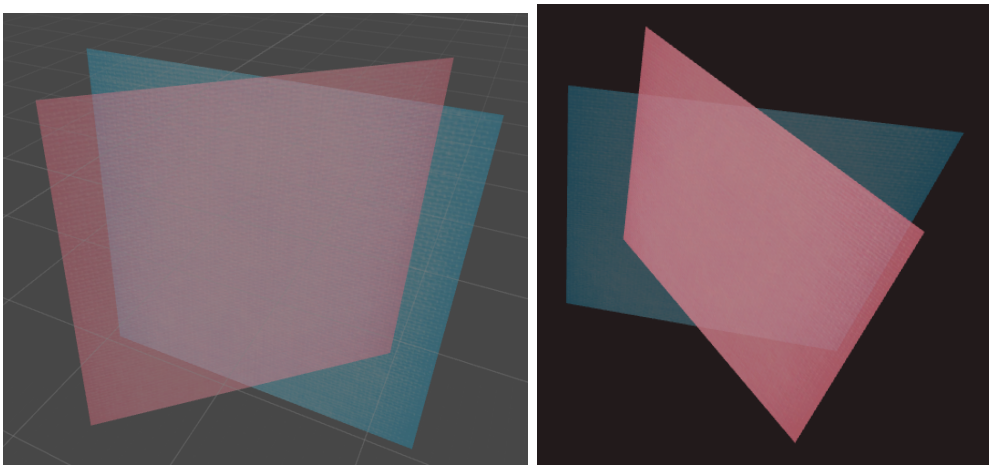
Una posible solución a este problema pasa por desactivar la escritura de la componente z por parte del objeto. De esta manera no se puede saber cómo ordenarlos dentro de la Render Queue y por lo tanto uno de los dos pasará primero y el siguiente quedará aparentemente por encima. Esto es por supuesto una solución poco aceptable normalmente, pero que en algunos casos concretos puede llegar a ser bastante práctica, como por ejemplo en un caso en que los objetos que presentan estos problemas se encuentren bastante alejados de la cámara.

4.4 *Alpha cutoff*

La técnica *Alpha cutoff* es una técnica que se aplica a materiales puramente transparentes y que consiste en poder recortar una textura aprovechando los valores del canal alpha de una textura. Habitualmente en Unity los materiales que aplican este método tienen un valor en la interfaz con nombre *Cutoff* que permite seleccionar un valor de corte o *threshold* por el que recortar la textura.

En la figura 47a se muestra una textura preparada para utilizar *Alpha Cutoff* aplicada sobre un plano, y en 47b el canal alpha de esa misma textura.

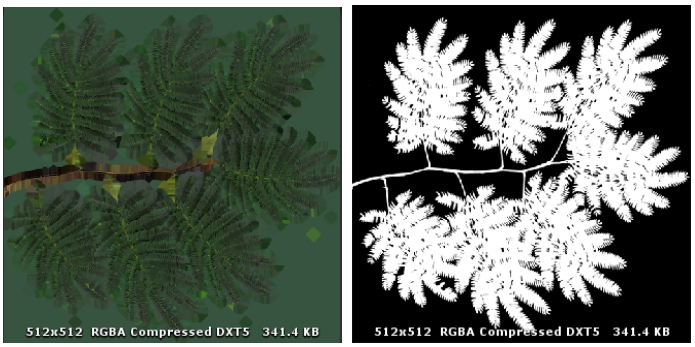
El proceso entonces es el siguiente. Dentro del *Fragment Shader* se resta el valor de corte (*Cutoff*) al valor del canal alpha de la textura. Esto para valores de corte



(a) Planos semitransparentes con la escritura de la componente z desactivada.

(b) En plano rojo está por encima del azul en apariencia.

Figura 46: Desactivación de la escritura de la componente z en los planos.



(a) Canales RGB de la textura.

(b) Canal alpha de la textura de 47a.

Figura 47: Textura propia de alpha cutoff.

superiores a 0 puede dar resultados negativos, por lo que se utiliza la instrucción *clip*, que tiene como entrada un valor de tipo float y que descarta un fragment si el valor recibido es menor que 0. Los resultados de aplicar este proceso están representados en la figura 48 donde se observa cómo a medida que aumenta el valor de corte un mayor número de fragments resultan descartados ya que hay más valores en la textura que restados al valor *Cutoff* dan un valor negativo. Hay que destacar que para que sea posible que vayan desapareciendo o apareciendo fragments a medida que se modifica el valor de corte tiene que haber una variedad de valores en el canal alpha de la textura.

(a) Cutoff = 0.0. (b) Cutoff = 0.1. (c) Cutoff = 0.5. (d) Cutoff = 1.0.

Figura 48: Resultados de aplicar alpha-cutoff con diversos valores.

5 Superfícies reflectivas y refractivas

5.1 Reflexiones

Los objetos del mundo real reciben luz de focos emisores como por ejemplo el Sol. A la hora de recibir esta luz, éstos absorben una parte dependiendo de sus propias características y emiten (o reflejan) el resto, siendo la luz reflejada el color que finalmente percibe el observador en el objeto. La luz emitida puede a su vez chocar con otras superfícies que en el caso de tener una alta capacidad para la reflexión, como por ejemplo un espejo o una superfície muy pulida de un material metálico, y hacer que se refleje de nuevo, con lo cual en el supuesto de que un observador recibiese esta luz reflejada, podría observar el objeto que ha emitido la luz a través de otra superfície reflectora. La figura 49 muestra una reflexión de la luz emitida por un objeto, en este caso una nube, representada con el vector $\vec{e}$ que se refleja en un espejo y llega al observador a través del vector reflejado $\vec{v}$.

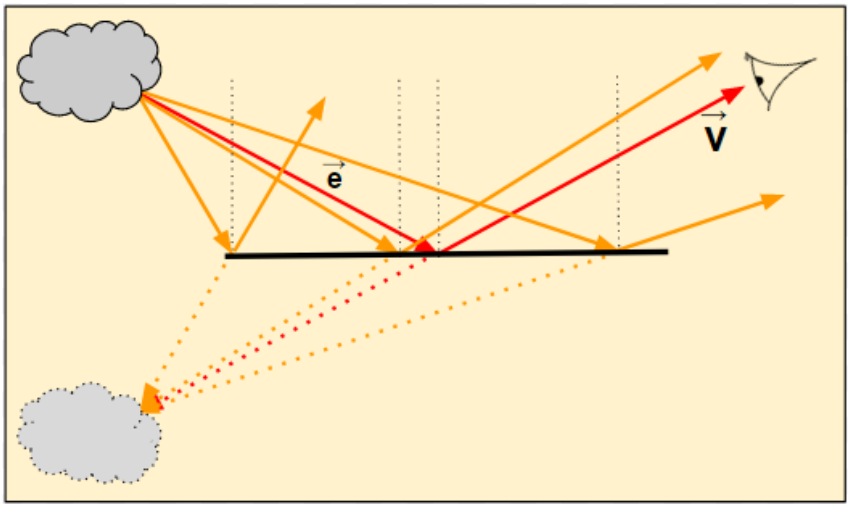


Figura 49: Esquema de las reflexiones sobre una superfície.

Explicado el funcionamiento de las reflexiones en el mundo real, se puede intuir la idea de generar reflexiones lanzando rayos desde el observador a la superfície, de manera que esos rayos se reflejasen y pudiesen devolver al observador el color de los objetos emisores. Esta idea se conoce como *RayTracing* pero representa un gasto computacional demasiado elevado ya que sería necesario lanzar un rayo por cada *fragment* de la superfície, por no hablar de que añadiría aún más complejidad analizar varias reflexiones en diversos objetos de una escena.

La solución que aporta Unity para realizar reflexiones pasa por la utilización de un método llamado *Environmental Mapping*. La técnica de *Environmental Mapping* permite, a partir de una geometría (por ejemplo un cubo o una esfera) con imágenes de toda la escena vista desde un punto, que un objeto pueda acceder a los colores que le envuelven en la escena. Para ejemplificar esto, se puede pensar en un cubo o *cubemap* con 6 texturas, una por cada cara. Un objeto puede muestrear las texturas presentes en ese *cubemap* aportando un vector tridimensional de forma

que la dirección de ese vector, con origen en el centro del cubo, atraviese una única muestra de la textura de una de las caras. La figura 50 muestra un ejemplo de *cubemap*.

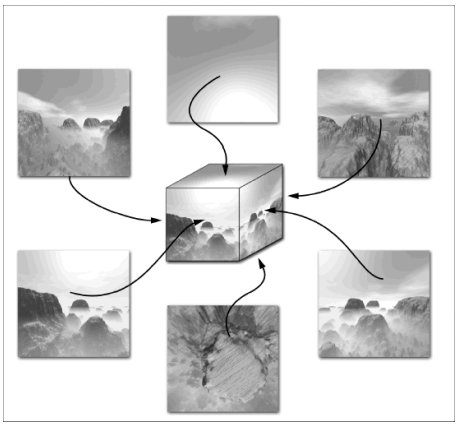


Figura 50: Partes de un *Cubemap*.

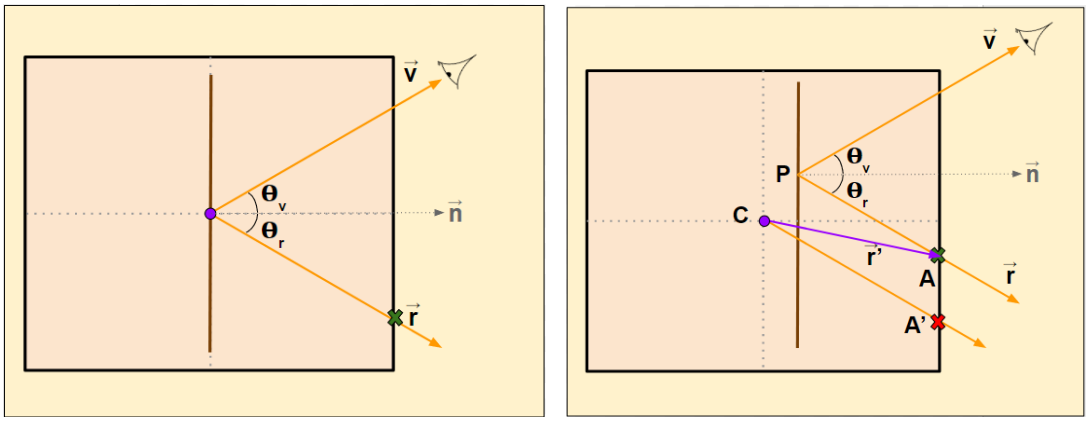
Sin embargo, la técnica del *Environmental Mapping* no implica que las texturas tengan que ser imágenes de la escena, sino que es sólo un medio para poder muestrear varias texturas en tres dimensiones. Lo cual significa que se necesita de un mecanismo que dado un punto de la escena y las medidas de un cubo, extraiga 6 texturas de la escena en las 6 direcciones y que además genere un *cubemap* con ellas. Ese mecanismo en Unity se llama *Light Probe*. Un *Light Probe* está compuesto por un punto central y una caja de influencia a su alrededor. Aquellos objetos que se encuentren dentro del área de la caja de influencia podrán utilizar los *cubemaps* que se generan en el *Light Probe*.

Estos dos elementos permiten que un objeto pueda mostrar la escena que le envuelve, sin embargo, no sólo basta con mostrar el entorno sino que debe cambiar en función de la dirección en la que mira el observador. La figura 51a simplifica a dos dimensiones los elementos básicos que forman una reflexión mediante *Environmental Mapping*. El cuadro exterior representa el *cubemap* generado por el *Light Probe* y la línea marrón la superfície reflectiva.

Teniendo en cuenta que $\theta_v = \theta_r$ se sabe que $\vec{r}$ es el vector $\vec{v}$ reflejado teniendo como eje el vector normal en el punto de la superfície. Por consiguiente, el vector que se debe utilizar para muestrear el *cubemap* es $\vec{r}$.

Este método tiene un problema que sale a la luz cuando se mueve el objeto del centro del *Light Probe*. A pesar de que el objeto se mueve, la posición que se utiliza para renderizar las imágenes que componen el *cubemap* es la misma, y por lo tanto la apariencia del objeto no varia. Dada la figura 51b que muestra un esquema del problema, se observa que el vector $\vec{r}$ tendría que ir a parar al punto A y en cambio va a parar a A'. Este problema se soluciona modificando el vector $\vec{r}$ de manera que llegue al punto A desde el centro.

Esto se consigue mediante la siguiente fórmula:



(a) Reflexión mediante *Light Probe* y *Environmental Mapping*.

(b) Error al realizar el muestreo sobre el *cubemap*.

Figura 51

$$\vec{r'} = d\hat{r} + (P - C) \tag{5.1}$$

donde $\vec{r'}$ es el vector $\vec{r}$ modificado, d es la distancia desde el punto P al punto A y $(P - C)$ es el vector que va desde el centro del cubo a la posición donde se encuentra el punto P en la superfície reflectora.

Para calcular d, antes es necesario saber la distancia entre la cara del cubo con la que primero intersecta el vector $\hat{r}$ y P, representada mediante las líneas discontínuas en la figura 52. En este caso la cara con la que primero se cruza es la de la derecha del cubo con lo cual el cálculo se puede realizar de la siguiente forma:

$$d_x = \frac{d_x}{cos(\hat{r}, \hat{x})} \tag{5.2}$$

$$d_y = \frac{d_y}{cos(\hat{r}, \hat{y})} \tag{5.3}$$

$$d = min(d_x, d_y) \tag{5.4}$$

En que los nombres de las variables son los que aparecen en la figura 52 y los vectores $\hat{x}$ e $\hat{y}$ son los vectores $(1, 0)$ y $(0, 1)$ respectivamente. Es posible utilizar $\hat{x}$ e $\hat{y}$ debido a que el cubo que delimita el área de influencia del *Light Probe* no se puede rotar.

Esto soluciona el problema de la posición del objeto dentro del *Light Probe*. Los *standard shaders*[6] de Unity utilizan esta aproximación para realizar reflexiones, pero aún existen fallos, ya que al tratarse de un cubo, las esquinas del mismo son una fuente de problemas. En la figura 53 se puede ver cómo la propia reflexión adopta la forma de un cubo en un espejo plano. Esto es debido a que en las esquinas de un cubo la distancia recorrida por ángulo es mayor lo que hace que se perciba un leve estiramiento alrededor de los bordes.

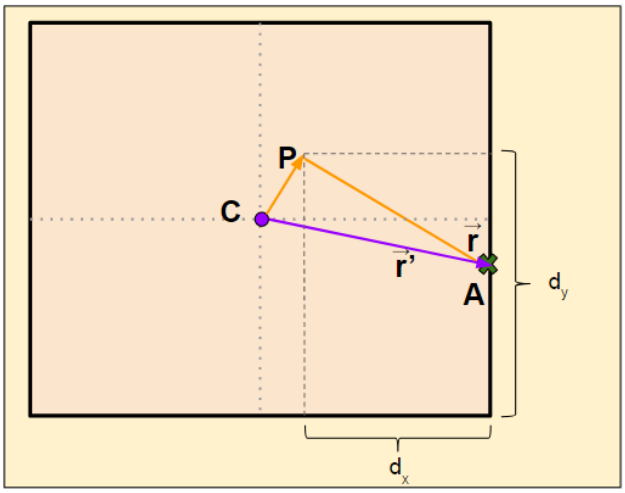


Figura 52: Esquema sobre el cálculo de $\vec{r'}$.

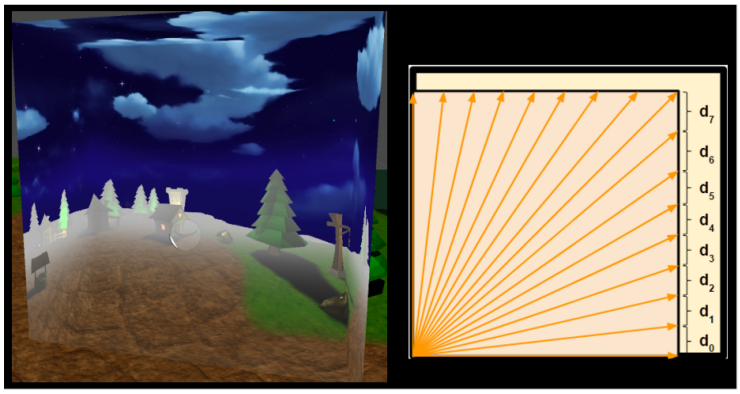


Figura 53: Muestra del error provocado por la geometría del cubo.

Para solucionar este problema se han implementado dos posibles alternativas, ambas basadas en el mismo principio: la única figura geométrica que no sufre un cambio de distancias por ángulo es la esfera.

La primera alternativa está presente en el esquema de la figura 54 y pasa por estirar el vector $\vec{r}$ utilizando una función cuadrática que cumpla las condiciones impuestas en 5.5. La función cuadrática se construye de la siguiente manera. Dado un valor x que comienza en cero en la parte inferior de la cara que se intersecta y es uno en la parte superior,

$$f(x) = ax^2 + bx + c \tag{5.5}$$

$$f(0) = c = 0 \tag{5.6}$$

$$f(\frac{1}{2}) = \frac{a}{4} + \frac{b}{2} = h \tag{5.7}$$

$$f(1) = a + b = 0 \tag{5.8}$$

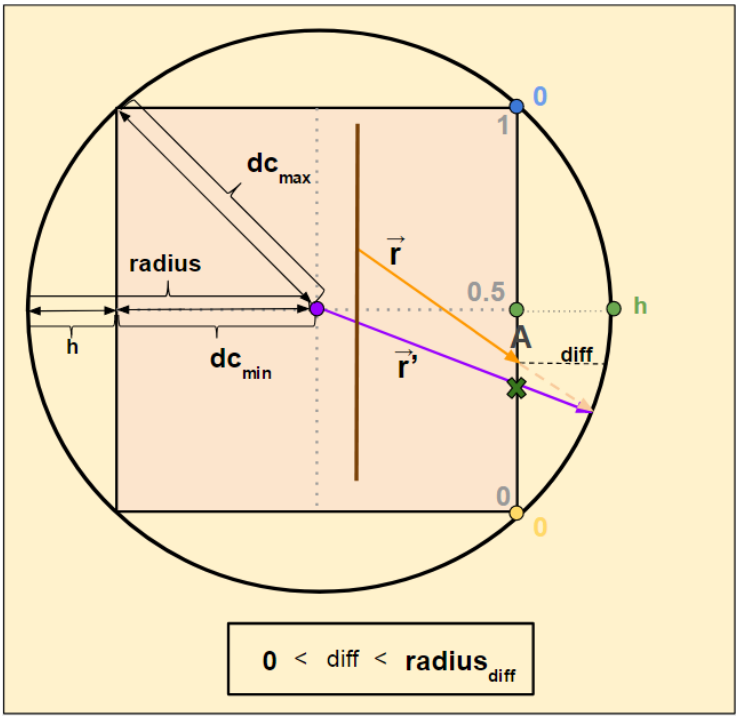


Figura 54: Esquema de la primera aproximación al problema de la apariencia de cubo de la reflexión.

Y resolviendo,

$$f(x,h) = -4hx^2 + 4hx = 4hx(-x+1) \tag{5.9}$$

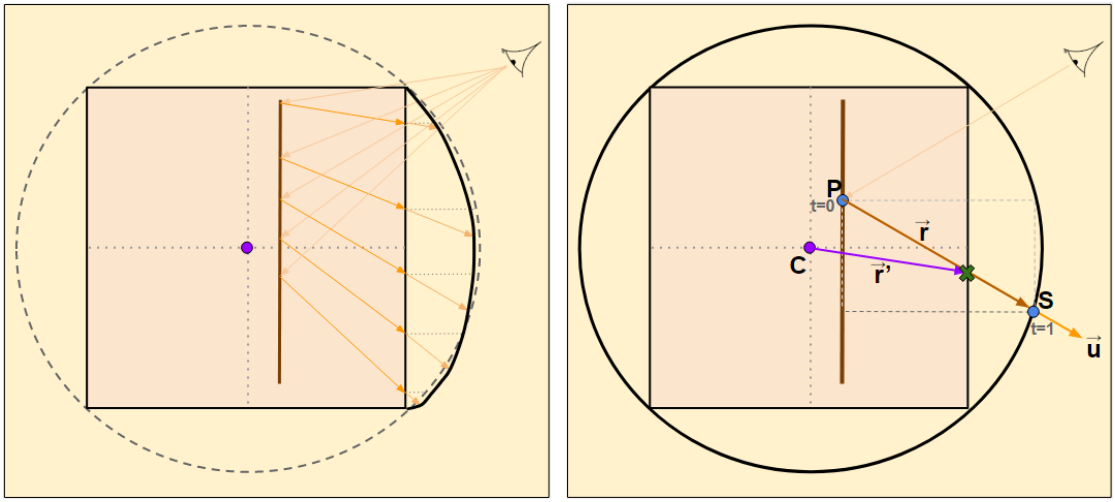
Esta aproximación se aplica con la precondición de haber encontrado previamente el punto A. Una vez conseguido esto, se comprueba en qué parte de la pared se encuentra B y ese valor es el que se pasa a la función f como x. Teniendo en cuenta que $radius = dc_{max}$, el valor de h se calcula haciendo la diferencia entre dc_{max} y dc_{min}. El valor obtenido es el que se utiliza más adelante para alargar el vector d con el que más adelante se calcula $\vec{r'}$ utilizando la fórmula 5.1.

Pero este método no es perfecto, ya que termina aproximando una esfera similar a la de la figura 55a.

Finalmente el segundo método y más exacto para simular una esfera a partir de un *cubemap* es utilizar la propia fórmula de la esfera,

$$(x - C_x)^2 + (y - C_y)^2 + (z - C_z)^2 = r^2 \tag{5.10}$$

siendo r el radio de la esfera y C el punto central de la misma. La figura 55 muestra que existe un vector $\vec{u}$ que va desde el punto P hasta el punto S, en la superfície de la esfera y cuyas componentes se definen de la siguiente forma,



(a) Muestra del esquema que representa el error en la superfície de la esfera aproximada.

(b) Esquema que muestra los componentes necesarios para el cálculo del mapeado esférico perfecto.

Figura 55

$$x(t) = (x_1 - x_0)t + x_0 \tag{5.11}$$
$$y(t) = (y_1 - y_0)t + y_0 \tag{5.12}$$
$$z(t) = (z_1 - z_0)t + z_0 \tag{5.13}$$

en que t es un parámetro limitado entre 0 y 1, donde si $t = 0$ el punto resultante es P y si $t = 1$ es S. Si se sustituyen los valores x, y y z en la fórmula 5.10 por $x(t)$, $y(t)$ y $z(t)$ y se resuelve se termina por obtener una función cuadrática en función del valor t que cumple lo siguiente:

$$a = (x_1 - x_0)^2 + (y_1 - y_0)^2 + (z_1 - z_0)^2 \tag{5.14}$$
$$b = 2(x_1 - x_0)(x_0 - x_c) + 2(y_1 - y_0)(y_0 - y_c) + 2(z_1 - z_0)(z_0 - z_c) \tag{5.15}$$
$$c = (x_0 - x_c)^2 + (y_0 - y_c)^2 + (z_0 - z_c)^2 - r^2 \tag{5.16}$$

$$t = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \tag{5.17}$$

El valor t obtenido significa el valor entre 0 y 1 en el cual $\vec{u}$ se cruza contra la superfície de la esfera, lo que permite multiplicar el vector original por t para conseguir finalmente $\vec{r}$, con el que más tarde mediante la fórmula 5.1 se consigue $\vec{r'}$

5.1.1 Implementación en Unity

Una vez explicada la teoría se va a proceder a exponer cómo se han integrado esas reflexiones en Unity. El *shader* que se ha generado tiene por nombre *Custom/-*

Reflections/Simple dentro del sistema de *shaders* de Unity, y está guardado en *Assets/Shaders/Others/SimpleReflection.shader*.

En primer lugar es necesario generar un *Light Probe* y colocarlo en la escena. La figura 56 muestra un *Light Probe* en la escena del proyecto, donde se pueden observar claramente los límites del cubo, así como el centro de este, representado por una esfera.

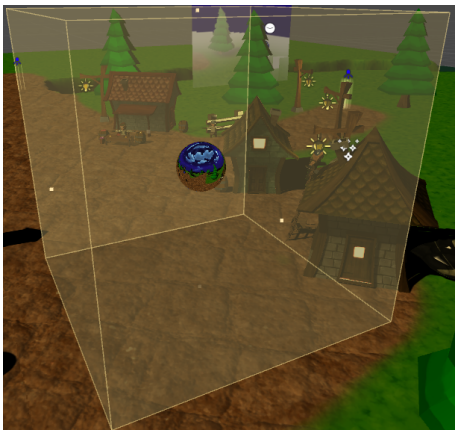


Figura 56: *Light Probe* en la escena.

Con el *Light Probe* situado en la escena, el *shader* puede ya realizar reflexiones. Para ello es necesario que este *shader* contenga como mínimo un paso *ForwardBase*. El cálculo de $\vec{r}'$ se puede realizar tanto dentro del *Vertex Shader* para más tarde pasarlo mediante el interpolador al *Fragment Shader*, como dentro del propio *Fragment Shader*. Con el fin de poder escoger cual de las opciones escoger, el *shader* contiene la variante *REFLECTION_OPTIMIZE*. En el caso que esta variante sea puesta a uno, el cálculo tendrá lugar en su modo optimizado, es decir, con $\vec{r}'$ calculado dentro del *Vertex Shader* y guardando un espacio en la estructura de salida al *Fragment Shader*, en caso contrario el cálculo se realizará directamente en el *Fragment Shader*. El cálculo del $\vec{r}'$ está contenido dentro de la función *ComputeReflexDirection*, presente dentro del fichero *UtilsReflection.cginc*.

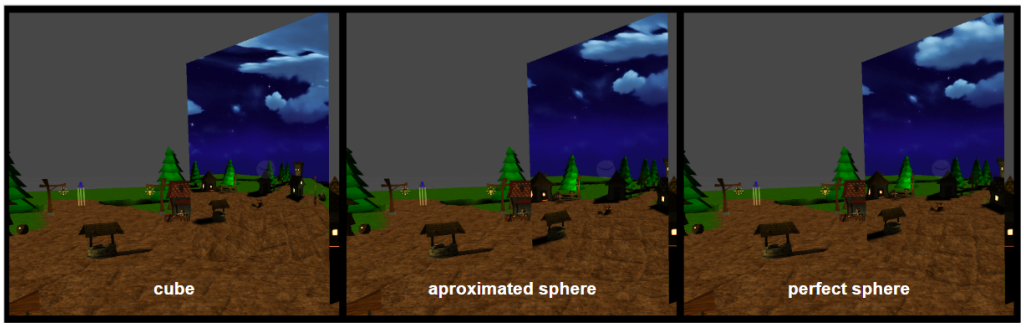
Además de la variante *REFLECTION_OPTIMIZE*, se han introducido cuatro variantes más. Tres de ellas están destinadas a distinguir qué tipo de método se utiliza para aproximar $\vec{r}'$. Son, en orden en que se han explicado, *_REFLECTION_CUBE*, *_REFLECTION_APPROXIMATION* y *_REFLECTION_PERFECT*. Estas tres variantes se utilizan dentro de la función *ComputeReflexDirection* para realizar el cálculo del vector reflejado, ya que éste cálculo depende del método que se utilice. La variante que falta lleva por nombre *_REFLECTION_NONE*, que sirve principalmente para saber cuando las reflexiones están desactivadas.

Así como el cálculo de $\vec{r}'$ se puede realizar tanto dentro del *Vertex Shader* como del *Fragment Shader*, el muestreo al *cubemap* se realiza únicamente dentro del *Fragment Shader* mediante la función de Unity *UNITY_SAMPLE_TEXCUBE_LOD*, que se encarga de coger la muestra dado un vector, un *cubemap*, y un valor de detalle, gracias al cual se puede simular un espejo poco nítido. La figura 57 muestra cómo el valor de nitidez afecta al resultado de una reflexión sobre una esfera.

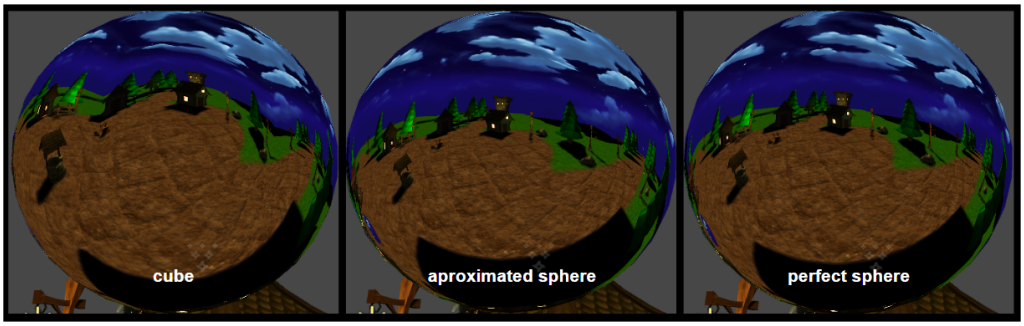
Figura 57: Diferentes valores de detalle en la muestra del *cubemap*.

Tanto para calcular $\vec{r}'$ como para realizar el muestreo, se necesitan varios valores que Unity deja utilizar al programador. Para realizar los cálculos en los que se calcula el vector $\vec{r}'$ es necesario conocer tanto el punto central del cubo que forma el *Light Probe*, como el tamaño y la posición de sus paredes. Estos valores están contenidos en la variable *unity_SpecCube0_ProbePosition* para saber el centro y *unity_SpecCube0_BoxMin* y *unity_SpecCube0_BoxMax* para saber la posición de las paredes del cubo. En el caso del muestreo, el *cubemap* está contenido en la variable *unity_SpecCube0*

Las imágenes de la figura 58a muestran los resultados de aplicar los métodos explicados sobre la escena del juego con objetos diferentes.



(a) Resultados de aplicar reflexiones a un plano.



(b) Resultados de aplicar reflexiones a una esfera.

Figura 58: Resultados de aplicar los distintos tipos de reflexiones a diferentes objetos.

5.2 Materiales metálicos y dieléctricos

Hasta ahora los materiales que se han visto en la sección 2 acostumbran a tener luces especulares de un color claro, similar al de la luz que las está provocando. Este tipo de materiales son los más habituales de encontrar y tienen como nombre materiales dieléctricos. Algunos ejemplos pueden ser por ejemplo un vaso de plástico o una pelota de goma. Por otro lado, existen otro tipo de materiales que debido a la composición de su superfície, suelen absorber más rápido la luz que incide sobre su superfície al mismo tiempo que evita que esta penetre demasiado en el objeto. Debido a esto, los materiales metálicos acostumbran a tener unas reflexiones especulares más nítidas teñidas por el color del propio objeto. Algunos ejemplos de materiales metálicos son, por ejemplo, el oro y el bronce de los intrumentos de viento como la trompeta o la tuba.

(a) Reflexiones dieléctricas en un recipiente de plástico.

(b) Reflexiones metálicas en una tuba.

Figura 59: Comparación entre materiales dieléctricos y metálicos.

5.2.1 Implementación de materiales metálicos y dieléctricos utilizando reflexiones

Con el fin de poder crear materiales metálicos y dieléctricos, se ha creado un *shader* con el nombre *"Custom/Reflections/Complex"* dentro del sistema de *shaders* de Unity, guardado en la carpeta *"Assets/Shaders/Others/"* con el nombre *Complex-Reflection.shader*. En la figura 60 se muestra el inspector del material.

