

# Editor de materiales para modelos en 3D basado en nodos.



Grado en Ingeniería Multimedia

Trabajo Fin de Grado



Universitat d'Alacant  
Universidad de Alicante

Autor:

Santiago Palacio Caro

Tutor/es:

Rafael Molina Carmona

Septiembre 2015



---

*“All it takes is for the rendered image to look right”*

*-Jim Blinn*

*“Let the form of an object be what it may, light, shade, and perspective  
will always make it beautiful”*

*– John Constable*

*“A good picture is equivalent to a good deed”*

*- Vincent Van Gogh*

---



---

*Dedicado a Matha D. Jaramillo, querida abuela cuya luz se extinguió  
durante este viaje. Descansa en paz.*

---



# Agradecimientos

En primer lugar agradecer a mi familia por haber sido incondicionales durante todo este proceso. Por saber sacar lo mejor de mí y darme siempre el impulso extra que he necesitado en los momentos más difíciles.

A mi profesor Rafael Molina Carmona por haberme dado la grandísima oportunidad de trabajar en un campo tan fascinante como el de los gráficos por computador y por haberme dado su respaldo y consejo durante todo el proceso.

A Kristina Metodieva por motivarme y animarme cada día durante estos últimos cuatro años a conseguir todas mis metas.

A mis compañeros y amigos de universidad por permitirme compartir estos años de carrera y hacerlos muchísimo más llevadores.

A mis allegados y amigos cercanos que de una forma u otra me han ayudado soportado y que también han contribuido para que haya podido llegar hasta aquí.





# Índices.

1	Introducción .....	1
2	Justificación y Motivación .....	3
3	Objetivos .....	5
3.1	General .....	5
3.2	Específicos .....	5
4	Estado del arte y contexto .....	7
4.1	Fenómeno Visual .....	7
4.2	Radiometría y Colorimetría. ....	8
4.2.1	Ley de Lambert .....	11
4.3	Fuentes de luz .....	11
4.3.1	Luces direccionales: .....	11
4.3.2	Luces puntuales: .....	12
4.4	BRDF .....	13
4.4.1	Modelo de Lambertiano: .....	14
4.4.2	Modelo de Phong .....	14
4.4.3	Modelo de Blinn-Phong .....	15
4.4.4	Modelo de Cook-Torrance: .....	15
4.5	The Rendering Equation .....	16
4.6	Material .....	17
4.7	Shading .....	17
4.8	Dibujando en 3D .....	18
4.8.1	OpenGL pipeline .....	21
4.9	Texturas .....	22

4.9.1 Mapas de textura difusos. ....	22
4.9.2 Mapas de textura normales. ....	23
4.9.3 Mapas de texturas Especulares.....	24
4.10 Physically Based Rendering.....	25
4.10.1 Características de un sistema PBR. ....	26
4.11 Estado del arte.....	27
4.11.1 Editor de materiales .....	27
4.11.2 Altamente integrado con el motor gráfico VS Independientes.....	27
4.11.3 Para renderizado en tiempo real VS renderizado offline. ....	28
4.11.4 Tipo de interfaz: Basada en Nodos VS Clásica.....	29
4.11.5 Editor de materiales de UE4. ....	30
4.11.6 Editor de materiales 3ds Max .....	32
4.11.7 Shader Forge .....	35
4.11.8 Marmoset Toolbag.....	38
4.12 Discusión .....	39
5 Metodología y herramientas.....	41
5.1 Metodología de desarrollo de Software. ....	41
5.2 Control de versiones .....	41
5.3 Herramientas de desarrollo .....	42
5.3.1 Visual Studio .....	42
5.3.2 Qt .....	43
5.3.3 GLEW (OpenGL Extension Wrangler Library) .....	45
5.3.4 Assimp. ....	45
5.3.5 GLM (OpenGL Mathematics) .....	45
5.3.6 NotePad++ .....	45
5.3.7 Photoshop y Gimp.....	45

5.3.8 3D Max 2016.....	45
6 Modelo Propuesto .....	47
6.1 Análisis y especificación.....	47
6.1.1 Descripción general del sistema.....	47
6.1.2 Editor de nodos.....	48
6.2.1 Motor de dibujado y controles del visor 3D.....	59
6.2.2 Uso e interacción .....	64
6.2.3 Requisitos. ....	67
6.2.4 Planificación: .....	73
6.3 Diseño e implementación del sistema .....	76
6.3.1 Modulo GUI: .....	77
6.3.2 Módulo de renderizado.....	81
6.5.1 Resolviendo el grafo de nodos. ....	87
7 Experimentación.....	91
7.1 Creando un material básico. ....	91
7.1.1 Implementación: .....	91
7.1.2 Implementado en UE4. ....	93
7.2 Creando materiales usando mascaras.....	94
7.2.1 Implementación.....	95
7.2.2 Implementación con UE4. ....	98
7.3 Crear materiales con transparencia.....	99
7.3.1 Implementación.....	100
8 Conclusiones y trabajos futuros. ....	103
8.1 Aportaciones .....	103
8.2 Comparación con los objetivos.....	103
8.3 General.....	104

8.4 Específicos .....	104
8.5 Trabajos futuros .....	105
8.5.1 Mejoras en el programa. ....	105
8.6 Conclusión personal.....	106
9 Bibliografía y referencias .....	109

# Resumen

Durante el proceso de creación de contenidos como videojuegos o animaciones en 3D, nos encontramos con tareas de diversos tipos. Los procesos de creación de estos contenidos son una combinación de disciplinas traídas de campos como la tecnología, el arte, y las ciencias.

El aspecto es uno de los puntos más importantes a tratar en cualquier proyecto multimedia, ya que éste proporciona una identidad visual. En términos de gráficos por computador, el aspecto de los objetos que se dibujan es consecuencia del comportamiento de sus superficies frente a la luz. Estos comportamientos propios de los objetos, modelan el concepto de “material” al igual que lo hacen los materiales en el mundo real.

El trabajo aquí expuesto propone el desarrollo de una herramienta que posibilite la creación de materiales para objetos en 3D. Esta aplicación cuenta en su diseño con una interfaz visual basada en nodos la cual enriquece el flujo de trabajo haciendo de dicha tarea un proceso cómodo y sencillo mediante la manipulación de los nodos y sus conexiones.

Dadas las tecnologías actuales definir el aspecto de un objeto es en definitiva definir el Fragment Shader (pequeño programa que controla la etapa de asignación de color a los objetos dentro del proceso de dibujado) que dota al objeto de dicho aspecto, por lo tanto la tarea de crear un material en definitiva se deriva a la de crear un Fragment Shader. Para la creación de estos Shaders el programa se basa en el grafo de nodos que se va creando a medida que se trabaja con el editor para de esta forma generar un flujo de programa que luego será reconvertido a un Fragment Shader escrito en el lenguaje de programación GLSL(OpenGL Shading Language).



# 1 Introducción

En la actualidad pasamos por grandes periodos de cambio, la tecnología nos sorprende cada vez más. Su avance parece no detenerse y su potencial parece no agotarse; así mismo, el campo de los gráficos por computador parece uno de los más ricos y prolíferos. Si bien ha crecido de manera notable en las últimas tres o cuatro décadas es evidente el impacto que tienen sobre este crecimiento determinados sectores económicos que manejan volúmenes de capital considerables como los videojuegos, el cine y el ocio digital en general.

La tecnología avanza, esto implica que los dispositivos son cada vez mejores y las técnicas que antes servían para hacer una cosa, hoy en día son mucho más depuradas, optimizadas y en general mejores que antaño. No obstante, hay que recalcar que el software avanza en función del hardware, así pues, cosas que antes no podíamos hacer, hoy en día con las tarjetas gráficas modernas podemos hacerlas en nuestros propios hogares.

Han sido innumerables los aportes al campo de los gráficos por computador. Se han hecho grandes motores gráficos que son capaces de simular efectos y fenómenos, tan reales como si de una fotografía se tratara. Y es precisamente ahí, en el aspecto y acabado final de los objetos en donde este trabajo pretende versar.

En gran medida, el realismo se consigue aplicando el color correcto –o más aproximado– sobre las superficies de los objetos que son expuestos a alguna fuente de luz. La forma de determinar dicho color es calculada basándose en distintos parámetros. El material es un concepto que encapsula varios de estos parámetros y define una forma general de comportarse bajo la incidencia de la luz para diferentes tipos de superficies o bien, **materiales**.

Definir un material en el mundo real puede ser algo sencillo, podríamos decir que con mencionar la palabra “metal”, nos referimos a que algún objeto esta hecho de metal. Lo mismo sucede con el cuero, o la piedra; no obstante, para un artista que trabaja en gráficos 3D, estas descripciones resultan planas e insuficientes. Si bien un artista debe tener en mente a que material quiere convertir un objeto (cuero, metal, etc.), también se preocupa

por entender cómo funciona la luz sobre una superficie, y así saber qué parámetros son los que hacen que un material se comporte como tal. De esta forma el artista puede emular dicho material y conseguir el aspecto más realista posible.

Dicho lo anterior, este proyecto intenta aportar una solución para los artistas que necesitan integrar un material en un entorno 3D usando un sistema sencillo, usable y entendible. Todo ello mediante el uso de técnicas modernas de gráficos y los estándares actuales.



## 2 Justificación y Motivación

Diferentes son los motivos por los cuales se ha decidido realizar este trabajo. Cabe volver mencionar que se trata de un trabajo de fin de grado, por lo tanto, es un requisito indispensable para todos los estudiantes del régimen estudiantil, el presentar y superar dicha prueba para obtener el título de ingenieros que este conlleva. No obstante, el enfoque que se tiene de éste no es el de un obstáculo, sino por el contrario, la de una grandísima oportunidad para demostrar las habilidades conseguidas durante todo el periodo estudiantil y que mejor forma de hacerlo que afrontando un proyecto tan desafiante como lo es éste. Un desafío nunca mejor recibido y que se afrontó con toda la pasión y esfuerzos posibles.

Dicho lo anterior, también se deben mencionar motivaciones más concretas fuera de lo que representa el trabajo de fin de grado como requisito para la obtención de un título universitario.

El estado actual del arte y de la industria involucrada en mantener sistemas 3D es cada vez más complejo, aportando soluciones y métodos únicos que permiten obtener resultados cada vez más interesantes y logrados. La realización del trabajo aquí descrito es también motivada por el gran interés que me despiertan las mencionadas tecnologías. Así pues, uno de los principales focos de atención dentro de un entorno de gráficos en 3D es la posibilidad de que dichos sistemas puedan aportar a modelos 3D un aspecto único, un material. Un material que le dota de un aspecto en ocasiones realista, estilizado o abstracto, según la finalidad; pero en definitiva es donde recae la belleza visual y la verdadera esencia artística y tecnológica.

Es en este punto donde el arte y la tecnología se unen permitiendo a artistas e ingenieros gráficos el lograr grandes creaciones a través de su trabajo en combinación. Es aquí donde este trabajo busca conseguir unificar de una forma nueva y más sencilla la combinación entre estos dos mundos. Por eso la decisión tomada fue la de crear una herramienta para crear materiales para objetos en 3D, sencilla y fácil de usar, que permita al usuario que lo use entender el proceso por el cual pasan los modelos en 3D hasta ser dibujados de una

forma más intuitiva y más aún, entender el estado de Shading en donde se le asignan los colores a las superficies de los modelos teniendo en cuenta tanto parámetros del material como parámetros de iluminación.

Una vez tomada la decisión de desarrollar una herramienta para crear materiales y por consiguiente **Shaders** (pequeño programa capaz de controlar determinados estados del proceso de dibujado o *graphic pipeline*), se detectó un modelo frecuente de este tipo de aplicaciones en los principales productos de la industria actual – motores gráficos, herramientas de modelado, etc.-. Dichas aplicaciones tienen en común que todas son capaces de generar materiales usando una interfaz basada en nodos. Esto simplifica de una manera considerable la forma en que se generan dichos materiales, entre otros motivos, porque se sustenta de las virtudes de un **VPL** (visual programming language) como el fácil entendimiento del funcionamiento del programa debido a que está basado en gráficos y grafos intuitivos, entre otros.

Una aplicación basada en nodos y que genere las operaciones necesarias para un determinado material en función del grafo de nodos que se haya diseñado de antemano permitiría a los artistas sin conocimientos de programación, gráficos en profundidad, ni HLSL/GLSL (lenguajes de programación) poder aportar todo su talento sin problema alguno.

En definitiva, este trabajo es considerado como un buen – y desafiante- proyecto a desarrollar como parte del TFG ya que implica tener los suficientes conocimientos que conlleva este tipo de aplicaciones. Conocimientos que han sido adquiridos durante los últimos años de carrera universitaria y que han sido puestos a prueba aquí.

## 3 Objetivos

### 3.1 General

El objetivo general de este proyecto es el de realizar una herramienta capaz de generar Shaders que definan el aspecto de objetos en 3D. La herramienta debe funcionar bajo una interfaz basada en nodos.

### 3.2 Específicos

- Desarrollar un modelo de iluminación parametrizable y capaz de ser alimentado por la información que produzca el usuario usando el grafo de nodos.
- Construir un editor de nodos que permita crear y eliminar nodos así como también crear y eliminar conexiones entre los nodos.
- Implementar una aplicación usable e intuitiva que permita al usuario llevar a cabo su tarea de la forma más sencilla posible proporcionándole soluciones para problemas típicos como la redundancia, las repeticiones y en definitiva mejorar el flujo de trabajo.
- Implementar una aplicación capaz de generar un Shader bajo los estándares de GLSL basado en el grafo de nodos definido por el usuario. Dicho Shader debe ser fácil de integrar en sistemas externos, o bien se le ha de dar usuario algún método para hacerlo.
- Obtener la capacidad para hacer uso de tecnologías nunca antes vistas haciendo uso del sentido de investigación y de aprendizaje que los años de carrera han aportado como parte de la experiencia educativa.