Los campos que se destacan son los siguientes:

- Los campos *Albedo Texture* y *Color* sirven para añadir color de albedo al objeto que utilice el *shader*. Este es el color del componente difuso para los materiales dieléctricos, pero también es el color de la luz especular en los materiales metálicos.
- El campo *Normal map* permite añadir una textura de normal para distorsionar tanto las reflexiones como la interacción difusa con la luz.

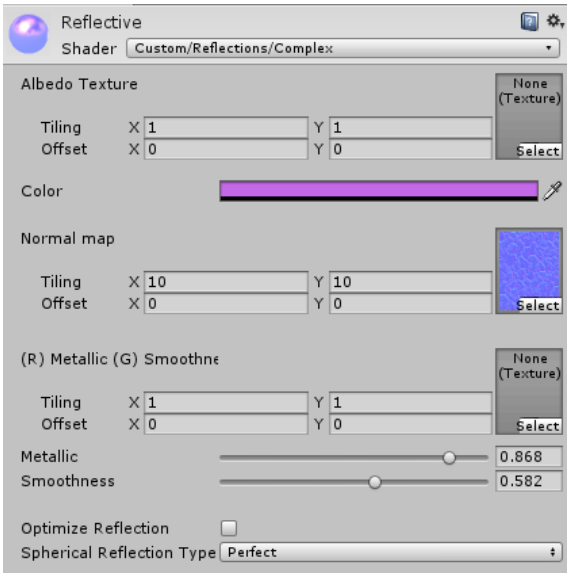


Figura 60: Inspector de un material metálico.

- La textura *(R) Metallic (G) Smoothing* es la textura extra. Como ya se explicita en el nombre del campo, el canal rojo establece el valor de metal del objeto mientras que el canal verde se asocia al valor de suavizado, es decir, lo lisa que es la superfície en ese punto.
- Los campos *Metallic* y *Smoothness* sólo están visible cuando no hay ninguna textura extra en el campo anterior. Permiten cambiar respectivamente el valor de metal y el valor de suavizado.
- Por último los campos *Optimize Reflection* y *Spherical Reflection Type* tienen la misma función que ya se ha explicado al final del apartado 5.1.1.

Para simular estos dos tipos de materiales se necesitan tanto una componente difusa como una especular.

Para conseguir la componente difusa hay que tener en cuenta que a medida que aumenta la suavidad de la superfície este componente se irá progresivamente mezclando con el reflejo. Cuando esto ocurre en metales, la componente difusa desaparece completamente dando paso al reflejo, teñido del mismo color que la luz especular, mientras que, en el caso de los materiales dieléctricos, la componente difusa no llega a desaparecer del todo, mezclandose con el reflejo. Por tanto, la componente difusa se calcula mediante una interpolación lineal entre la interacción difusa de la luz y el reflejo en función del peso o *weight*,

$$weight = 0.25 \cdot Smoothness + 0.75 \cdot Metallic \tag{5.18}$$

de esta forma se consiguen los valores de la tabla 1.

Smoothness	Metallic	weight
0	0	**0.00**
0	1	**0.75**
1	0	**0.25**
1	1	**1.00**

Taula 1: Peso entre la componente difusa y la reflexión.

donde cuanto más alto es *weight* mayor es el nivel de reflexión. Se puede observar, por tanto, que cuando el material es metálico y completamente liso, la reflexión es total, mientras que un material dieléctrico también liso tiene una reflexión de tan solo el 25%.

En cuanto a las luces especulares, no se ha utilizado la aproximación descrita en la sección 2.4, debido a que se buscaba conseguir unas luces especulares más intensas. Para conseguir esto, se ha seguido la aproximación utilizada por el *Standard shader* de Unity. Esta aproximación utiliza la textura *unity_NHxRoughness*[3] mostrada en la figura 61 para simular la intensidad de la interacción especular entre la superfície y una luz.

Dado que las luces especulares que se han explicado en la sección 2 son una aproximación de las reflexiones para superfícies rugosas, a medida que la rugosidad del objeto se aproxima a cero, la luz especular debe desaparecer. Como se puede observar en la figura 61 , la textura *unity_NHxRoughness* también tiene en cuenta eso ya que para valores de *roughness* bajos, es decir, superfícies muy lisas, la textura sólo aporta intensidades especulares nulas.

Para conseguir la intensidad, se muestrea utilizando el coseno entre el vector $\vec{h}$, calculado mediante la fórmula 2.1 y el vector normal a la superfície $\vec{n}$ elevado a cuatro y multiplicado por 16. Se han escogido estos valores porque son los mismos que utiliza Unity para calcular la intensidad de las luces especulares.

Este código se encuentra dentro de la función *GetSpecularIntensityFromLUT*. Para conseguir el color de la luz especular en función de el valor de metal que se haya especificado a través del inspector del material, se ha creado la función *GetSpecularColorByMetallic* que realiza una interpolación lineal entre los colores del albedo del objeto y la luz que recibe teniendo en cuenta el valor de metal del material como peso. Cuando el valor de metal es 1, la luz especular es del color del albedo, mientras que si por el contrario, es cero, el color es el de la luz que recibe la superfície.

Si se utiliza el campo de la textura extra, los valores de metal y suavizado de la superfície se determinan a partir de ésta. Aplicando una textura extra sobre un cilindro, los resultados son los de la figura 62.

[3]Situada en el fichero *UnityStandardBRDF.cginc* dentro del sistema de ficheros de Unity.

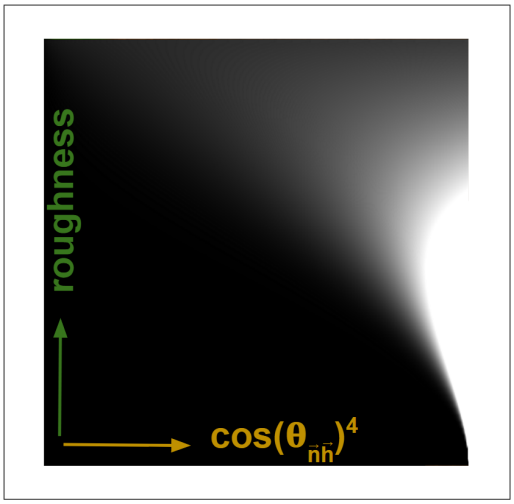


Figura 61: Textura utilizada para calcular la intensidad de la luz especular.

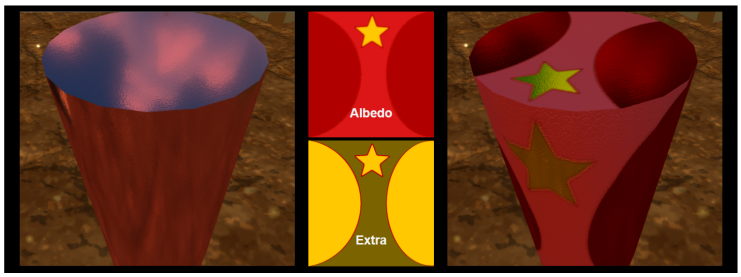


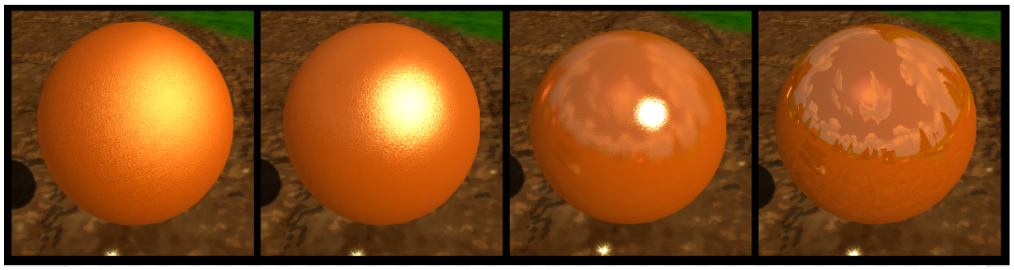
Figura 62: Resultado de aplicar valores de metal y suavizado sobre un cilindro mediante una textura extra como la de el centro abajo. Los dos primeros canales de la textura representan respectivamente el valor de metal (R) y el suavizado (G).

Los resultados de aplicar este *shader* a una esfera se pueden observar en la figura 63.

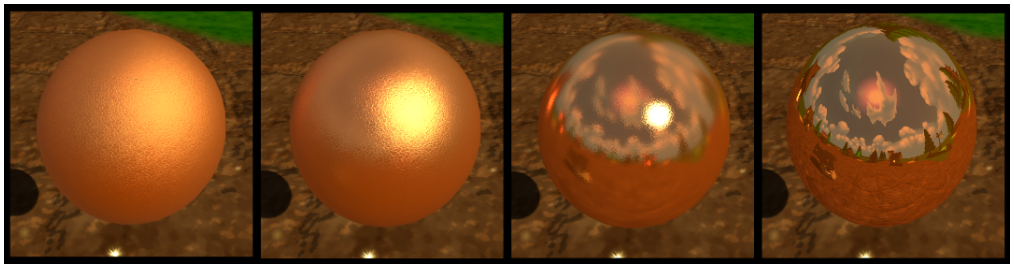
5.3 Refracciones

Como ya se ha visto en el apartado de reflexiones, existen objetos que, debido a las características de su superfície, tienen la capacidad de reflejar la luz que proviene de otros objetos. Los objetos con superfícies refractivas, por otra parte, dejan pasar la luz a través de ellas simplemente alterando la dirección del rayo incidente. El proceso por el cual un rayo de luz altera su dirección al pasar de un medio con unas características concretas de propagación de la luz a otro medio con características diferentes se llama refracción. En la figura 64a se muestra cómo cuando un rayo de luz pasa de un medio a otro su dirección se altera.

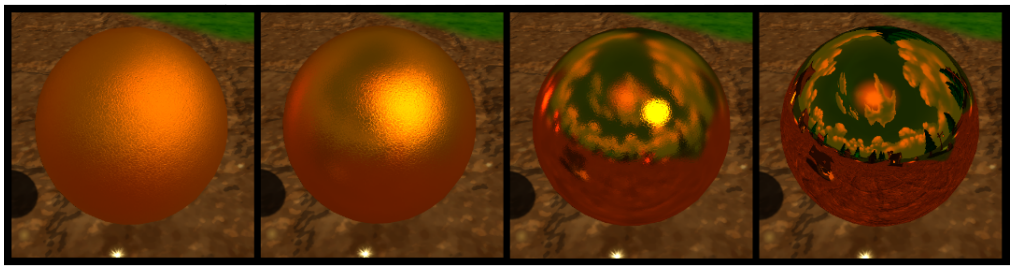
La propiedad que define la magnitud de esta alteración es el índice de refracción n. Cuanto mayor es este índice, mayor refracción presentan los rayos. El índice de refracción se calcula teniendo en cuenta que cada medio limita la velocidad v a la que la luz puede propagarse a través de él. La fórmula con la que se calcula es la siguiente:



(a) Esfera con valor de metal 0 y suavidad 0.25, 0.5, 0.85 y 1.



(b) Esfera con valor de metal 0.5 y suavidad 0.25, 0.5, 0.85 y 1.



(c) Esfera con valor de metal 1 y suavidad 0.25, 0.5, 0.85 y 1.

Figura 63: Resultados de aplicar diversos valores de metal y de suavizado sobre una esfera.

$$n = \frac{c}{v} \tag{5.19}$$

donde c es la velocidad de la luz en el vacío, con un valor de 299792458 m/s.

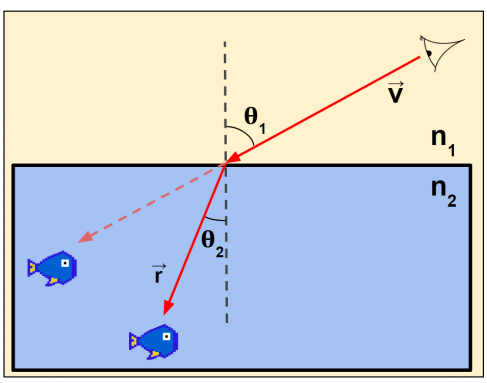
A través de esta fórmula se puede observar cómo a medida que aumenta la velocidad de propagación en el medio, el índice de refracción disminuye, hasta el punto en que cuando los dos medios son el vacío ($v = c$) no existe refracción.

Tanto el rayo emitido por el observador $\vec{v}$ como el rayo refractado $\vec{r}$ forman respectivamente los ángulos θ_1 y θ_2 con el vector normal $\hat{n}$ de la superfície.

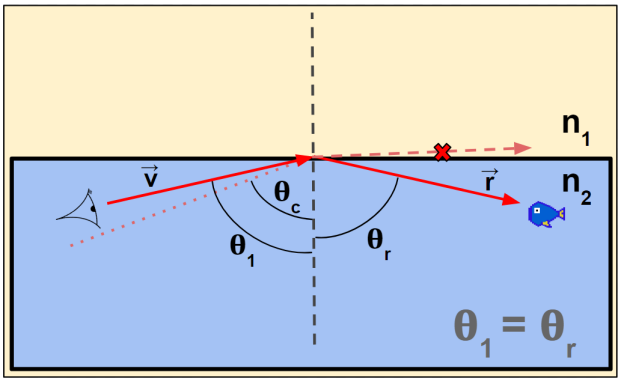
Dado el vector $\vec{v}$, el vector normal a la superfície $\vec{n}$ y el coeficiente η que representa el ratio entre el valor de n en el medio del observador (n_1) y el valor de n en el segundo medio (n_2), el cálculo del vector $\vec{r}$ se realiza del siguiente modo.

Mediante la ley de *Snell*[4] que establece,

[4]https://en.wikipedia.org/wiki/Snell%27s_law



(a) Comportamiento de una refracción.



(b) Comportamiento de una refracción cuando llega al ángulo critico.

Figura 64

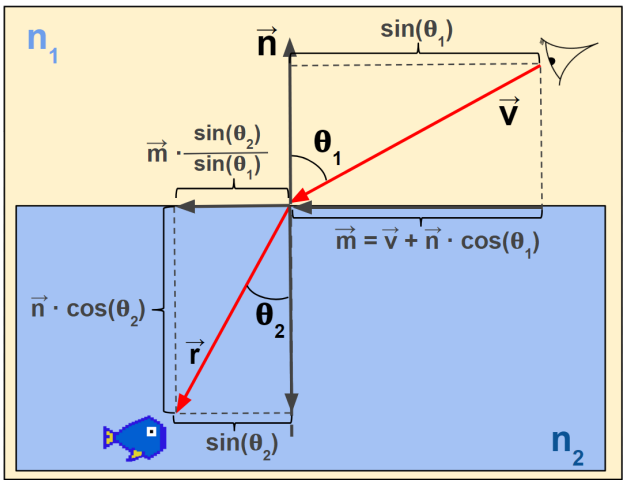


Figura 65: Construcción de $\vec{r}$.

$$n_1 \cdot sin(\theta_1) = n_2 \cdot sin(\theta_2) \tag{5.20}$$

$$\eta = \frac{n_1}{n_2} = \frac{sin(\theta_2)}{sin(\theta_1)} \tag{5.21}$$

se puede extraer $\vec{r}$ mediante la siguiente fórmula utilizando los vectores mostrados en la figura 65.

$$cos(\theta_2) = \sqrt{1 - \eta^2 sin^2(\theta_1)} \tag{5.22}$$

$$\vec{r} = \eta \cdot (\hat{v} + \hat{n} \cdot cos(\theta_1)) - \hat{n} \cdot cos(\theta_2) \tag{5.23}$$

Esta fórmula aparece explicada considerando el vector $\vec{v}$ girado en [15].

El comportamiento de la refracción básica es simple, sin embargo, no todos los rayos que inciden en una superfície refractiva son refractados. Una parte de

estos se refleja en la superfície. El porcentaje de rayos que son reflejados en la superfície se denomina **reflectancia**, mientras que el porcentaje que define aquellos que son refractados es el de transmitancia. Estos dos valores son complementarios, es decir, cuando la reflectancia es igual a r, la transmitancia es $1 - r$. El valor de la reflectancia en un punto de una superfície refractiva depende de tres valores: el índice de refracción del medio que transmite el rayo antes de la colisión con la superfície o n_1, el índice de refracción del medio que se quiere atravesar o n_2 y el ángulo que forman entre ellos $\hat{v}$ y $\hat{n}$, θ_1.

La figura 66 es una imagen real de lo que se llama *Efecto Fresnel*, donde se muestra cómo la reflectancia sobre un medio con agua aumenta cuando crece el ángulo entre la dirección en la que mira el observador y la dirección normal del agua. La reflectancia se calcula utilizando las ecuaciones de *Fresnel*[5].

$$R_s(\theta_1, n_1, n_2) = \left| \frac{n_1\ cos\theta_1 - n_2\ \sqrt{1 - (\frac{n_1}{n_2} sin\theta_1)^2}}{n_1\ cos\theta_1 + n_2\ \sqrt{1 - (\frac{n_1}{n_2} sin\theta_1)^2}} \right|^2 \tag{5.24}$$

$$R_p(\theta_1, n_1, n_2) = \left| \frac{n_1\ \sqrt{1 - (\frac{n_1}{n_2} sin\theta_1)^2} - n_2\ cos\theta_1}{n_1\ \sqrt{1 - (\frac{n_1}{n_2} sin\theta_1)^2} + n_2\ cos\theta_1} \right|^2 \tag{5.25}$$

$$R(\theta_1, n_1, n_2) = \frac{1}{2}(R_s(\theta_1, n_1, n_2) + R_p(\theta_1, n_1, n_2)) \tag{5.26}$$

donde R es el valor de la reflectancia.

Figura 66: Aplicación real de la reflectancia en agua.

Sin embargo, estas ecuaciones resultan muy costosas a la hora de calcular la reflectancia de manera que es habitual utilizar una aproximación llamada aproximación de *Schlick*. Este método aparece en [16], donde se define la aproximación como,

$$R_{Schlick}(c, \vec{l}, \vec{n}) = c + (1 - c)(1 - \vec{l} \cdot \vec{n})^5 \tag{5.27}$$

donde $\vec{l}$ en este caso es $\vec{v}$ ya que el la dirección en la que el observador recibe la luz, $\vec{n}$ es el vector normal y c es un valor que corresponde al valor de la reflectancia calculado sobre el ángulo $\theta_1 = 0$. Por lo tanto,

[5]https://en.wikipedia.org/wiki/Fresnel_equations

$$c = R_s(0°, n_1, n_2) = R_p(0°, n_1, n_2) = R(0°, n_1, n_2) = \left|\frac{n_1 - n_2}{n_1 + n_2}\right|^2 \tag{5.28}$$

$$R_{Schlick} = \left|\frac{n_1 - n_2}{n_1 + n_2}\right|^2 + \left(1 - \left|\frac{n_1 - n_2}{n_1 + n_2}\right|^2\right) \cdot (1 - cos\theta_1)^5 \tag{5.29}$$

Suponiendo tres materiales diferentes se ha realizado una comparación entre los resultados de las equaciones de *Fresnel* y los conseguidos mediante la aproximación de *Schlick*. Se considera que el medio inicial es aire, por tanto n_1 igual a 1, mientras que los medios que reciben el rayo de luz son, por orden de índice de refracción, agua (1.31), vidrio (1.52) y diamante (2.42)[6]. Esta comparación se muestra en la figura 67.

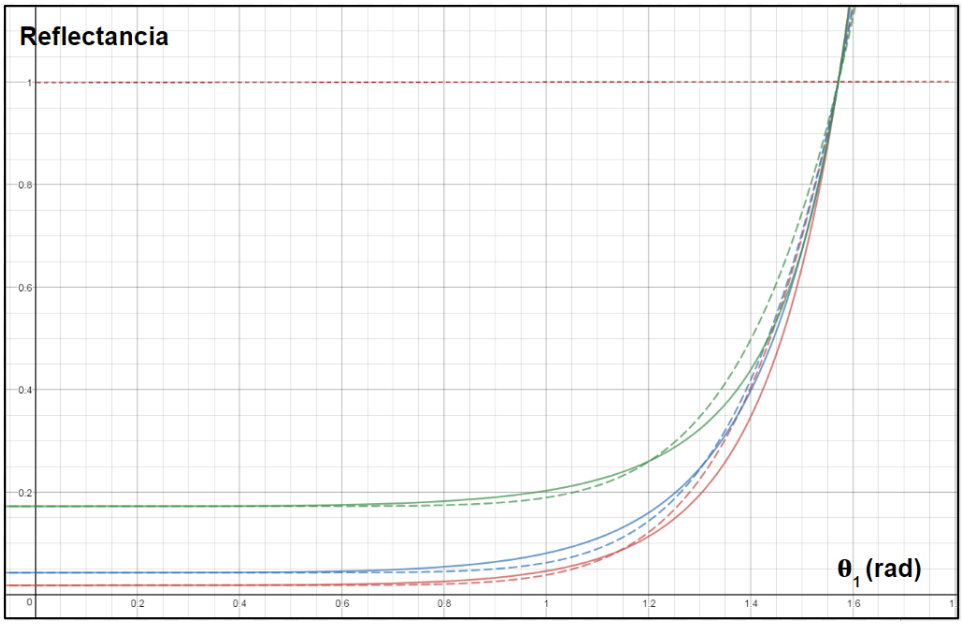


Figura 67: Comparación entre la función real de reflectancia (línea continua) y la aproximación de *Schlick* (línea discontinua) para primer medio aire y segundo medio agua (rojo), vidrio (azul) y diamante (verde).

Debido a que existe, aunque débil, una parte de reflexión, es necesario calcular el vector reflejado de la superfície utilizando el método explicado en el apartado 5.1 de reflexiones. El color resultante debe ser una interpolación lineal en la que el peso sea la reflectancia.

5.3.1 Implementación de superfícies refractivas en Unity

Para que un material sea capaz de realizar refracciones tiene que poder saber qué objetos tiene detrás o como mínimo qué colores. El método seguido para conseguir

[6]https://en.wikipedia.org/wiki/List_of_refractive_indices

esto ha sido utilizando un *Light Probe*, es decir, el mismo método que en el apartado de reflexiones, pero realizando el muestreo mediante un vector refractado y no reflejado.

Se ha creado un nuevo *shader* con el nombre *"Custom/Transparent/Refraction"* en el sistema de *shaders*. El archivo se encuentra dentro de la carpeta *"Assets/Shaders/Refraction"*. En la figura 68 se muestra el inspector de este *shader*.

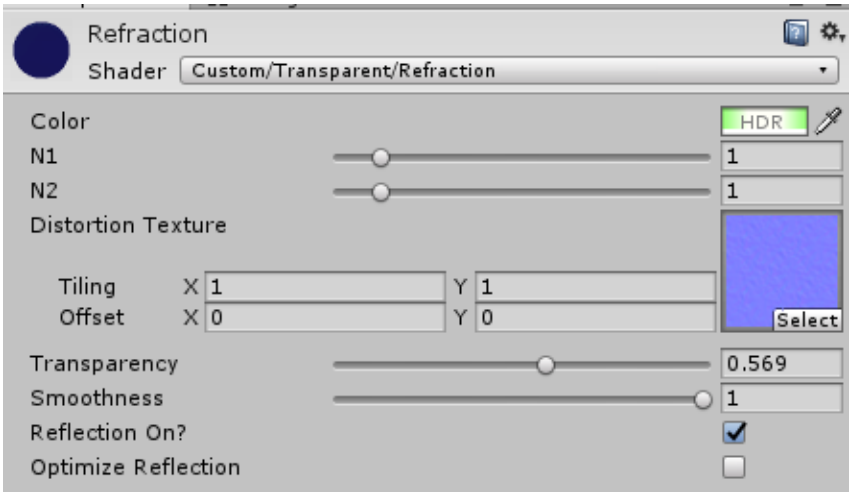


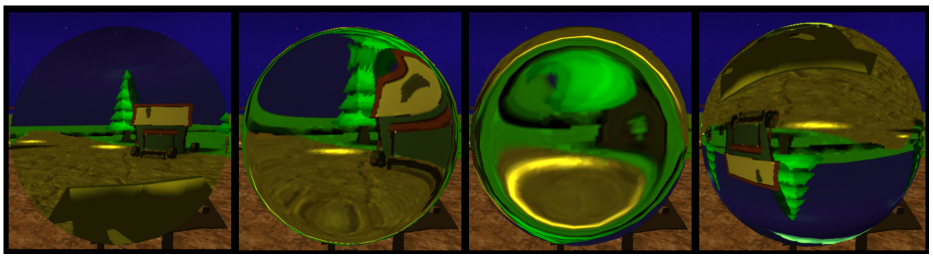
Figura 68: Inspector del *shader* refractive.

El material consta de los siguientes campos:

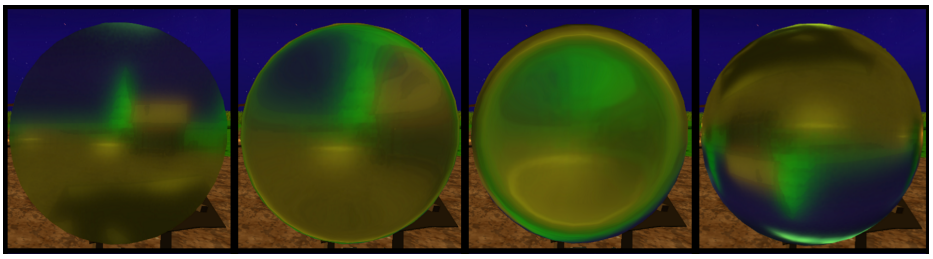
- El campo *Color* donde se puede modificar el tinte del material.
- Los campos *N1* y *N2* permiten modificar los índices de refracción de los medios. El valor de *N1* representa el índice de refracción del medio a través del cual la luz llega hasta la superfície del objeto. Habitualmente este valor es 1 ya que lo más corriente es que entre el objeto y el observador haya aire. El valor de *N2* representa el índice de refracción del objeto en sí.
- El campo *Distorsion Texture* permite añadir una *bump texture* para así poder modificar las normales y aparentar cierto grado de rugosidad.
- El campo *Transparency* permite modificar la calidad de la muestra recogida del *cubemap* de manera que cuanto menor sea este valore más sensación de opacidad dé el material.
- El campo *Smoothness* tiene dos propósitos. El primero de ellos es, de la misma forma que en el campo *Transparency*, modificar la calidad de la muestra, pero en este caso sobre las reflexiones, ya que cuanto más irregular es una superfície menos claras son sus reflexiones. El segundo propósito es el de modificar el nivel de distorsión de las normales mediante la textura del campo *Distorsion Texture*.

- Por último, los campos *Reflection On?* y *Optimize Reflection* permiten escoger si el material debe utilizar reflexiones o no y optimizar el cálculo del vector de la reflexión calculando éste dentro del *Vertex Shader* para más tarde pasarlo interpolado al *Fragment Shader* dentro de la estructura de entrada.

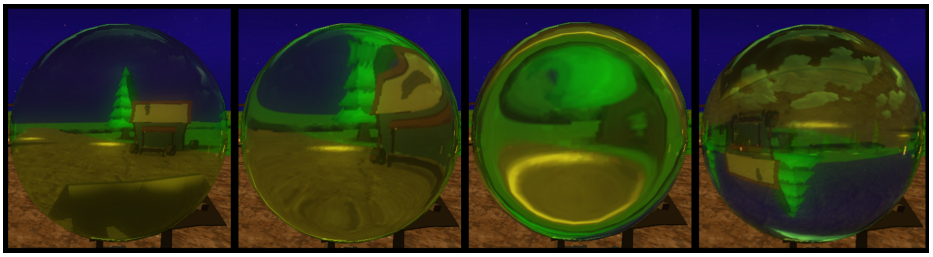
Aplicando este *shader* a una esfera y modificando algunos de los campos explicados se han obtenido los resultados de la figura 69.



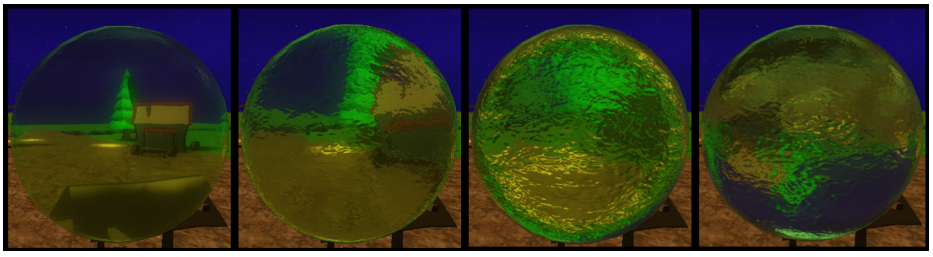
(a) Esfera con índices de refracción 1, 1.15, 1.31 y 2.42.



(b) Esfera con los índices de refracción en 69a y nivel de transparencia de 0.3.



(c) Esfera con los índices de refracción en 69a, nivel de transparencia de 0.75 y reflexiones utilizando la aproximación de *Schlick*.



(d) Resultado de aplicar rugosidad a la esfera de la figura 69c.

Figura 69: Resultados de aplicar diferentes valores a una esfera con una superfície refractiva.

6 Visualización de sombras en algoritmos proyectivos

Las sombras son fundamentales en imágenes, animaciones y juegos debido a que permiten al observador tener una referencia visual que le ayuda a situar en el espacio un determinado objeto, siendo por esto, un elemento imprescindible a la hora de desarrollar aplicaciones con dispositivos de realidad virtual. Este tema se desarrolla en profundidad en el apartado de sombras de [4].

6.1 Visualización de sombras en la GPU

Uno de los métodos más utilizados por su rapidez y eficiencia para calcular sombras es la aproximación descrita por Williams en [1], que tiene como nombre *Shadow Mapping*.

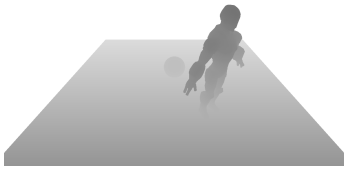


Figura 70: Mapa de profundidades o *shadow map*.

Este método consta de dos pasos principales:

- El primer paso consiste en conseguir desde el punto de vista de la luz que está provocando la sombra y mediante un buffer especializado llamado *ZBuffer*, una textura que contenga únicamente las profundidades a los objetos de la escena, como se muestra en la figura 70. Este mapa de profundidades se llama *shadow map*.
- El segundo paso parte de la precondición de tener guardadas las profundidades en el mapa de profundidades o *shadow map*. Se trata de transformar aquellos puntos o *fragments*, visibles desde el punto de vista del observador en la escena final, a coordenadas de luz. De esta manera es posible comprobar si en el punto que se está evaluando existe una sombra o no, ya que en caso que la distancia guardada en el paso previo fuese menor significaría que existe un objeto que obstaculiza el rayo de luz.

La figura 71 muestra un cuadrado que recibe luz en una dirección $\vec{l}$ mientras un observador mira la superfície bajo el cuadrado. En este caso la luz que obstruye el cuadrado no llega a la superfície y, por lo tanto, cuando el observador mira esa zona percibe una falta de luz, es decir, sombra. La línea azul representa el *shadow map* donde se guardan las profundidades desde el punto de vista de la luz. Siguiendo la técnica del *Shadow Mapping*, a la hora de renderizar la superfície se compara la distancia entre el punto y la luz contra la distancia guardada en el *shadow map*.

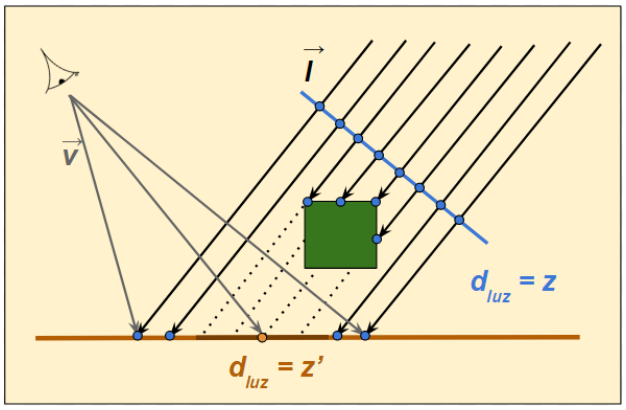


Figura 71: Esquema básico de la generación de una sombra mediante la técnica *Shadow Mapping*.

De esta forma, considerando que, como en la figura, el *shadow map* tiene guardada una distancia z y en el punto de la superfície la distancia es z', existirá sombra si $z < z'$.

6.2 Visualización de sombras en Unity

En Unity los dos pasos que constituyen la técnica del *Shadow Mapping* se pueden identificar con dos fases, la de proyección de sombras y la de recepción de sombras. Por ejemplo, un objeto puede proyectar sombras sobre otra superfície, pero no proyectar sombras en su propia superfície, es decir, no recibirlas.

Para que Unity sea capaz de detectar un *GameObject* como un obstáculo que pueda proyectar sombras, éste objeto tiene que proporcionar su posición dentro de un paso marcado con la etiqueta *"Lightmode = ShadowCaster"*. Esta etiqueta es la que permite al sistema de sombras de Unity identificar los objetos que obstruyen la luz y, por lo tanto, sólo aquellos *GameObjects* cuyo *shader* implementa un paso con una etiqueta así aparecen en el *shadow map*. El paso tiene que contener un programa *Vertex Shader* y otro programa *Fragment Shader*. Como se verá más adelante, estos dos programas van a ser diferentes para cada tipo de luz, por lo que es necesario añadir la directiva de compilación

#pragma multi_compile_shadowcaster

que añade las variantes necesarias para poder comprobar dentro del paso qué tipo de luz es la que está proyectando una sombra.

Unity contiene algunas herramientas que permiten modificar algunas propiedades de las sombras. A continuación se enumeran las más relevantes:

- Dentro del menú *QualitySettings* está el apartado *Shadows*.
- Dado que las sombras también dependen de la luz, existen opciones para modificar éstas desde el inspector de cualquier *GameObject* que tenga un

componente *Light*.

Ambas utilidades se muestran en la figura 72.

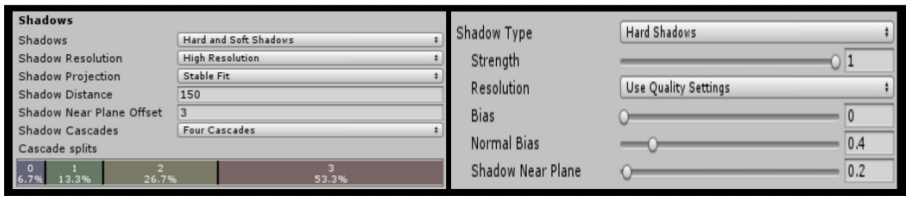


Figura 72: Izquierda: el apartado Shadows en la pestaña de *QualitySettings*. Derecha: el apartado de sombras del componente *Light*.

La proyección de sombras explicada hasta ahora puede parecer un proceso simple, sin embargo, el procedimiento para llevar a cabo esto cambia según el tipo de luz. Dada la escena de la figura 73 sin sombras formada por tres *GameObjects* (un robot, una esfera y un plano) se explicarán a continuación los pasos que sigue Unity tanto para proyectar sombras sobre otros objetos como para recibir sombras de otros objetos en función del tipo de luz que las proyecte.

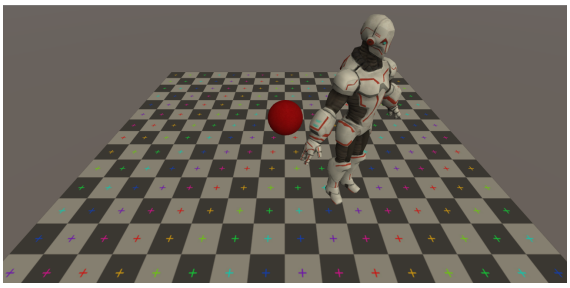


Figura 73: Escena sin sombras.

6.2.1 Sombras con luces direccionales

Como ya se ha explicado en la sección 2.1 las luces direccionales están formadas por una intensidad y una dirección igual para todos los rayos. En la figura 6 de esa sección se muestra un esquema de una luz direccional.

A la hora de generar sombras a partir de luces direccionales se sigue el principio del método explicado en el apartado anterior, sin embargo se utiliza desde una aproximación ligeramente diferente.

Lo primero a destacar es que Unity activa dos tareas cuando una luz direccional produce sombras:

- La primera es la tarea *RenderShadowMap*. Esta tarea se encarga de generar un *shadow map* a partir de la información proporcionada por el objeto que obstruye la luz.

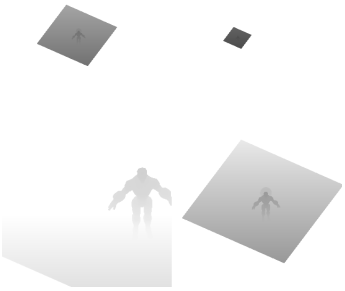


Figura 74: *Shadow map* resultante de la tarea *RenderShadowMap*.

- La segunda tarea es la llamada *ShadowCollector*. Esta tarea se encarga de traducir el *shadow map* generado en el paso anterior en una textura en escala de grises desde el punto de vista de la cámara como la de la figura 75. Esta textura se encuentra en coordenadas de observador, a diferencia de los *shadow maps*, lo que permite que simplemente se tenga que realizar un muestreo sobre esta textura para saber si existe sombra en un determinado *fragment* visible.

  Esta textura se genera a partir del *shader* interno de Unity *"Hidden/Internal-ScreenSpaceShadows"*, donde para cada *fragment* entrante en coordenadas de observador y a través de un cambio a coordenadas de luz, se hace la comparación entre las distancias guardadas en los *shadow maps* y las distancias reales en esas mismas coordenadas. El resultado de esa comparación es una textura de la escena desde el punto de vista del observador en escala de grises dependiente únicamente con sombras, como la mostrada en la figura 75.

Figura 75: Resultado de la tarea *ShadowCollector*.

6.2.1.1 Proyección de la sombra

Como ya se ha explicado anteriormente, las luces direccionales utilizan únicamente una dirección y por lo tanto todos los rayos provenientes de una luz direccional son paralelos entre sí. Esto es importante cuando, siguiendo el proceso de *Shadow Mapping*, se transforman los puntos de la escena de coordenadas de mundo a coordenadas de luz, ya que debido a esto, la transformación se hace mediante una proyección ortográfica.

Las sombras que proceden de luces direccionales son las más simples de calcular. Únicamente necesitan tener como entrada al *Vertex Shader* una estructura que

contenga la posición y la normal en coordenadas de objeto. La función del *Vertex Shader* en este caso es transformar esta posición a coordenadas de *clip* utilizando la matriz *MVP*. Es posible que el resultado sólo con esto experimente problemas de ruido cuya solución se explicará más adelante en el apartado 6.2.5.1. Estos problemas se solucionan con dos funciones ya implementadas por Unity que añaden dos tipos de *bias*[7].

- La primera de ellas es *UnityClipSpaceShadowCasterPos* que, además de añadir un tipo de *bias* llamado *bias* de normal, se encarga de transformar la posición a coordenadas de *clip*.
- La segunda es la función *UnityApplyLinearShadowBias* que añade otro tipo de *bias* llamado *bias* lineal a partir de las coordenadas de clipping anteriores.

El *Fragment Shader* en este caso no necesita devolver nada ya que lo único que se requiere es la posición.

6.2.1.2 Recepción de la sombra

Partiendo de la precondición que los *shadow maps* han sido generados y la textura con las sombras de toda la escena se ha creado a partir de éstos se procede a muestrear la textura con las sombras. Éste tipo de texturas son texturas de pantalla, es decir, están extraídas a partir de la información de toda la escena y por lo tanto deben ser muestreadas utilizando las coordenadas de pantalla de los objetos. Las coordenadas de pantalla son coordenadas que tienen sus componentes xy a cero en la parte inferior izquierda de la pantalla y a uno en la parte superior derecha de la pantalla. Las componentes z y w son las mismas que las de la posición en coordenadas de *clip* y, por lo tanto, la componente z corresponde a un valor $\frac{z}{d} \in [0, 1]$ y la componente w es igual a d, donde d es la distancia entre el punto y la cámara.

El muestreo sobre una textura a nivel de pantalla recibe el nombre de *projective sampling*. Lo que hace este tipo de muestreo es dividir las componentes x e y de las coordenadas de pantalla, entre la componente w, para de esta forma mantener la perspectiva de la escena.

Con el fin de realizar ésta muestra Unity dispone de tres funciones básicas ya definidas dentro del fichero *"AutoLight.cginc"* que debe ser incluido previamente. Estas tres funciones tienen siempre el mismo nombre, sin embargo, su contenido varía según el tipo de luz que provoque las sombras. En el caso de luces direccionales el proceso que se sigue es el siguiente:

- *SHADOW_COORDS* Debe ser utilizado dentro del *struct* de salida del *Vertex Shader*. Crea un campo para las coordenadas de la sombra con el *semantic TEXCOORD* y el número proporcionado.

[7]Ambas están incluidas dentro del fichero *"UnityCG.cginc"* que es necesario importar al principio del programa con la directiva *"#include UnityCG.cginc"*.

- *TRANSFER_SHADOW* Esta función se utiliza dentro del *Vertex Shader*. Asocia a las coordenadas de sombra el valor de las coordenadas de pantalla del objeto.

- *SHADOW_ATTENUATION* Esta función tiene que ser llamada dentro del *Fragment Shader*. Utilizando las coordenadas de pantalla almacenadas por la instrucción *TRANSFER_SHADOW*, se realiza un muestreo a la textura de las sombras de la escena en escala de grises utilizando *projective sampling*.

Una vez ya se han utilizado estas tres funciones en un paso, ese paso tiene la capacidad de recibir sombras. El proceso seguido para realizar sombras mediante este tipo de luces se muestra en la figura 76.

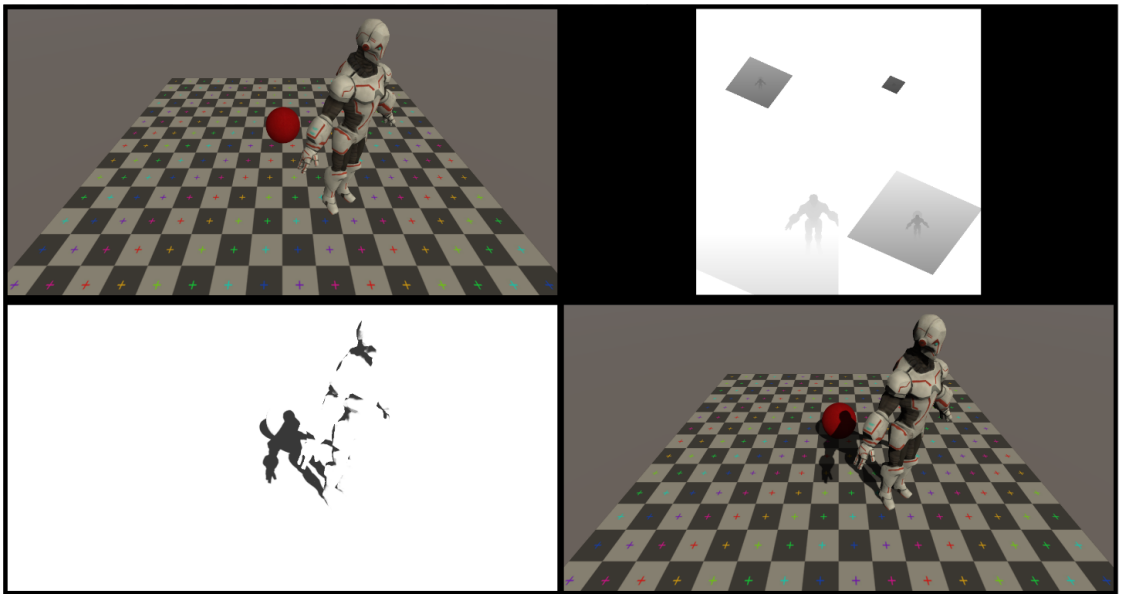


Figura 76: Proceso de proyección de sombras mediante luces direccionales.

6.2.2 Sombras con luces puntuales

En la sección sobre tipos de luces se ha explicado cómo funcionan las luces puntuales. Como se observa en la figura 8 de esa sección, éstas luces dispersan rayos de luz en todas direcciones, lo cual implica que se realiza una proyección perspectiva, en vez de ortográfica, al pasar a coordenadas de luz, a diferencia de las luces direccionales. Pero no es ésta la única diferencia. Para las luces puntuales Unity utiliza un *cubemap* para almacenar los *shadow maps*, ya que de esta manera hay un *shadow map* para todas las direcciones del entorno del cubo. Son por tanto estas luces las más costosas a la hora de proyectar sombras.

6.2.2.1 Proyección de la sombra

Para conseguir proyectar sombras sobre otros objetos a partir de una luz puntual

se ha de seguir una aproximación diferente a la utilizada con las luces direccionales. En primer lugar dentro del paso con etiqueta *ShadowCaster*[8] debe existir una estructura de entrada al *Vertex Shader* que contenga la posición del vértice.

Así como en el caso de las luces direccionales basta con devolver la posición en coordenadas de *clip* en el *Vertex Shader* para que Unity genere el *shadow map*, no es posible realizar lo mismo con *cubemaps*. Por tanto, en vez de dejar que Unity calcule la profundidad a partir de la posición, esta profundidad tiene que ser calculada dentro del paso. En concreto, la profundidad debe ser calculada dentro del *Fragment Shader* y por lo tanto el *Vertex Shader* tiene que tener como salida una estructura para introducir el vector de la luz $\vec{l}$, calculado a partir de la resta entre la posición del vértice y la posición de la luz, ambas en coordenadas de mundo[9].

El cálculo de la distancia tiene lugar dentro del *Fragment Shader* para evitar perder precisión utilizando el vector interpolado proveniente del *Vertex Shader*. De la misma forma que en el caso de la luz direccional, se pueden producir errores en la proyección de la sombra que se solucionan añadiendo un *bias* a la distancia calculada. En este caso el *bias* se encuentra dentro de la componente x de la variable *unity_LightShadowBias* que en luces de tipo puntual es el valor del deslizador de nombre *Bias* en el componente *Light* de la figura 72.

Como ya se ha observado en el caso de la luz direccional, los *shadow maps* están formados por valores situados entre 0 y 1 mientras que la magnitud del vector normalmente es mayor. Para contener los valores dentro de este intervalo se multiplica la magnitud del vector $\vec{l}$ por el factor $\frac{1}{r}$, donde r es el radio de la luz puntual[10]. De esta forma el valor de la distancia queda a 1 en los extremos de la circumferencia y a 0 en el centro.

Sólo con esto ya se puede devolver la distancia en el *Fragment Shader*, sin embargo, se recomienda utilizar la función *UnityEncodeCubeShadowDepth* que se encarga de codificar la distancia de tipo *float* en una textura *RGBA* de cuatro canales de 8 bits en caso de necesitarlo. Esto es debido a que algunas plataformas no soportan texturas de profundidad de tipo flotante y prefieren utilizar profundidades codificadas en texturas.

6.2.2.2 Recepción de la sombra

La recepción de sombras producidas por luces puntuales parte de la precondición de que el *cubemap* con los *shadow maps* ha sido generado. En este caso no se genera ninguna textura en espacio de pantalla con las sombras ya calculadas, por consiguiente, cada *fragment* tendrá que comprobar, como ya queda representado en la figura 70, si la profundidad guardada dentro del *shadow map* es mayor o menor a la existente entre el *fragment* y el centro de la luz.

[8]No es necesario crear un nuevo paso si se utiliza la palabra variante *SHADOWS_CUBE* dentro de la directiva *" #if defined(KEYWORD_VARIANT)"*.

[9]La variable *_WorldSpaceLightPos0* contiene la posición de la luz. Se puede acceder a ella incluyendo el fichero *"UnityCG.cginc"*

[10]La variable *_LightPositionRange* contiene la posición de la luz en sus componentes xyz y su radio dentro de la componente w.

En el caso de las luces puntuales el proceso que se sigue es el siguiente:

- *SHADOW_COORDS* Debe ser utilizado dentro del *struct* de salida del *Vertex Shader*. Crea un campo para las coordenadas de la sombra con el *semantic TEXCOORD* y el número proporcionado.
- *TRANSFER_SHADOW* Esta función se utiliza dentro del *Vertex Shader*. Asocia el vector $\vec{l}$ a las coordenadas de sombra.
- *SHADOW_ATTENUATION* Esta función tiene que ser llamada dentro del *Fragment Shader*. A partir del vector $\vec{l}$ almacenado en las coordenadas de sombra se realiza el muestreo del *cubemap* para conseguir la profundidad almacenada. Por otra parte también se tiene que calcular de nuevo la magnitud de $\vec{l}$ dentro del intervalo $[0, 1]$. Una vez se tienen estos dos valores se procede a la comparación. Si la distancia almacenada en el *shadow map* resulta menor que la distancia entre el fragment y el centro de la luz, existe sombra y se devuelve un 0, en caso contrario se devuelve un 1.

El proceso seguido para realizar sombras mediante luces puntuales está representado en la figura 77.

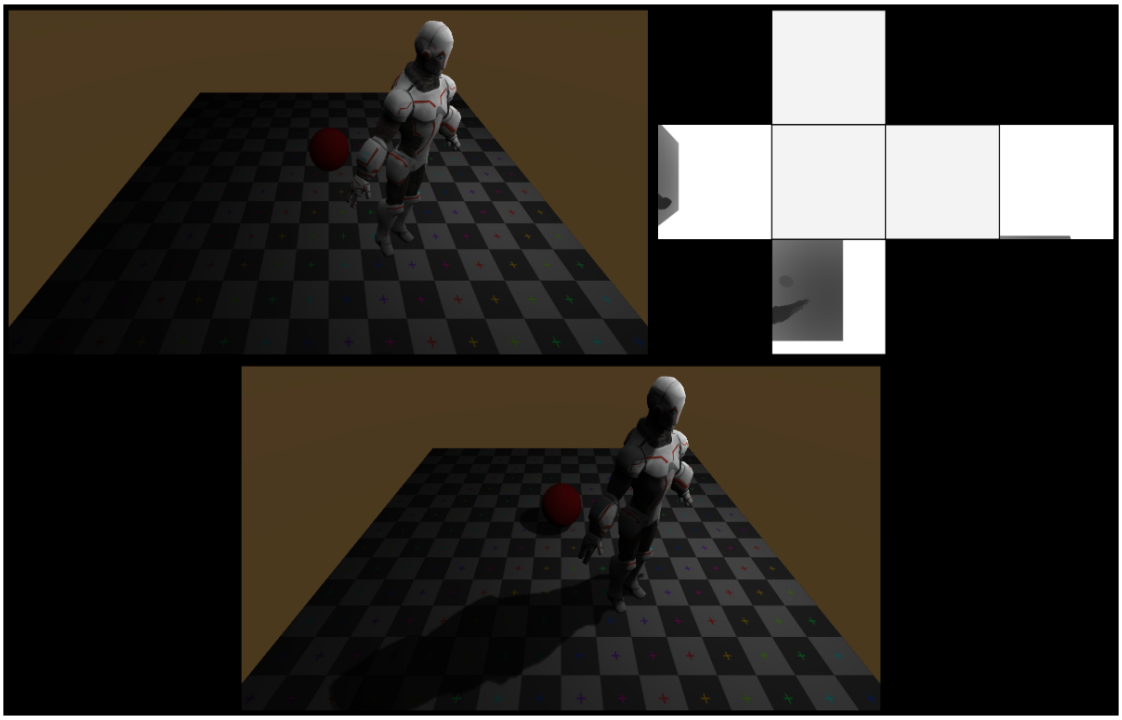


Figura 77: Proceso de proyección de sombras mediante luces puntuales.

6.2.3 Sombras con *Spot Lights*

Las *Spot Lights* están formadas por una posición, un rango, un ángulo y una intensidad. En esquema de una *Spot Light* puede verse en la figura 9. A continuación se explica cómo se han introducido sombras proyectadas por una *Spot Light*.

6.2.3.1 Proyección de la sombra

A la hora de proyectar sombras, las *Spot Lights* combinan elementos de los dos tipos de luces explicadas en los apartados anteriores: las luces direccionales y las luces puntuales.

En la figura 9 de la sección 2.1 se observa cómo las *Spot Lights* tienen un punto de origen y varias direcciones para cada rayo, igual que las luces puntuales. También se observa cómo, al contrario que las luces puntuales y de la misma forma que las luces direccionales, las *Spot Lights* no irradian luz en todas direcciones. Esto hace que, a la hora de proyectar sombras, las *Spot Lights* no utilicen un *cubemap* y por lo tanto sólo tengan que indicar la posición en coordenadas de *clip* como las luces direccionales.

Por otra parte, dado que cada rayo forma un ángulo diferente respecto de la dirección en la que apunta la luz, a la hora de trasladar las posiciones de los objetos al espacio de coordenadas de luz se utiliza una matriz similar a la utilizada por las luces puntuales, es decir, perspectiva. Por consiguiente, la similitud entre luces puntuales y *Spot Lights* está en el traslado a coordenadas de luz.

Sin embargo, esta última similitud no afecta a la implementación, ya que la realiza Unity internamente. Por tanto, la implementación del paso será la misma que en el caso de las luces direccionales.

6.2.3.2 Recepción de la sombra

La recepción de sombras generadas por *Spot Lights* también parte de la precondición de haber calculado previamente los *shadow maps*. De la misma forma que en la recepción de sombras generadas por luces puntuales, el sistema de sombras no genera una textura que ya tenga las sombras calculadas y por lo tanto el sistema a seguir es muy similar. Las únicas diferencias existentes son aquellas debidas a la utilización de *cubemaps* por parte de las luces puntuales.

El proceso de recepción de sombras se lleva a cabo utilizando las funciones:

- *SHADOW_COORDS* Debe ser utilizado dentro del *struct* de salida del *Vertex Shader*. Crea un campo para las coordenadas de la sombra con el *semantic TEXCOORD* y el número proporcionado.
- *TRANSFER_SHADOW* Esta función se utiliza dentro del *Vertex Shader*. Convierte la posición del vértice a coordenadas de luz mediante la matriz *unity_WorldToShadow[0]* y asocia esas coordenadas a las coordenadas de sombra.
- *SHADOW_ATTENUATION* Esta función tiene que ser llamada dentro del *Fragment Shader*. Aprovechando que tanto el *shadow map* generado como las coordenadas de sombra están representados en coordenadas de luz, se puede realizar la comparación entre la distancia guardada en el *shadow map* y la distancia real entre el punto de la superfície y la luz.

La figura 78 muestra el proceso que seguido para realizar sombras proyectadas por *Spot Lights*.

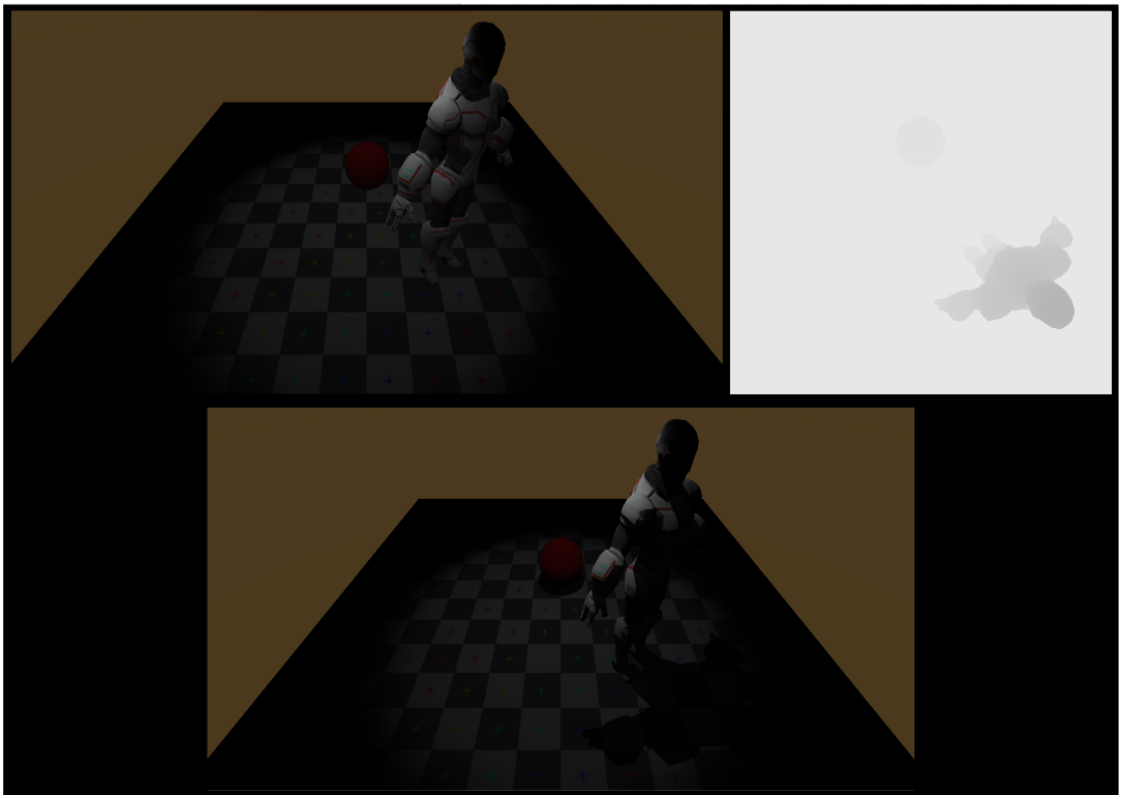


Figura 78: Proceso de proyección de sombras mediante *Spot Lights*.

6.2.4 Sombras semitransparentes

En la sección 4.3 se han visto los materiales semitransparentes. Como ya se explica en ese apartado, la semitransparencia se consigue realizando una interpolación lineal entre los colores del objeto y los colores que ese mismo objeto tiene detrás siguiendo la dirección en la que mira el observador. A pesar de que mediante este método se consigue el resultado esperado sobre en el objeto, éste sigue teniendo una forma definida. A la escena de los ejemplos anteriores se ha añadido un plano con semitransparencias y una textura de humo.

Esta escena está representada en la figura 79, donde se observa cómo, a pesar de que el plano es semitransparente, la sombra es uniforme y define la silueta del plano original.

La solución que se acostumbra a utilizar para resolver éste problema es la siguiente. La figura 80 muestra una serie de texturas de 4x4 píxeles con patrones dibujados. Estas texturas, llamadas *dither textures*, están ordenadas en función de la cantidad de puntos negros que tienen, empezando por 0 y terminando por 16, donde cada textura tiene un punto negro más que la textura anterior. Partiendo de que la probabilidad de encontrarte un punto negro al azar pasa de un 0% a un 100% entre la primera y la última textura y que el nivel de transparencia de un

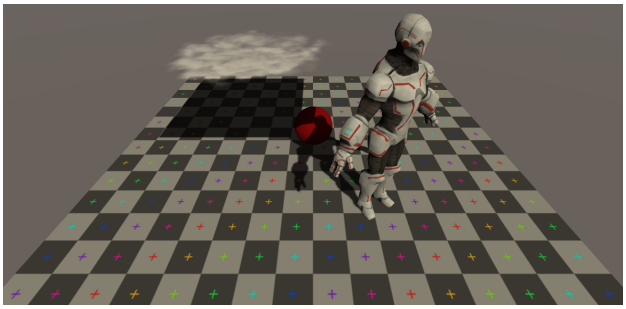


Figura 79: Escena con un plano semitransparente con humo que proyecta una sombra uniforme.

punto está también entre estos dos porcentajes, se puede establecer una relación entre estos dos valores.

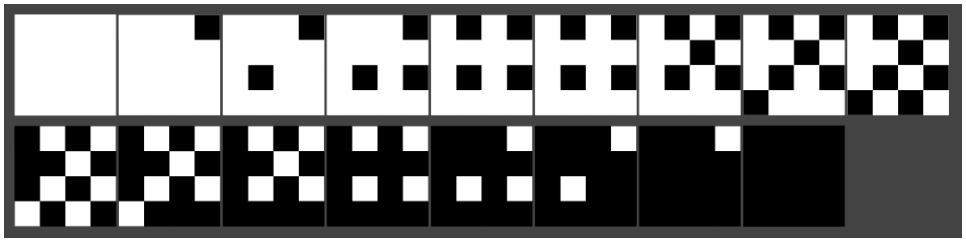


Figura 80: *Dithering textures.*

El método es el siguiente:

- Suponiendo una superfície semitransparente con una textura con diferentes valores en el canal *alpha*. Antes de transformar la posición del objeto a coordenadas de luz para construir el *shadow map* se obtiene el valor del canal *alpha* de la textura en ese punto.
- Se realiza un muestreo sobre las *dither textures* utilizando el valor del canal *alpha*, de manera que, teniendo en cuenta que el numero total de *dither textures* es de 17 y asumiendo un valor de *alpha* igual a x, se realice el muestreo sobre la *dither texture* número $16(x - x \ mod \ \frac{1}{16})$.
- El valor resultante del muestreo de la textura puede ser tanto 0 como 1. Restando a este valor un número muy bajo, una intrucción *clip* se encarga de descartar aquellos puntos que resulten en 0.

Para implementar este método en Unity se ha definido una variable macro con el nombre *SEMITRANSPARENT_SHADOWS*. Cuando esta variable está activada la estructura de entrada al *Vertex Shader* añade un campo para almacenar las coordenadas de la textura con la transparencia en el canal *alpha* que da color al albedo del objeto.

Dentro del *Vertex Shader* se define la posición en coordenadas de *clip*, asociándola a la palabra reservada o *semantic SV_POSITION*. Sin embargo, para realizar la

muestra sobre las *dithering textures* se necesita la posición de la pantalla. Esta posición se consigue utilizando el *semantic VPOS*. Debido a que la muestra se realiza dentro del *Fragment Shader* debe existir un campo dentro de la estructura de entrada al *Fragment Shader* que utilice esta palabra. No obstante, los dos *semantics* no pueden estar dentro de la misma estructura de entrada al *Fragment Shader* ya que Unity lo entiende como dos maneras diferentes de asociar la posición al objeto en la misma estructura. La solución a este problema es generar una nueva estructura de entrada al *Fragment Shader* que contenga los mismos campos que la otra, pero utilizando el *semantic VPOS*. De esta manera, la posición en coordenadas de *clip* se asocia en la salida del *Vertex Shader*, mientras que al *Fragment Shader* entra la estructura con las coordenadas de pantalla del objeto, ya calculadas por Unity.

La textura que contiene todas las *dithering textures* tiene por nombre *_DitherMaskLOD* y es en realidad lo que se conoce como una textura 3D. Una textura 3D es simplemente una manera de almacenar varias texturas 2D en diversos niveles. Las texturas 2D que forman la textura 3D *_DitherMaskLOD* se corresponden con las mostradas en la figura 80.

Los resultados obtenidos se muestran en la figura 81.