## 4 Estado del arte y contexto

Este trabajo pretende construir un modelo de iluminación no muy complejo y fácilmente alimentable por los datos que el usuario genere, datos que finalmente serán los que se le pasen al modelo como parámetros del mismo.

A continuación se hará un análisis a grandes rasgos de los fundamentos teóricos y técnicos necesarios para entender la consecución de un sistema de dibujado 3D en tiempo real, tomando en cuenta los fenómenos físicos y también los procedimientos modernos más comunes en el campo de los gráficos.

### 4.1 Fenómeno Visual

Cuando se estudia el dibujado por ordenador es útil primero entender algunos comportamientos que caracterizan las imágenes que percibimos del mundo real y su posterior técnica para simularla en sistemas computacionales.

---

*“Like so many challenges in computer science, a great place to start is by investigating how nature works”. (Fletcher & Parberry, 2011)*

---

En primera instancia la luz es un fenómeno físico, que se puede entender como una onda que se mueve en el espacio, o bien, se puede entender como un cuerpo minúsculo que viaja a gran velocidad en línea recta (fotón).

La luz es emitida por cuerpos como el sol o las luces artificiales que encontramos en nuestro entorno. Esta luz interactúa con objetos, una parte de dicha luz es absorbida y otra parte es dispersada. Son estos dos fenómenos los que alteran el aspecto de un objeto cuando son recogidos por un sensor, bien sea el ojo humano, una cámara fotográfica, un sensor eléctrico etc.

La cantidad de luz dispersada junto con la cantidad de luz absorbida son el resultado de la interacción física que tiene el material con el rayo de luz en definitiva, y quien por lo tanto es capaz de determinar el aspecto de las superficies. En pocas palabras, la cantidad de luz reflejada y absorbida dota a una superficie de un aspecto visual determinado. Este comportamiento podría ser modelado generando un conjunto de propiedades (en determinados casos, véase tipos de BRFD en apartado 4.4) que determinan eventualmente la forma en que se manejan las superficies bajo una fuente de luz, a este conjunto de propiedades podemos llamarla a este punto, **Material**. Más adelante hablaremos en profundidad de los materiales.

## 4.2 Radiometría y Colorimetría.

Cabe valorar a este punto la importancia de hacer un análisis sobre la radiometría y la colorimetría. Si bien son dos conceptos que pueden resultar ajenos a la informática y a los gráficos, son fuentes importantes de estudio en términos de medir y modelar el comportamiento de la luz y su percepción sobre nuestros ojos, pues ofrecen valiosas herramientas que facilitan su entendimiento.

“La radiometría es la ciencia que estudia la medición de la radiación electromagnética. Su campo abarca las longitudes de onda del espectro electromagnético (frecuencias entre  $3 \times 10^{11}$  y  $3 \times 10^{16}$  Hz), al contrario de la fotometría que solo se ocupa de la parte visible del espectro, es decir, la que puede percibir el ojo humano.” (Wikipedia, n.d.). Para cada término radiométrico existe un término similar en fotometría, no obstante para el estudio de gráficos en ocasiones resulta más conveniente (y común) trabajar bajo términos de radiometría por razones como que los dispositivos RGB funcionan de una forma más radiométrica que fotométrica debido a diferencias en la definición de términos como la *radiant flux* (energía por unidad de tiempo en radiometría) y la *luminous flux* (energía por unidad de tiempo en fotometría). Debido a esto, en tareas como convertir imágenes a escala de grises o calcular el brillo de un pixel es conveniente el uso de términos fotométricos mientras que para dibujar imágenes RGB resulta más natural trabajar en términos radiométricos que fotométricos[7].

El concepto popular que todos tenemos sobre el color es que este es una combinación de en alguna medida la luz roja, verde y azul (RGB). No obstante, este concepto no es del todo correcto, pues bien, la luz puede tomar cualquier frecuencia en el ancho de banda del espectro visible, o una combinación de ellas. El color es un fenómeno de percepción humana y no es lo mismo que una frecuencia. De hecho diferentes frecuencias de luz pueden ser percibidas como diferentes colores, a este fenómeno se le conoce como *metamers* [4].

“La colorimetría es la ciencia que estudia la medida de los colores y que desarrolla métodos para la cuantificación del color, es decir, obtener valores numéricos del color.” (Wikipedia, n.d.). La colorimetría se encarga de proporcionar métodos para la clasificación y la reproducción de los colores. Los distintos sistemas de interpretación del color son llamados **espacios de color**. La reproducción un color depende en gran medida del dispositivo sobre el cual se quiere reproducir y del espacio de color.

Para describir la distribución espectral de la luz hace falta una función continua. Sin embargo para la percepción humana de la luz es suficiente con los tres números R, G y B [4].

Existen varios espacios de color, y no necesariamente el RGB es el mejor, pero por varios motivos resulta el más convenientes es campos como la informática y los gráficos. Principalmente el espacio RGB saca ventaja en el hecho de que la mayoría de dispositivos de visualización hoy por hoy están basados en este estándar.

Al fin y al cabo lo que interesa es poder medir la intensidad de la luz en términos de la radiometría. Así el primer término en entrar a escena es la **energía radiante** (*radiant energy*) que no es más ni menos que la energía electromagnética en radiometría. Sus unidad en el Sistema Internacional (SI) es el *julio* (J). Además interesa conocer la medida para cantidad de energía por unidad de tiempo, es decir la potencia. En el SI, la potencia viene dada por *watt* o *vatio* que es un *julio* por segundo ( $1\text{ W} = 1\text{ J/s}$ ). La potencia en energía electromagnética es llamada **flujo radiante** (*radiant flux*) [4]. De esta forma el flujo radiante mide la cantidad de energía que incide, emite o fluye sobre una superficie. Por otro lado si consideramos una superficie de  $2\text{ m}^2$  que emite una cierta cantidad de flujo radiante y otra superficie de  $20\text{ m}^2$  emitiendo la misma cantidad, la superficie con menos área va a parecer mucho más brillante que la de mayor. La densidad de energía por unidad de área es también conocida como

**radiosity** cuyas unidades en el SI son el W/m. En ocasiones el termino radiosity puede cambiar en función de la dirección del rayo de luz, así pues, si el rayo está incidiendo sobre la superficie es usado el termino **irradiancia** (*irradiance*), si el rayo está siendo emitido por la superficie el término usado es *exitance* o **emitancia radiante** (*radiant emittance*) y por otro lado si el rayo está dejando la superficie (después de una reflexión p. ej.) se usa el termino radiosity.

Para medir la cantidad de brillo de un rayo de luz (lo cual es importante a la hora de construir un modelo físico) se debe medir la cantidad flujo por unidad de área proyectada. El término usado es **radiancia** (*radiance*).

La Tabla 4.1 enumera los términos radiométricos y sus unidades en el SI.

**Unidades del SI utilizadas en radiometría**

Magnitud física	Símbolo	Unidad del SI	Abreviación	Notas
<b>Energía radiante</b>	Q	julio (unidad)	J	energía
<b>Flujo radiante</b>	$\Phi$	vatio	W	Energía radiada por unidad de tiempo. Potencia.
<b>Intensidad radiante</b>	I	vatio por estereorradián	$W \cdot sr^{-1}$	Potencia por unidad de ángulo sólido
<b>Radiancia</b>	L	vatio por estereorradián por metro cuadrado	$W \cdot sr^{-1} \cdot m^{-2}$	Potencia. Flujo radiante emitido por unidad de superficie y por ángulo sólido
<b>Irradiancia</b>	E	vatio por metro cuadrado	$W \cdot m^{-2}$	Potencia incidente por unidad de superficie
<b>Emitancia radiante</b>	M	vatio por metro cuadrado	$W \cdot m^{-2}$	Potencia emitida por unidad de superficie de la fuente radiante



<b>Radiancia espectral</b>	$L_\lambda$	vatio por estereorradián por metro cúbico o	$W \cdot sr^{-1} \cdot m^{-3}$	Intensidad de energía radiada por unidad de superficie, longitud de onda y ángulo sólido. Habitualmente se mide en $W \cdot sr^{-1} \cdot m^{-2} \cdot nm^{-1}$
	o	vatio por estereorradián por metro cuadrado por hercio	o	
	$L_\nu$		$W \cdot sr^{-1} \cdot m^{-2} \cdot Hz^{-1}$	
<b>Irradiancia espectral</b>	$E_\lambda$	vatio por metro cúbico o	$W \cdot m^{-3}$	Habitualmente medida en $W \cdot m^{-2} \cdot nm^{-1}$
	o	vatio por metro cuadrado por hercio	o	
	$E_\nu$		$W \cdot m^{-2} \cdot Hz^{-1}$	

**Tabla 4.1.** Unidades del SI utilizadas en radiometría. (Fuente: Wikipedia)

### 4.2.1 Ley de Lambert

La cantidad de irradiancia que produce un rayo al incidir sobre una determinada superficie es determinada por el ángulo con el cual se está produciendo dicha incidencia. Cuando un rayo de luz incide sobre una superficie este genera una cantidad de irradiancia, si la el ángulo formado por el rayo y la superficie es perpendicular, este generará más irradiancia que el generado por ejemplo por un ángulo más cerrado (*glazing angle*).

## 4.3 Fuentes de luz

Las fuentes de luz son capaces de emitir energía a distintas direcciones. Estas se pueden clasificar según distintas características. A continuación se va a realizar una breve descripción de estas en términos de gráficos por computador.

### 4.3.1 Luces direccionales:

Las luces producidas por objetos como el sol u objetos que se perciben como generadores de luz masivos son fácilmente representadas con una luz direccional. El método para llevar a cabo el efecto de una luz direccional sobre una escena consiste en mantener una dirección asociada a la luz y una energía generada. Este tipo de luces no representan gran complicación a la hora de ser implementadas, no obstante esto puede variar en función de la complejidad del modelo de iluminación que se haya elegido.

### 4.3.2 Luces puntuales:

Son luces que se representan en un punto del espacio desde el cual se genera luz en todas las direcciones. Este tipo de luces también son llamadas *omni lights*.

El vector dirección ( $l$ ) y la contribución de irradiación ( $E_L$ ) de este tipo de luces depende de la posición de la superficie sobre la que se esté calculando la intensidad de luz:

$$\begin{aligned} r &= \|\mathbf{p}_L - \mathbf{p}_S\|, \\ l &= \frac{\mathbf{p}_L - \mathbf{p}_S}{r}, \\ E_L &= \frac{I_L}{r^2} \end{aligned} \tag{4.1}$$

Donde  $\mathbf{p}_L$  es la posición de la luz y  $\mathbf{p}_S$  la posición de la superficie.

Típicamente este tipo de luz es simulada bajo un valor de atenuación que le dota de realismo y que hace que no se iluminen todos los objetos de una escena con la misma luz e intensidad que los demás. Para calcular el valor de atenuación se pueden usar diferentes funciones. Es común que distintos desarrolladores usen cada uno su función ya que simular físicamente estos valores puede llegar a ser más complejo.

#### Spotlight

Este tipo de luz genera irradiación en una dirección circular pero acotada por un ángulo que define la amplitud del cono que produce la misma. Históricamente, este tipo de luces se pueden calcular con funciones incorporadas en las APIS de dibujo OpenGL y DirectX.

La cantidad de irradiancia  $I_L$  es calculada con el vector dirección de la luz  $\mathbf{s}$  y el ángulo  $\theta_s$  de amplitud. Ya que el cálculo se realiza sobre la superficie de los objetos el vector  $\mathbf{l}$  se requiere así como también  $-\mathbf{l}$  vector dirección opuesto. En OpenGL el cálculo de la irradiancia es calculado mediante la

$$I_L = \begin{cases} I_{Lmax}(\cos \theta)^{s_{exp}}, & \text{donde } \theta_s \leq \theta_u, \\ 0, & \text{donde } \theta_s > \theta_u, \end{cases} \tag{4.2}$$

$$I_L = \begin{cases} I_{Lmax}(\cos \theta)^{s_{exp}}, & \text{donde } \theta_s \leq \theta_u, \\ 0, & \text{donde } \theta_s > \theta_u, \end{cases} \quad (4.2)$$

## 4.4 BRDF

Como se ha visto anteriormente, la emitancia radiante es generada por los cuerpos que son fuentes de luz. Esta llega en forma de irradiancia a las superficies de los objetos que quedemos dibujar. La manera en que dicha irradiancia es reflejada o tratada por la superficie es definida por lo que se denomina **Bidirectional Reflectance Distribution Function** (BRDF). Una BRDF es una función en términos de radiometría, de cuatro variables reales que definen como la luz es reflejada sobre una superficie opaca [12].

$$f(x, \hat{w}_{in}, \hat{w}_{out}, \lambda) \quad (4.3)$$

La función toma como parámetros a  $x$  como el punto donde se produce la incidencia, a los vectores unitarios  $\hat{w}_{in}$  y  $\hat{w}_{out}$  direcciones desde donde se emite y hacia donde se refleja el rayo respectivamente y a  $\lambda$  como la longitud de onda el rayo de luz en cuestión. Así este último parámetro recuerda que BRFD recibe cualquier tipo de longitud de onda, es decir, bien sea lo proveniente de fuentes de luz como la reflejada por otros objetos con superficies reflectantes.

Cuando una superficie obedece principios físicos (rendering avanzado) la cantidad de luz reflectada debe ser igual a la cantidad de luz entrante. De esta forma se establece un principio de reciprocidad entre la luz entrante y la luz saliente:

$$f(x, \hat{w}_1, \hat{w}_2, \lambda) = f(x, \hat{w}_2, \hat{w}_1, \lambda) \quad (4.4)$$

A este principio le llamamos **Reciprocidad de Helmholtz** [4].