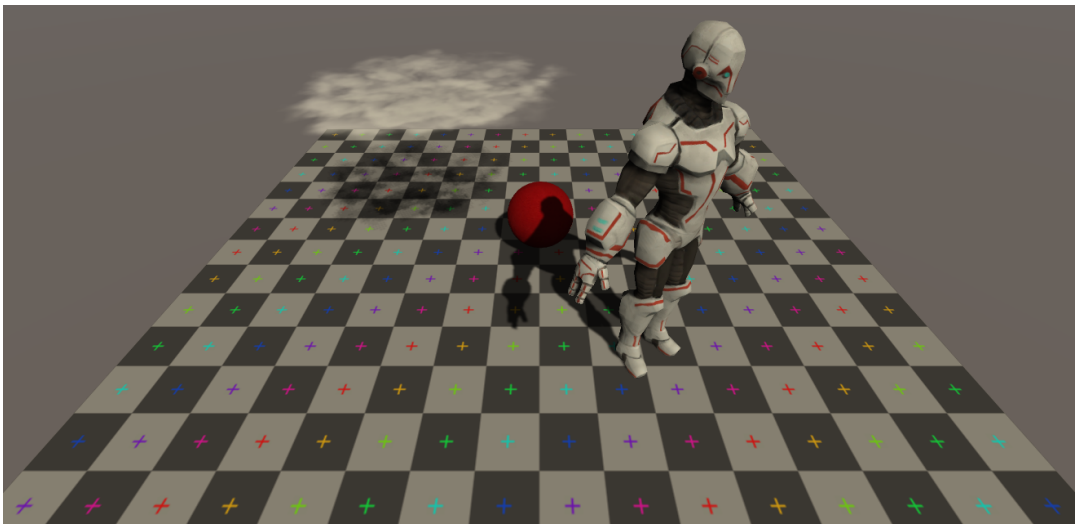


Figura 81: Resultado de la correcta proyección de sombras por parte de objetos semitransparentes.

Se debe tener en cuenta que el hecho de descartar *fragments* de la superfície que obstruye la luz hace que la sombra no sea uniforme y por lo tanto se observen patrones como los de la figura 82. Otro de los problemas que causa este método es debido a la utilización de las coordenadas de cámara para realizar la muestra sobre las *dithering textures*. En una escena donde, o bien la cámara se mueve, o bien se mueve el objeto, éstas coordenadas cambian continuamente, y por lo tanto se observa un efecto por el que los píxeles aparecen y desaparecen. Este es uno de los motivos por el que este tipo de sombras no se suelen utilizar, no obstante, pueden resultar útiles para objetos estáticos y lejanos.

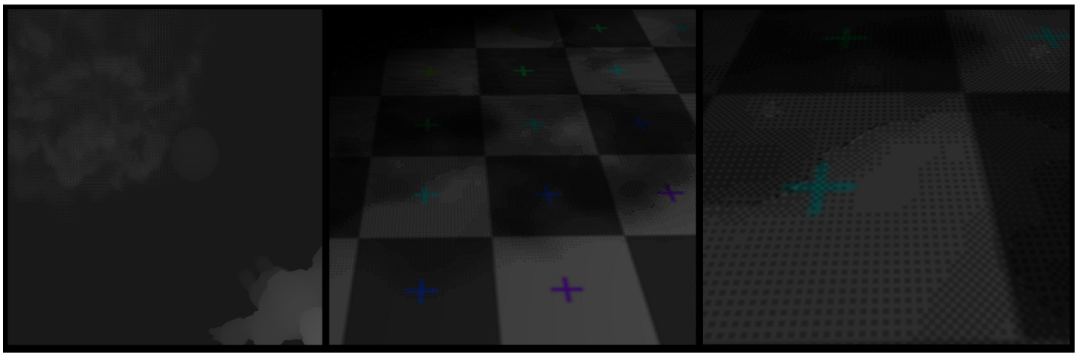


Figura 82: Resultado de proyectar con *Spot Lights* sombras semitransparentes utilizando *dither textures*. Izquierda: *shadow map*. Centro y derecha: muestra de la sombra sobre la escena.

6.2.5 Problemas y soluciones

6.2.5.1 *Aliasing* de perspectiva

Un problema recurrente a la hora de generar sombras es la aparición de *aliasing* en la parte exterior de éstas. Como en todo tipo de *aliasing*, el problema se debe a la naturaleza discreta del muestreo. Cuanto menor sea la resolución del *shadow map* mayor *aliasing* existirá.

Una solución a este problema cuando se utilizan luces direccionales en Unity pasa por utilizar una cascada de *shadow maps*, cada uno con una resolución diferente. Como se puede ver en la figura 74, esta tarea genera varios *shadow maps* dentro de la misma textura desde varias distancias en lo que se denomina un atlas de cascadas. En la figura también se puede observar como cuanto mayor es la distancia a la que se ha generado el *shadow map*, más elementos entran en este. Sin embargo, cuantos más elementos entren menor será la resolución por cada objeto.

A la hora de realizar la muestra sobre la cascada de *shadow maps*, en función de la distancia de la cámara a la que se encuentre la superfície que recibe la sombra, se realizará la muestra sobre un *shadow map* o sobre otro. De esta manera, los *fragments* de una superfície que estén más cerca de la cámara realizarán el muestreo sobre el *shadow map* con mayor resolución, mientras que aquellos *fragments* lejanos respecto de la cámara lo harán del *shadow map* con menor resolución. Ésta técnica, por tanto, hace que las sombras más cercanas al observador tengan menos *aliasing* en el exterior y por lo tanto más calidad mientras que aquellas más alejadas tengan menos.

El hecho de que las luces direccionales sean el único tipo de luz que soporta cascadas es debido a que es también, de los tres tipos de luz explicados, el único que no parte de un punto de origen. De esta forma, un *shadow map* puede ser llevado a mucha distancia de la escena sin que esto afecte realmente al propio *shadow map*.

Unity tiene una distancia máxima que puede ser modificada por el usuario a través de la variable *Shadow Distance* del apartado *Quality Settings* que se muestra en la figura 72. No se verán sombras más allá de esa distancia. Está dividida en n

partes, donde n es el número de cascadas que se han introducido en el desplegable *Shadow Cascades* del mismo apartado, que pueden ser 0, 2 o 4. Existe también el elemento *Cascade Splits*, que permite modificar la distancia a la que se van a realizar los *shadow maps* de la cascada. La distancia a la que se toman los diferentes *shadow maps* viene marcado por el final de cada sector siendo siempre el último sector del elemento la distancia máxima.

Cuanto más cerca se saquen los *shadow maps* más resolución tendrán estos y por lo tanto menos *aliasing*. Sin embargo a medida que la resolución se hace pequeña con las cascadas el efecto del *aliasing* es mayor. La figura 83 muestra la mejora que hay al cambiar una resolución alta de *shadow map* por otra más baja.

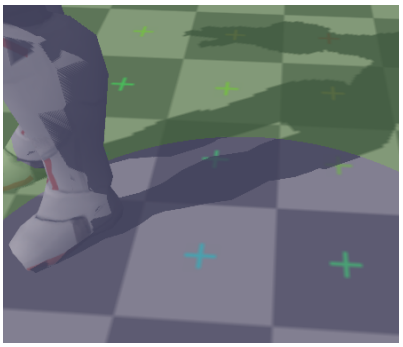


Figura 83: Muestra del cambio entre resoluciones de *shadow map*.

A pesar de que utilizar una cascada de *shadow maps* produce buenos resultados si las distancias de la cascada se equilibran bien, esta solución sólo es válida para luces direccionales. Otro método para mejorar el *aliasing* que sí funciona para todos los tipos de luces es el que se conoce como *soft shadows*. Para activar este método simplemente hay que marcar esa opción en el desplegable *Shadow Type* del componente *Light*, mostrado en la imagen de la derecha en la figura 72. Activar este modo hace que una superfície, a la hora de recibir sombras, realice una media entre los cuatro pixeles que envuelven diagonalmente al punto que se debería muestrear de manera que los bordes de la sombra queden suavizados, simulando una penumbra. Los resultados de aplicar este método para luces direccionales se muestran en la figura 84.

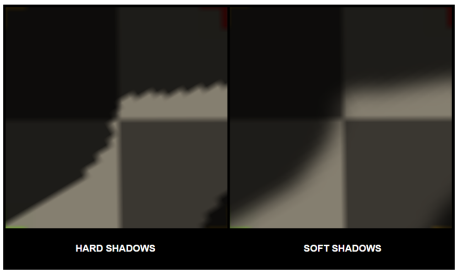


Figura 84: Izquierda: *Aliasing* con *hard shadows*. Derecha: *Aliasing* con *soft shadows*.

6.2.5.2 *Shadow acne*

En la figura 86a se pueden observar algunos puntos de ruido donde no debería haber ninguno. Este problema es conocido como *shadow acne*, explicado en [17]. El *shadow acne* aparece en la comparación de la profundidad por dos motivos:

- El primer motivo es la precisión de los valores de profundidad. Puede darse que, o bien las profundidades guardadas en el *shadow map* o bien las profundidades reales, estén calculadas utilizando una precisión muy baja, de manera que a la hora de realizar la comparación de las profundidades ese pequeño error haga que algunos pixeles estén sombreados cuando no deberían.

- El segundo motivo se debe a la naturaleza discreta del *shadow map*. El *shadow map* es una textura, y por lo tanto tiene una resolución. Cuanto más baja es ésta resolución, más *fragments* de la escena representa cada pixel del *shadow map*, cuando lo ideal es una proporción de $1:1$. En la figura 85, la dirección de la luz forma un ángulo de θ grados con la normal de la superficie, donde θ está cerca de los 45°. La muestra del *shadow map* representa sólo la distancia al medio del conjunto de *fragments*, pero en los extremos de ese conjunto puede estar representando más o menos distancia de lo que en realidad es, lo cual acaba haciendo que haya zonas oscuras y zonas claras.

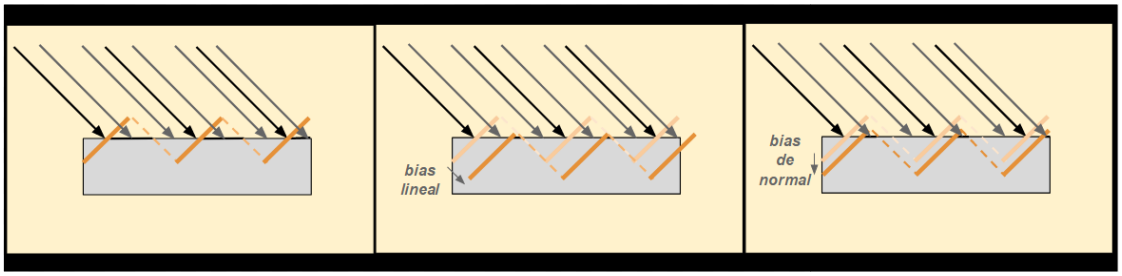


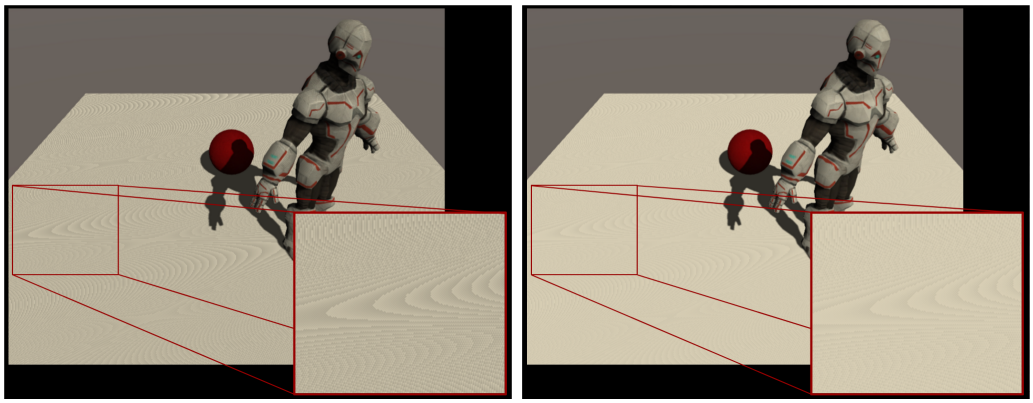
Figura 85: Esquema de la aplicación de bias de normal.

La solución a este problema pasa por añadir un *bias* que fuerce que tenga que existir una mayor diferencia entre las distancias del *shadow map* y la distancia real para poder considerar que haya sombra en un punto. Existen dos tipos de *bias*:

- El primer tipo de *bias* es el *bias* lineal. El *bias* lineal desplaza la superfície del objeto que obstaculiza la sombra en la dirección de la luz. Como ya se ha explicado en el apartado 6.2.1.1, la función de Unity *UnityApplyLinearShadowBias* aumenta las componente z de las coordenadas de *clip* de manera que más adelante cuando Unity traslada la posición a coordenadas de luz la distancia almacenada aumenta en la dirección de la luz como se muestra en la figura 85.

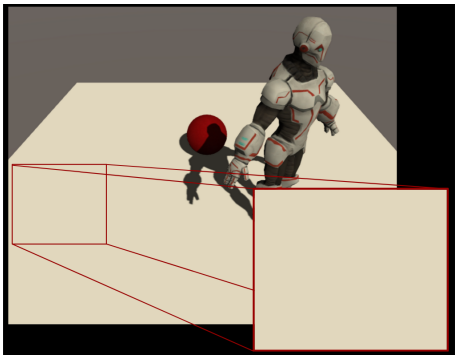
- El segundo tipo de *bias* es el *bias* de normal. El *bias* de normal utiliza el vector normal a la superfície del objeto para introducir la superfície del objeto que obstruye la luz hacia dentro como muestra la figura 85. Como se ha

explicado en el apartado 6.2.1.1 La función de Unity *UnityClipSpaceShadowCasterPos* implementada dentro del paso *ShadowCaster* aplica este tipo de *bias*. Dentro la función utiliza el seno entre el vector de la luz y la normal (0° cuando el ángulo entre éstos sea 0 y 1 cuando sea 90°) junto con la variable *unity_LightShadowBias.z*, que está relacionada con el slider *"Normal Bias"* que aparece en el inspector de una luz direccional, para desplazar en dirección contraria a la normal la posición, como se muestra en la figura 85.



(a) Escena sin bias en la que se observa *shadow acne*.

(b) Bias de normal = 0.3.



(c) Bias de normal = 0.3 y bias lineal = 0.01.

Figura 86: Resultados obtenidos modificando el bias lineal y el bias de normal de la luz direccional.

6.2.5.3 *Peter Panning*

Precisamente la solución del problema anterior nos lleva a otro problema: el *peter panning*. Al intentar solucionar el problema del *shadow acne*, podemos acabar introduciendo otro problema, que aunque menos grave, no deja de ser molesto y puede llegar a influir en la percepción del espacio del jugador. El *peter panning* se produce cuando se introduce un *bias* demasiado alto, de manera que las sombras se acaban desplazando como muestra la figura 87. La solución pasa por reajustar el *bias* de manera que se equilibre el *shadow acne* con el *peter panning*.

Figura 87: Error de *peter panning* usando un valor de *bias* lineal de 0.6.

7 Visualización de terrenos

Los terrenos son una parte fundamental en un juego ya que forman aquella superfície sobre la que se sostiene toda la escena. En un terreno es posible ver diferentes tipos de elementos propios de la naturaleza como hierba, agua o niebla. Estos elementos son precisamente los que se han tratado de implementar en este proyecto utilizando los conceptos explicados en apartados anteriores.

7.1 Creación de hierba

A la hora de crear hierba para situarla en el terreno del juego se han comparado dos métodos diferentes. La primera de ellas es utilizando un *Geometry Shader* y la segunda utilizando planos con texturas.

7.1.1 Hierba con *Geometry Shader*

7.1.1.1 Geometry Shader

Para comprender mejor cómo se ha conseguido realizar este tipo de hierba, primero es necesario entender qué es un *Geometry Shader*. Un *Geometry Shader* es un programa *shader* ejecutado en la GPU similar al *vertex* o al *Fragment Shader* y que se ejecuta entre la salida y la entrada de estos dos programas. Es decir, recibe grupos de vértices procesados por el *Vertex Shader* en forma de punto, linea o triángulo, según se indique, y tiene como salida uno o más triángulos que más tarde se rasterizarán y pasarán por el *Fragment Shader*. La figura 88 muestra la situación del *Geometry Shader* dentro del proceso de renderización de un triángulo.

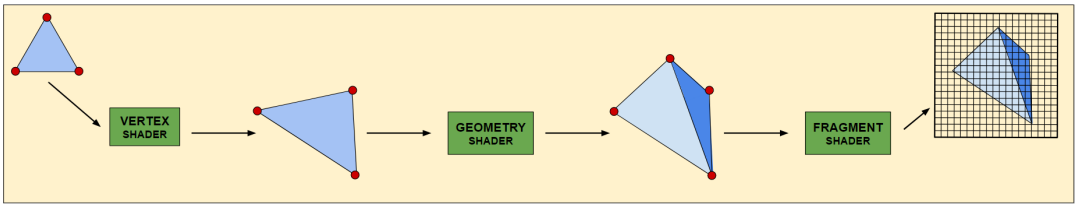


Figura 88: Esquema del proceso de renderizado.

7.1.1.2 Modelado de hierba

Explicado esto, ya se puede proceder a describir cómo se ha generado esta modalidad de hierba. El proceso que se ha seguido es el siguiente.

Se ha creado un *Geometry Shader* tal que dado un modelo tridimensional, formado por vértices que a su vez forman triángulos, se quiere generar una pirámide por cada triángulo que exista en ese modelo. Esta pirámide haría las veces de hebra y por lo tanto un modelo que estuviese formado por n triángulos, tendría n hebras de hierba. En el caso de este proyecto el modelo escogido es un plano formado por

200 triángulos. La información que llega al *Geometry Shader* en este caso son tres vértices en forma de triángulo con la información sobre la posición del vértice en coordenadas de objeto, y deben salir a través de un *stream* los triángulos que se generen con la información necesaria para el *Fragment Shader* en cada punto del triángulo saliente. Entre esa información necesaria se encuentra el vector normal al triángulo en coordenadas de mundo. Este vector se consigue realizando un producto vectorial entre los vectores que forman los vértices del triángulo, como se muestra en la figura 89. Es necesario indicarle al *Geometry Shader* el numero máximo de vértices que pueden salir de él. En el shader que se ha creado para este tipo de hierba se ha añadido una variable entera con el nombre *_Sections* para seleccionar el número de divisiones que se puedan llegar a hacer a una hebra.

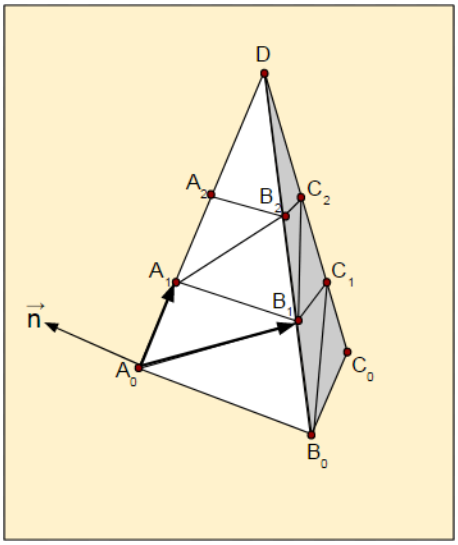
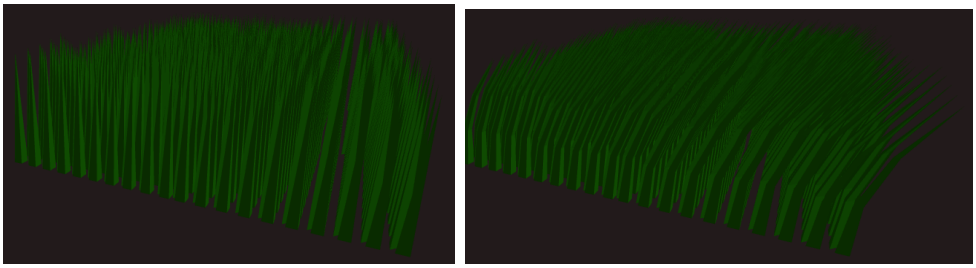


Figura 89: Esquema que muestra el cálculo de las normales en la pirámide.

La figura 90 muestra imágenes de una hebra formada por una y tres secciones respectivamente. Consiste en, a partir de los tres vértices que forman el triángulo inicial o base, generar los tres siguientes sumando a estos el vector que va desde cada vértice al centro de los tres dividido entre el número de secciones, y para que la hebra crezca se suma a la componente y la altura dividida entre el número de secciones. Además también se suma una vez creados el vector de movimiento $\vec{w}$ que se explicará más adelante cómo se ha calculado. Con estos seis vértices se generan los seis triángulos que se necesitan por cada sección que no finalice en punta. Una vez pasados estos seis triángulos al *stream* los vértices de la base pasan a ser los tres que se habían generado antes y se calculan los siguientes. Este proceso se repite hasta que se llega a la última sección, donde simplemente se forman tres triángulos que convergen en forma de pirámide.

El *shader* que se ha implementado para realizar este tipo de hierba está situado dentro de la carpeta *"Assets/Shaders/Others/Grass"* que lleva por nombre *GeometryGrass* y que tiene como dirección o *path* *"Custom/Grass/GeometryGrass"* dentro del sistema de *shaders* de Unity . El algoritmo de división de la hierba está implementado dentro de la función *geom* de este *shader*.



(a) Conjunto de hierba con una única sección.

(b) Conjunto de hierba con tres secciones.

Figura 90: Conjunto de hierba con diferente número de secciones.

7.1.1.3 Movimiento de la hierba

El comportamiento de la hierba es de tipo sinusoidal y por tanto se utilizará una función de la forma $f(x) = A \cdot sin(\omega x + \varphi)$, donde A representa la amplitud del seno, o cuanto de lejos puede desplazarse la hebra, ω representa la velocidad angular, o cómo de rápido irá de un lado a otro la hebra y finalmente φ, que es la fase inicial del sinusoide, permitirá desplazar el seno según la posición de la hebra, dando así el efecto que tiene un golpe de aire que atraviesa un campo de hierba.

Con el fin de que todas las hierbas no se muevan al unísono es necesario introducir un elemento de aleatoriedad. Una de las posibilidades, ya que la función *random* del lenguaje *CG* no se puede utilizar dentro del *Vertex Shader* ni el *Geometry Shader* es introducir una textura de ruido que pueda dar un valor diferente a cada vértice, sin embargo hacer un muestreo en un shader que no sea el *Fragment Shader* es una mala idea, ya que la calidad de la textura muestreada es demasiado baja debido a que se hace sólo una muestra por vértice. Esta opción queda, por tanto, descartada. La solución que se ha seguido ha sido generar un número que dependa de las componentes x y z de la posición en coordenadas de objeto del centro del triángulo, de manera que cada hebra del plano tenga un único valor. Los elementos que forman la función que se ha utilizado son,

$$\varphi = abs(P_x + P_z) mod \quad 2\pi \tag{7.1}$$

$$\omega = 1 + \frac{\varphi}{2\pi} \tag{7.2}$$

$$A = h \cdot w_i(\omega - 1) \tag{7.3}$$

El valor de φ se ha escogido de forma que se pueda definir como $\varphi \in [0, 2\pi]$ en función de la posición (P) de cada hebra. Por otra parte ω está incluida en el rango $\omega \in [1, 2]$ y finalmente la amplitud depende de la altura (h), la intensidad del aire (w_i) y un valor que añada una relativa aleatoriedad comprendido entre 0 y 1. Es imprescindible que la amplitud dependa de la altura ya que de esta forma se evita que se muevan las raices de la hebra, con altura cero, y se consigue por otra parte que se produzca una curvatura a medida que ésta aumenta.

Finalmente el vector $\vec{w}$ que representa el vector de la fuerza ejercida por el viento

sobre la hierba queda de la forma,

$$\vec{w} = \vec{w_d} h \cdot w_i \cdot (1 + \frac{A}{h} \sin(\omega t + \varphi)) \tag{7.4}$$

Donde t es el tiempo que ha pasado desde que ha empezado el juego. Se puede observar que el vector $\vec{w}$ se divide en dos partes en que la primera parte es $\vec{w_d} \cdot h \cdot w_i$ y la segunda es $A \sin(\omega t + \varphi)$. La primera se encarga de doblar las hebras en función de la altura, la intensidad del aire y la dirección. La segunda parte tiene en cuenta a la primera pero además añade la función sinusoidal que hace que las hebras se muevan. En el proyecto se considera que el aire sopla únicamente en el plano xz y por lo tanto al vector $\vec{w}$ resultante de la fórmula anterior se le desactiva la componente y. En la figura 91 están representadas las dos fórmulas con las que se construye $\vec{w}$.

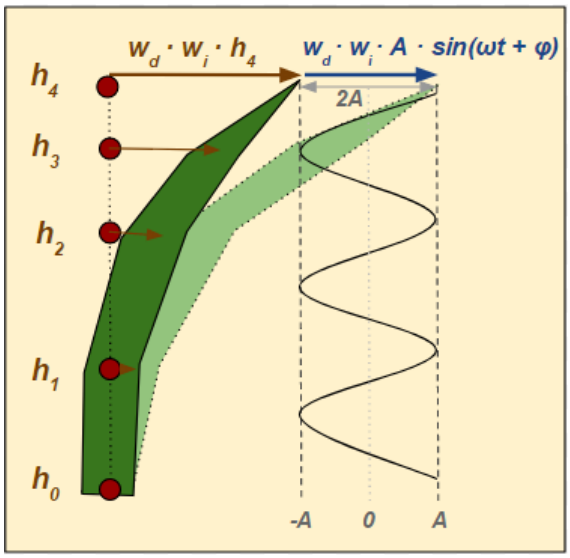
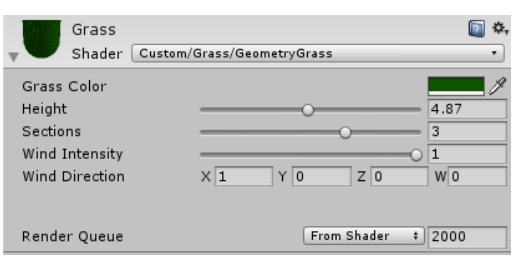


Figura 91: Esquema de los componentes del movimiento en la hierba.

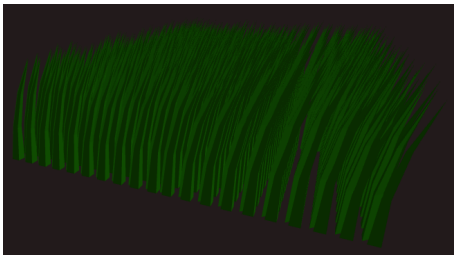
Con el fin de simular el movimiento que provoca el aire en la hierba de la forma explicada en este apartado se han introducido dos propiedades al *shader* con el objetivo de controlar la dirección y la intensidad. Estas dos propiedades tienen como nombre *_WindDirection* y *_WindIntensity* respectivamente.

Este método tiene, sin embargo, algunas desventajas, como es por ejemplo la dependencia que tiene respecto al número de triángulos del modelo. Otro problema relacionado con los triángulos de los modelos pasa por la regularidad con la que se generan las hebras, lo cual se debe a la regularidad inherente a la estructura de triángulos de un modelo. Por otra parte es un método costoso ya que se generan varios triángulos aumentando este número aún más cuando se fracciona en secciones. Es por estas razones que no se ha considerado como una opción viable para la versión final del proyecto.

La figura 93 muestra el resultado de este tipo de hierba en la escena.



(a) Inspector del material *Geometry-Grass*.



(b) Resultado del *geometry grass* con tres secciones.

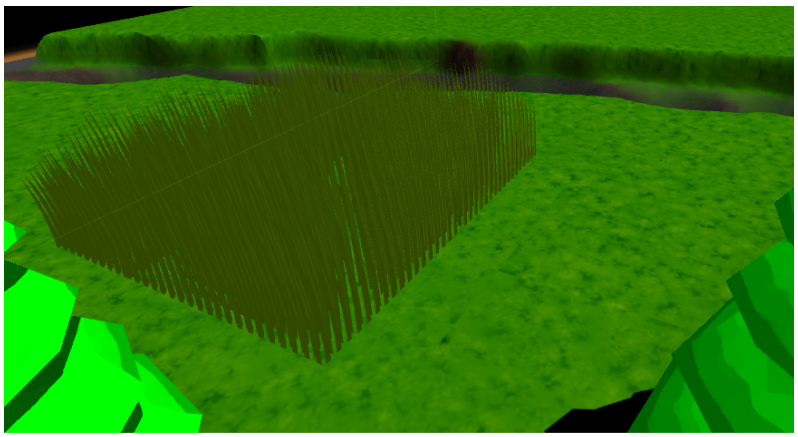


Figura 93: Hierba de tipo *Geometry* integrada en la escena.

7.1.2 Hierba con textura

El proceso para generar hierba mediante un *Geometry Shader* es muy costoso, por el contrario, el método para crear un campo de hierba sólo con texturas es bastante ligero como se va a ver a continuación. Este método consiste simplemente en crear un plano y proyectar una textura de hierba con transparencias sobre él. Las texturas utilizadas se muestran en la figura 94.

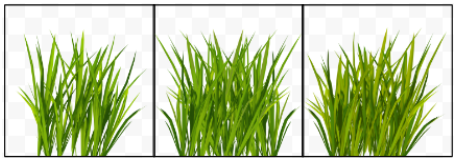


Figura 94: Texturas utilizadas para simular hierba.

Para entender cómo se ha implementado hierba mediante texturas antes se ha de explicar una técnica llamada *Billboard shading*.

7.1.2.1 Billboard shading

El *Billboard shading* es una técnica que se utiliza para hacer que un objeto plano dé la impresión de tener un volumen. En la figura 95 izquierda se muestra cómo en función de dónde se coloca el observador (cámara) se ve claramente que un plano

simple no tiene volumen. Sin embargo, en la misma figura a la derecha, se ve cómo si el observador está mirando desde otra posición, no se percibe que el plano no tiene profundidad. Si se consigue que el plano se gire automáticamente cuando cambia la posición del observador de manera que siempre quede de frente, el observador no percibirá que el plano no es grueso. Este es el efecto que produce el *Billboard shading*.

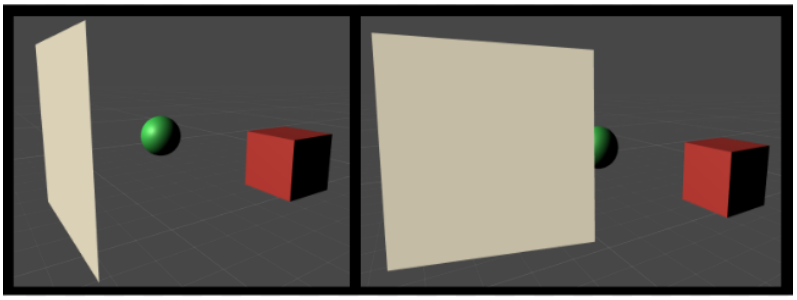


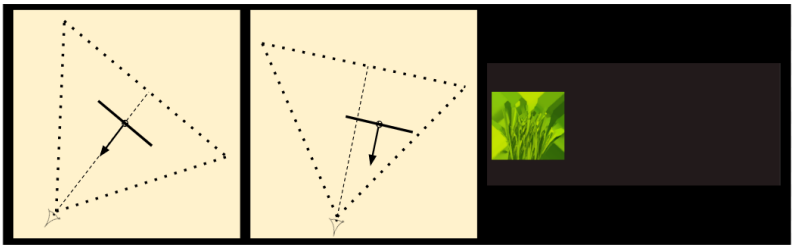
Figura 95: A la izquierda un plano que no implementa *billboard*. A la derecha un plano que implementa *billboard*.

El *Billboard shading* se puede implementar por dos métodos diferentes en función del vector normal a la superfície del plano:

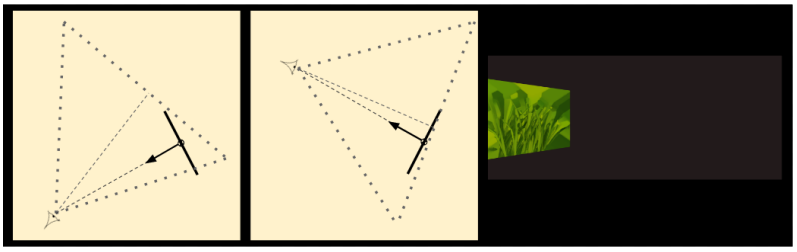
- En el primer método el plano tiene como vector normal un vector que depende de la dirección en la que mira el observador. El plano que implementa *Billboard shading* por este método tiene un problema que se debe a que el observador no sólo ve aquello que se encuentra en la dirección en la que mira, representado por una línea discontinua en las imágenes, sino que también puede ver lo que sucede en los extremos del cono, de manera que cuando la cámara está cerca del plano pero a un lado, se puede llegar a distinguir cómo sólo se trata de un plano. En la figura 96a se muestra este método para calcular el vector normal del plano.

- En el segundo método el plano tiene como normal el vector que forma la resta entre la posición del centro del plano y la posición del observador. En este proyecto se ha utilizado el método que tiene en cuenta la posición, ya que de esta manera el objeto siempre mirará hacia el observador. La figura 96b muestra el resultado de utilizar este método de cálculo del vector normal del plano.

Para realizar el segundo tipo de *Billboard* se parte del centro del modelo, representado por el punto $(0, 0, 0)$ en coordenadas locales de objeto, para transformarlo a coordenadas de mundo multiplicando este punto por la matriz modelo. De esta manera se consiguen las coordenadas del centro del objeto en la escena. Una vez se ha conseguido el centro, este se tiene que desplazar en cada vértice ya que si no todo el objeto estaría concentrado en el punto central del plano. Este desplazamiento se consigue sumando las componentes de la posición del vértice en coordenadas de objeto a la posición central en coordenadas de mundo. A pesar de realizar operaciones entre vectores representados en diferentes sistemas de coordenadas, esta es



(a) Muestra de un plano con *billboard* que gira en función de la dirección de visión del observador.



(b) Muestra de un plano con *billboard* que gira en función de la posición del observador.

Figura 96

una buena opción teniendo en cuenta que las coordenadas de objeto representan la distancia respecto al centro del modelo. Sin embargo, este tipo de coordenadas, no cambian en función de la escala, de manera que en caso que se quisiera modificar el tamaño de un objeto que utiliza la técnica del *billboard* se tendría que hacer mediante un valor introducido por el usuario final. Respecto a las direcciones en las que se tienen que desplazar los vértices, estas conforman la base del plano *billboard* y dependen tanto de la posición de la cámara respecto del objeto en la escena como de dos de los vectores que definen la rotación de la cámara.

La cámara de una escena está orientada según tres vectores que se muestran en la figura 97. El vector $\vec{v}$ o *view* es el que representa la dirección en la que la cámara está mirando, el vector $\vec{u}$ o *up* es el que representa la dirección en la que está girada la cámara respecto al eje y de la escena y finalmente el vector $\vec{r}$ o *right* apunta hacia el lado derecho de la cámara y es el resultado de realizar el producto vectorial entre los dos primeros vectores.

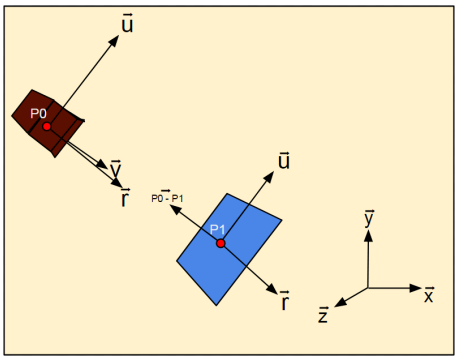


Figura 97: Vectores de una cámara.

Los vértices se desplazan en función del valor de las componentes x e y en coordenadas de objeto siguiendo la dirección de los vectores $\hat{u}$ y $\hat{r}$. La fórmula que define este proceso es la siguiente:

$$x' = P_1 + s_x x \cdot \hat{r} \tag{7.5}$$
$$y' = P_1 + s_y y \cdot \hat{u} \tag{7.6}$$

donde P_1 es el punto central del plano, s_x y s_y son las componentes que sirven para escala el plano final, x e y son la posición del vértice en coordenadas de objeto, los vectores $\hat{r}$ y $\hat{u}$ *right* y *up* respectivamente y x' e y' la posición final del vértice en coordenadas de mundo.

Hasta ahora el plano que implementa la técnica *Billboard shading* está orientado siempre de cara a la cámara, pero este hecho, que supone algo bueno si el observador se encuentra a la misma altura que el plano, se convierte en algo poco deseable si este mira el plano desde la altura. Esto por supuesto depende de aquello que se quiera representar con ésta técnica, no obstante en el caso de la hierba, produce los resultados expuestos en la figura 98a, donde el observador, pese a estar mirando a la hierba desde arriba, puede llegar a distinguir que cómo ésta está girada entorno a él.

(a) El vector y del plano es $\hat{u}$.

(b) El vector y del plano es $(0, 1, 0)$.

(c) El vector y del plano es la media entre $\hat{u}$ y $(0, 1, 0)$.

Figura 98: Resultados de calcular el vector y del plano mediante los métodos explicados.

Para este fallo hay dos posibles soluciones que pasan por modificar el vector y del plano. La primera se muestra en la figura 98b y consiste en utilizar como vector y en vez de $\hat{u}$ el vector $(0, 1, 0)$. Utilizando esta solución el plano sólo persigue al observador en el plano xy, sin embargo, esta solución sólo es válida si la cámara no está situada a demasiada altura ya que si es este el caso resulta demasiado obvio que se trata de un plano. La segunda solución se muestra en la figura 98c, y pasa por intentar encontrar un vector medio entre $\hat{u}$ y $(0, 1, 0)$. Éste vector medio se construye siguiendo la fórmula,

$$\vec{y} = \hat{u} + (0, 1, 0) \tag{7.7}$$

$$\hat{y} = \frac{\vec{y}}{||\vec{y}||} \tag{7.8}$$

Esta parte afecta al vector y del plano, no obstante también será necesario modificar el vector x de cara a la implementación que se ha realizado en este proyecto, y es que más adelante el observador tendrá que ver el plano con *billboard* girado en ciertos ángulos, lo cual implica realizar una rotación sobre $\hat{r}$. La fórmula para realizar esta rotación es la siguiente. Considerando θ un ángulo con radianes como unidad, la matriz de rotación R_y se construye como,

$$R_y = \begin{pmatrix} cos(\theta) & 0 & sin(\theta) \\ 0 & 1 & 0 \\ -sin(\theta) & 0 & cos(\theta) \end{pmatrix} \tag{7.9}$$

Y por lo tanto considerando que $\hat{r'}$ es el vector *right* modificado,

$$\hat{r'} = R_y \cdot \hat{r} \tag{7.10}$$

Una vez rotado el vector $\hat{r'}$, es necesario recalcular el vector normal de la superfície, ya que si el plano gira sobre su eje y la normal inevitablemente cambiará. Para ello se realiza el producto vectorial entre el nuevo vector $\hat{r'}$ y $\hat{y}$.

$$\vec{n} = \hat{r'} x \hat{y} \tag{7.11}$$

$$\hat{n} = \frac{\vec{n}}{||\vec{n}||} \tag{7.12}$$

7.1.2.2 Modelado

La hierba que se ha creado en este proyecto está formada por un plano y una textura que aplica transparencias utilizando el método *AlphaToMask* explicado en la sección 4.2. Este método está descrito en la página de documentación de Unity como específico para casos como la hierba debido a la cantidad de planos con transparencias que hay en un campo lleno de plantas. El resultado de aplicar las transparencias a las tres texturas que se han mostrado anteriormente se muestra en la figura 99.

Un posible paso para conseguir un campo repleto de hierba es añadir varios planos con texturas de hierba y *billboard* en línea o de forma que quede un campo que tenga una apariencia de ser denso. Esto es correcto y produce buenos resultados, pero la parte inferior del plano puede resultar muy plana y por lo tanto puede perder el efecto del volumen. Una posible solución a esto es que cada unidad de hierba no esté

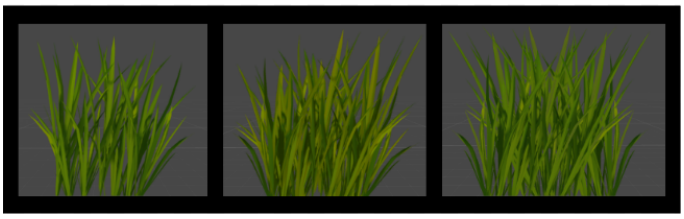


Figura 99: Texturas en la figura 94 aplicadas sobre el plano.

sólo formada por un plano, si no que esté formada por varios planos cruzados entre sí. Considerando que una unidad de hierba está formada por n planos cruzados entre sí por el centro, el ángulo θ que existente entre los planos es $\frac{360}{2n}$ grados. En las figuras 100 y 101 muestra los resultados obtenidos según el número de planos que se han cruzado.

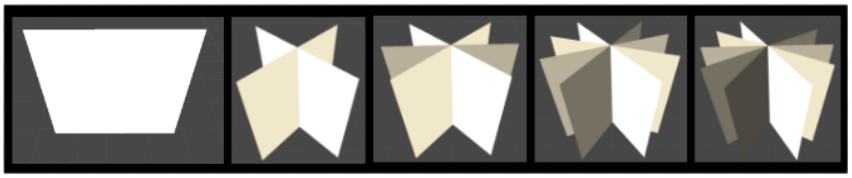


Figura 100: Muestra de los planos cruzados sin texturas.

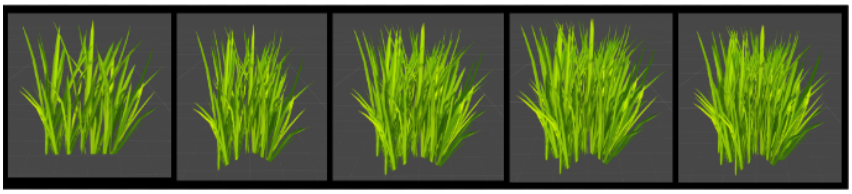


Figura 101: Muestra de los planos cruzados con texturas.

Como se puede ver con más de un plano existe un mayor efecto de volumen, no obstante, a partir de tres planos el efecto que se logra es prácticamente el mismo y por lo tanto lo más óptimo es utilizar entre uno y tres planos.

7.1.2.3 Movimiento de la hierba

El cálculo del movimiento de la hierba se realiza de forma similar a la utilizada en la hierba creada con el *Geometry Shader*, explicada en la sección 7.1.1.3. Una de los principales problemas a la hora de generar un movimiento en la hierba viene dado al uso de texturas y diversos planos para una sola unidad. Una de las principales diferencias frente al anterior tipo de hierba es que, al contrario que en la *geometry grass* esta hierba aplica aleatoriedad sin tener en cuenta la posición de cada vértice en coordenadas de mundo, ya que como cada vértice tiene una posición ligeramente diferente en coordenadas de mundo, el plano se puede llegar a deformar demasiado, llevando a efectos no deseados de texturas distorsionadas.

Para implementar este tipo de hierba en Unity se ha creado un *shader* con el nombre *"Grass_CrossBillboard"* con las propiedades que se observan en la figura 102,

además de la dirección e intensidad del viento, ocultas para el *Inspector*. Este shader está formado por un *Vertex Shader* en el que se realiza la mayoría de cálculos y un *Fragment Shader* que calcula sólo la reflexión difusa de la luz. Dentro del *Vertex Shader* se realizan los cálculos tanto del *billboard shading* como de la dirección en la que se moverá la hierba. Respecto el cálculo del *billboard shading* existe una diferencia entre la teoría que se ha explicado y la aplicación en Unity. Cuando en Unity se crea un plano, éste se crea en el plano xz y por lo tanto a la hora de desplazar los vértices del centro del plano en vez de utilizar la componente y del plano en coordenadas de objeto se utiliza la componente z.

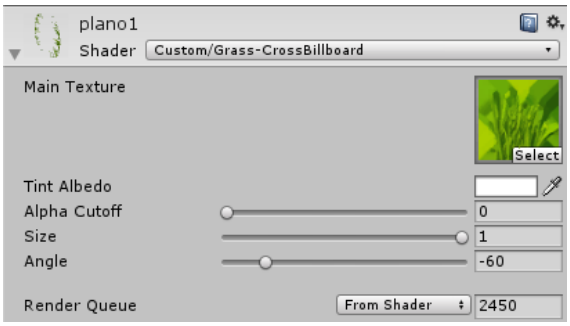


Figura 102: Muestra del inspector del material.

Son dos los valores aleatorios que se utilizan dentro del *shader*. Se generan desde un *script* externo y se aplican a unas propiedades ocultas dentro de este con nombres *_Seed* y *_SinSeed* declaradas como *uniform*, de manera que todos los vértices del mismo plano compartirán estos valores. Pero no sólo los de un mismo plano, ya que estos valores aleatorios se generan una única vez, en la creación de la unidad, son únicos para todos los planos que formen dicha unidad. De esta forma se evita que cada plano se mueva hacia un lado diferente o con una fase distinta al resto. El valor *_Seed* es un valor situado en el conjunto $[0, 1]$ que se utiliza dentro del cálculo de la amplitud de la onda, mientras que el valor *_SinSeed* es un valor que depende de las componentes x y z de la posición en la escena del centro de cada unidad y como se utiliza únicamente en el cálculo de la fase (ϕ) de la onda puede ser tan grande como sea necesario. La velocidad angular (ω) se ha decidido que sea igual a 1. La fórmula, por lo tanto queda como,

$$\phi = seed_{sin} \tag{7.13}$$
$$\omega = 1 \tag{7.14}$$
$$A = I + seed \tag{7.15}$$

$$s = \frac{A}{2} sin(\omega \cdot t + \phi) \tag{7.16}$$
$$s = \frac{I + seed}{2} sin(t + seed_{sin}) \tag{7.17}$$

donde s es el valor de la función sinusoidal, I es la intensidad del viento y t es la variable que marca el tiempo.

7.1.2.4 Generación automática

Para generar un campo de hierba automáticamente se ha creado un *script Grass-Generator* en lenguaje *c#* que asociado a un plano genera unidades de hierba que rellenan su área. Este *script* es el responsable de asociar tanto las propiedades ocultas en el inspector de los materiales, el valor aleatorio y las propiedades relacionadas con el viento, como los aspectos generales del generador. Entre estos aspectos generales se encuentran aquellos que se pueden modificar desde el generador de hierba que se muestra en los inspectores de la figura 103. Los valores modificables del inspector se dividen en los que se pueden modificar mientras el juego está en marcha y las que no. Las variables que sólo son modificables cuando el juego no está en ejecución son el valor *Density*, que representa la densidad del campo, la variable *Wellness* representando lo cuidada que está la hierba y está asociada a la propiedad del shader *_TintColor*, *Variety* , que indica el grado de mezcla de las tres texturas de hierba que se utilizan y la variable *number of planes* que utiliza la propiedad *_Angle* dentro del *Vertex Shader* para orientar los planos según el número de éstos por los que estén formadas las unidades. Aquellas variables que se pueden modificar en la ejecución son la variable *Scale*, que aunque no recomendable, puede modificar en ejecución el tamaño general de los planos de las texturas utilizando la propiedad *_Scale*, las variables *Wind Direction* y *Wind Intensity* que modulan el movimiento de la hierba simulando el movimiento provocado por el viento en la hierba y finalmente, la variable *Billboard Y* activa el modo que decide cual de los tres tipos de vector *y* del plano se va a utilizar. En la figura 103 se muestran varias configuraciones del generador de hierba junto con sus resultados.

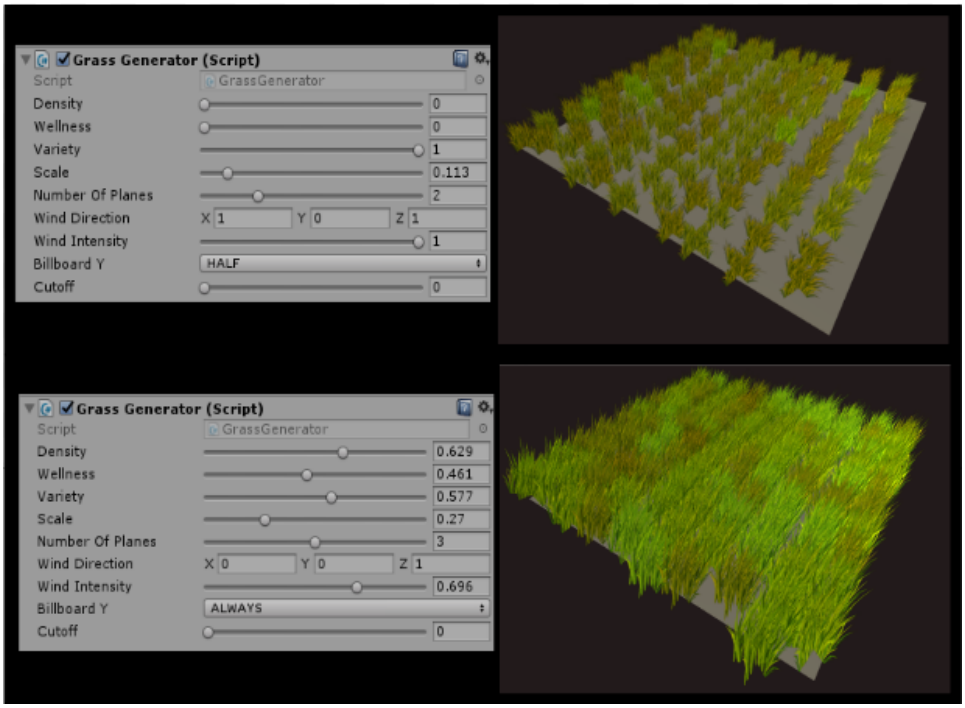
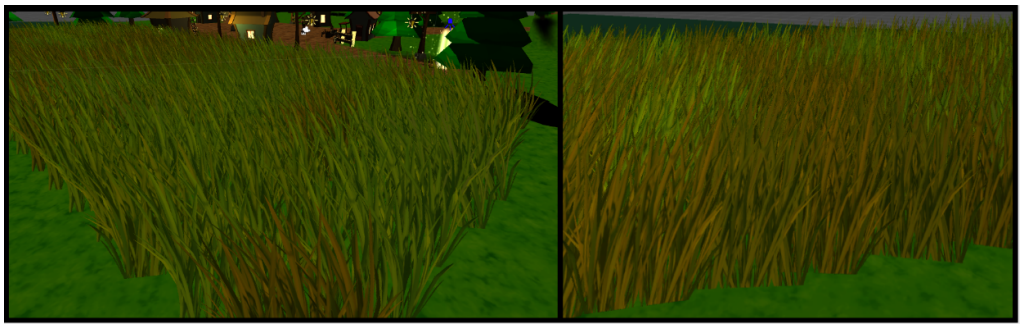


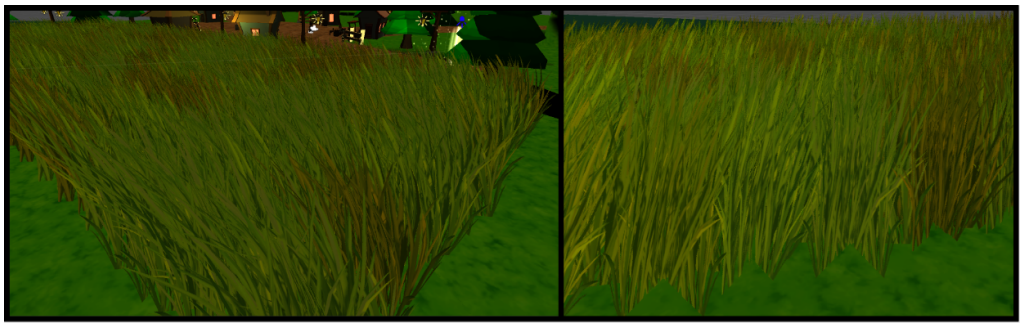
Figura 103: Muestra del inspector del generador de hierba junto con los resultados.

El resultado de la integración de este tipo de hierba en la escena se presenta en

las figuras 104 y 105.



(a) Hierba utilizando un único plano.



(b) Hierba utilizando tres planos cruzados.

Figura 104: Resultados de integrar *billboard grass* en la escena.

Figura 105: Vista general desde arriba de la escena con la hierba.

7.2 Creación de agua

En la escena del juego que se ha modificado aparecía un poblado rodeado por agua. En este proyecto, por tanto, ya existía agua antes de realizar las modificaciones, sin embargo, este agua era demasiado realista y visualmente desentonaba con el resto de la escena. Tomando como referencia el agua ya existente se ha generado

un *shader* que, aplicado a un objeto, generalmente un plano, simula agua con un estilo menos realista que encaja más dentro de la estética desenfadada del juego.

Como ya se ha explicado en el apartado de refracciones, el agua es un medio a través del cual la luz se mueve a menor velocidad que por el vacío y, por lo tanto, su índice de refracción n es mayor que uno, en concreto 1.31. Siguiendo las indicaciones marcadas en ese apartado, sería lógico realizar refracciones utilizando un *Light Probe*, sin embargo, no es una opción viable. La utilización de *Light Probes* sobre planos con el fin de explorar el entorno de una escena no es una opción recomendable, ya que el punto de vista desde el cual el *Light Probe* extrae las texturas para el *cubemap* hace que tanto las reflexiones como las refracciones no sean fieles a la realidad cuando el punto de vista cambia. En la figura 106 se muestra este problema aplicado al agua.

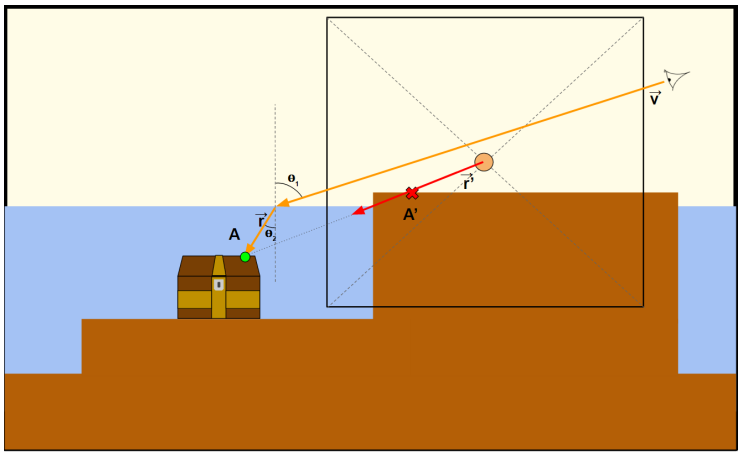


Figura 106: Error en el cálculo de el vector $\vec{r}$.

Para realizar las refracciones en el agua, se va a utilizar una textura calculada desde la cámara que representa un fotograma o *frame*, es decir, aquello que el jugador está viendo. Lo que se persigue con ésto, es conseguir que el plano pueda mostrar los objetos que tiene detrás, siguiendo la dirección en la que el observador mira el plano, como si éste fuese transparente. Una vez el plano muestra lo que tiene detrás, simplemente tiene que distorsionar la textura del plano que contiene los colores utilizando una textura de *bump mapping* que represente las olas del mar, de manera que en el resultado final no se detecte el hecho de no estar practicando refracciones.

Debido a que la textura que procede de la cámara muestra la imagen de la pantalla, el muestreo en el plano sobre ésta textura debe realizarse utilizando las coordenadas de pantalla de cada *fragment* del plano mediante *projective sampling*.

En las sección de refracciones también se ha explicado que los medios refractivos también practican reflexiones. A la hora de realizar estas reflexiones, se ha decidido utilizar el método de reflexión con *Light Probe* explicado en la sección 5.1. El hecho de utilizar ésta técnica a pesar de no haberse considerado óptima para las refracciones tiene que ver con la situación concreta de la escena. En la escena se observa un pequeño poblado completamente rodeado de mar, de manera que

siempre que el jugador mira el mar, normalmente lo único que ve es el reflejo del cielo. En algunos casos, como por ejemplo el de la figura 107, el jugador puede observar cómo no existe ningún reflejo de la otra parte del río, pero considerando que estos casos son pocos y que el agua que se busca no exige demasiado realismo, ésta solución es aceptable.

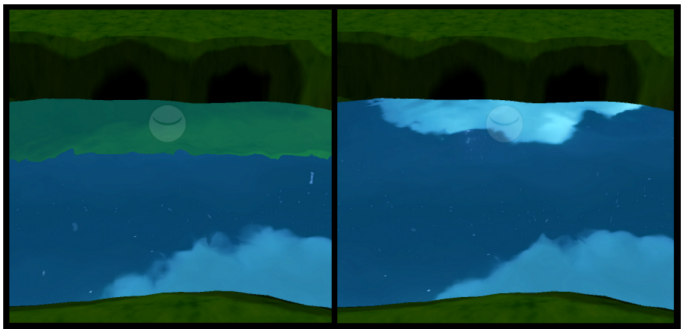


Figura 107: Izquierda: Reflexión del otro lado del río utilizando un *Light Probe*. Derecha: No hay reflexión del otro lado del río.

Las reflexiones son una parte muy importante en el resultado final del agua, puesto que cuando el jugador observa el horizonte o en general las partes más alejadas del agua, debido a que el ángulo de incidencia es mayor, prácticamente lo único que ve son estas reflexiones. Por lo tanto, para conseguir un estilo menos realista en el agua, utilizar métodos para quitar realismo a las reflexiones puede ser un buen punto de partida. Una posibilidad a la hora de quitar realismo a una imagen que no implique procesar imágenes en efectos de postprocesado es discretizar sus colores, como propone [18]. Para conseguir esto se puede utilizar una función escalonada similar a la mostrada en la figura 108 para cada uno de los canales del color.

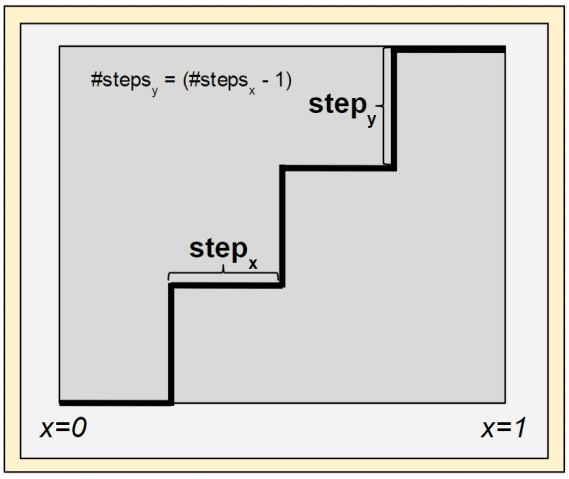


Figura 108: Función escalonada discretizadora.

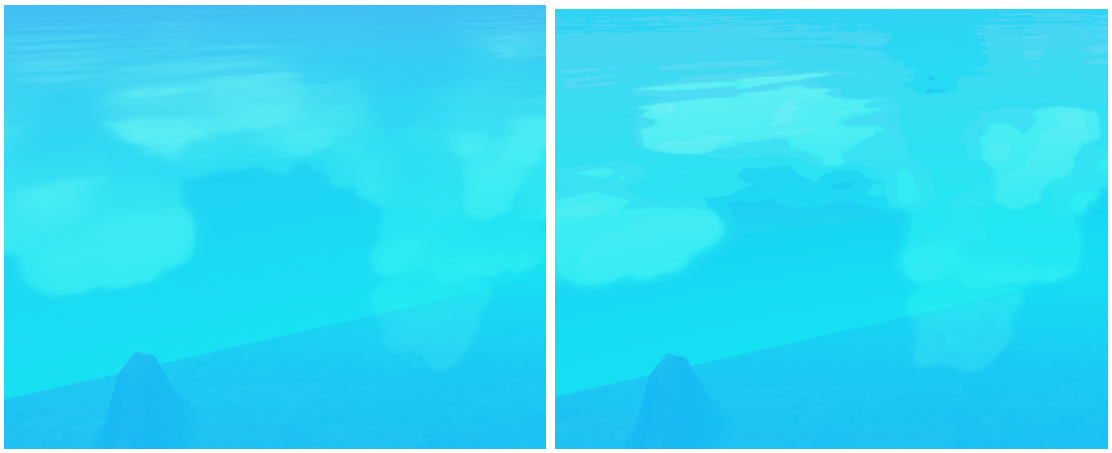
Dados tanto el valor x como la medida de $step_x$, la función se construye de la siguiente forma:

$$step_y = \frac{step_x}{1.0 - step_x} \tag{7.18}$$

$$x = x - x \ mod \ step_x \tag{7.19}$$

$$y = \frac{x \cdot step_y}{step_x} \tag{7.20}$$

En este caso concreto el valor x es uno de los canales del color reflejado, el valor $step_x$ es un número entre el 0 y el 1 que representa la longitud de los escalones en el eje de abscisas y el resultado es el valor x discretizado. Los resultados de modificar los colores usando éste método son los que se pueden observar en la figura 109b.



(a) Agua sin discretización.

(b) Agua con discretización utilizando $step_x = 0.15$.

Figura 109: Comparación entre agua sin discretización del reflejo (izquierda) y con discretización (derecha).

Una vez se tienen tanto las refracciones como las reflexiones se procede a realizar el cálculo de la reflectancia utilizando la aproximación de *Schlick*. Las formas de las funciones de reflectancia real y aproximada se encuentran en el gráfico de la figura 67. Teniendo en cuenta que el agua siempre va a tener el mismo valor de refracción, considerando el aire como el medio original y asumiendo que el ángulo entre el vector $\hat{v}$ y el vector $\hat{n}$ va a estar siempre entre 0° y 90°, se puede definir una función $F_{agua}(u)$ que, a partir de la fórmula aproximada 5.29 y teniendo en cuenta los índices de refracción del aire y del agua, se define como:

$$u = cos(\theta_1) \tag{7.21}$$

$$F_{agua}(u) = \left(\frac{1 - 1.31}{1 + 1.31}\right)^2 + \left(1 - \left(\frac{1 - 1.31}{1 + 1.31}\right)^2\right) \cdot (1 - u)^5 = 0.018 + 0.982 \cdot (1 - u)^5 \tag{7.22}$$

El cálculo de la aproximación de *Schlick*, aunque muy optimizado respecto al de la función real, contiene el cálculo de una potencia a la 5. Para tratar de optimizar

ésto se genera una textura a partir de esta función. Las texturas utilizadas con el fin de almacenar datos o representar una función reciben el nombre de *ramp textures*. Sustiyendo en un programa de diseño como *Gimp* los valores de la función por colores, se puede obtener una textura como la de la figura 110.