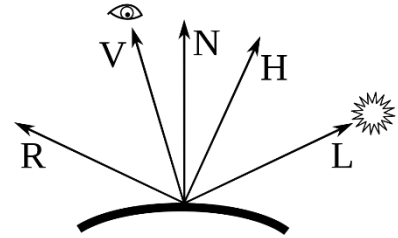
A continuación se describirán algunos modelos de BRDF.

#### 4.4.1 Modelo de Lambertiano:

Es aquel que representa superficies perfectamente difusas (mates) mediante una BRDF constante.

#### 4.4.2 Modelo de Phong

Se trata de un modelo empírico propuesto por Bui Tuong Phong en el año 1973. El modelo de Phong define la cantidad de luz saliente de una superficie como la suma de 3 componentes básicas: Componente ambiental, componente difusa y componente especular. La expresión para calcular la Irradiancia en el modelo de Phong tiene la forma de la ecuación (4.5).



**Ilustración 4.1.** Vectores utilizados para el cálculo del modelo de Phong (Fuente: Wikipedia 2015)

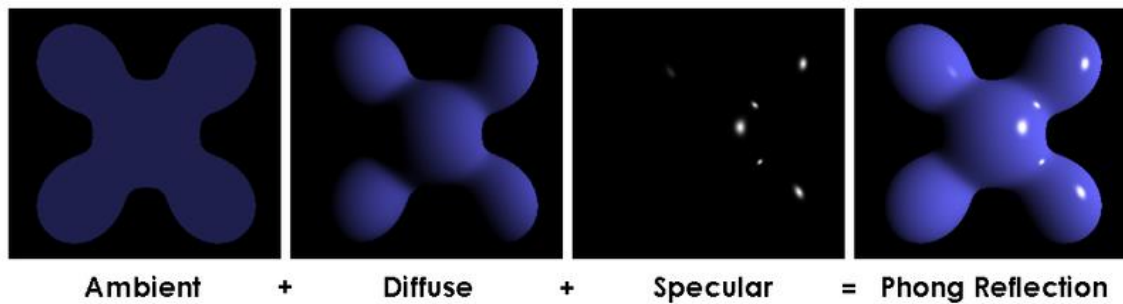
$$I_p = k_a i_a + \sum_{m \in \text{lights}} (k_d (\hat{L}_m \cdot \hat{N}) i_{m,d} + k_s (\hat{R}_m \cdot \hat{V})^\alpha i_{m,s}). \quad (4.5)$$

Componente ambiente es calculado como la suma de la contribución ambiental de luz de todas las luces de la escena ( $i_s$ ). Los componentes  $i_s$  y  $i_d$  son definidos como intensidades especulares y difusas de las fuentes de luz.

Los valores K son constantes definidas por el material, de modo que:  $k_s$  es la constante especular del material que regula la proporción de reflexión especular de la luz entrante,  $k_d$  es la constante difusa del material que regula la proporción de luz difusa de la luz entrante y  $k_a$  es la constante ambiental del material que regula la cantidad de luz presente en todos los lugares de la escena. Además el exponente  $\alpha$  es el término de brillo o *Shininess*. Este exponente es mayor para superficies esmaltadas y menos para superficies más opacas. En definitiva este exponente regula cuan condensado son los brillos especulares del

**Ilustración 4.2.** Suma de los componentes del modelos de Phong (Fuente: Wikipedia 2015: [https://en.wikipedia.org/wiki/Phong\\_reflection\\_model](https://en.wikipedia.org/wiki/Phong_reflection_model))

material. La Ilustración 4.2 muestra visualmente la suma de los componentes en el modelo de Phong.



#### 4.4.3 Modelo de Blinn-Phong

El modelo de Blinn-Phong es una optimización en términos de rendimiento del modelo de Phong, por lo tanto se basa en el mismo principio que dicho modelo. La principal peculiaridad del modelo de Blinn-Phong es que este utiliza el vector  $H$  para el cálculo del componente especular reemplazando el término  $R \cdot V$  el cual es un por  $N \cdot H$  donde  $N$  es el vector normal de la superficie y  $V$  es el vector que apunta hacia la posición del observador. La mejora radica en que de esta forma no hace falta calcular el vector reflexión  $R = 2(L \cdot N)N - L$  lo cual es una operación costosa, en cambio, se calcula el vector  $H = \frac{L+V}{|L+V|}$  que es el vector normalizado bisectriz entre  $L$  y  $V$  [11].

Este modelo es el elegido por su fácil implementación y por su mencionado rendimiento para satisfacer las necesidades de este proyecto. Así pues, el modelo de Blinn-Phong fue el implementado aquí, en combinación del modelo de Shading de Phong.

#### 4.4.4 Modelo de Cook-Torrance:

Se trata de un modelo más cercano a la realidad física que el modelo de Phong, Es usado para simular la reflectancia especular de distintos materiales. Se trata de un modelo que simula mejor ciertos tipos de materiales del mundo real como son los metales y los materiales con altos índices de reflectividad. El modelo trata cada superficie como un conjunto de muchas microfacetas, es decir, pequeñas facetas que reflejan la luz entrante. En superficies ásperas, las pendientes de estas microfacetas varían considerablemente, mientras que en superficies lisas las microfacetas están orientadas en una dirección similar.

Este modelo tiene en cuenta fenómenos naturales y muy presentes en la realidad como el factor Fresnel que es un tipo de reflexión que varía en función del ángulo de visión con de la superficie [8].

## 4.5 The Rendering Equation

La ecuación de renderizado es una integral en la cual el equilibrio de radiancia saliente de un punto dado es la suma de la radiancia emitida y la radiancia reflejada bajo una aproximación geométrica de la óptica (rol del perceptor de la escena) [16].

Si bien la BRFD nos permite calcular la radiancia saliente de un punto dado en una desde una dirección  $\hat{w}_{in}$  hacia una dirección  $\hat{w}_{out}$ , la ecuación de renderizado realiza la suma de dicha radiancia más la cantidad de radiancia que es reflejada a cualquier luz que es emitida desde la superficie hacia nuestra dirección.

La forma de la ecuación de renderizado es la siguiente:

$$L_o(\mathbf{x}, \omega_o, \lambda, t) = L_e(\mathbf{x}, \omega_o, \lambda, t) + \int_{\Omega} f_r(\mathbf{x}, \omega_i, \omega_o, \lambda, t) L_i(\mathbf{x}, \omega_i, \lambda, t) (\omega_i \cdot \mathbf{n}) d\omega_i \quad (4.6)$$

Para resolver la ecuación de renderizado hacer falta tener en cuenta un tipo de iluminación mucho más compleja y que aporta mucho más realismo al dibujado la cual es la iluminación global. Si bien la BRFD puede definir la cantidad de radiancia local, que es reflejada en una dirección la cantidad de radiancia es una escena con varios objetos suele ser el resultado de varias radiancias entrantes y salientes producto de las distintas reflexiones que se suceden en la escena. En este se hace insuficiente calcular solo la porción de radiancia que proporciona la BRFD.

Muchos sistemas de renderizado más complejos y realistas intentan resolver esta ecuación y llevar un dibujado más técnico cuantificando de forma real la cantidad total de energía que hay en la escena.

Dado que esta ecuación toca aspectos que este trabajo no puede cubrir, no se hará más énfasis en el tema, pero queda nombrado como algo a considerar si se desea desarrollar sistemas más complejos en un futuro no muy lejano.

## 4.6 Material

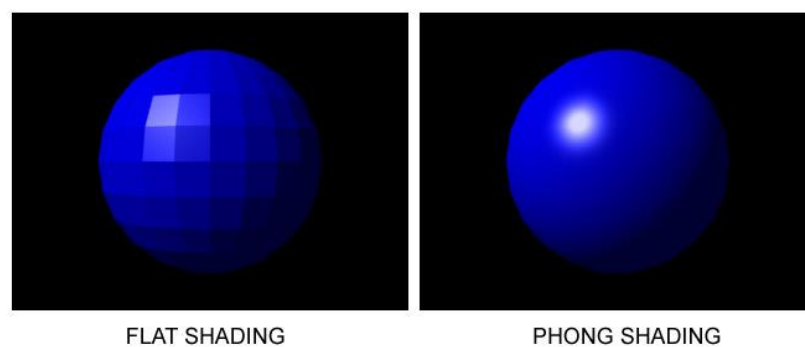
Un material define un conjunto de propiedades que caracterizan y definen un aspecto determinado. Un material es el encargado en generar los datos que luego son transmitidos a la BRFD para computar la cantidad de radiancia de la superficie y posteriormente realizar el cálculo del color de la superficie. La forma en que están modelados los materiales varía en función del modelo de iluminación que se quiera satisfacer. En el caso de este proyecto los materiales están pensados para ser suministrados al modelo de reflexión de Phong. Más adelante se hablará del diseño de los materiales para este proyecto.

## 4.7 Shading

Shading es el proceso de usar una función BRFD para computar la radiancia saliente  $L_o$  a lo largo del rayo de visión,  $\mathbf{v}$ , basándose en las propiedades del material y de la fuente de luz. Es decir, shading es proceso para darle un color a un objeto basándose en una BRFD y en unas propiedades de materiales.

Existen modelos de shading más comunes que otros, por ejemplo, el modelo usado por este proyecto, el Modelo de Blinn-Phong, que se trata de una versión modificada y optimizada del modelo original de Phong, el cual usa método de interpolación de normales entre fragmentos, en combinación con el modelo de reflexión de Phong (BRFD). Hablar de Shading y de BRFD no es lo mismo, si bien un BRFD propone matemáticamente un solución para determinados tipos de superficies, el proceso de Shading es el definido como el de obtener un color para un determinado fragmento basándose en una BRFD determinada. También es común confundir el modelo de reflexión de Phong (BRFD) con el Shading de Phong. Si bien el modelo de reflexión de Phong define como se ha de calcular una determinada irradiancia sobre una superficie, el Shading de Phong no es más que una

técnica ligada a las condiciones de los gráficos por computador y más a los modelos 3D, en donde mediante interpolación se obtiene valores para la normal en puntos donde no se encuentra un vértice geométrico (los cuales son los únicos que almacenan la información de la normal) y así obtener resultados más continuos visualmente. En contraste a este tipo de Shading se encuentra el Flat Shading (plano) donde la normal es igual para todo un polígono y el resultado es un objeto facetado, o el Shader de Gouraud donde no se interpola la normal pero sí se calcula el color a nivel de Fragment y donde se obtienen resultados menos facetados aparentemente, pero evidentemente facetados para modelos con bajos polígonos y en zonas donde la luz especular está presente. Como ha de esperarse, los modelos con mejor resultado visual son mucho más costosos en términos de tiempo de computación y suelen requerir de un procesamiento extra en la etapa de Fragment Shader (etapa en el proceso de dibujado donde se calcula el color del pixel a partir de un pequeño programa – *Fragment Shader*).



**Ilustración 4.3.** Comparación entre Flat Shading y Phong Shading (**Fuente:** Wikipedia 2015)

## 4.8 Dibujando en 3D.

Hoy en día las técnicas para obtener una imagen de objetos en 3d están prácticamente depuradas y extendidas, incluso podríamos hablar de estándares. Esta práctica, en gran medida, y como muchas otras cosas en el campo de la informática, ha crecido basándose en las propuestas de personas o grupo de personas que han sabido encarar de la mejor forma posible el problema en cuestión. Resultado de esto, hoy en día tenemos una forma bastante competente de hacer las cosas para conseguir pintar objetos en 3D. Paralelamente, y muy importante, ha sido el rol que los fabricantes de hardware para gráficos en 3d han tenido



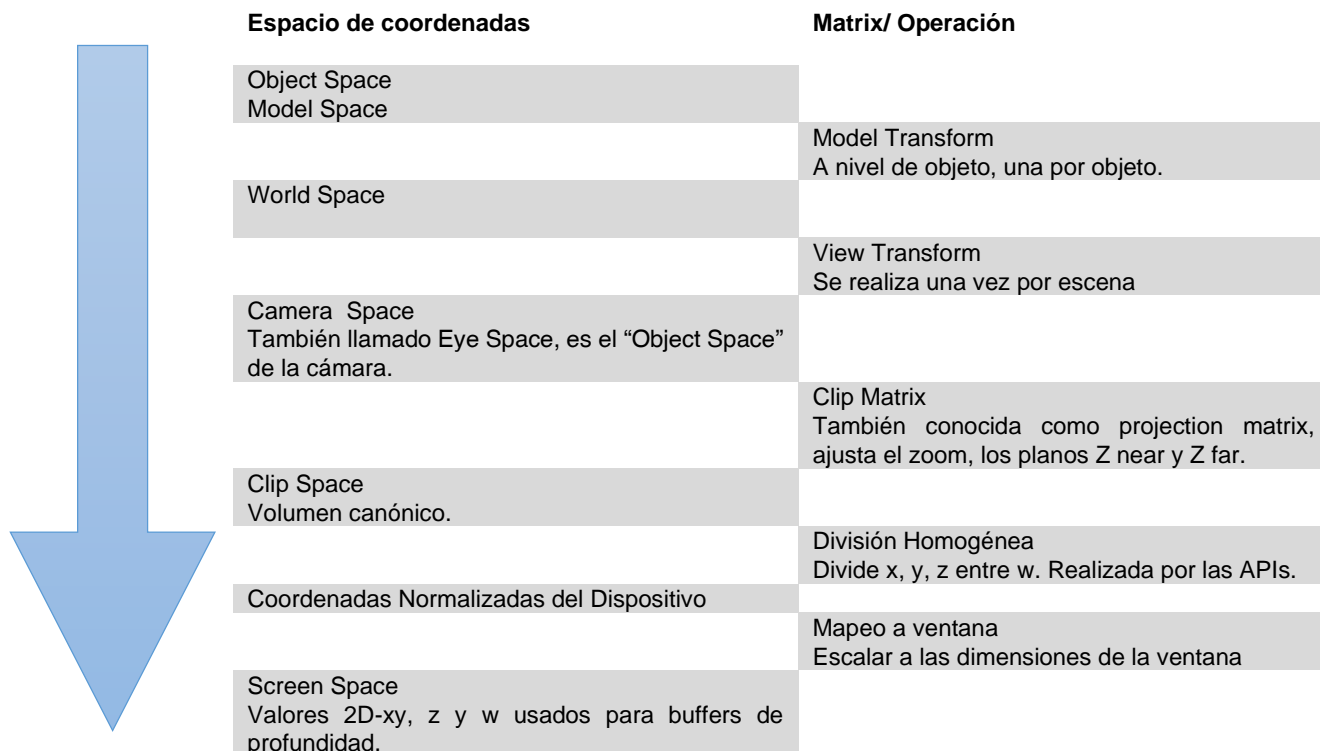
en esta evolución y bien, en gran medida, a cómo la tecnología software y hardware se han sabido complementar para crear arquitecturas bien definidas y altamente extendidas.

Pintar objetos en 3d hoy en día (sin tener en cuenta modelos de iluminación) es la una combinación de varios factores. En primer lugar deben existir dispositivos para el dibujado y para los cálculos que se realizan hasta conseguir una imagen plana. Dadas estas dos premisas hoy en día nos encontramos con un elemento fundamental: las APIs de dibujado. OpenGL y DirectX son un conjunto de funciones a bajo nivel que permiten a los programadores el poder abrir contextos visuales y crear imágenes en 3D de objetos, entre otras cosas. OpenGL es un proyecto open source multiplataforma, mientras que DirectX es un software propietario y es desarrollado por la compañía Microsoft e incorporado en dispositivos con sistema operativo Windows. Ambas son herramientas que sirven para dibujar. Dicho esto, estas APIs están altamente cohesionadas con las arquitecturas hardware actuales. Su forma de actuar es razonablemente entendible si se mira desde el punto de vista hardware y se atiende a la premisa de que quien logra altos índices de velocidad de computación – gracias a su arquitectura paralela SIMD o SIMT (NVidia) – son las tarjetas gráficas actuales.

Aprovechando las instrucciones en paralelo de estas tarjetas podemos por ejemplo, realizar el cálculo de millones de píxeles de forma eficaz y así determinar el color que le corresponde dadas ciertas reglas antes comentadas como la luz, el material etc.

Aunque sean dos APIs distintas ambas manejan relativamente el mismo “protocolo” para dibujar en pantalla. A este protocolo se le denomina pipeline (tubería). Tanto el pipeline de OpenGL como el DirectX son similares es varias etapas, incluso en ocasiones usando términos distintos para denominar cosas esencialmente similares (algo nada nuevo). OpenGL es la API usada para el desarrollo de este proyecto.

A continuación se hará una breve descripción del proceso llevado a cabo para dibujar mallas de vértices en pantalla. Esto es así porque se ha considerado que este proceso determinado no representa un hito a superar en este proyecto ya que resulta algo trivial en el dibujado 3D. Además en cualquier fuente de información relacionada con el tema de gráficos se presentan de una mejor y más detallada manera las explicaciones referentes a este tópico.



**Tabla 4.2.** Conversión de coordenadas de vértices a través del graphic-pipeline. A la izquierda el sistema de coordenadas y a la derecha la matriz que se aplica para convertir a los sistemas. **Fuente:** (Fletcher & Parberry, 2011)

Mediante una valiosa herramienta, el álgebra lineal, se tiene la capacidad de generar imágenes planas haciendo uso de distintas transformaciones geométricas. Estas transformaciones constituyen una forma de conseguir proyectar diferentes objetos sobre un plano abstracto como lo es una pantalla o monitor. Para llegar a esto, primero se han de pasar por diferentes etapas y ahí es donde los pipelines de cada API entran en juego, dando la posibilidad de controlar lo que pasa en cada etapa. Las transformaciones básicas como la rotación, el escalado, la translación, la proyección etc. son llevadas a cabo mediante operaciones matemáticas basadas en matrices. De esta forma podemos aplicar diferentes transformaciones a, p. ej., un vértices de una malla 3D simplemente haciendo uso de una matriz preparada para dicha transformación.

### 4.8.1 OpenGL pipeline

OpenGL tiene un pipeline denominado OpenGL Pipeline el cual posee a grandes rasgos las mismas etapas que proceso descrito anteriormente. Tanto DirectX como OpenGL dan la libertad al desarrollador de controlar determinadas fases. Dichas fases son implementadas en términos de programación por pequeños programas (*Shaders*) escritos en un lenguaje de programación diseñado para este tipo de tareas. El lenguaje de programación de OpenGL es llamado GLSL (OpenGL Shading Language). Las etapas programables en el pipeline de OpenGL son:

**Vertex Shader:** Se procesan operaciones básicas sobre cada vértice, estas operaciones suelen llevar los vértices a Clip Space mediante transformaciones y también existe la posibilidad de que el usuario genere información de salida que puede ser recibida en posteriores etapas como la dirección de las luces, o las normales.

**Tessellation:** Las primitivas pueden ser teseladas usando dos Shader, Tessellation Control Shader (TCS) que define la cantidad de teselación y Tessellation Evaluation Shader (TES) que se encarga de la interpolación y de otras operaciones definidas por el usuario.

**Geometría Shader:** Esta fase recibe un número determinado de primitivas y puede devolver cero (descartar vértices) o más primitivas. El Shader que controla esta fase es llamado Geometry Shader.

**Fragment Processing:** Los datos de cada fragmento en el estado de rasterización son procesados por el Fragment Shader. La salida de esta etapa da como resultado una lista de colores que se almacenan en el *FrameBuffer*, en el *DepthBuffer* (valores de profundidad) y en el *StencilBuffer* (el stencil value).

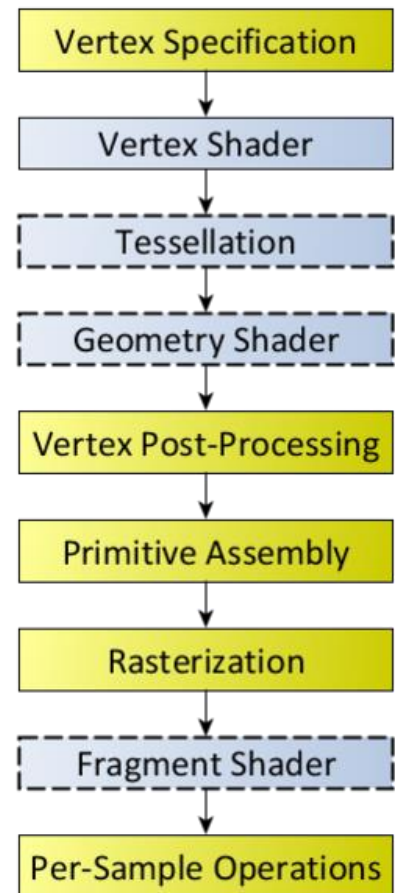


Ilustración 4.4. OpenGL Pipeline, los cuadros con líneas discontinuas representan etapas opcionales (Fuente: Wikipedia)

## 4.9 Texturas

Las texturas son una poderosa herramienta en el campo de los gráficos, no solo para ser aplicadas sobre los objetos, un uso más común, sino también para realizar tareas más diversas como cálculos de iluminación basada en mapas de luz (texturas con información para el cálculo de luz) o el cálculo de reflexiones con cube maps. Algunos mapas de texturas son creadas de antemano por programas de edición de imágenes como Photoshop mientras que otros son generados mediante procedimientos en tiempo de carga o de edición (precalculados). Ejemplos de esto último son los cube maps generados automáticamente para el cálculo de reflexiones donde el color de la radiancia entrante es una porción calculada proporcionada por el cube map, de esta forma se pueden conseguir efectos como las reflexiones propias de un espejo, o las de un material altamente reflectivo.



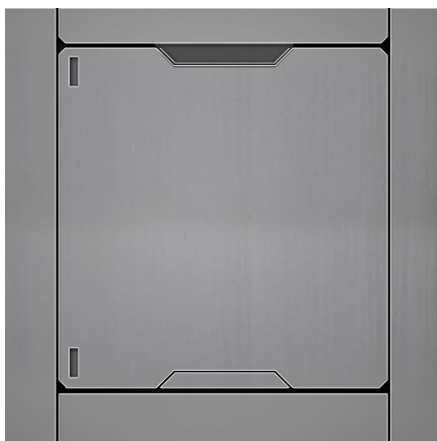
**Ilustración 4.5.** Reflexiones calculadas con environment cube maps.

(Fuente: <https://developer.valvesoftware.com/wiki/Cubemaps>)

No obstante, el uso de mapas de texturas que más interesa para la finalidad de este proyecto son las que proporcionan información para establecer parámetros de los materiales. Algunos ejemplos de estas texturas son los mapas de textura difusos, maps de normales, mapas especulares, mapas de valores de transparencia (alpha maps), e incluso Shininess maps.

### 4.9.1 Mapas de textura difusos.

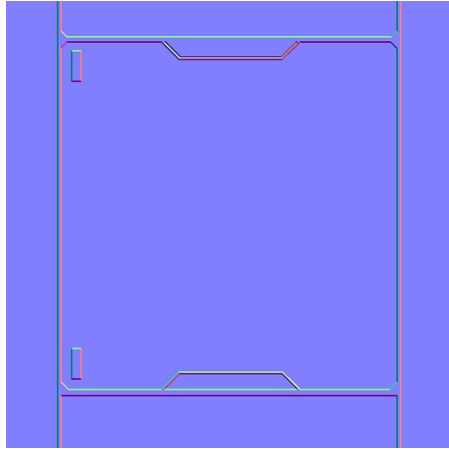
Estos mapas contienen la información del color difuso del material, en ocasiones los artistas suelen incluir valores de iluminación en este tipo de materiales (sombreado artificial o *Ambient Occlusion*). La Ilustración 4.6 muestra un ejemplo de este tipo de mapas.



**Ilustración 4.6.** Ejemplo de mapa de textura difusa. (**Fuente:** Elaboración propia)

#### **4.9.2 Mapas de textura normales.**

La técnica consiste en almacenar la dirección de la normal en una imagen. Posteriormente el valor mapeado es leído para cada Fragment y ese valor de la normal es usado para el cálculo del color con el modelo de shading que corresponda. Una particularidad de esta técnica es que los valores almacenados en la textura están en términos del espacio tangencial (tangent space. Sistema de coordenadas donde sus ejes son definidos por los vectores locales normal, binormal y tangente). Es por este motivo que los cálculos para obtener el color son realizados en dicho espacio de coordenadas, esto obliga a que los vectores dirección de la luz, dirección del observador y todo aquellos que entren en juego a la hora de realizar el cálculo estén en espacio tangente. Para llevar un objeto a espacio tangente lo que se suele realizar es componer una matriz TBN (Matriz Tangent-Binormal-Normal) calculada a partir de las componentes normal, tangencial y binormal de los vértices y posteriormente realizar la multiplicación con esta matriz para hacer el cambio de coordenadas. La Ilustración 4.7 muestra un ejemplo de este tipo de mapas.



**Ilustración 4.7.** Ejemplo de mapa de textura normal. (**Fuente:** Elaboración propia)

En definitiva, si se quiere usar esta técnica, los cálculos de la iluminación han de hacerse en espacio tangencial ya que es la forma en la que los estos mapas guardan la información de la normal.

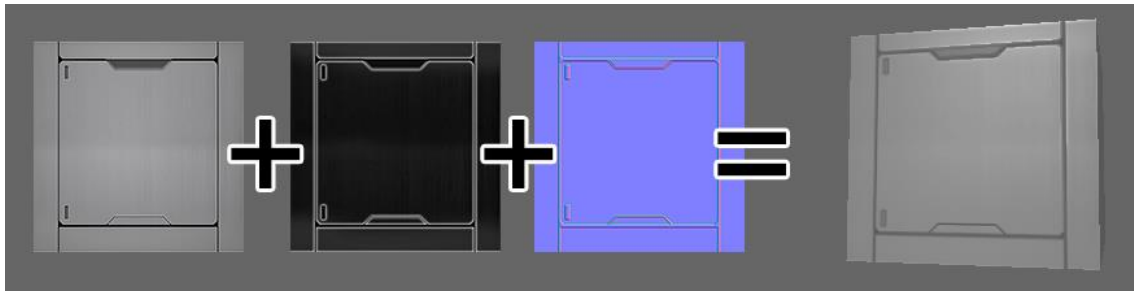
### 4.9.3 Mapas de texturas Especulares

Este tipo de mapa almacena la intensidad del valor especular del material. Este mapa suele estar en escala de grises, aunque existe la posibilidad que cuente con color lo que produciría brillos de color. La Ilustración 4.8 muestra un ejemplo de este tipo de mapas.



**Ilustración 4.8.** Ejemplo de mapa de textura especular. (**Fuente:** Elaboración propia)

Si asignamos los mapas correspondientes al modelo difuso, especular y ambiental con perturbación de las normales obtenemos un resultado como el siguiente:



**Ilustración 4.9.** Uso de mapas de textura difusa, especular y normal para el cálculo del modelo de Blinn-Phong. (**Fuente:** Elaboración propia).

## 4.10 Physically Based Rendering.

Actualmente existe una tendencia por la cual está pasando la industria de los videojuegos. La mayoría de grandes empresas desarrolladoras empezaron a darle una nueva aproximación a sus sistemas de renderizado, o por lo menos, un más enfoque mucho más científico. Propiciado por los avances en temas de hardware, se han podido modelar sistemas mucho más complejos y realistas que los antiguos. Así pues se ha empezado a popularizar el termino **Physically Based Rendering (PBR)**.