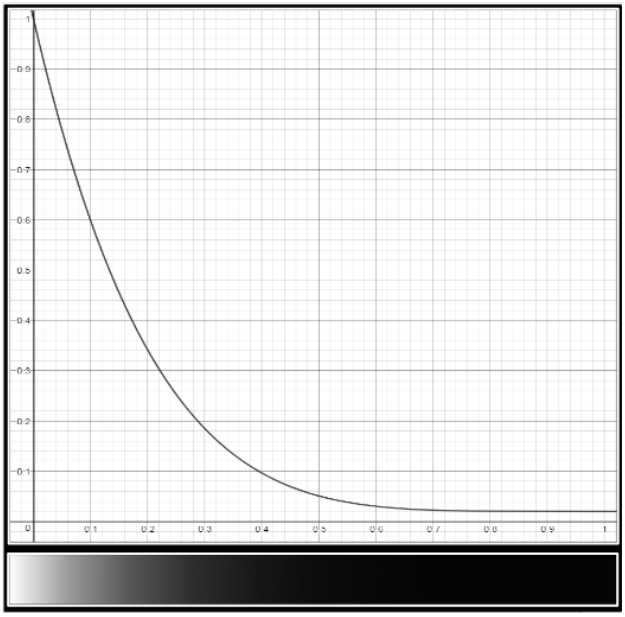


Figura 110: *Ramp texture* que representa la reflectancia.

Otra de las características del agua de un entorno natural es que no acostumbra a ser cristalina, sino que contiene partículas que hacen que al mirar a través no se pueda llegar a ver el fondo. Para conseguir que el agua tenga este efecto se ha utilizado una *depth texture* desde el punto de vista de la cámara, es decir una textura de toda la pantalla, igual que la utilizada para la parte de la refracción, pero con las distancias entre los objetos y la cámara en lugar de los colores. La figura 111 muestra el área de visión o *frustum* de una cámara perspectiva. Como se puede apreciar, esta área tiene la forma de una pirámide seccionada por dos planos, el plano *near*, que corta la punta, y el plano *far* que secciona la base de la pirámide.

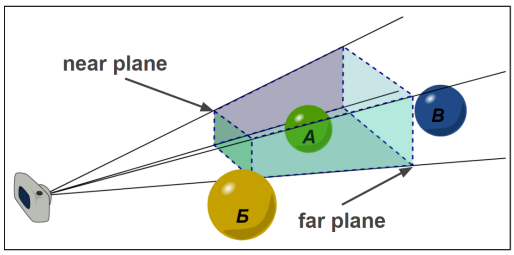
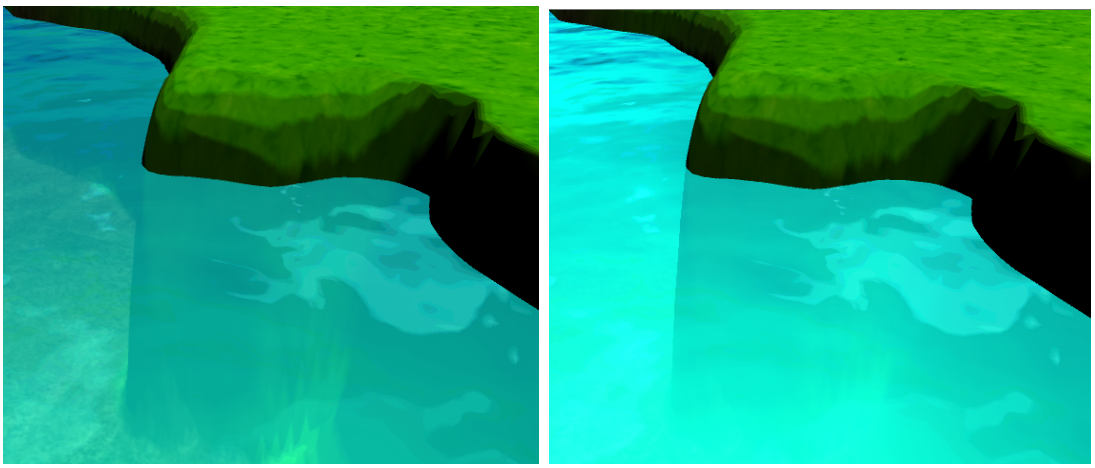


Figura 111: *Frustum* de una cámara.

Realizando una muestra mediante *projective sampling* se obtiene el valor de la profundidad. En la *depth texture*, si un objeto se encuentra a una distancia *near* del observador, su profundidad será de cero, mientras que si por el contrario el objeto se encuentra a una distancia *far* del observador, su profundidad será de uno.

Asumiendo que el agua tiene un color definido, para conseguir el efecto de densidad de las partículas del agua, o niebla de profundidad, se puede realizar una interpolación lineal entre el color refractado y el color del agua, utilizando como peso la profundidad muestreada. En la figura 112 se puede distinguir claramente la diferencia entre utilizar el método y no utilizarlo. El agua de la figura 112a, que no implementa este método, permite que se observe el fondo del agua, mientras que en la figura 112b, donde sí se utiliza, se puede ver cómo a medida que un objeto o el terreno en este caso, está más alejado del observador, es más difícil reconocer su color o su silueta.



(a) Agua sin niebla de profundidad.

(b) Agua con un valor de niebla de profundidad igual a 0.5.

Figura 112: Resultado de aplicar una niebla que oculte la parte más profunda del agua.

7.2.1 Implementación del agua en Unity

Para implementar agua en Unity, en primer lugar se ha creado un *shader* con el nombre *"Custom/Nature/Water"* dentro del sistema de *shaders* de Unity. El fichero se encuentra dentro de la carpeta *"Assets/Shaders/Refraction/Water"*. Una vez creado, se ha escrito un único paso sólo para ser utilizado por luces direccionales. Esto se hace con el fin de ahorrar cálculos que tampoco van a influir demasiado en el resultado final. Dentro del paso, como ya se ha explicado, es necesario conseguir las texturas en espacio de pantalla con la información de los colores y otra con las profundidades. Ambas texturas pueden ser obtenidas utilizando recursos de Unity.

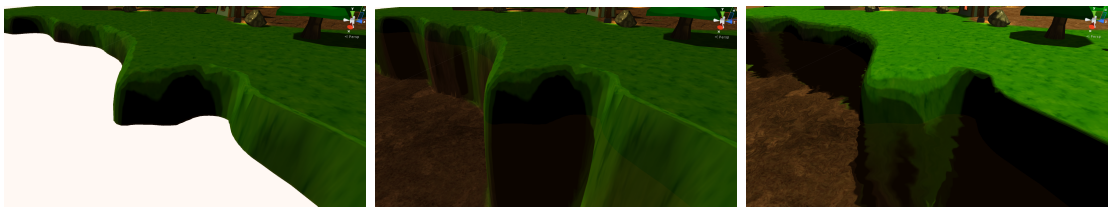
La primera de ellas, que contiene los colores de todo el *frame*, se consigue utilizando un paso especial extra llamado *GrabTexture*. Este paso especial le permite a Unity saber que el objeto que va a ser pintado necesita una textura con los objetos de la escena que estén ya pintados justo antes de que pase ese objeto por el renderizador. Por lo tanto, un material que espere mostrar todo lo que haya detrás de él, como es el caso del agua, tendrá que utilizar este paso después de que todos los objetos o el máximo número posible estén pintados en la escena. Por consiguiente,

como se explica en la sección 4.1, es necesario asociar al material un valor elevado de *Render Queue*.

Para poder utilizar la textura generada mediante el paso *GrabTexture* se necesita, además de haber explicitado el paso, la variable *_GrabTexture* de tipo *sampler2D*, que es la variable que contiene la textura. Dentro del *Vertex Shader* se calculan las coordenadas de pantalla del vértice que se está procesando para, a través de un campo en la estructura de entrada al *Fragment Shader*, donde se dividirán entre la distancia tras haber sido interpoladas.

Con estas coordenadas ya es posible realizar la muestra sobre la textura *_GrabTexture* para mostrar la escena escondida detrás del objeto, el plano de agua en este caso. Para conseguir las perturbaciones sólo es necesario añadir un pequeño desplazamiento utilizando la variable *_Time* que el propio sistema de *shaders* de Unity proporciona y una textura con la que se realiza *bump mapping* sobre las normales, además de, como se verá en adelante, un valor asociado a la intensidad de las olas.

Para mejorar el movimiento de las olas se pueden utilizar dos texturas de normal en vez de sólo una. El problema cuando las olas se generan a partir de una sola textura es que cuando se añade movimiento al agua, las olas se mueven en una única dirección. Las olas del mar se generan cuando dos masas de agua con diferentes direcciones chocan entre sí. De esta manera, al utilizar dos texturas con diferentes parámetros de movimiento, da una impresión más realista. En vez de dos texturas también es posible utilizar solamente una, pero realizando dos muestras con coordenadas diferentes. El efecto conseguido por las olas se muestra en la figura 113c.



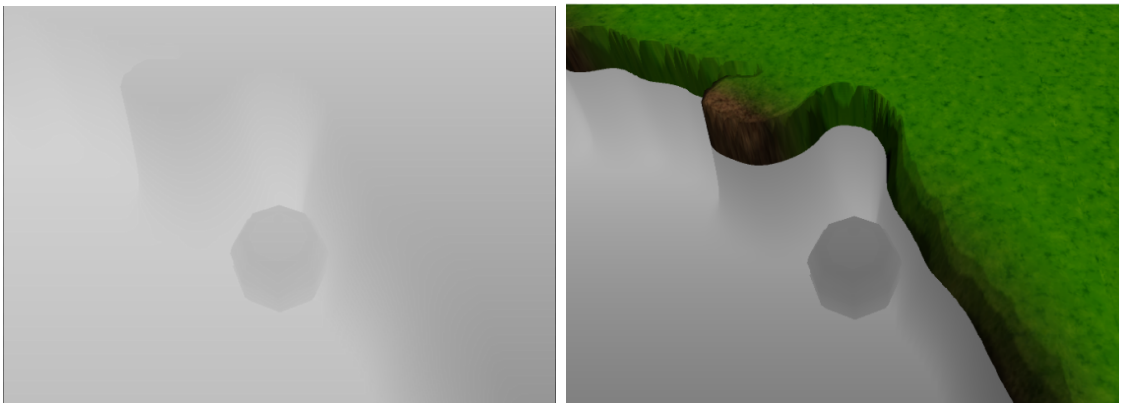
(a) Plano sin transparencia. (b) Plano con transparencia sin refracciones. (c) Plano con distorsiones.

Figura 113: Parte refractiva del agua.

La segunda textura que se necesita para implementar éste tipo de agua es la textura en espacio de pantalla con las profundidades. Ésta es necesaria para la simulación de la niebla de profundidad. De la misma forma que en el caso de la textura con los colores de la escena, la textura con las profundidades también puede ser obtenida del sistema de *shaders* utilizando la variable *_CameraDepthTexture* de tipo *sampler2D*. Con el fin de conseguir las coordenadas de pantalla se debería haber podido utilizar las mismas coordenadas ya calculadas para la textura anterior, sin embargo, Unity presenta algunos problemas cuando se trata de utilizar algunas texturas extraídas de una cámara, conocidas como *Render Textures*, y pueden estar giradas verticalmente. Para solventar este problema, es necesario girar las coordenadas en caso de ser necesario, mediante la componente x de la variable

_ProjectionParams. Debido a que este giro puede llegar a afectar al muestreo de la otra textura, es recomendable utilizar dos coordenadas diferentes para cada una.

Con la muestra obtenida de esta textura, se realiza una interpolación lineal entre el color del agua y el color de la refracción utilizando como peso el valor de la profundidad. El resultado de aplicar la textura de la figura 114a sobre el plano del agua se muestra en la figura 114b.

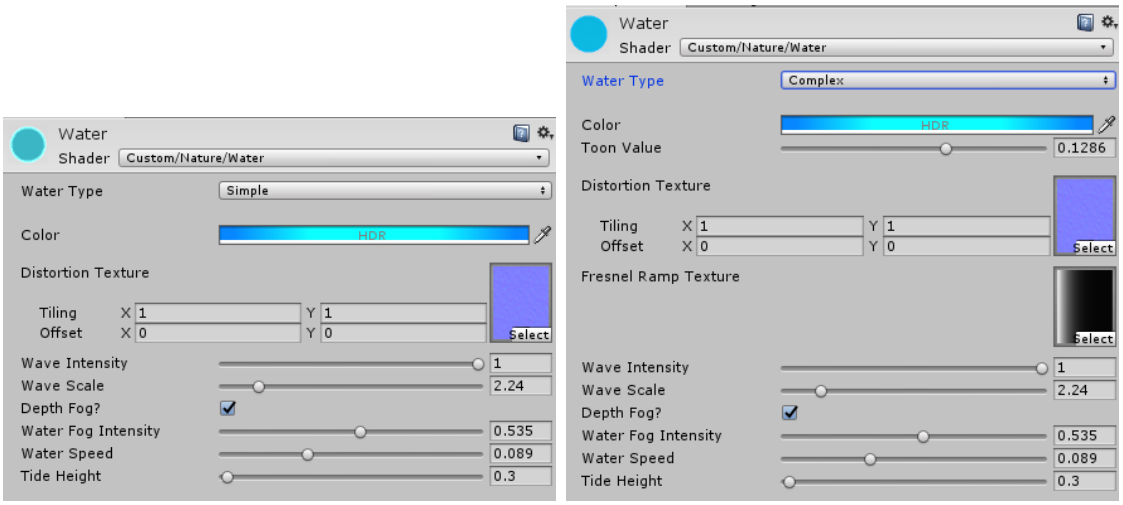


(a) Textura con las profundidades extraída de la cámara que representa al observador.

(b) Muestreo utilizando *projective sampling* sobre la textura de profundidades de la figura 114a.

Figura 114

Dos tipos diferentes de agua se han implementado en este proyecto: una simple llamada *Simple* y otra compleja de nombre *Complex*.



(a) Inspector para el agua simple.

(b) Inspector para el agua compleja.

Figura 115

El agua simple sólo realiza refracciones y utiliza como base un color plano teniendo también en cuenta el color de los objetos por debajo del agua. Utilizando las normales modificadas, realiza $\vec{v} \cdot \hat{n}$ para más adelante discretizarlo en 3 fran-

jas. El inspector de este tipo de agua se muestra en la figura 115a. Éstas son las propiedades de las que dispone esta variación de *shader*:

- La propiedad *Water Type* permite seleccionar el tipo de agua que se desea utilizar. Tiene como opciones *Simple* y *Complex*.
- La propiedad *Color* permite cambiar el color base del agua.
- La textura *Distorsion Texture* hace posible añadir una textura de normal con la que generar las olas.
- El valor *Wave Intensity* aumenta o disminuye la intensidad de las olas modificando el valor extraído de la textura de normal.
- La propiedad *Wave Scale* sirve para aumentar o disminuir el tamaño de las olas multiplicando este valor por las coordenadas de la textura de normal. Resulta útil para calibrar el tamaño de las olas cuando se modifica la escala del plano.
- Los campos *Depth Fog* y *Water Fog Intensity* permiten, activar o desactivar la niebla de profundidad mediante el primer campo, y con el segundo modificar la intensidad de ésta.
- El valor *Water Speed* modifica la velocidad de las olas sumando este valor a las coordenadas de la textura de normal.
- Finalmente, la propiedad *Tide Height* sube y baja el plano en la dirección de su normal simulando la marea del mar.

Utilizando en la escena el agua simple se obtienen los resultados mostrados en la figura 116.

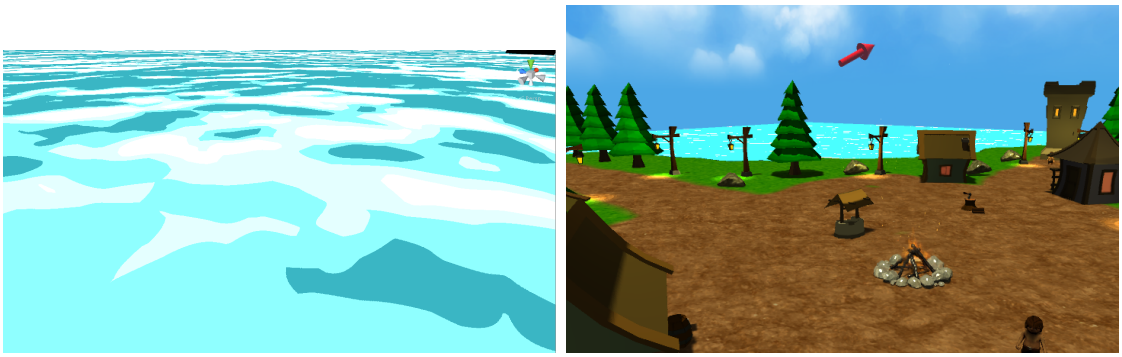


Figura 116: Escena con agua simple.

El agua compleja utiliza tanto reflexiones como refracciones. Para las refracciones tiene en cuenta tanto el color del agua como el color de la refracción en sí, pero, a diferencia del agua simple, el agua compleja no realiza ninguna discretización es ellas. En cuanto a las reflexiones, para intentar conseguir un aspecto menos

realista se utiliza el método de discretización de los colores de la reflexión explicado previamente.

El inspector de este tipo de agua se muestra en la figura 115b. Introduce dos propiedades nuevas respecto el inspector del agua simple. La primera de ellas es el valor *Toon Value*, que sirve para aumentar o disminuir la discretización de la imagen reflejada en el agua. Cuanto más grande este valor, mayor discretización se aplica sobre el reflejo. El segundo campo introducido es *Fresnel Ramp Texture*, que permite introducir una ramp texture como la de la figura 110 para controlar la reflectancia del agua.

El agua compleja da los resultados mostrados en la figura 117.

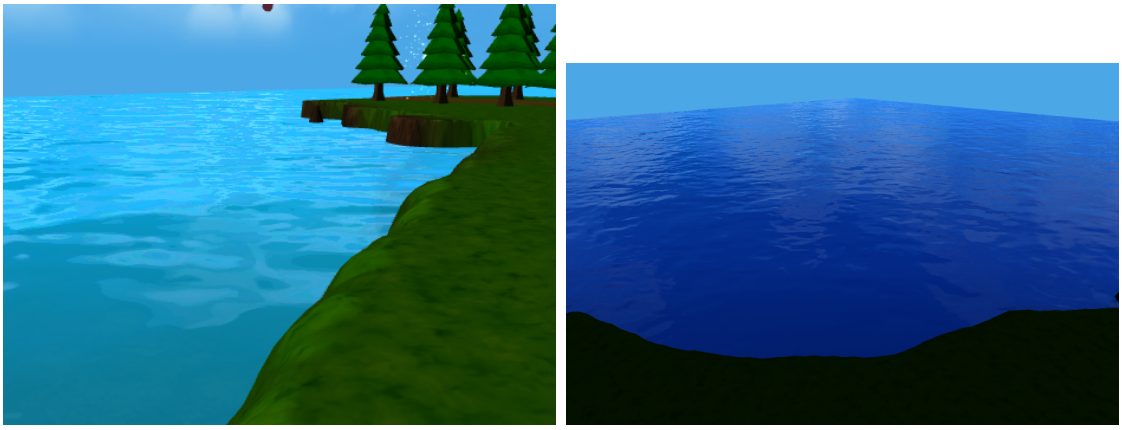


Figura 117: Escena con agua compleja.

7.3 Aplicación de niebla

Para introducir niebla en el proyecto se ha optado por añadir al mismo tiempo dos tipos de niebla, un tipo que tenga movimiento que represente las nubes que suelen aparecer a ras de suelo y otro tipo que represente la niebla a nivel global en toda la escena.

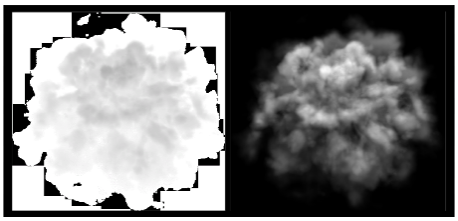


Figura 118: Textura utilizada para simular niebla. A la izquierda los canales RGB. A la derecha el canal *alpha*.

El primer tipo de niebla que se ha introducido está implementada dentro del *shader Custom/Fog*. Se trata de un shader que representa un material semitransparente con la textura que se observa en la figura 118. Este shader utiliza los conceptos que se han explicado en el apartado de transparencias (sección 4.3) y se aplica a un plano situado ligeramente por encima del terreno. El plano se sitúa a ras de suelo para minimizar el efecto de la figura 119 (izquierda), donde se observa claramente que el plano atraviesa la casa de la escena.

La principal diferencia existente entre este *shader* y el *shader* explicado en el apartado 4.3 es que este pertenece a una cola de renderizado más baja que el resto de materiales semitransparentes y por lo tanto pasa antes por el renderizador. Esto se hace debido a que este plano se encuentra prácticamente tocando el terreno, y por lo tanto, hay más probabilidades de que otros objetos semitransparentes se encuentren por encima. Por lo tanto, cualquier otro objeto semitransparente que estuviese situado por encima del plano debería tener la niebla en cuenta y por consiguiente es necesario que pase después por el renderizador.

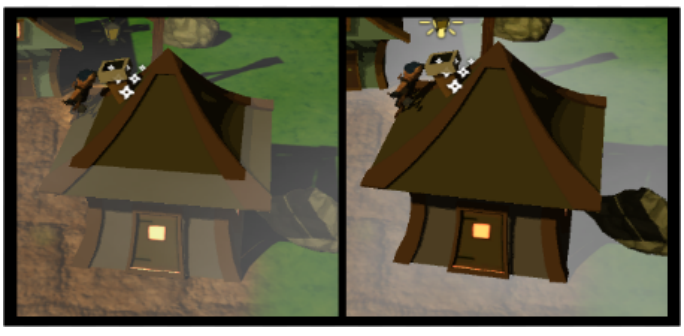
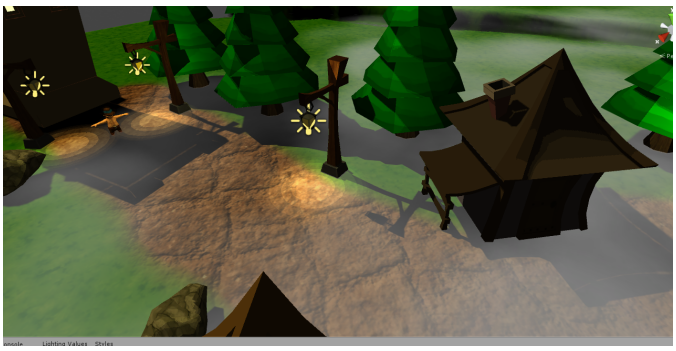


Figura 119: Muestra del cambio de color entre el plano con la niebla y una casa.

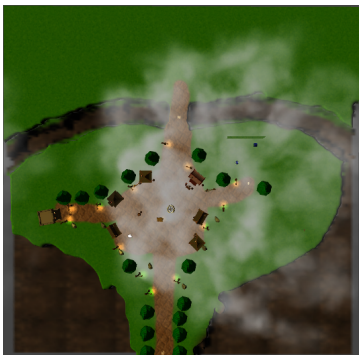
Para que el plano represente más fielmente la niebla, se ha utilizado dentro del *Vertex Shader* la variable interna *_Time* para desplazar las coordenadas de la textura con el paso del tiempo. Esto reproduce el efecto de la niebla moviéndose

a causa del viento. Además, dentro del *Fragment Shader* se ha hecho uso de la variable *_SinTime*, que contiene los valores del tiempo alterados por una función sinusoidal, para modificar el valor *cutoff* del shader de manera que la niebla aparezca y desaparezca con el tiempo.

La figura 120 muestra la escena del proyecto cubierta por el plano desde diferentes perspectivas.



(a) Vista del poblado desde cerca.



(b) Vista del poblado desde arriba.

Figura 120: Muestra de los resultados de aplicar el plano con niebla desde diferentes perspectivas.

Sin embargo, en las imágenes anteriores se puede observar cómo por ejemplo las casas no se ven afectadas. Esto hace que el efecto en sí no parezca realista y por lo tanto hace necesario que se implemente una niebla a nivel global que afecte a todos los objetos de la escena. Una niebla de estas características debe tener en cuenta que a medida que aumenta la distancia entre un observador y un objeto que se quiera mirar a través de una atmósfera llena de niebla también aumenta la cantidad de partículas de niebla entre los dos puntos. Por lo tanto, considerando que la absorción de la luz en todo punto de la niebla es constante, se puede afirmar que cuanto mayor es la distancia, mayor es la dificultad para ver a través de la niebla, o dicho de otra manera, con la distancia se produce una mayor atenuación de la luz. La dificultad para ver un objeto en una atmósfera con niebla depende de dos factores: la distancia y la densidad de la propia niebla. Por consiguiente existe una función f que depende de los valores d y t que son respectivamente distancia y densidad (o *thickness*) y que devuelve la intensidad i de la niebla. Se puede por tanto representar f como:

$$i = f(d, t) \tag{7.23}$$

Para poder utilizar esta fórmula por supuesto hay que calcular previamente la distancia d, que no es otra que la distancia entre la cámara y el objeto que se está observando. Este cálculo de la distancia se puede realizar teniendo en cuenta la componente z de la posición del objeto en coordenadas de *clip* así como la distancia mínima y máxima de los planos de la cámara.

La posición de un objeto representada en coordenadas de *clip* se consigue multiplicando cada uno de los vértices de ese objeto en coordenadas de objeto por la matriz *MVP*. En Unity, la componente z de una posición que pasa por este proceso está contenida entre *near* y 0, donde *near* es la distancia entre el observador el *near plane*. El valor 0 corresponde a la distancia del *far plane* mientras que *near* es el valor de un punto situado a la misma distancia del observador que el *near plane*. Es decir, aumenta a medida que se acerca al observador. El esquema de la figura 121 muestra visto desde arriba los dos planos que forman el área de visión de la cámara.

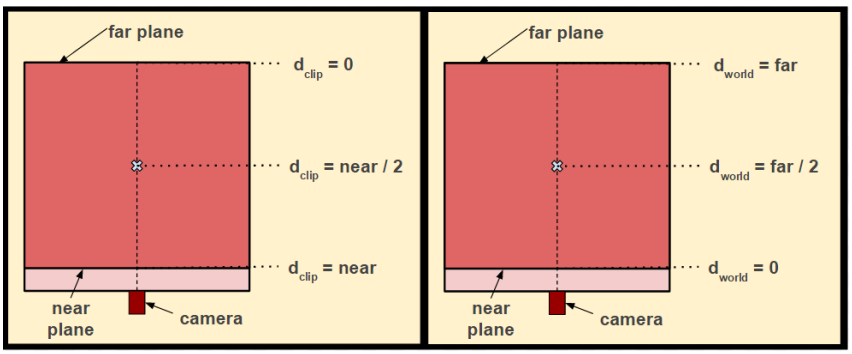


Figura 121: Esquema de las partes de la cámara.

Sin embargo por el momento lo único que se tiene es un valor d_{clip} entre 0 y *near* mientras que el valor que se espera conseguir no es otra cosa que la distancia en coordenadas de mundo o escena, o d_{world}. La fórmula,

$$d_{world} = max(f(1 - \frac{d_{clip}}{n}), 0) \tag{7.24}$$

$$d_{clip} = 0 \implies d_{world} = f \tag{7.25}$$

$$d_{clip} = near \implies d_{world} = 0 \tag{7.26}$$

consigue situar el valor de d_{clip} dentro del conjunto $[0, f]$, sin embargo el valor 0 no representa la posición del observador, sino la distancia al *near plane*, de manera que es recomendable no dar un valor demasiado alto al *near plane*. Calcular la distancia entre la posición de la cámara y un punto de la escena siguiendo este método es más eficiente que con el método habitual ya que se evita de esta forma calcular una raíz cuadrada. Por este motivo se ha utilizado esta forma de calcular d_{world} a la hora de introducir una niebla global en el proyecto.

Una vez ya se ha obtenido la distancia d_{world} y se dispone de un factor que indique el nivel de densidad de la niebla ya se puede utilizar la funcion $f(d, t)$ para así poder calcular la intensidad ésta en cada vértice de cada objeto. Suponiendo que esa intensidad tenga un valor que se encuentre representada por el conjunto $[0, 1]$, puede servir para determinar a través de una interpolación lineal el color final del objeto, siendo en 0 el color del objeto en sí, es decir, como si no existiera la niebla, y 1 cuando sólo se tenga en cuenta el color de la niebla. De esta manera,

suponiendo que la función f intenta simular el comportamiento real de la niebla, $i = 0$ cuando la distancia entre la cámara y el objeto es cercana a 0 e $i = 1$ cuando la distancia es elevada.

Unity dispone de un apartado mostrado en la figura 123 dentro de la ventana *Lighting* de configuración de la luz en el que es posible indicar el tipo de niebla que se quiere aplicar. Este apartado está formado por dos campos principales. El primer campo *Fog Color* corresponde al color de la niebla que se va a utilizar en la interpolación. El segundo campo *Fog Mode* es el que determina la función f que va a ser empleada. En el caso de Unity existen tres modos predefinidos: *Linear*, *Exponential* y *Exponential Squared*. Dentro del sistema de shaders de Unity y en concreto del fichero *UnityCG.cginc* se encuentra la función *UNITY_CALC_FOG_FACTOR_RAW* que calcula la intensidad a partir de los parámetros introducidos en el apartado *Fog* en función del modo que se encuentre seleccionado. Estas tres funciones se definen de la siguiente forma:

$$f_{linear}(d, s, e) = \frac{e - d}{e - s} \tag{7.27}$$

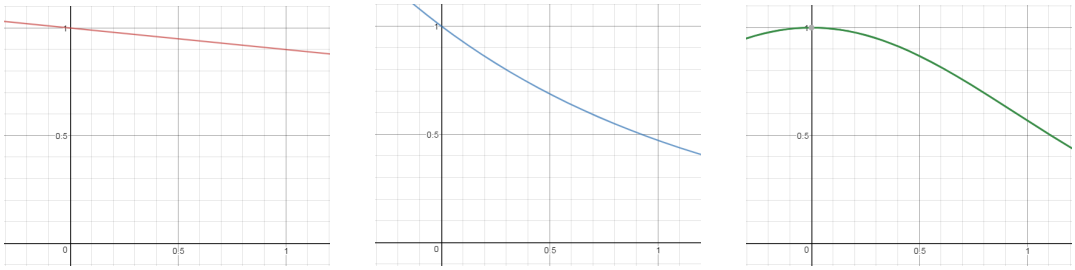
$$f_{exp}(d, t) = e^{-td} \tag{7.28}$$

$$f_{exp2}(d, t) = e^{-(td)^2} \tag{7.29}$$

En el caso de la función f_{linear} los valores de entrada son diferentes. En vez recibir un factor de densidad de la niebla, reciben un valor de inicio (s o *start*) y un valor de fin (e o *end*). Con esto se construye la función que se muestra en 122c. Esta función no utiliza directamente la densidad si no que la construye a partir de los valores s y e. La figura 122 muestra las tres funciones que definen los tres modos de niebla que Unity implementa.

Estas fórmulas, como se observa en la figura 122, disminuyen a medida que aumenta la distancia, cuando debería ser al contrario. Este problema sin embargo tiene fácil solución, ya que simplemente intercambiando los colores a la hora de realizar la interpolación la intensidad calculada ya actúa de forma correcta.

No obstante para poder utilizar la función *UNITY_CALC_FOG_FACTOR_RAW* se ha de poder distinguir desde el *shader* qué modo de niebla está activado en cada momento, además de los valores empleados para calcular la intensidad. Para esto es necesario, al principio del paso *ForwardBase* (únicamente se aplica en este paso), utilizar la instrucción *#pragma multi_compile_fog* que introduce las variables *FOG_LINEAR*, *FOG_EXP* y *FOG_EXP2*, las cuales se corresponden con los tres modos existentes de niebla. Unity se encarga de activarlas y desactivarlas así como de modificar los valores de entrada de la función, como la densidad, almacenados dentro de la variable *unity_FogParams*.



(a) Modo *Linear* (7.27) con $e = 10$ y $s = 0$.

(b) Modo *Exponential* (7.28) con $t = 0.75$.

(c) Modo *Exponential Squared* (7.29) con $t = 0.75$.

Figura 122: Diferentes fórmulas que utiliza Unity para calcular la intensidad de la niebla.

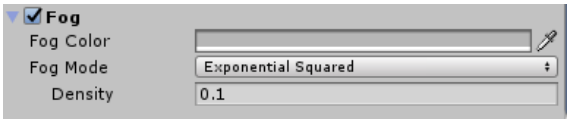


Figura 123: Apartado *Fog* en la ventana *Lighting* de Unity.

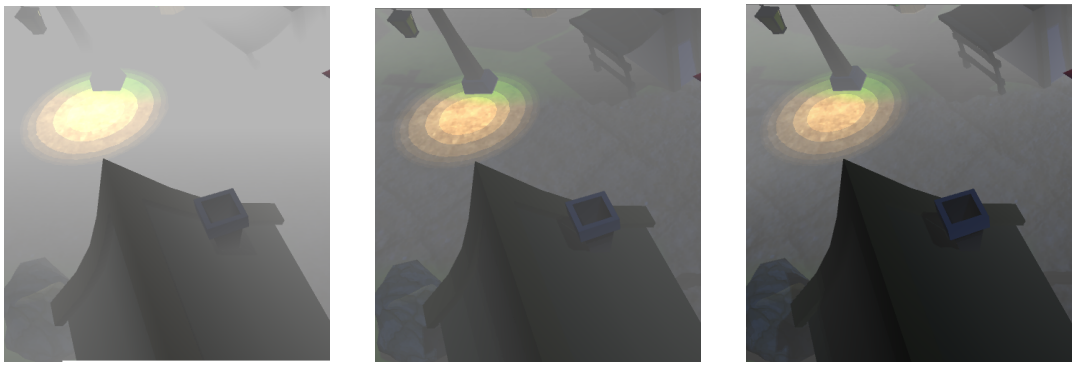
El cálculo de la intensidad tiene lugar dentro del *Vertex Shader*, para evitar calcular las exponenciales prseentes en las fórmulas 7.28 y 7.29. El resultado, al tratarse de un sólo valor de tipo flotante, se ha pasado por el interpolador dentro de la componente w de la variable que representa la posición del vértice en coordenadas de mundo. De esta manera se evita crear un interpolador más.

Dentro del *Fragment Shader* lo único que se tiene que realizar es una interpolación lineal entre el color de la niebla y el color del objeto una vez ya se calculado toda la interacción de la luz con el objeto en cuestión. Como ya se ha explicado antes el peso en la interpolación lineal es la intensidad calculada en el *Vertex Shader* y ya interpolada en el *Fragment Shader*.

A la hora de integrar la niebla se ha creado un *script* en lenguaje *c#* con tres variables, que son las que se muestran en la figura 124. La primera de ellas, la propiedad *Intensity* modifica la intensidad de la niebla, tanto la niebla implementada dentro del plano como la niebla global. La propiedad *Speed* afecta solamente a la niebla del plano y modifica la velocidad a la que se mueve esta. Finalmente, la propiedad *Fog Mode* permite cambiar entre los tres tipos de modos de niebla que se han explicado anteriormente.

Figura 124: Interfície del generador de niebla.

En la figura 125 se muestran los resultados de aplicar este tipo de niebla a toda la escena con los tres modos.



(a) Modo *Linear* activado. (b) Modo *Exponential* activado. (c) Modo *Exponential Squared* activado.

Figura 125: Una parte del poblado con los tres tipos de niebla a nivel global aplicados.

8 Visualización no realista

Este proyecto ha girado entorno a conseguir a través de la creación de *shaders* una visualización del juego no realista que lo haga más amigable para los niños. A continuación se explican los cambios que se han intentado introducir para conseguir llegar a esta meta.

8.1 Toon shading

El *Toon shading* o *Cel shading* es un tipo de renderizado que busca reproducir un material de forma poco realista, como por ejemplo en un cómic o en dibujos animados. Para lograr este efecto se pueden utilizar tres métodos distintos.

8.1.1 *Steep Toon Shading*

El *Steep Toon Shading* es un tipo de *Toon Shading* que hace que los objetos de una escena estén pintados por franjas en función de la dirección de la luz, tal y como se observa en la figura 126.

Estas franjas que se observan son resultado de una discretización de la interacción difusa de la luz con la superfície del objeto. El modo de realizar esto es calculando el producto escalar entre el vector normal a la superfície y el vector de la luz, tal y como se enseña en la sección 2.2 referente a la reflexión difusa, para más tarde discretizar ese resultado utilizando condiciones.

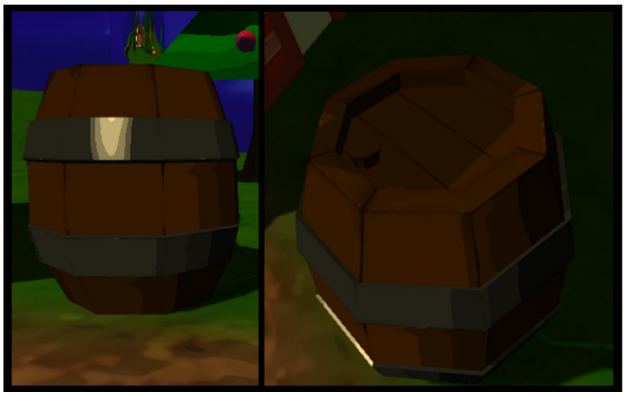
Limitando el resultado del producto escalar $\vec{n} \cdot \vec{l}$ entre 0 y 1, se han escogido las siguientes franjas:

$$\begin{aligned} [0.00, 0.45] &\longrightarrow 0.2 \\ (0.45, 0.55] &\longrightarrow 0.5 \\ (0.55, 0.65] &\longrightarrow 0.7 \\ (0.65, 1.00] &\longrightarrow 1.0 \end{aligned}$$

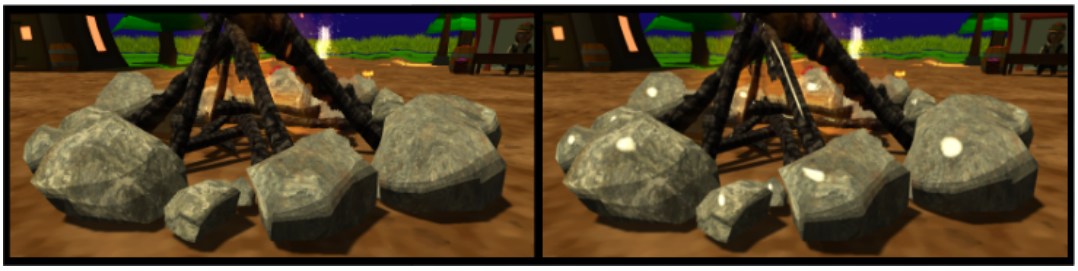
siendo, por lo tanto, el valor 0 el correspondiente a $\vec{n} \cdot \vec{l} = -1$. También se observa cómo la parte que habitualmente está más oscura tiene en realidad un valor mayor que 0, con lo que siempre se verá la textura del objeto en esa zona. Se ha implementado de esta forma para hacer que, por mucho que el valor resultante de $\vec{n} \cdot \vec{l}$ sea negativo, siempre exista una luz ambiental.

Los resultados de aplicar estos rangos sobre un barril y una hoguera en la escena se observan en la escena 126.

Este método genera unos resultados aceptables, sin embargo, existen otros tipos de *Toon Shading* que ayudan a obtener visualizaciones más artísticas.



(a) Muestra de un barril con *Steep Toon shading*.



(b) Muestra de una hoguera con *Steep Toon shading*

Figura 126

8.1.2 *Flat Toon Shading*

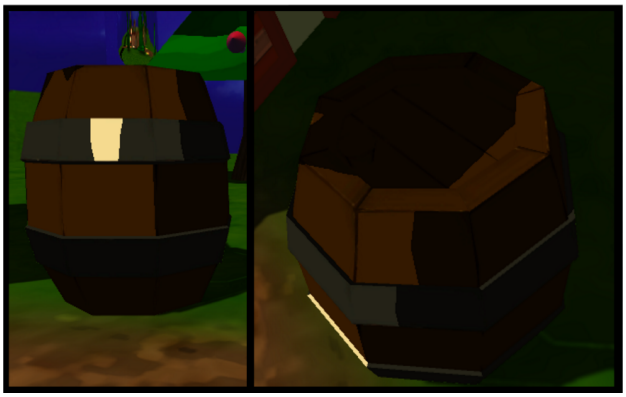
El *Flat Toon shading* es otro tipo de *Toon Shading* que consiste en crear una única franja de separación alrededor del *terminator*, que es el nombre por el que se conoce a la parte de un objeto en que el producto escalar entre la normal y la dirección de la luz es igual a cero. El proceso en este caso es prácticamente el mismo que la discretización que se lleva a cabo en el *Steep Toon Shading*, sin embargo, en este caso sólo existe un único valor de corte. Este método suele estar acompañado por una luz especular con unos bordes muy marcados generados a partir de otro proceso de discretización.

En el caso del *Flat Toon Shading* es común tener una variable que permita suavizar el contorno tanto de la componente difusa como el de la componente especular, produciendo los resultados observables en la figura 127b.

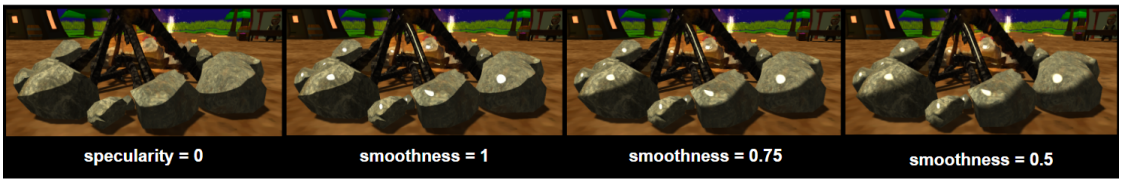
8.1.3 *Toon Shading* con texturas

El último tipo de *Toon shading* se realiza en base al mismo principio de discretización, pero se diferencia en que se utiliza una *ramp texture* para conseguir esa discretización. Un ejemplo de la *ramp texture* utilizada se muestra en la figura 128.

Teniendo en cuenta que $\vec{n} \cdot \vec{l}$ da como resultado un número contenido en el conjunto $[-1, 1]$, es necesario convertirlo a un rango $[0, 1]$ mediante la fórmula,



(a) Muestra de un barril con *Flat Toon shading*.



(b) Hoguera iluminada mediante *Flat Toon Shading* con diferentes valores de suavizado.

Figura 127

Figura 128: *Ramp texture*.

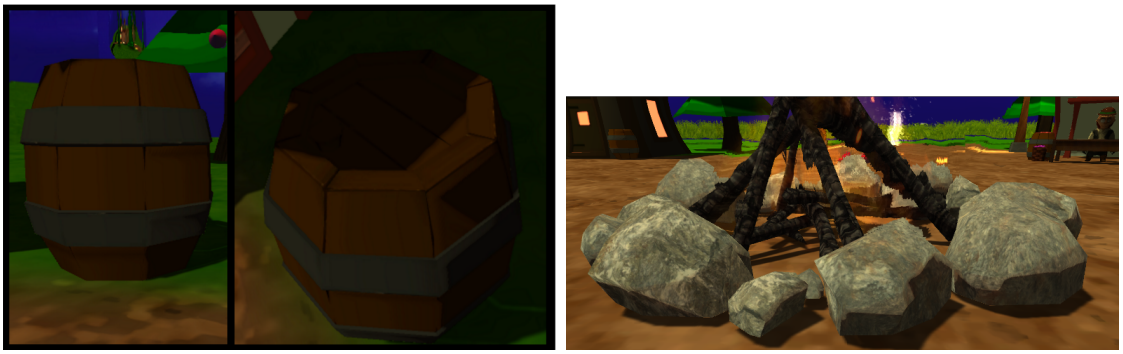
$$diffuse = \frac{\vec{n} \cdot \vec{l}}{2} + \frac{1}{2} \tag{8.1}$$

de esta forma, el valor de $diffuse$ puede utilizarse para muestrear la *ramp texture*. Esta técnica normalmente requiere de un equipo de diseño que genere las texturas, con lo cual, aunque no es la más indicada para este proyecto, es posible conseguir muy buenos resultados gracias a esta técnica. En [19], referente al juego *Team Fortress 2*, se explica cómo es en buena parte gracias a ésta técnica que se ha conseguido que éste juego tenga una estética tan reconocible. En la figura 129 se muestran los resultados obtenidos en el proyecto utilizando como *ramp texture* la textura de la figura 128.

Las tres opciones son válidas a la hora de representar un tipo de visualización no-realista que se asemeje a el tipo de dibujo presente en cómics o dibujos animados.

8.1.4 Implementación de *Toon Shading* en Unity

Con el fin de aplicar este tipo de visualización en todos los objetos de la escena se han generado dos *shaders* que cualquier objeto pueda utilizar para su material. Estos dos *shaders* tienen por nombre *CelShader* y *ComplexCelShader*. Como los nombres indican, el primero de ellos realiza una serie de funciones básicas pensadas para los objetos pequeños que en general pasan desapercibidos en la escena, como

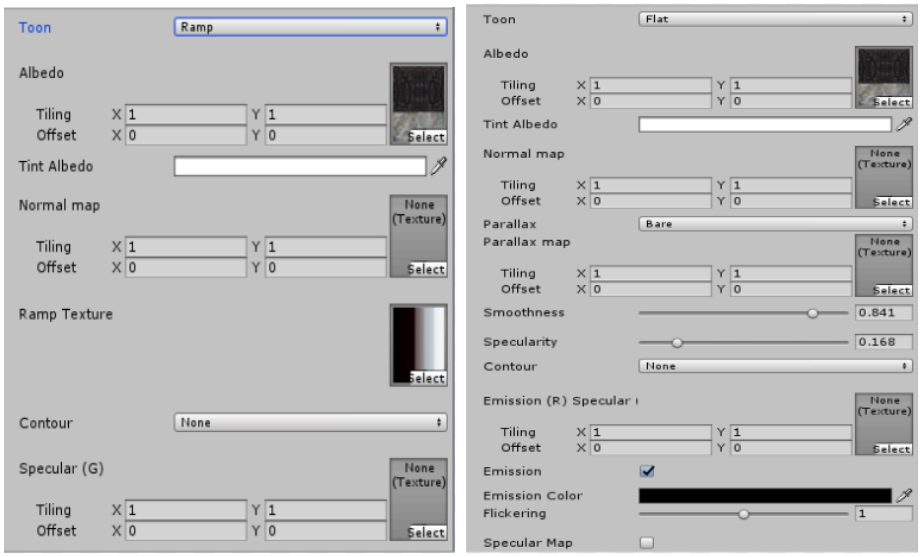


(a) Muestra de un barril utilizando la *ramp texture* de la figura 128.

(b) Muestra de una hoguera utilizando la *ramp texture* de la figura 128.

Figura 129

un barril o un hacha, mientras que el segundo se utiliza en objetos más grandes que pueden necesitar de más complejidad. Mediante *scripts* en lenguaje *c#* de Unity se han generado dos inspectores independientes para cada *shader* con la finalidad de adaptarlos a las necesidades de cada uno. La figura 130 muestra estos dos inspectores.



(a) Inspector de *CelShader*.

(b) Inspector de *ComplexCelShader*.

Figura 130

En el inspector del *shader* *CelShader* se puede ver lo siguiente:

- En primer lugar hay un campo con el nombre *Toon* que permite escoger el tipo de *Toon Shading* que se desea utilizar. Las opciones son *None*, *Flat*, *Stepped* y *Ramp*. En la figura fig:toon-inspectorcelshader aparece activado el modo *Ramp*.
- A continuación hay dos apartados (*Albedo* y *Tint Albedo*) dedicados al albedo del objeto, uno para la textura y otro para el color del que se quiera teñir.

- La propiedad *Normal Map* permite introducir una textura para realizar la técnica *bump mapping* explicada en la sección 3.1.
- Debido a que el modo de *Toon Shading* activado en la figura fig:toon-inspectorcelshader es el modo *Ramp* existe una sección (*Ramp Texture*) en la que indicar la *ramp texture* que se desea utilizar.
- La propiedad *Contour*, que se explicará en el apartado 8.2. Aplica contornos.
- Finalmente existe la sección *Specular (G)*, donde la letra *G* indica el canal de la textura que se va a utilizar, sirve para introducir una textura que actúa como mapa para restringir las partes del objeto especular que pueden mostrar luces especulares. Un ejemplo de ello se muestra en el barril de la figura 127a, donde sólo la parte metálica del barril tiene reflexiones especulares.

En el inspector del *shader ComplexCelShader* se permiten realizar las mismas acciones además de algunas más. En este caso el modo de *Toon Shading* seleccionado es *Flat Toon Shading*, con lo que el apartado para introducir la *ramp texture* ya no es necesario. Una de las propiedades que se introduce en este *shader* es la técnica *parallax mapping*, explicado en la sección 3.2. El apartado *Parallax* permite escoger entre los tres métodos explicados, y el apartado *Parallax Map* permite introducir un *height map* que contenga las alturas con las que realizar la técnica. Además de *parallax mapping* se puede representar mediante el canal *R* de la textura del apartado *Emission(R)* las partes de la textura que pueden emitir luz, como la casa de la figura 131. También permite por otra parte la opción de cambiar el color de la luz, así como hacer que su intensidad varíe con la propiedad *Flickering*.

Figura 131: Casa con el *shader ComplexCelShader* emitiendo luz.

Existen además dos propiedades, *Specularity* y *Smoothness*, inherentes al modo *Flat* que modifican respectivamente la intensidad de la reflexión especular y la suavidad del contorno en las transiciones mostradas en 127b.

8.2 Contornos

Es común ver en juegos o incluso en películas de dibujos animados una estética similar al *toon shading* junto con un contorno de color negro que marca la silueta exterior de los personajes y los edificios. Esto se ha intentado aplicar utilizando dos métodos diferentes.

8.2.0.1 Contornos con *Geometry Shader*

El primer método consiste en crear una nueva superfície del objeto en cuestión utilizando un *Geometry Shader*, desplazando ésta superfície en la dirección del vector normal de cada vértice de sus triángulos como muestra la figura 132. Si sólo se realiza esto la nueva superfície oculta a la antigua, sin embargo, existe un proceso que permite descartar triángulos de una superfície en función de si están orientados hacia el observador o al contrario. Este proceso se llama *culling* y tiene lugar después de que se generen los triángulos en el *Vertex Shader* o, en este caso en el *Geometry Shader*. Si mediante este proceso se descartan aquellos triángulos que miran al frente, los que están orientados en sentido contrario dan la impresión de formar un contorno alrededor del modelo 3D.

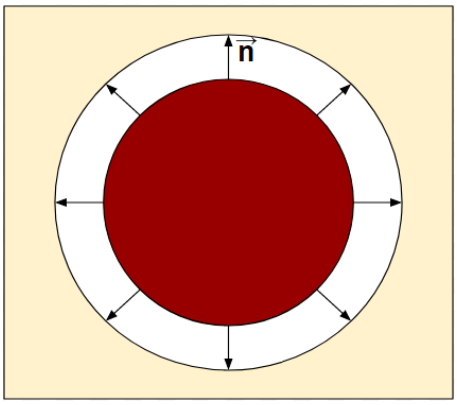
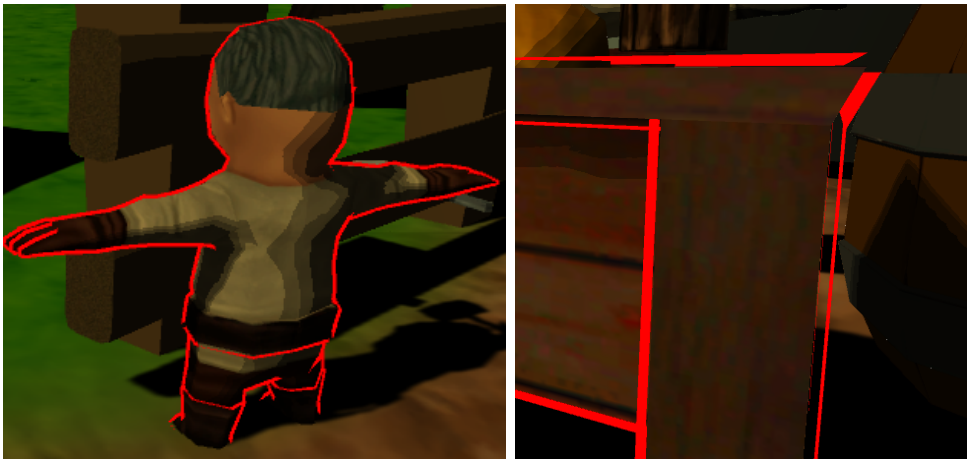


Figura 132: Formación de un contorno mediante una nueva superfície.

En Unity este proceso se ha construido de la siguiente manera. En primer lugar se ha creado un nuevo paso sin ninguna etiqueta y con la instrucción *Cull Front* antes de comenzar el programa, que realiza el proceso de *culling* sobre los triángulos orientados hacia la cámara. En este paso se utiliza un *Geometry Shader* donde se genera la nueva superficie creando nuevos triángulos desplazados según la normal de cada vértice. Para ello sólo es necesario sumar el vector normal, modificando su magnitud con una variable a la que se ha llamado *_OutlineWidth*, a la nueva posición de cada vértice del nuevo triángulo. El cálculo de la posición de la nueva superfície se realiza en coordenadas de mundo, sin embargo la posición final debe estar en coordenadas de *clip* por lo que es necesario realizar el cambio de coordenadas después de generar la nueva superfície. Una vez creada, lo único que resta por hacer es devolver el color del contorno dentro del *Fragment Shader*.

Este método, sin embargo, tiene fallos, como se muestra en la figura 133a. Si el

modelo contiene una cantidad elevada de vértices, no se suelen advertir problemas graves, pero, si por el contrario el objeto está formado por pocos vértices, y además contiene triángulos o cuadrados alargados como en la figura 133b, en una vista lateral es obvio el proceso que se ha seguido, y por lo tanto no se ha considerado que sea un método aceptable para este proyecto.



(a) Contorno de color rojo sobre un objeto con suficientes vértices.

(b) Contorno de color rojo sobre un objeto con pocos vértices.

Figura 133

8.2.0.2 Siluetas con *rim lighting*

Viendo que el método para introducir contornos mediante un *Geometry Shader* da resultados pobres se ha decidido intentar realizar una silueta utilizando los principios del *rim lighting* explicados en la sección 2.5.

Siendo el *rim lighting* un proceso por el que se suma un color a la parte exterior de un objeto y queriendo que esa silueta sea de color negro, es decir, un valor igual a 0, es necesario modificar este proceso para que el valor obtenido del *rim lighting* sea multiplicado en vez de sumado. En la figura 134 se muestran los resultados de aplicar éste método.

Figura 134: Formación de una silueta *rim lighting*.

Primera variación

Uno de los problemas de esta aproximación es que, a pesar de que pinta el exterior

del objeto, no reproduce el efecto de una silueta ya que el color se va degradando poco a poco, y una silueta normalmente no se degrada. Es por ello que se ha implementado una segunda aproximación, que consiste en utilizar un valor límite o *threshold* según el cual, si el valor es superior a ese límite el resultado es 1 y si no lo supera 0. Los resultados de esta aplicación son los siguientes.

Figura 135: Formación de una silueta utilizando un valor de corte sobre el *rim lighting*.

Los resultados de este método tampoco son los esperados ya que por lo general los modelos 3D de la escena están formados por muy pocos polígonos, de manera que en superfícies grandes y planas como la mesa de la figura 135, es decir, planas y alargadas, la silueta oscurezca la mitad de la superfície súbitamente.

Segunda variación

Con el fin de tratar de paliar este efecto se ha pensado en realizar una discretización del valor del *rim lighting* de forma similar al *Stepped Toon Shading*. Para realizar ésta discretización se ha utilizado una función escalonada en la que se pueda modificar el suavizado entre escalones.

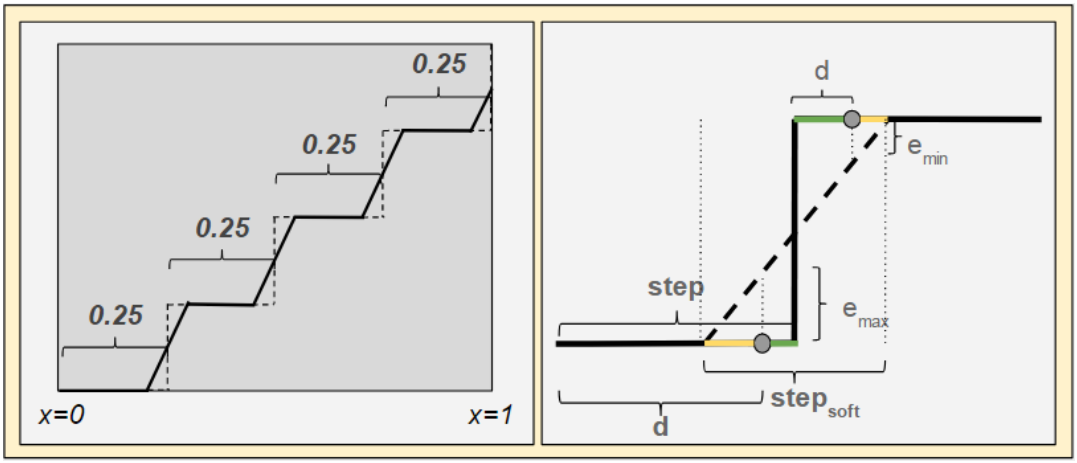


Figura 136: Función escalonada con suavizado.

Dados los valores x, $step$ y $step_{soft}$ de la figura 136 derecha, donde $max(step_{soft}) = step$, se puede construir una función como la que se muestra en la imagen de la izquierda. La función escalonada sin suavizado, con la línea discontínua, se construye mediante la fórmula 7.18 de la sección de creación de agua.

Para realizar el suavizado, en primer lugar se comprueba si x está dentro de una sección de transición. En el supuesto que una de estas dos comprobaciones resulte cierta se realiza una interpolación lineal entre los valores 0 y $-step$ o $+step$ en función de si la transición es al inicio o al final del *step*. Las fórmulas para calcular los valores de la interpolación son las siguientes:

$$weight_{min} = \frac{\frac{step_{soft}}{2} - d}{step_{soft}} = \frac{1}{2} - \frac{d}{step_{soft}} \tag{8.2}$$

$$weight_{max} = \frac{\frac{step_{soft}}{2} - (step - d)}{step_{soft}} = \frac{1}{2} - \frac{step - d}{step_{soft}} \tag{8.3}$$

$$\begin{cases} e_{min} = -step \cdot weight_{min}, & \text{if } d < \frac{step_{soft}}{2} \\ e_{max} = +step \cdot weight_{max}, & \text{if } (step - d) < \frac{step_{soft}}{2} \end{cases} \tag{8.4}$$

Para calcular el valor final de la función simplemente es necesario sumar los valores e_{min} y e_{max} a la variable x.

Sin embargo, como se observa en las imágenes de la figura 137 los resultados no son demasiado óptimos en superfícies alargadas, además de provocar que hayan demasiadas franjas de iluminación junto con el *Stepped Toon Shading* lo cual puede resultar molesto a la vista.

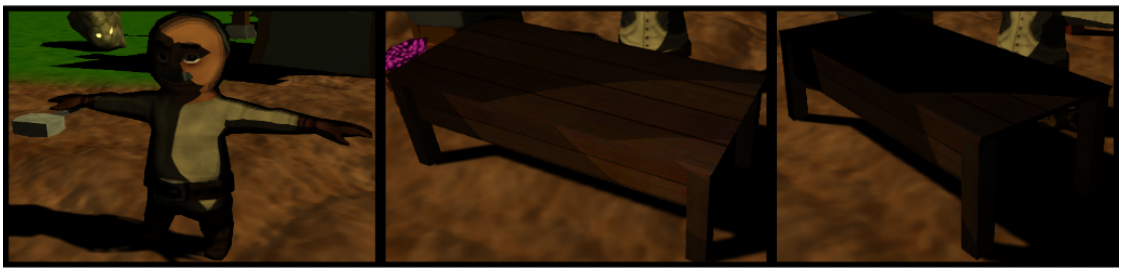


Figura 137: Aproximación discretizando el valor del *rim lighting*.

Tercera variación

La última aproximación buscada ha sido intentar conseguir una silueta definida como la segunda aproximación pero con una transición suavizada pero rápida de 0 a 1 en función del valor de *rim* utilizando la fórmula $x = 50x^2$ representada en la figura 138. A pesar de que, como se muestra en la figura ésta es probablemente la mejor solución al problema, sigue sin ser aceptable para superfícies largas y planas, como se muestra en la figura 139.

Los resultados obtenidos a la hora de conseguir siluetas en los objetos de la escena utilizando éstos métodos han sido peores de lo esperado debido en parte a la baja cantidad de polígonos que forman los modelos 3D de la escena, sin embargo, pueden resultar útiles para otro tipo de objetos, y por tanto, finalmente se ha decidido integrar en los *shader CelShader* y *ComplexCelShader* mediante el *popup Contour*.

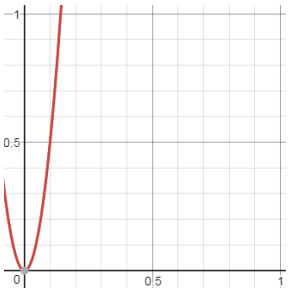


Figura 138: Fórmula de suavizado del *rim lighting*.

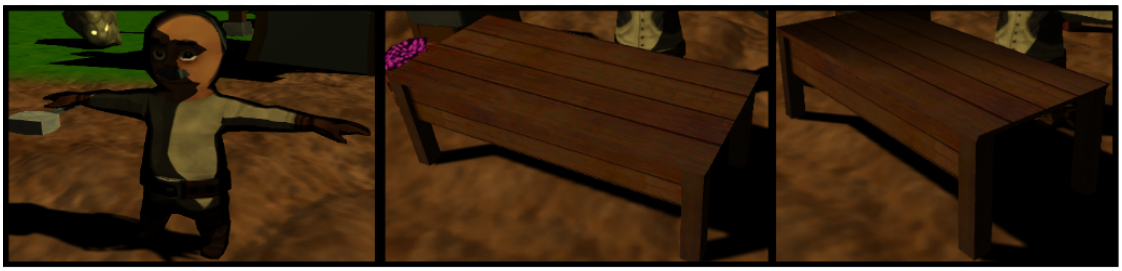


Figura 139: Aproximación suavizando el valor del *rim lighting*.

9 Realidad Virtual en Unity

La realidad virtual es un sistema de visionado que, en un juego adaptado a esta tecnología, permite que un jugador pueda explorar en primera persona los distintos escenarios que conforman una escena de un juego, de manera que la escena lo envuelva en las tres dimensiones. Uno de sistemas de realidad virtual más conocidos en la actualidad es *Occulus Rift*[3].

Para poder emplear esta tecnología, tanto en el desarrollo de un juego como a la hora de jugarlo, se necesitan dispositivos que intercambien información sobre la posición y rotación de la cabeza del jugador a tiempo real con la máquina que ejecuta la escena. En la figura 140 están presentados los dispositivos que se han utilizado en este proyecto para poder adaptar el juego a la realidad virtual.