El PBR no es algo nuevo, en el mundo del renderizado off-line ha estado presente desde hace mucho tiempo. No obstante, es un tópico nuevo para renderizado en tiempo real ya que hasta hace muy poco no se podían realizar gastos computacionales considerables sin afectar el *frame rate* de dibujado, y es ahora cuando al parecer, se han podido generar nuevas técnicas (aun distantes a las del renderizado off-line) que permiten de cierta forma modelar sistemas de renderizado donde se conservan magnitudes físicas como lo son la cantidad de energía radiante que se encuentra en una escena. El avance del hardware propicia muchos de los avances en la materia.

A día de hoy no existe un estándar de PBR, inclusive cada desarrollador está llevando los conceptos como bien considera. No obstante en conferencias de renombre como SIGGRAPH (Special Interest Group on GRAPHics and Interactive Techniques) se exponen siempre enunciados y trabajos con el ánimo de reflejar un criterio unificado y conseguir así hacer del PBR un estándar único. Cada quien intenta realizar el sistema más competitivo posible, no obstante, empiezan a surgir determinadas similitudes y conceptos comunes. Un ejemplo de la necesidad de estas semejanzas radica en hechos como que los artistas que

trabajan en diferentes proyectos posean un método de trabajo que igual, sea cual sea el motor al que se destinará determinado Asset (un objeto o recursos de un proyecto de tipo videojuego). De no ser así, el coste de tener artistas entrenados para determinado motor podría ocasionarles problemas.

#### **4.10.1 Características de un sistema PBR.**

Un sistema de PBR tiene como objetivo principal crear un dibujado fotorrealista tan preciso como sea posible. En PBR un objeto no puede reflejar más energía de la que recibe (conservación de la energía). A continuación se presentan algunos parámetros que pueden alimentar un PBR y sus diferencias con términos tradicionales.

El término “difuso” es sustituido por un término más preciso y fiel llamado Albedo. Un mapa de albedo define el color de luz difusa. Una de las grandes diferencias entre un mapa Albedo y un tradicional mapa difuso es la falta de luz direccional o “ambient occlusion” (AO. Técnica usada para aproximar un efecto de iluminación ambiental). Esto es debido a que bajo ciertas condiciones la luz direccional que incluye el AO resultan incorrectas. Por eso los valores de AO deben ser añadidos como nuevo parámetro de material y no combinado en el canal difuso como se hacía tradicionalmente.

**Microsurface** (micro superficie) define cuan áspero es la superficie. A este punto es cuando los principios de la conservación son afectados por la micro superficie del material. Superficies más ásperas mostraran reflexiones especulares más amplias mientras que superficies más lisas mostraran brillos especulares más definidos asemejándose al aspecto de un espejo.

**Reflectividad** es el porcentaje de luz que una superficie refleja. Todos los tipos de reflectividad están incluidos en este parámetro: especular, metalness (metalicidad), etc. La reflectividad define como de reflectivo es el material cuando es observado de frente, mientras que el parámetro Fresnel define como de reflectiva es la superficie en ángulos cerrados (glazing angles).

Algunos motores gráficos como UE4 nombran este parámetro como Metalness.



**Fresnel** define la cantidad de reflectividad de la superficie en ángulos cerrados. La mayoría de materiales debería establecer este parámetro a 1, pues la mayoría de objetos tienen reflectividad del 100% en ángulos cerrados.

Otros mapas como mapas de Ambient Occlusion y de Cavity (cavidad) complementan el modelo.

Una de las grandes ventajas de trabajar con PBR es la que la definición de un material resulta mucho más intuitiva para los artistas, viéndose favorecido su flujo de trabajo de forma considerable [9].

## **4.11 Estado del arte**

### **4.11.1 Editor de materiales**

Un editor de materiales es una interfaz que permite de alguna forma editar y suministrar valores para los distintos parámetros de los modelos de iluminación que se esté empleando. En la mayoría de enfoques un material es una representación abstracta que se hace visualmente evidente cuando es aplicada a una malla.

El término “editor de materiales” es un concepto bastante extendido en la actualidad en contextos de gráficos por computador y en motores gráficos. Si bien existen gran variedad de herramientas que permite suministrar los parámetros que caracterizan un determinado material, cabe destacar algunas características que son diferenciadoras entre unos y otros editores de materiales.

### **4.11.2 Altamente integrado con el motor gráfico VS Independientes.**

En determinados entornos, los editores de materiales son parte de un sistema mucho más complejo y funcionan como sub-sistema dentro de un motor gráfico o sistema más generales. Dichos editores funcionan de forma complementaria al sistema de renderizado de sus motores, estableciendo integridad en la parametrización de valores de cara a su ecuación de renderizado a varios niveles de complejidad. En otras palabras, existe una alta cohesión entre lo ofrecido por el editor de materiales y su puesta en escena por parte del

motor, donde otros procesos entran en juego como la iluminación global, el sombreado de la escena y como otros efectos más avanzados (desenfoque de movimiento basado en mapeado de movimientos, la iluminación volumétrica etc.). Todos ellos hacen parte del dibujado final de la escena pero no así de las propiedades físicas del material.

Un ejemplo claro de este tipo de herramientas son las ya incorporadas en motores comerciales de alto desempeño como el motor gráfico Unreal Engine 4 (UE4) de Epic Games Inc. o el motor gráfico CryEngine de Crytek.

Por otra parte, si bien hemos dicho que existen editores pensados para un determinado motor de renderizado en tiempo real, también se da el caso de editores de materiales que por el contrario no están diseñados para un sistema de renderizado específico, manejando conceptos y parámetros más genéricos y que se aproximan más al “estándar” más extendidos de gráficos por computador permitiendo dibujar los materiales producidos bajo distintos motores de renderizado. Un caso claro de esto son los editores de materiales incluidos en las aplicaciones de edición 3d. En estas es común realizar el trabajo de edición de los materiales y luego elegir el motor de renderizado. Un buen ejemplo se da en el programa 3ds Max, donde podemos elegir por defecto entre el motor de renderizado *Scanline* o el motor de renderizado *MentalRay*. Además existen motores de terceros como el motor *V-Ray* que pueden ser añadidos y usarse para el renderizado todo dentro de 3ds Max.

Llegados a este punto cabe puntualizar que aunque estos editores manejen parámetros más generales, también pueden disponer de otros más avanzados en función del motor de renderizado que se va a utilizar, pero nunca abandonando los valores estándar. En el caso de 3ds Max, es frecuente ver como el editor de materiales se ve modificado añadiéndole nuevas interfaces que permiten el poder trabajar con dicho motor gráfico y sus parámetros específicos.

#### **4.11.3 Para renderizado en tiempo real VS renderizado offline.**

Una importante característica más a destacar, es también el hecho de que el destino de algunos de estos editores sean sistemas de renderizado en tiempo real o bien de renderizado offline. Esto puede ser importante ya que ambos sistemas difieren

sustancialmente en aspectos clave como la calidad del resultado o el tiempo de respuesta final.

Un claro ejemplo de esto se da en los parámetros básicos como de factor de especularidad de una superficie, o su valor de componente difusa. Si bien la forma en que un editor para renderizado en tiempo real los manejan respecto a otro offline puede resultar similar, a la hora de realizar el dibujado final, sus procesos internos son radicalmente distintos, ya que en la mayoría de los casos, un sistema offline usa métodos basados en *Raytracing*, los cuales son inviables para renderizado en tiempo real debido a su gran costo computacional.

#### **4.11.4 Tipo de interfaz: Basada en Nodos VS Clásica.**

Una tendencia que se está empezando a extender bastante es la de pasar del clásico editor de materiales con casillas de texto y campos para las texturas, a un nuevo enfoque mucho más diverso y con más potencial. Se trata de las interfaces basadas en nodos. El principio básico en la mayoría de casos, es el aprovechamiento de operaciones realizadas sobre los colores directamente, en combinación con las texturas y las fuentes de datos como los vectores, creando conexiones entre nodos y otros nodos que representan operaciones. En este punto, el editor de materiales hace referencia más a un creador de Fragment Shaders (de alguna forma) que de simplemente asignar unos valores a unas entradas. La potencia que proporciona es notable. Constituye un sistema VPL (Visual programming Language) que se aprovecha de las bondades de estos sistemas como la capacidad de manipular elementos gráficamente para generar programas, en este caso, para generar un Shader. Los nodos en combinación con operaciones generan nuevos resultados y al final lo que se obtiene es una pieza de código capaz de definir un material.

A continuación se hará un análisis de los editores de materiales que más han influido este trabajo y sus principales características.

#### 4.11.5 Editor de materiales de UE4.

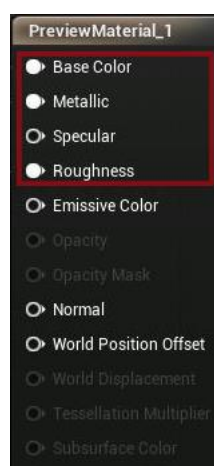
Este editor permite la creación de Shaders para aplicarlos sobre alguna superficie geométrica determinada. Dicho editor viene incorporado en el motor gráfico UE4 y es parte de uno de sus subsistemas.

Su interfaz gráfica basada en nodos, es una de sus características más destacables. Es frecuente en UE4 ver dicho tipo de interfaces basada en nodos.

El editor de materiales de UE4 permite la creación de una variedad considerable de nodos que bien representan operaciones las cuales encapsulan una función determinada, o bien nodos que representan algún tipo de información como valores de color, o de mapas de texturas. Cada nodo está provisto de entradas y salidas a las cuales se les puede conectar otros nodos ya bien sea como entradas o como salidas. El número de entradas depende del tipo de nodo.

Este tipo de interfaz es común en otros subsistemas del UE4 como lo son los *Blueprints*, en donde la finalidad no es crear materiales sino modificar o crear características del gameplay, cámaras y otros tipos de elementos de una forma más intuitiva y rápida, más aun para gente que no está especializada en programación.

UE4 utiliza un modelo de shading PBR. UE4 define un material basándose en 4 propiedades básicas: BaseColor, Metallic, Roughness y Specular.



Nodo principal del panel de grafo en UE4. Por defecto este nodo está siempre presente. A él son conectados los inputs creados en el editor para generar el resultado final.

La propiedad **BaseColor** define un color total para el material del objeto. Recibe como parámetro de entrada un vector de tres dimensiones donde cada componente representa los valores de intensidad para el estándar de color RGB. Estos valores son restringidos al rango [0-1].

Por su parte, la propiedad **Roughness** controla el nivel de aspereza del material. Un material con un alto nivel Roughness refleja luz en más direcciones que uno con poco nivel dejando una luz especular más difusa y borrosa, alternativamente un material con un nivel de Roughness más bajo refleja luz parcialmente en una misma dirección y tiene una luz especular mucho más definida tendiendo a imitar el comportamiento de un espejo.

La propiedad **Metallic** controla el nivel de metalizado de la superficie. Fenómenos como la cantidad de luz que es absorbida por el material son controladas por este parámetro contribuyendo a definir el nivel de metalizado del material. Un material con un valor de 1 se comporta como totalmente como un metal mientras que un material con un valor de 0 no poseería ninguna característica metálica (rocas, madera, etc.).

La propiedad **Specular** recibe un valor entre 0 y 1 y se utiliza para escalar la cantidad actual de especularidad en superficies no metálicas. Es decir, no tiene ningún efecto sobre los metales. Por defecto, si no se le pasa un valor, el valor inicial es de 0.5.

Si se modifica el valor de especularidad, se hace para añadir micro oclusiones o sombreado a pequeña escala, representadas en el mapa de normales (mapa de cavidades).

Geometría a pequeña escala, especialmente detalles solo presentes en el modelos *high poly* y luego generados (*baked*) sobre el mapa de normales, no son recogidas por las sombras del renderizador en tiempo real. Para capturar estas sombras, el motor UE4, genera un mapa de cavidades (*Cavity*), el cual es típicamente un mapa AO (*Ambient Occlusion*) con poca distancia de trazado. Este mapa es multiplicado por el *BaseColor* y posteriormente multiplicado por 0.5 (especularidad por defecto) como la salida final de la propiedad **Specular**. Es decir:  $BaseColor = Cavity * OldBaseColor$ ,  $Specular = Cavity * 0.5$

#### 4.11.6 Editor de materiales 3ds Max

El editor de materiales de 3ds Max ofrece dos versiones en su interfaz de usuario: La versión compacta y la versión de pizarra (Slate editor). Ambas opciones producen la misma calidad de resultados en términos técnicos. No obstante la diferencia entre ambos es grande.

##### **Versión compacta:**

La versión compacta es representada en una pequeña ventana modal, invocada desde la interfaz principal. Los materiales generados se ven en la parte superior de la ventana en forma de grid. Por defecto 3ds Max ofrece una serie de materiales a partir de los cuales se pueden empezar a alterar propiedades. Sin embargo el más típicamente usado en términos generales es el llamado “Standard”. Dicho perfil de material posee las propiedades más típicamente usadas y entendidas por gráficos por computador.

La interfaz gráfica de 3ds Max, en su mayoría, está estructurada por **rollouts** (grupos de controles de interfaz de usuario). El modelo estándar tiene como principales rollouts a:

- **Shader basic Parameters:** Aquí se puede definir el mediante qué función de reflexión se ha de interpretar el material. Entre sus opciones más destacadas se encuentran: Blinn, Phong, Metal etc.
- **Basic Parameters:** Sus opciones cambian según el modelo de reflexión elegido en el rollout anterior. Por lo general presenta los valores más típicos del material:

**Ambient:** Constante de color que intenta simular la contribución lumínica de las reflexiones globales de la escena. Recibe un color rgb como parámetro de entrada.

**Diffuse:** Se trata del color base del material. La cantidad de luz que es reflejada en direcciones uniformes. Recibe un color rgb como parámetro de entrada.