(a) *Head Mounted Display (HMD).* (b) Sensor de movimiento y rotación. (c) Mando de *XBOX One* junto con el adaptador *wireless*.

Figura 140: Dispositivos necesarios para poder utilizar realidad virtual en juegos.

Estos dispositivos son los siguientes:

- ***Head Mounted Display* o *HMD*:** El *HMD* es un dispositivo que se coloca en la cabeza y que permite al jugador observar las imágenes de la escena a través de las gafas que tiene en la parte delantera. Este dispositivo también dispone de un sistema que calcula la rotación de la cabeza en los ejes x, y y z.
- **Sensor:** El sensor es el dispositivo que envía información sobre la posición y la rotación del *HMD* a la máquina en la que se ejecuta el juego con el fin de poder modificar los parámetros de la cámara de la escena para de que ésta gire junto con la cabeza del jugador y se mueva cuando el jugador se aparte hacia alguna dirección. El sensor también ejerce la función de punto de referencia, de manera que la rotación del dispositivo sea de 0° en los tres ejes cuando el *HMD* esté orientado con la parte donde se encuentran las gafas mirando frontalmente al sensor.
- **Mando de *XBOX One*:** Es el método de entrada escogido. Consta de el mando en sí y de un adaptador con USB sin cables. A través del mando se puede mover al personaje por la escena.

9.1 Funcionamiento de la realidad virtual en Unity

En primer lugar, para poder utilizar el sistema de realidad virtual como desarrollador, es necesario haber instalado correctamente los tres dispositivos presentados al comienzo de esta sección. En el apéndice 11 apartado 11.1.2, dentro de los pasos para poder utilizar la realidad virtual en el proyecto, se explica el proceso de instalación de los dispositivos.

Normalmente, en un juego sin realidad virtual, la cámara principal de la escena pinta la parte que está grabando dentro de una textura denominada una *Render Texture*. Esta textura es el resultado de haber pasado toda la escena visible a través del *pipeline* gráfico, es decir, a través de todo el proceso de cambios de coordenadas y *shaders* que llevan a la imagen que ve el usuario final. Una vez la *Render Texture* se ha generado, ésta se puede redirigir tanto al dispositivo de salida de vídeo, como por ejemplo puede ser la pantalla del ordenador, conocido como *Target Display*, como a una textura (que más tarde puede ser utilizada dentro de un *shader*).

Cuando en un proyecto de Unity se marca la casilla *Virtual Reality Supported* (*Edit* > *Project Settings* > *Player* > *Other Settings*), Unity automáticamente cambia el *Target Display* a las gafas de *HMD* en vez de la pantalla del ordenador. Además, debido a que los dos ojos no se encuentran en la misma posición, sino que están separados por lo que se denomina distancia interpupilar, la escena se debe pintar desde dos puntos de vista diferentes para cada ojo. Esto se consigue utilizando matrices Vista (de coordenadas de mundo a coordenadas de cámara) y Proyección (de coordenadas de cámara a coordenadas de *clip*) diferentes para cada ojo. Cuando se activa la realidad virtual, Unity modifica estas matrices internamente de manera automática teniendo en cuenta los datos de rotación y posición detectados por el sensor, además del campo visual (*field of view*) de la cámara, o la distancia interpupilar, ajustable desde el propio *HMD*.

Por tanto, el proceso de creación de imágenes de una escena mediante realidad virtual es bastante más costoso que siguiendo el proceso normal sin ella, ya que se pinta la misma escena dos veces desde puntos de vista diferentes, una para cada ojo. Para reducir el coste de todo el proceso, Unity utiliza *Render Textures* de baja resolución, lo cual disminuye la definición de la escena en general. En caso que se desee mejorar la calidad final se puede aumentar la escala de las *Render Textures*. Cuando se aumenta esta escala, aumenta también la resolución final, mientras que el rendimiento baja debido al aumento del coste de pintar un mayor número de píxeles. Cuando, por el contrario, se disminuye la escala, disminuye también la resolución final de la imagen, reduciéndose la cantidad de cómputo total.

9.2 Integración de realidad virtual en el proyecto

Con el fin de adaptar la escena del poblado, alrededor de la cual gira este proyecto, a un sistema de realidad virtual, se ha dividido esta integración en diversas subtareas.

La primera de ellas es poder alternar entre el juego normal y el juego con realidad virtual. Para realizar este cambio, se ha creado un nuevo *script* de comportamiento

en lenguaje *c# ToggleVR.cs* y se ha asociado a un objeto vacío de la escena. Dentro de este *script*, a cada *frame* se comprueba si se ha pulsado la tecla de espacio. En caso de ser así, se llama a una función que realiza los cambios entre los dos sistemas de entrada.

La configuración con realidad virtual de la escena presenta algunos cambios respecto de la escena con el sistema normal de visualizado. En primer lugar, se ha decidido cambiar la perspectiva de la cámara. Cuando se explora la escena sin el sistema de realidad virtual la cámara utiliza una perspectiva en tercera persona, sin embargo, lo que se ha querido implementar con la realidad virtual es que el jugador pueda ser el personaje principal, y por lo tanto se debe implementar una perspectiva en primera persona.

La implementación de este cambio de perspectiva pasa por generar un enum *CAMERA_PERSPECTIVE* con los valores *FIRST_PERSON* y *THIRD_PERSO* dentro del controlador de la cámara *MainCameraController.cs*, asociado al *GameObject* que contiene la cámara. Cuando desde la clase *ToggleVR* se activa la perspectiva en primera persona[11], se modifican tanto la posición como la rotación del *GameObject* que contiene la cámara, bajando a ésta hasta aproximadamente la cabeza del personaje. Para evitar que el personaje pueda interferir en la visión del jugador, desde la función *SetHeroVisibility* en la clase *ToggleVR* se desactivan los componentes *MeshRenderer* y *SkinnedMeshRenderer* sin los cuales no se puede pintar al personaje.

Otra de las subtareas, además de cambiar la perspectiva, también es cambiar el método de entrada. En el juego de *Fracsland* sin realidad virtual es el ratón el que indica hacia dónde se debe dirigir el personaje a la dirección que marca el mando, sin embargo, no es posible (o como mínimo nada recomendable) utilizar el ratón como método de entrada, por tanto se ha decidido utilizar el mando como dispositivo de entrada. El movimiento del personaje con el mando se muestra en la figura 141.

Figura 141: Movimientos del personaje con el mando.

Dentro de la clase *c# Hero* (donde también se comprueba la entrada del ratón), se ha creado la función *JoystickInput*, donde se comprueba la entrada del mando. Unity habitualmente reconoce el *joystick* izquierdo de los mandos, pero no el de-

[11]Mediante la instrucción SetCameraPerspective(CAMERA_PERSPECTIVE.FIRST_PERSON.)

recho. Para poder recoger la entrada de este *joystick* se han creado dos nuevos ejes (*axis*) *JoystickRightX* y *JoystickRightY* en la ventana *InputManager* de Unity, como muestra la figura 142.

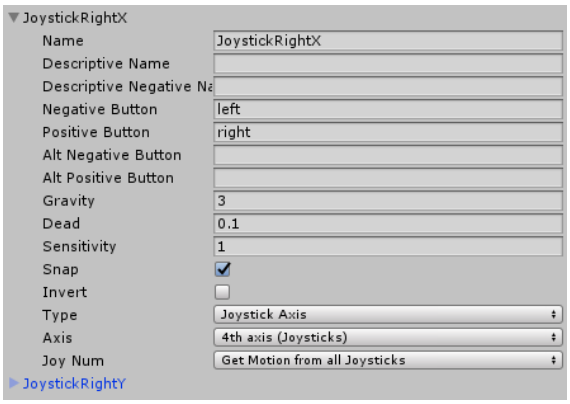


Figura 142: Propiedades del eje.

Los movimientos de translación y rotación no se realizan sobre la cámara principal directamente, ya que Unity no lo permite con el fin de evitar que se sobreescriban los valores de posición y rotación, actualizados constantemente, obtenidos de la información que recibe la cámara del sistema de realidad virtual, si no que se realizan sobre el *GameObject* que contiene a la cámara, cambiando así la posición y la rotación global de la cámara, propiedades asociadas al *GameObject* padre.

9.2.1 Adaptación de la interfície de usuario a la realidad virtual

Quizá la mayor diferencia entre los juegos que utilizan realidad virtual y los que no es la manera de interactuar con el jugador, es decir, la interfície de usuario.

En el juego *Fracsland*, la información se muestra al jugador a través de una serie de paneles superpuestos al resto de la escena, de manera que siempre están situados al frente. Sin embargo, con realidad virtual esto no es recomendable ya que los humanos no estamos acostumbrados a tener un objeto siempre delante cuando movemos la cabeza, y por tanto, puede resultar molesto o incluso producir mareos. Los juegos de realidad virtual generalmente integran los elementos informativos como un objeto más dentro de la escena, de manera que el jugador pueda verlo sólo si mira en la dirección en la que se encuentra.

En la escena del juego original hay dos personajes secundarios (*NPC*) con los que el personaje principal puede interactuar. Uno de ellos es *Lord Barus* y el otro es el vendedor del poblado. El primero te permite escoger una misión mientras que el segundo te ofrece mercancía. Con tal de diferenciar las funciones entre los dos personajes se han diseñado dos iconos diferentes. Dentro de los personajes se ha creado un objeto *Canvas*, donde se pueden colocar diversos elementos de la interfície de usuario, pero se marca con la opción *World Space* dentro de el desplegable *Render Mode*, en vez de la opción *Screen Space*, como se puede ver en la figura 143. La diferencia entre estos dos tipos de interfície se muestra en la figura 144.

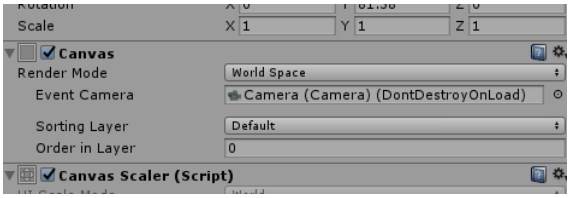
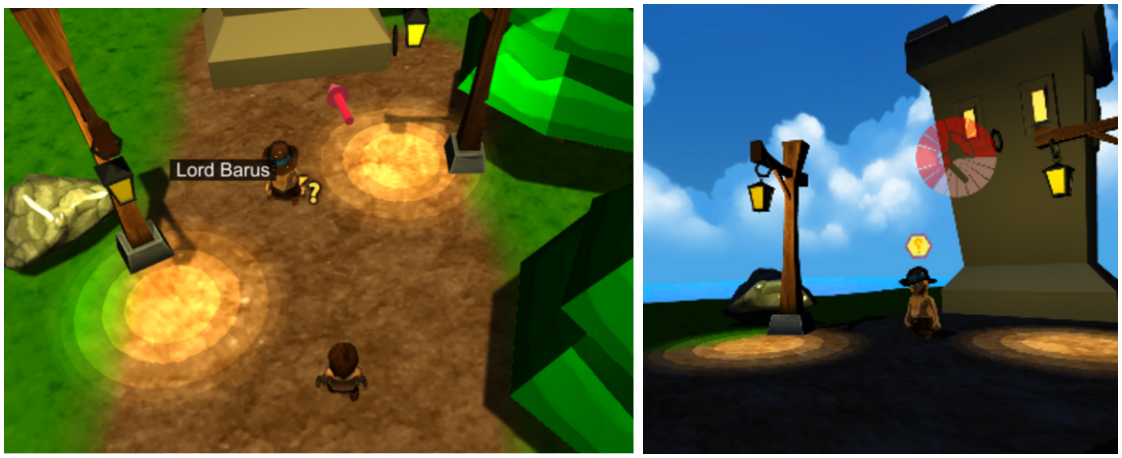


Figura 143: Generación del elemento de la interfície en coordenadas de mundo.



(a) Información con perspectiva de tercera persona.

(b) Información con perspectiva de primera persona.

Figura 144: Diferencia entre la interfície de usuario con realidad virtual y sin ella.

A pesar de que se ha mencionado que no es recomendable tener elementos fijos en la interfície cuando se está utilizando realidad virtual, hay casos específicos en que la situación justifica tenerlos. Es el caso del medidor de vida. El jugador debe saber en todo momento la vida que tiene y no se le puede forzar a girar la cabeza en alguna dirección concreta para poder verlo. Tiene que estar siempre a la vista. Para realizar este medidor se ha implementado un *shader* con el que mostrar la vida del personaje protagonista en la pantalla. Este *shader* está situado dentro del sistema de *shaders* como *"Custom/VR/FixedHealth"* y está en *"Assets/Shaders/VR/FixedHealth"*. La figura 145 muestra su inspector.

El *shader* del medidor de vida está formado por las siguientes propiedades o campos:

- El campo *UI Texture* actúa como máscara mediante el canal rojo. El color final se multiplica por este valor, afectando de esta manera también al canal *alpha*, y por lo tanto, volviendo el resultado transparente cuando el canal rojo de la textura está a cero.
- Los colores *Color Full* y *Color Empty* representan los colores que tienen las secciones cuando están llenas y cuando están vacías respectivamente.
- El valor *Visibility* esconde (0) o muestra completamente (1) el medidor, multiplicando el valor del canal *alpha* por el valor de *Visibility*. Esta variable

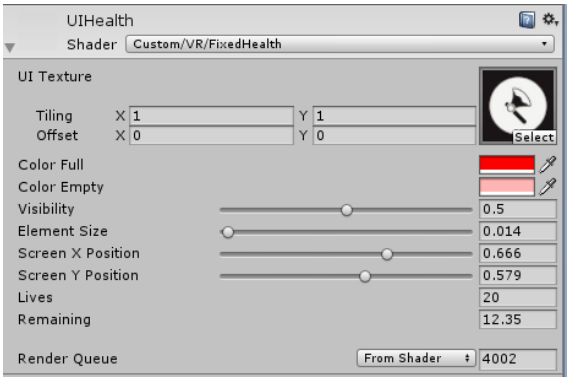


Figura 145: Inspector del *shader* que genera el medidor de vida.

permite que se pueda ver lo que hay detrás del medidor, evitando bloquear todos los objetos que haya detrás. De esta manera se minimizan las opciones de provocar mareos.

- El campo *Element Size* aumenta o disminuye el tamaño del medidor.
- Las posiciones *Screen Position X* y *Screen Position Y* sitúan el medidor en la pantalla, donde $(x, y) = (0, 0)$ abajo a la izquierda de la pantalla y $(x, y) = (1, 1)$ arriba a la derecha.
- Finalmente, los valores *Lives* y *Remaining* representan respectivamente el número de vidas total y el número de vidas que le quedan al personaje.

En primer lugar para poder representar un objeto delante de la cámara con una posición fijada en la pantalla, es necesario que siempre aparezca delante del resto de objetos en la escena. Esto se consigue utilizando la etiqueta *Queue* con el valor *Overlay* (valor 4000) al principio del *shader* que crea el material del objeto. En concreto, para el medidor de salud se ha escogido un valor de 4002 para asegurar que aparezca por delante de otros objetos con un valor de cola de 4000. Para que también el material pueda tener semitransparencias se ha utilizado la instrucción *Blend SrcAlpha OneMinusSrcAlpha* con la que se pide a Unity que realize la interpolación explicada en la sección de semitransparencias 4.3.

La estructura de entrada al *Vertex Shader* es muy simple, tan sólo está formada por las coordenadas de textura y la posición en coordenadas de objeto. Con esta información se puede realizar el cálculo directo de las coordenadas de *clip* que más adelante se pasarán al *Fragment Shader* teniendo en cuenta las variables globales especificadas en los campos *Screen Position X*, *Screen Position Y* y *Element Size*, además de las coordenadas de objeto de la estructura de entrada.

Teniendo en cuenta que la posición en coordenadas de *clip* para las componentes x e y debe estar dentro del rango $[-1, 1]$, donde (-1,-1) representa la esquina inferior izquierda de la pantalla y (1,1) la esquina superior derecha, y que las variables *Screen Position X*, *Screen Position Y* se encuentran dentro del rango [0,1], el proceso para calcular correctamente las coordenadas tiene lugar de la siguiente manera:

$$x = 2(x + s \cdot obj_x) - 1 \tag{9.1}$$
$$y = ar \cdot (2(y + s \cdot obj_y) - 1) \tag{9.2}$$

donde s es el valor del campo *Element Size*, obj_x y obj_y son las componentes x e y de las coordenadas de objeto y ar es el *aspect ratio* de la pantalla, es decir, la altura dividida por la anchura. Este último valor evita que el objeto se estire con la forma de la pantalla.

Dentro del *Fragment Shader* tiene lugar el cálculo que permite que el objeto aparezca segmentado en varias secciones, que representan las vidas. El proceso que se ha seguido para realizar este efecto se explica a continuación.

Las coordenadas de la textura, de la misma forma que las de pantalla, tienen sus componentes a cero en la esquina inferior izquierda y a uno en la esquina superior derecha. En primer lugar, se deben transformar estas coordenadas de manera que el punto $(0, 0)$ quede situado en el centro de la textura multiplicando las coordenadas por 2 para luego restar 1. De esta manera el rango de las coordenadas pasa de $[0, 1]$ a $[-1, 1]$. Una vez realizada esta corrección, el punto en coordenadas de textura corregidas pasa a ser un vector, que a partir de ahora se llamará $\vec{t}$. El siguiente paso es calcular el ángulo entre el vector $(1, 0)$ y $\vec{t}$. El ángulo se calcula utilizando el coseno (mediante producto escalar) entre estos dos vectores normalizados y usando más tarde una función *acos* para obtener el ángulo resultante.

Una vez se ha obtenido el ángulo, cada vector, es decir, cada punto, debe saber en cuál de las secciones se encuentra para poder poder pintarse con el color de una sección llena o con el color de una vacía. Utilizando las variables globales *Lives* y *Remaining* se calcula la sección en la que se encuentra el punto, dividiendo el ángulo entre el paso, que es el resultado de dividir 360° entre el número total de vidas. También utilizando el módulo entre estos dos valores se puede establecer un límite a partir del cual el punto se encuentre en un espacio de transición entre sección y sección.

El color final se forma multiplicando el color al que pertenece el ángulo (*Color Full* y *Color Empty*) por la muestra extraída de la textura *UI Texture* y la pertenecia o no a un espacio de transición entre secciones. Los resultados obtenidos utilizando varias configuraciones del *shader FixedHealth* se pueden observar en la figura 147.

9.2.2 Resultados

Algunos de los resultados obtenidos en la escena cuando se utiliza el dispositivo de realidad virtual de *Occulus Rift* se presentan en la figura 148.

A pesar de no haber introducido demasiados elementos de interfície de usuario, se han introducido dos maneras diferentes dar información al usuario en realidad virtual. Además también se ha integrado un sistema de movimiento diferente con el mando, junto con una nueva perspectiva en primera persona.

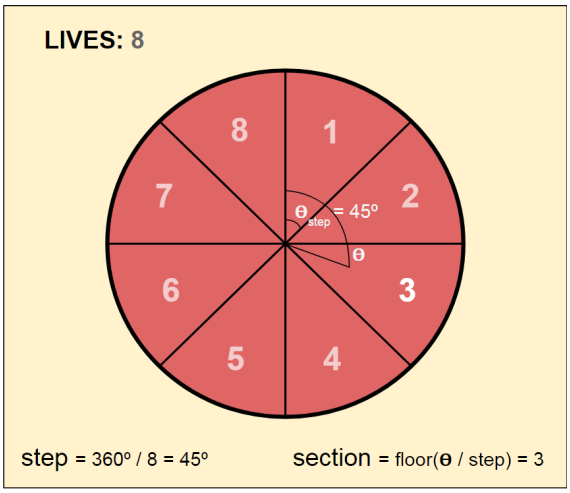
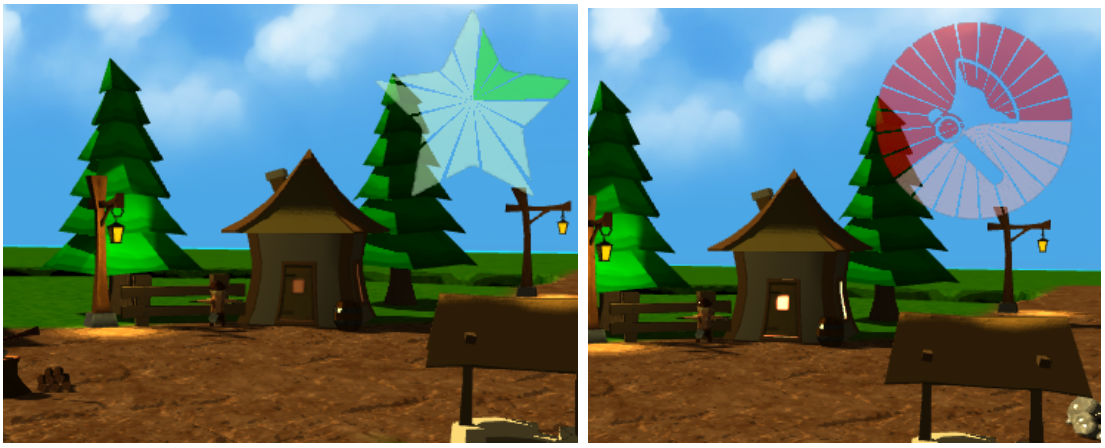


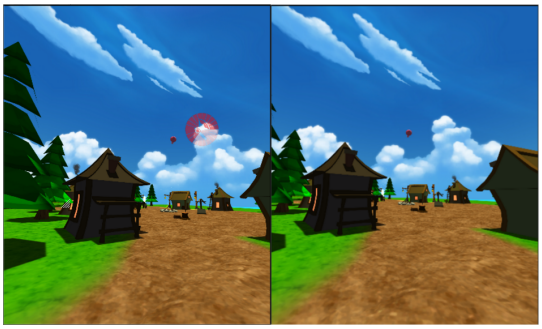
Figura 146: Esquema que muestra cómo se calculan los valores *step* y el número de sección del medidor de vida.



(a) Medidor de vida con 3 vidas restantes de 13.

(b) Medidor de vida con 15 vidas restantes de 25.

Figura 147: Diferentes configuraciones del medidor de vida.



(a) Imagen estereoscópica de la escena utilizando realidad virtual.

(b) Imagen estereoscópica de la escena utilizando realidad virtual.

(c) Imagen en 360° de la escena utilizando realidad virtual.

Figura 148: Resultados de la introducción de realidad virtual en el poblado.

10 Conclusiones y trabajo futuro

Teniendo en cuenta los objetivos de este proyecto, la mayoría de ellos se han cumplido satisfactoriamente. Después de haber modificado los programas de sombreado de los objetos del poblado, la escena tiene una estética más infantil con colores más intensos. A pesar de los resultados generales positivos del proyecto, no se ha podido introducir en la escena ninguna técnica realmente aceptable de introducción de contornos. En cuanto a la integración del sistema de realidad virtual, aunque no completada la interacción del usuario con el juego, muestra el poblado desde la perspectiva del personaje principal, permitiéndole moverse por toda la escena, lo cual ya implica una primera aproximación dentro de este ámbito, que era el objetivo marcado.

Personalmente, he quedado satisfecho con las técnicas de *shading* aprendidas, ya que me ha permitido conocer una base sólida para poder entender prácticamente todo tipo de *shaders* en un futuro. Además, esta memoria ha dejado constancia de todas estas técnicas y su implementación en Unity, además de explicaciones sobre algunos de los procesos internos, no demasiado conocidos, que realiza Unity para generar los resultados observables en sus *Standard shaders*, como es el caso de las sombras, las reflexiones especulares o las reflexiones.

Como trabajo futuro ha quedado pendiente la introducción de efectos de post-procesado como por ejemplo introducción de un contorno alrededor de los objetos de la escena, el efecto *Bloom*, que hace que la escena se difumine con la luz o en general efectos que deformen la imagen final de la escena. También tendría cabida dentro del proyecto la introducción de elementos de luz volumétrica. También queda pendiente realizar *shaders* que utilicen la técnica del *deferred shading*[12], con la que se consiguen renderizar escenas con un número de luces muy elevado con un coste muy bajo.

[12]https://es.wikipedia.org/wiki/Deferred_shading

11 Apéndice: Manual técnico

11.1 Guía de instalación para desarrolladores

11.1.1 Requisitos mínimos

Para poder instalar el proyecto sin Realidad Virtual es necesario cumplir los siguientes requisitos:

- Versión instalada del motor gráfico Unity3D igual o superior a la 5.5.0f3.
- Sistema Operativo *Windows 7 SP1 (64bit)* o superior.
- Tarjeta gráfica con *Direct X 9 (DX9)* o *Direct X 11 (DX11)*.
- Disponer de un editor de código compatible con Unity. Dos de los editores más utilizados son *Visual Studio*[13] y *MonoDevelop*[14].

Para poder instalar el proyecto con Realidad Virtual se requiere cumplir los siguientes requisitos adicionales:

- Disponer del set de *Occulus Rift* formado por el *Head Mounted Display (HMD)*, el sensor de movimiento y rotación, un mando de *XBOX One* con pilas no descargadas y un adaptador wireless para el mando o en su defecto cable adaptador.
- Tarjeta gráfica *NVIDIA GeForce GTX 1060* o superior.
- CPU equivalente o superior a *Intel Core i5- 4590*.
- Memoria RAM de 8GB.
- Una salida HDMI 1.4.
- Tres puertos USB 3.0.
- Controlador de dispositivo (driver) para Oculus 361.91 o más reciente.

11.1.2 Instalación del proyecto

Partiendo de la precondición de que se cumplen los requisitos mínimos listados, se procede a exponer el proceso de intalación del proyecto. Para instalar este proyecto sin realidad virtual se deben realizar los siguientes pasos:

1. Descargar (y extraer si necesario) el proyecto de la página *"https://github.com/rperezde11/fracs town.git"*.
2. Abrir el programa Unity3D.
3. Pulsar la opción *Open* o *Abrir*.

[13]Enlace de descarga: https://www.visualstudio.com/es/downloads/
[14]Enlace de descarga: http://www.monodevelop.com/download/

4. Seleccionar la carpeta descargada.
5. Dentro del proyecto desplazarse en la pestaña *Project* la carpeta *"Assets/Resources/Scenes/"* y abrir el archivo *Town*.
6. En caso de recibir muchos errores al cargarse la escena, cerrar Unity y volverlo a abrir cargando la escena *Town* de nuevo.

Para poder utilizar el sistema de realidad virtual dentro de proyecto es necesario realizar los siguientes pasos.

1. Descargar e instalar la aplicación configuración de *Occulus Rift*. Esta aplicación se puede descargar en *"https://www3.oculus.com/en-us/setup/"*. Cuando para continuar la instalación sea necesario introducir los dispositivos del *Occulus*, seguir los siguientes pasos:

    1.1. Conectar el cable HDMI del HMD de *Occulus* a la entrada HDMI de la tarjeta gráfica que cumpla los requisitos mencionados anteriormente. Conectar también el cable USB a una de las 3 entradas USB 3.0 requeridas.
    1.2. Conectar el cable USB del sensor de *Occulus* a una de las 3 entradas de USB 3.0 disponibles. Orientar el dispositivo apuntando a la posición en la que va a estar el HMD.
    1.3. Introducir el adaptador *wireless* para el mando de *XBOX One* en la última ranura USB 3.0.

2. Una vez concluida la instalación de los elementos, si el proceso se ha realizado correctamente, el HMD debería mostrar la tienda de *Occulus* en realidad virtual.
3. Entrar al proyecto de Unity y dentro de la escena *Town* pulsar el botón de *play* situado en la parte superior central de la aplicación, siguiendo la distribución habitual de ventanas. Esto ejecutará el juego.
4. Cuando se esté ejecutando el juego pulsa la tecla *Space*. Realizado esto el HMD debería mostrar el poblado en realidad virtual. Si esto no ocurre:

    4.1. Reiniciar el proyecto en Unity saliendo y volviendo a entrar. Volver a realizar una vez más los pasos 3 y 4. Si sigue sin funcionar, pasar a 4.2.
    4.2. Comprobar que en la ventana *Edit > Project Settings > Player > Other Settings* la casilla *Virtual Reality Supported* está marcada. En caso de no estarlo, marcarla y volver a realizar los pasos 3 y 4.

Referències

[1] "Unity3D's website"https://unity3d.com/

[2] "CrazyBump v1.2.2"http://www.crazybump.com/

[3] "Occulus Rift's webpage"https://www.oculus.com/rift/

[4] "Catlikecoding website, Unity shaders tutorial"http://catlikecoding.com/unity/tutorials/

[5] "Forward Rendering Path Details."https://docs.unity3d.com/Manual/RenderTech-ForwardRendering.html (última visita 4 de Mayo 2017).

[6] "Unity standard shaders download page", https://unity3d.com/es/get-unity/download/archive

[7] Gouraud, H. (1971). Computer display of curved surfaces. 1-80. UTEC-71-113; UTEC-CSc-71-113

[8] B. T. Phong, Illumination for computer generated pictures, Communications of ACM 18 (1975), no. 6, 311–317.

[9] Lambert, J H (1760). "Photometria, sive de Mensura et gradibus luminis, colorum et umbrae".

[10] James F. Blinn (1977). "Models of light reflection for computer synthesized pictures". Proc. 4th annual conference on computer graphics and interactive techniques: 192–198. CiteSeerX 10.1.1.131.7741 Freely accessible. doi:10.1145/563858.563893

[11] James F. Blinn. (1978). "Simulation of wrinkled surfaces.Ïn Proceedings of the 5th annual conference on Computer graphics and interactive techniques (SIGGRAPH '78). ACM, New York, NY, USA, 286-292. DOI=http://dx.doi.org/10.1145/800248.507101

[12] Kaneko, Tomomichi, Toshiyuki Takahei, Masahiko Inami, Naoki Kawakami, Yasuyuki Yanagida, Taro Maeda, and Susumu Tachi. (2001). "Detailed Shape Representation with Parallax Mapping.Ïn Proceedings of the ICAT 2001 (The 11th International Conference on Artificial Reality and Telexistence), Tokyo, December 2001, pp. 205–208.

[13] McGuire, M. and McGuire, M. (2005). "Steep Parallax Mapping."I3D 2005 Poster.

[14] Brawley, Z., and Tatarchuk, N. (2004). "Parallax Occlusion Mapping: Self-Shadowing, Perspective-Correct Bump Mapping Using Reverse Height Map Tracing.Ïn Shader X3: Advanced Rendering with DirectX and OpenGL, Engel, W., Ed., Charles River Media, pp. 135–154.

[15] HECKBERT, Paul S. HANRAHAN, Pat. (1984) “Beam tracing polygonal objects.În: ACM SIGGRAPH Computer Graphics. ACM, 1984. p. 119-127.

[16] Schlick, C. (1994). “An Inexpensive BRDF Model for Physically-based Rendering.Çomputer Graphics Forum. 13 (3): 233. doi:10.1111/1467-8659.1330233

[17] Piórkowski, Rafał and Radosław Mantiuk. “Automatic Detection of Shadow Acne and Peter Panning Artefacts in Computer Games.”(2015).

[18] Dade, Kevin , “Toonify: Cartoon Photo Effect Application”, Department of Electrical Engineering Stanford University

[19] Jason L. Mitchell , Moby Francke , Dhabih Eng. (2007) “Illustrative rendering in Team Fortress 2”, ACM SIGGRAPH 2007 courses, August 05-09, 2007, San Diego, California [doi¿10.1145/1281500.1281666]