**Specular (Color):** Define el color con el que las luces especulares son interpretadas. Recibe un color rgb como parámetro de entrada.

**Specular Level:** Indica el nivel de especularidad del material. Un material con un valor especular igual a 0 no reflejara luces especulares hacia la cámara, mientras que un material

con un valor especular mucho más alto tendrá mucha riqueza de dichas luces en su superficie.

**Glossiness:** Define el la fuerza de las luces especulares. De ello depende cuán grande son las luces especulares que son reflejadas. Si el valor es muy bajo, se producen luces más grandes con gradientes mayores. Si el valor es alto, la luz especular se decrementa al igual que lo hace el gradiente, dejando luces mucho más definidas y puntuales.

**Maps:** En este rollout están presentes los slots a los cuales se pueden introducir distintos tipos de mapas para dotar al material de diferentes características. Por lo general, a cada mapa le es asignado un componente de tipo Bitmap, pues es una acción típica el importar mapas para diferentes acabados en ficheros de tipo imagen y que son generados, generalmente, en otro tipo de programas como Photoshop. Sin embargo a dichos slots se les puede asignar otro tipo de componente. 3ds Max ofrece gran variedad de mapas, algunos por ejemplo, procedurales capaces de generar mapas de ruido aleatoriamente como el componente “Noise”, u otros capaces de generar gradientes como el componente “Gradient”. Por lo tanto, estos valores no tienen por qué estar enteramente ligados a un mapa imagen, aunque suele ser lo más frecuente en el flujo de trabajo de la industria de gráficos en tiempo real como los videojuegos.

Entre los canales de mapas más utilizados se encuentran:

- **Diffuse Color:** Recibe un mapa (por lo general una textura), que aporta el color base a la superficie del material. Los valores de coordenadas UV también están a disposición del usuario para su asignación.
- **Specular Level:** Recibe un mapa de valores especulares. Este tipo de mapa, típicamente está en escala de grises y controla el comportamiento del índice especular a lo largo de todo el modelo.
- **Opacity:** Recibe un mapa de opacidad donde el valor de los pixeles define la visibilidad de la superficie.
- **Bump:** Recibe un mapa que, típicamente está en escala de grises y que define micro detalles y rugosidades a lo largo del material. Para ello, el mapa es capaz de modificar la dirección de las normales encapsulando valores

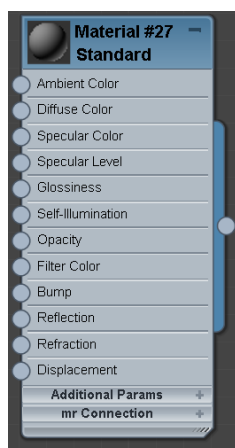
numéricos sobre cada pixel, normalmente, en espacio de coordenadas tangencial.

### **Versión Slate.**

Esta versión es una aproximación más cercana a la propuesta en este proyecto en su interfaz gráfica.

Su interfaz de usuario deja de ser la convencional interfaz compacta para pasar a ser una más actualizada respecto a las tendencias actuales de edición de materiales. Dicha interfaz está basada en nodos, lo cual la hace mucho más adecuada a las demandas y flujos de trabajo actuales de este tipo de edición.

El espacio de trabajo tiene como componente principal un área donde se pueden crear nuevos materiales bien sea arrastrándolos de una lista de perfiles o haciendo clic derecho sobre dicha área y eligiendo el perfil del material a iniciar. En ese momento se crea un nodo (Ilustración 4.10), este nodo principal contiene una lista de inputs a los cuales se les pueden conectar nuevos nodos. Los inputs representan las propiedades expuestas para dicho material en dicho perfil.

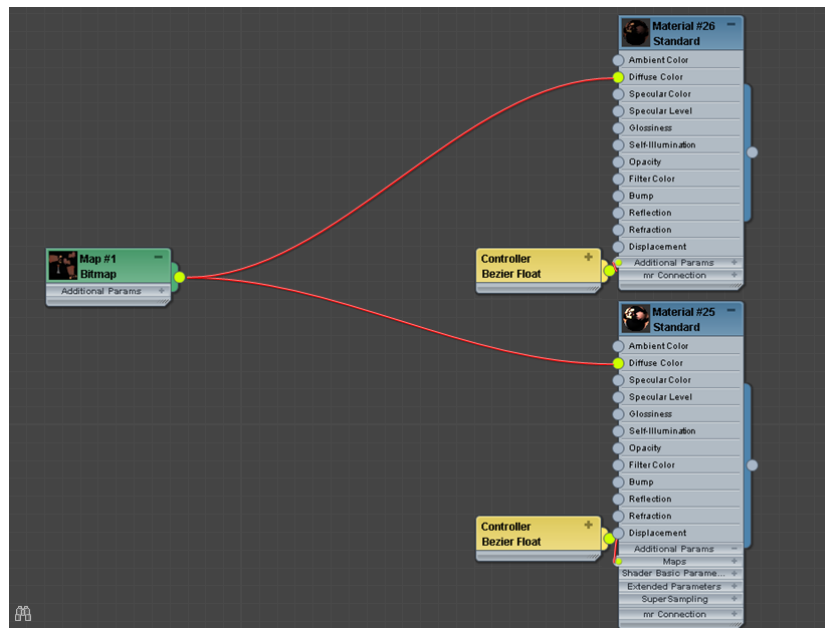


**Ilustración 4.10.** Nodo principal correspondiente a un material de tipo Standard. (**Fuente:** Captura desde 3ds Max 2012)

En el mismo espacio de trabajo se pueden generar más materiales permitiendo al usuario reutilizar nodos o resultados de nodos entre distintos materiales que conviven en el espacio



de trabajo. Lo cual ofrece mucha más potencia a la herramienta al igual que escalabilidad. La Ilustración 4.11 muestra un ejemplo de lo anterior.



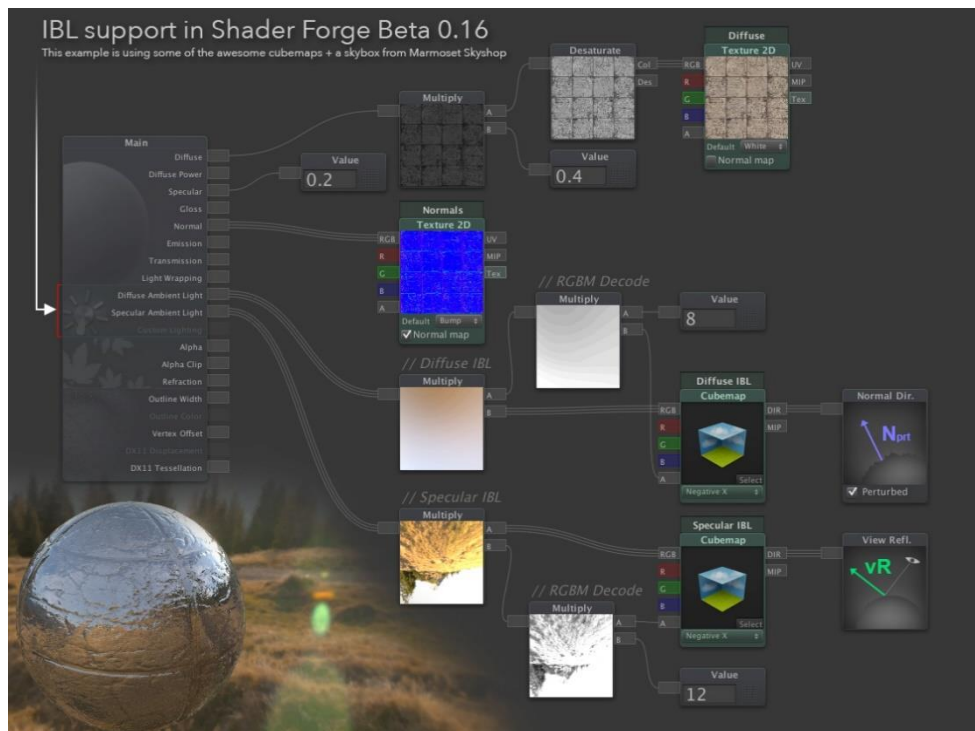
**Ilustración 4.11.** Dos materiales compartiendo y reutilizando una textura representada por un nodo de tipo Bitmap. (Fuente: Capturado desde 3ds Max 2012)

### 4.11.7 Shader Forge

Se trata de un editor de Shaders basado en nodos. Está diseñado para que funcione bajo el motor gráfico Unity, de tal forma se presenta como una extensión de pago del motor. La licencia es adquirida a través de la Unity Asset Store – tienda de contenidos y extensiones de Unity-.

Shader Forge trabaja con modelos de iluminación basado en físicas (PBR) y shading de conservación de la energía haciendo uso de los modelos Blinn-Phong o Phong.

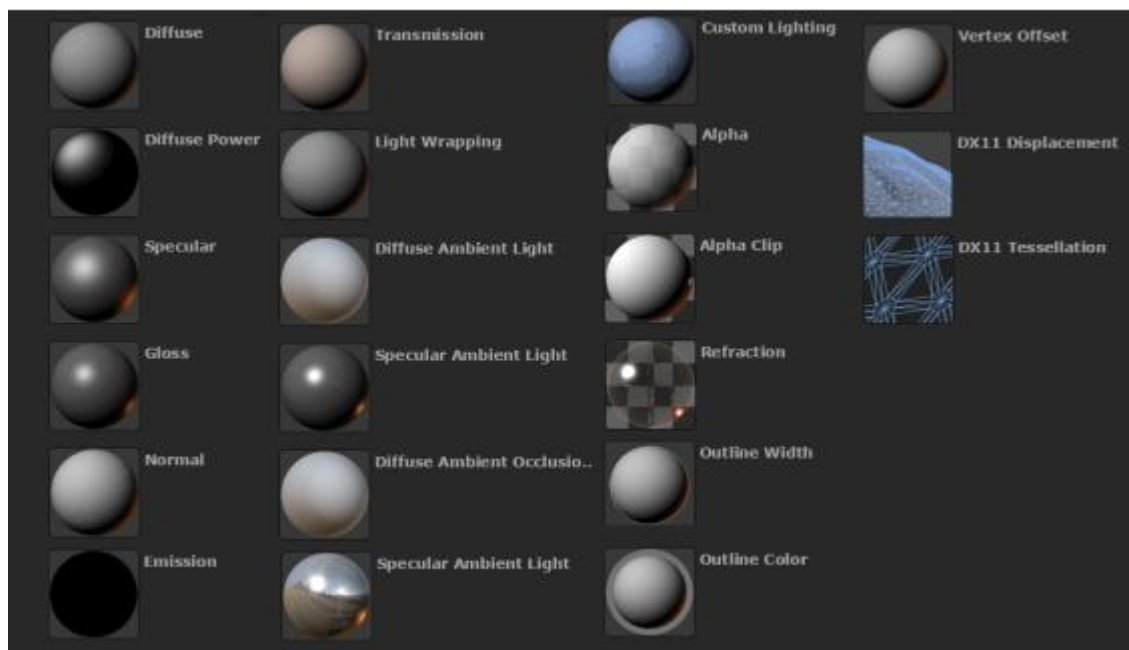
Además Shader Forge tiene la capacidad de poder trabajar con *Image Based Lighting* lo que le permite trabajar con técnicas como la de *Cube Maps* para visualizar la forma en que se iluminan los objetos, y generar valores para *Diffuse Ambient Light* y *Specular Ambient Light*. Gracias a esto Shader Forge es compatible con shaders y herramientas de la familia Marmoset (Marmoset Skyshop, Ilustración 4.12).



**Ilustración 4.12.** Ejemplo de un Shader creado por Shader Forge usando Skyboxes de Marmoset Skyshop.  
(Fuente: <http://acegikmo.com/shaderforge/> Recuperado en Agosto de 2015)

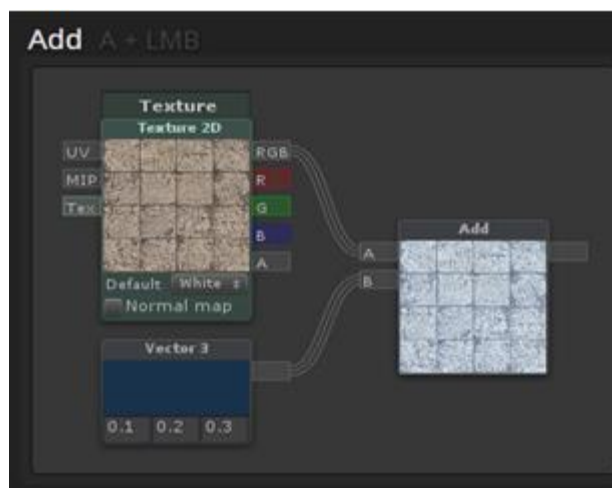
Su flujo de trabajo intuitivo es perfecto para artistas quienes no está muy habituados a programar ni a generar shaders mediante programación de texto.

El nodo principal de Shader Forge está compuesto por 21 inputs (Ilustración 4.13) a los que se les puede pasar un valor que modificara los parámetros del material y de la iluminación del objeto al que se le aplique el Shader. Entre los 21 inputs se encuentran algunos inputs clásicos como Diffuse, Specular y Normal, así como también algunos más personalizados como *Custom Lighting* el cual le permite al usuario definir un comportamiento de iluminación personalizado sin atender a las físicas de iluminación preestablecidas por el modelo de Shader Forge.



**Ilustración 4.13.** Conjunto de inputs de Shader Forge. (Fuente: <http://acegikmo.com/shaderforge/nodes/> Recuperado en Agosto de 2015).

Shader Forge dispone de un número de operaciones considerable que permiten la creación de nuevos valores resultado de operaciones realizadas sobre otros mismos. Operaciones como suma, resta, interpolación, mínimo, máximo, valor absoluto, clamp, blend (fundido), etc. son típicas en el desarrollo mediante Shader Forge.



**Ilustración 4.14.** Ejemplo en Shader Forge de una operación de suma entre un nodo de tipo textura y un vector representando un color del tipo RGB. (Fuente: <http://acegikmo.com/shaderforge/nodes/> Recuperado en Agosto de 2015)

#### 4.11.8 Marmoset Toolbag

Marmoset Toolbag es una herramienta principalmente de renderizado de alta calidad basado en físicas (PBR).

Marmoset Toolbag permite a artistas visualizar su trabajo en un render de alta calidad sin necesidad de utilizar un motor gráfico para lograrlo. Esto proporciona un estado de estandarización más amplio ya que muchos artistas pasan antes por Marmoset Toolbag que por el motor y empieza a convertirse en un referente de testeo para materiales en PBR.

La calidad de su render es la principal característica de Marmoset Toolbag (Ilustración 4.15). Está pensado para visualizar escenas generalmente pequeñas, con iluminación basada en mapas (Cubemaps y Skyboxes) para modelar el comportamiento en el que la luz refleja determinado modelo. Además de poseer una gran gama de características de renderizado que le dotan de gran vistosidad y realismo en sus escenas.



Ilustración 4.15 Modelo 3D renderizado con Marmoset Toolbag en tiempo real (Fuente: Colt Python or Benjamin Turner | turnededges.com)

Entre los parámetros que define un material en Marmoset Toolbag se encuentran: **Albedo**, (define el color base del material), **Specular**, **Gloss** y **Normales**. Otras características de materiales que se pueden encontrar en Marmoset Toolbag son:

- Dynamic Tessellation
- PN Triangle mesh smoothing
- Height & Vector Displacement modes
- Detail Normal Maps
- Parallax Occlusion Maps
- "Metalness" Maps
- GGX, Blinn-Phong, & Anisotropic Reflections
- Secondary Reflections
- Skin Diffusion
- Microfiber ("Fuzz") Diffusion
- Occlusion and Cavity Maps
- Emissive Maps
- Allegorithmic Substances
- Dithered Supersampled Transparency

## 4.12 Discusión

Como se ha podido ver, existen bastantes herramientas que permiten modificar el aspecto de los objetos de alguna manera. Pese a todo ello, se ha detectado una carencia general en todos ellos. Si bien se ha interpretado de antemano que crear un material es crear un Shader, no todos estos editores lo interpretan así. Por ejemplo, el caso de Marmoset o 3ds Max, demuestra que su finalidad es solo la de alimentar un modelo de material que existe de antemano, es decir, solo sirve para pasar los parámetros de valores del material y dejar el proceso de Shading a otro sistema diferente ya preparado.

Para casos como los de Shader Forge y el editor de materiales de UE4 el enfoque sí corresponde al de crear un Shader para crear un material, -al igual que se enfocará en este proyecto -, no obstante, dichos Shader resultan poco útiles de cara al usuario común ya que la complejidad que entrañan los convierten en Shaders tan específicos que solo bajo sus complejos sistemas pueden funcionar.

Dicho esto, tanto como si se crea o no el Shader, la utilidad de estos radica, a vista de un usuario, en ningún sitio más allá de en los complejos motores gráficos para los que se han generado.

En definitiva, la mencionada carencia descrita se pretende resolver mediante la creación de Shaders a partir del editor de nodos, y que **sí** sirvan para ser integrados o tratados por los usuarios de forma fácil y sencilla controlando en todo momento el proceso de dibujado de los objetos. En cualquiera de los casos, el Shader generado será siempre pensado para que el usuario lo pueda leer y entender.

## **5 Metodología y herramientas**

### **5.1 Metodología de desarrollo de Software.**

El proyecto software se ha desarrollado bajo una planificación donde el estudiante visitaba periódicamente al tutor. En dichas visitas se acordaba qué nuevas tareas se habrían de realizar para la siguiente visita. Las visitas se realizaban semanalmente en la mayoría de los casos.

Es por esto que la metodología seguida es una similar a una metodología basada en Sprints pequeños de unos 7 días por lo general, en donde se presentaban nuevas funcionalidades cada vez.

Estrictamente no se ha llevado a cabo el seguimiento de una metodología de desarrollo como tal. Desarrollo en algunos momentos contó con periodos de revisión más prolongados debido a inconvenientes que surgieron para acudir a las visitas de evaluación y revisión.

### **5.2 Control de versiones**

Es de vital importancia contar con una herramienta que permita llevar a cabo un control de versiones por diferentes motivos: para controlar el desarrollo, para publicar, y más importante aún, para tener un respaldo siempre presente ante cualquier imprevisto. Dada las características del proyecto, y tratándose de un trabajo principalmente académico, la decisión en un principio fue la de elegir una plataforma académica y libre como lo era Google Code. Una buena ventaja de esta decisión es que Google Code se podía basar en Subversion (un tipo de control de versiones) lo cual era muy bien recibido debido a la experiencia adquirida en pasado proyectos sobre esta plataforma.

El desarrollo se llevó a cabo sobre este repositorio durante la primera etapa de existencia. Durante esta primera etapa de desarrollo (mediados de Marzo de 2015) Google anuncio el cierre de Google Code y el final del servicio de reposición. Debido a esto la necesidad de migrar el proyecto se hizo casi obligatoria.

Tras un tiempo de investigación y consulta, se decidió usar GitHub. El servicio prestado por GitHub es gratuito, lo que se ajusta perfectamente a las mis condiciones económicas. Su herramienta de control de versiones es más que suficiente para llevar a cabo el desarrollo del proyecto de forma íntegra. El pequeño coste es la curva de aprendizaje para usarlo, aunque, como todo, con práctica se consiguió. Además GitHub es realmente fácil de usar, con lo que la curva de aprendizaje se reduce bastante.

La aplicación para realizar los commits es el **cliente de GitHub de Windows**. En él es fácil crear un commit y subir los cambios sincronizando la versión local con la del servidor.

Al final de todo no se sabrá cuanto tiempo vaya a estar el proyecto bajo el repositorio. Se hará todo lo posible para que dure el máximo de tiempo online, pero como todo puede pasar no se puede asegurar que el repositorio este vigente siempre. No obstante, si algo llegase a ocurrir, se procurará dejar toda la información del nuevo paradero del proyecto en caso de que se mueva o se migre.

A día de hoy (Agosto de 2015) la dirección para encontrar el proyecto en GitHub es: <https://github.com/s4ntia60/spc-editor/>

## 5.3 Herramientas de desarrollo

### 5.3.1 Visual Studio

Como entorno de desarrollo se utilizó Visual Studio (VS) en su versión 2013. Visual Studio permite el desarrollo de aplicaciones bajo sistemas operativos Windows con lenguajes de programación como C++ y con la posibilidad de integrar librerías y frameworks de forma fácil. Cabe también destacar que la potencia ofrecida, el número de herramientas y más que todo la facilidad para integrar desarrollo de otro tipos de proyectos (como scripts para motores gráficos o desarrollos web) hacen de VS uno de los más potentes –y universales – IDEs del mercado.



### 5.3.2 Qt

Qt es un frameworks multiplataforma de desarrollo de aplicaciones ampliamente usado para desarrollo de aplicaciones que pueden ser ejecutadas en diferentes sistemas sin la necesidad de cambiar el código o cambiarlo de una manera radical.

Qt fue la herramienta elegida para crear la interfaz gráfica del proyecto. Qt permite crear contextos para el dibujado 3D en OpenGL, permite crear interfaces de usuario típicas como controles de texto e inputs numéricos, pero además, lo más significativo para este proyecto es que permite generar un espacio donde a partir de elementos visuales como cajas y líneas, se puede crear el editor basado en nodos requerido.

La instalación típica incluye un IDE propio de Qt y también herramientas para el diseño de ventanas y formularios. Esta instalación fue evitada dado que el IDE de desarrollo es Visual Studio. Los creadores de Qt dejan a disposición de los desarrolladores la opción de crear las mismas aplicaciones que se pueden crear desde su propio IDE, en Visual Studio. Para ello se ha publicado un plugin para VS. Este plugin realiza las configuraciones pertinentes para crear un proyecto Qt. Para usar configurar un proyecto Qt en VS principalmente se debe seleccionar el lugar de instalación de Qt en el disco para establecer la versión con la que se compilara y para generar el código enlazando desde esta ubicación. Al momento de compilar, el plugin crea toda una rutina para generar archivos necesarios para la ejecución. Estos archivos pueden ser ficheros de interfaces de usuario (.ui) o ficheros *moc\_* donde se definen propiedades de algunos controles y meta elementos que Qt gestiona para funcionalidades como Signals/Slots (se trata de una aproximación de gestión de eventos mediante funciones).

Además, para contribuir con la tarea de trabajar con VS, Qt permite la instalación del diseñador de ventanas y formularios (*QtDesigner*, Ilustración 5.1) de forma independiente, donde lo único que se generan son archivos .ui.

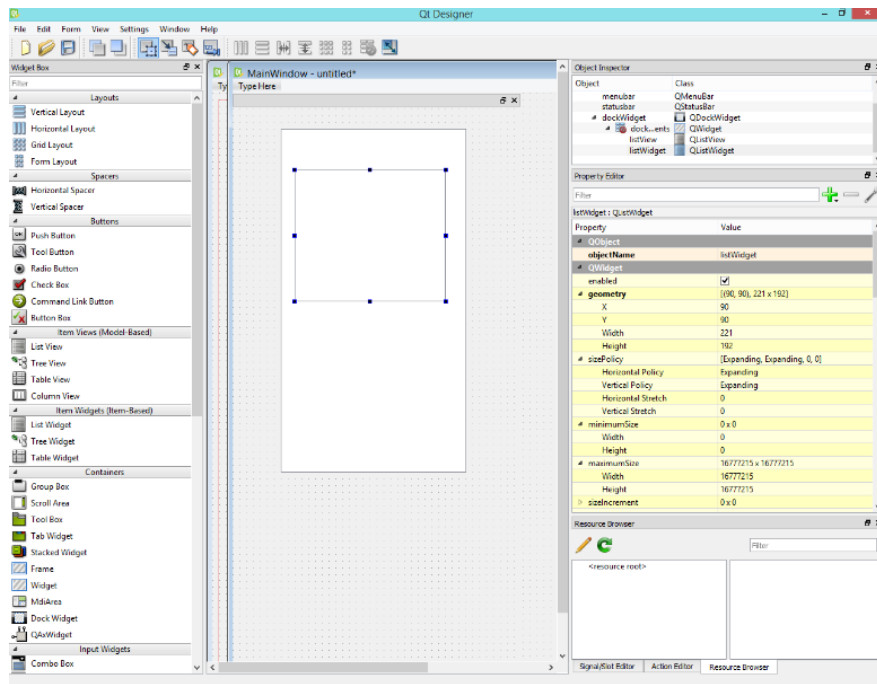


Ilustración 5.1. Qt Designer (Fuente: Captura desde QtDesigner)

Como consecuencia de usar Qt para el desarrollo, la curva de aprendizaje fue un nuevo desafío a superar. El mayor dolor de cabeza de todo el proyecto fue la implementación del editor de nodos como tal. Afortunadamente en la etapa de investigación y búsqueda de un elemento similar se encontró con el editor de bloques de **Stanislaw Adaszewski** [10]. Este editor era una implementación de Qt usando las clases de dibujo como *QGraphicsView* y *QGraphicsScene*. La implementación cuenta con los elementos básicos como nodos, inputs y conexiones. Además la captura de eventos p. ej. para realizar las conexiones mediante el uso del ratón o el cambio de posiciones mediante la acción de clic y arrastre del ratón. En definitiva este pequeño editor fue de gran ayuda en ausencia de experiencia con Qt y como ilustración para a partir de la misma desarrollar y modelar el sistema como se requiere.

El código de Stanislaw está bajo una licencia de tipo BSD. La reproducción, alteración y publicación están permitidas por lo que usar/modificar este código no supone ningún problema. Son pocas las restricciones que esta licencia obliga, como incorporar un cabecero en cada fichero que contenga su código donde se especifica los estatutos de dicha licencia [10].

### 5.3.3 GLEW (OpenGL Extension Wrangler Library)

GLEW es una librería multiplataforma que permite llamar a las funciones de OpenGL. Aunque OpenGL no es una librería como tal, pues está instalado y habita en las tarjetas gráficas, es necesario realizar la inclusión de una librería como GLEW para realizar consultas y cargar las extensiones de OpenGL. Sin GLEW no podríamos llamar a las funciones de OpenGL [14].

### 5.3.4 Assimp.

Se trata de una librería que permite la importación de varios de los elementos usados en gráficos. Principalmente se ha usado para importar las mallas usadas en la escena 3D del proyecto. Su principal ventaja es que puede importar mallas desde varios tipos de ficheros como .obj, .3ds, etc. Además assimp es un proyecto de software libre [2].

### 5.3.5 GLM (OpenGL Mathematics)

GLM es una librería matemática cuya principal ventaja es que permite el uso de tipos como vectores y funciones lineales de una forma muy parecida a la que GLSL los implementa. Además está ampliamente extendida y aunque no es un estándar, suele ser un elemento esencial en el desarrollo cotidiano de OpenGL.

### 5.3.6 NotePad++

Se trata de un editor de texto plano para desarrollo de software bastante ligero. Su uso principalmente fue para retocar el código de los shaders y su edición.

### 5.3.7 Photoshop y Gimp.

El uso de Photoshop fue principalmente para realizar los gráficos y texturas del proyecto, no solo para mapas de texturas sino también para crear esquemas y cuadros conceptuales. El uso de Gimp se basó en a partir del plugin *Normalmap* crear algunos mapas de normales usados en los ejemplos y en las pruebas del proyecto.

### 5.3.8 3D Max 2016

Se trata de una herramienta para el modelado en 3D. Se usó principalmente para generar mallas y modelos en formato .obj las cuales se podían importar al proyecto para su visualización. Además en este programa se pueden modificar las coordenadas de textura

de los modelos aplicando funciones de proyección distintas en caso de que las predeterminadas no satisficieran las necesidades eventuales.

## 6 Modelo Propuesto

### 6.1 Análisis y especificación

A continuación se realizará una descripción completa del sistema, repasando aspectos como requisitos funcionales y no funcionales, interfaz gráfica y otros comportamientos. Además se hará la descripción de la planificación que se seguirá durante el desarrollo.

Antes de empezar a describir en detalle el sistema se hará una definición del mismo en términos generales.

#### 6.1.1 Descripción general del sistema.

El Sistema es una aplicación de escritorio que permite al usuario poder manipular una serie de nodos para crear un grafo, las conexiones entre los nodos alimentan unas entradas básicas, especificadas como parámetros del modelo de iluminación a desarrollar. Una vez hecho esto, el sistema debe resolver las conexiones del grafo y generar un Shader (Fragment) que se compilará y se usará para dibujar. Por consiguiente, debe existir un visor en 3d en el cual los resultados de las modificaciones que se van realizando se hagan notorias.

Cada nodo del grafo representa una operación o una fuente de datos. Todos tienen al menos una salida y los nodos que requieren de algún argumento para realizar operaciones tienen entradas para dichas operaciones. Todos los nodos terminan alimentando de alguna forma u otra alimentando el nodo principal *MainNode*. Los nodos que no forman parte del grafo (aislados) no se toman en cuenta para la creación del Shader.

El sistema proporciona todas las interfaces necesarias para el suministro de datos por parte del usuario como son datos de vectores, imágenes y valores de los distintos nodos.

Además de las interacciones con los nodos, también están contempladas las realizadas sobre el visor 3D. Estas incluyen girar el modelo de muestra, acercar o alejar la cámara y abrir o cerrar el zoom de la cámara. Además se pueden ajustar parámetros de la luz como el color y la intensidad. Otra opción a disposición del usuario es la de poder cambiar el modelo a una serie de modelos proporcionados como muestras de visualización.

A continuación se hace un análisis a más profundidad de cada apartado del sistema.

### 6.1.2 Editor de nodos.

El editor de nodos será un área de trabajo donde se pueden manipular cajas que representan los nodos de funciones dentro del Fragment Shader que se generará.


#### Nodos

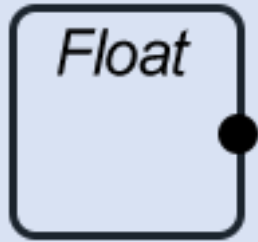
Los nodos son abstracciones visuales de operaciones existentes en OpenGL o creadas de forma personalizada. Cada nodo devolverá un valor capaz de ser interpretado por el nodo que solicite dicha salida (esté conectado a él). Además de devolver el resultado principal de sus operaciones, los nodos deben devolver de forma individual las componentes de color que lo forman. Así si un nodo tiene una salida del tipo vec4, éste contendrá una salida para obtener el valor compuesto del vec4 pero además tendrá otras 3 salidas para las componentes R, G, B y A. La principal motivación para esto es que devolviendo cada componente se puede lograr efectos muchísimos más variados como el enmascarado de colores usando texturas (Texture Masking) o la obtención de nuevos materiales sin nuevas fuentes de datos. Esta característica dota de potencial a cualquier editor de materiales basado en nodos.

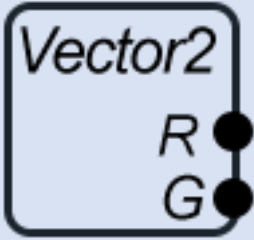
Al iniciar el sistema, se contará con un nodo principal el cual contiene solo inputs (MainNode). Este nodo contiene los inputs básicos a definir para construir un material nuevo. Estos inputs son:

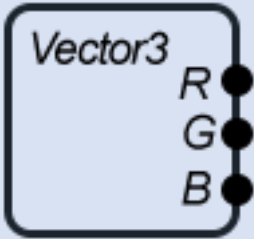
- **Color**  
Suministra el componente difuso del modelo.
- **SpecularColor**  
Suministra el componente especular del modelo.
- **Shininess**  
Regula el exponente de especularidad dentro del modelo.
- **Normal**  
Suministra la dirección de la normal para realizar el Shading.
- **Alpha**  
Sirve como mascara para modificar el valor de alpha. Y asignar transparencia.

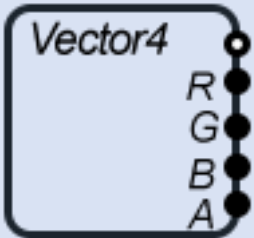
A continuación se hace una descripción de cada nodo disponible en el sistema donde se especifica el diseño que debe tener, las entradas y salidas que poseen y para qué sirven además de la operación que realizan:

<b>Nombre:</b>	<b>Main Node</b>
<b>Diseño</b>	
<b>Descripción:</b>	Recibe los valores finales para los parámetros del modelo de iluminación.
<b>Inputs:</b>	-Color (vec4) -Specular(vec4) -Shininess (vec4) -Normal (vec4) -Alpha (vec4).
<b>Outputs:</b>	No posee outputs

<b>Nombre:</b>	<b>Const Value Node</b>
<b>Diseño</b>	
<b>Descripción:</b>	Almacena un valor único en forma de float. Este nodo debe tener relacionado un campo numérico para la introducción del dato como tal.
<b>Inputs:</b>	Ninguno.
<b>Outputs:</b>	Devuelve un número en coma flotante (float).

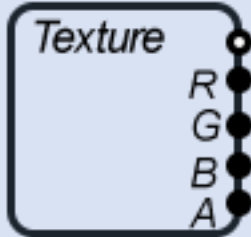
<b>Nombre:</b>	<b>Vector 2 Node</b>
<i>Diseño</i>	
<i>Descripción:</i>	Almacena un vector 2D con componentes RG.
<i>Inputs:</i>	Ninguno.
<i>Outputs:</i>	R: Devuelve el componente de rojo del vector (float). G: Devuelve el componente de verde del vector (float).

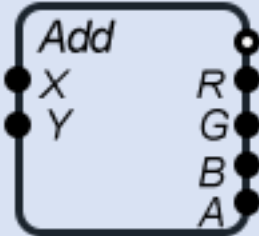
<b>Nombre:</b>	<b>Vector 3 Node</b>
<i>Diseño</i>	
<i>Descripción:</i>	Almacena un vector 3D con componentes RGB.
<i>Inputs:</i>	Ninguno.
<i>Outputs:</i>	R: Devuelve el componente de rojo del vector (float). G: Devuelve el componente de verde del vector (float). B: Devuelve el componente azul del vector (float).

<b>Nombre:</b>	<b>Vector 4 Node</b>
<i>Diseño</i>	

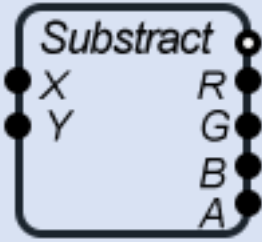


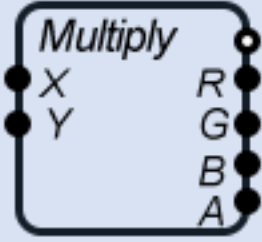
<i>Descripción:</i>	Almacena un vector 4D de componentes RGBA.
<i>Inputs:</i>	Ninguno.
<i>Outputs:</i>	<p><b>O</b> - Devuelve el vector4 que componen los valores RGBA de la textura.</p> <p><b>R</b>: Devuelve el componente de rojo del vector (float).</p> <p><b>G</b>:Devuelve el componente de verde del vector (float)</p> <p><b>B</b>: Devuelve el componente azul del vector (float).</p> <p><b>A</b>: Devuelve el componente alpha del vector (float).</p>

<i>Nombre:</i>	<b>Texture Node</b>
<i>Diseño</i>	
<i>Descripción:</i>	Almacena una textura.
<i>Inputs:</i>	Ninguno (Se ha de introducir la ruta de la textura)
<i>Outputs:</i>	<p><b>O</b> - Devuelve el vector4 que componen los valores RGBA de la textura.</p> <p><b>R</b>: Devuelve el componente de rojo del vector (float).</p> <p><b>G</b>:Devuelve el componente de verde del vector (float)</p> <p><b>B</b>: Devuelve el componente azul del vector (float).</p> <p><b>A</b>: Devuelve el componente alpha del vector (float).</p>

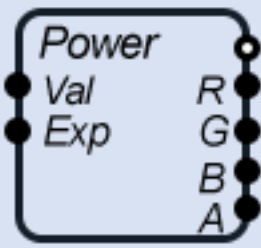
<i>Nombre:</i>	<b>Add</b>
<i>Diseño</i>	

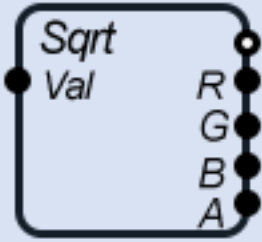
<i>Descripción:</i>	Calcula la operación de suma.
<i>Inputs:</i>	X (vec4 o float) y Y (vec4 o float): Operandos para realizar la suma.
<i>Outputs:</i>	<p><b>O</b> - Devuelve el vector4 que componen los valores RGBA de la operación.</p> <p>R: Devuelve el componente de rojo del vector (float).</p> <p>G: Devuelve el componente de verde del vector (float)</p> <p>B: Devuelve el componente azul del vector (float).</p> <p>A: Devuelve el componente alpha del vector (float).</p>

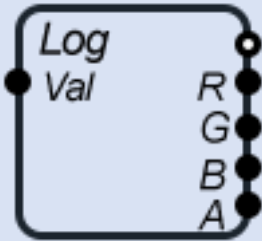
<b>Nombre:</b>	<b>Subtract</b>
<i>Diseño</i>	
<i>Descripción:</i>	Calcula la operación de resta.
<i>Inputs:</i>	X (vec4 o float) y Y (vec4 o float): Operandos para realizar la resta.
<i>Outputs:</i>	<p><b>O</b> - Devuelve el vector4 que componen los valores RGBA de la operación.</p> <p>R: Devuelve el componente de rojo del vector (float).</p> <p>G: Devuelve el componente de verde del vector (float)</p> <p>B: Devuelve el componente azul del vector (float).</p> <p>A: Devuelve el componente alpha del vector (float).</p>

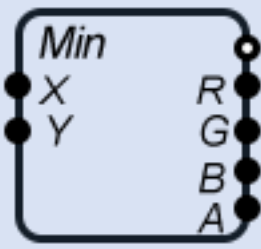
<b>Nombre:</b>	<b>Multiply</b>
<i>Diseño</i>	

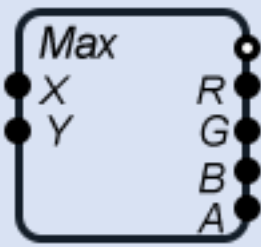
<i>Descripción:</i>	Calcula la operación de multiplicación.
<i>Inputs:</i>	X (vec4 o float) y Y (vec4 o float): Operandos para realizar la multiplicación.
<i>Outputs:</i>	<p><b>O</b> - Devuelve el vector4 que componen los valores RGBA de la operación.</p> <p>R: Devuelve el componente de rojo del vector (float).</p> <p>G: Devuelve el componente de verde del vector (float)</p> <p>B: Devuelve el componente azul del vector (float).</p> <p>A: Devuelve el componente alpha del vector (float).</p>


<i>Nombre:</i>	<b>Power</b>
<i>Diseño</i>	
<i>Descripción:</i>	Eleva un valor a la potencia especificada.
<i>Inputs:</i>	<p>Val (vec4 o float): Base de la potencia.</p> <p>Exp(vec4 o float): Exponente.</p>
<i>Outputs:</i>	<p><b>O</b> - Devuelve el vector4 que componen los valores RGBA de la operación.</p> <p>R: Devuelve el componente de rojo del vector (float).</p> <p>G: Devuelve el componente de verde del vector (float)</p> <p>B: Devuelve el componente azul del vector (float).</p> <p>A: Devuelve el componente alpha del vector (float).</p>

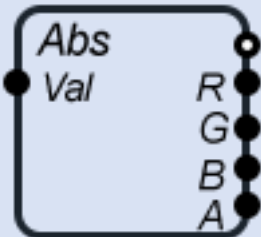
<b>Nombre:</b>	<b>Sqrt</b>
<i>Diseño</i>	
<i>Descripción:</i>	Realiza la operación de raíz cuadrada.
<i>Inputs:</i>	Val(vec4 o float): Valor por el que se calculara la raíz.
<i>Outputs:</i>	<p><b>O</b> - Devuelve el vector4 que componen los valores RGBA de la operación.</p> <p>R: Devuelve el componente de rojo del vector (float).</p> <p>G: Devuelve el componente de verde del vector (float)</p> <p>B: Devuelve el componente azul del vector (float).</p> <p>A: Devuelve el componente alpha del vector (float).</p>

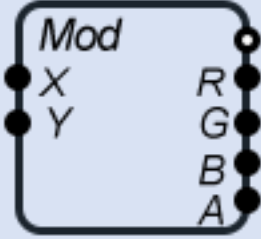
<b>Nombre:</b>	<b>Log</b>
<i>Diseño</i>	
<i>Descripción:</i>	Realiza la operación de logaritmo.
<i>Inputs:</i>	Val (vec4 o float): Valor con el que se calculara el logaritmo.
<i>Outputs:</i>	<p><b>O</b> - Devuelve el vector4 que componen los valores RGBA de la operación.</p> <p>R: Devuelve el componente de rojo del vector (float).</p> <p>G: Devuelve el componente de verde del vector (float)</p> <p>B: Devuelve el componente azul del vector (float).</p> <p>A: Devuelve el componente alpha del vector (float).</p>

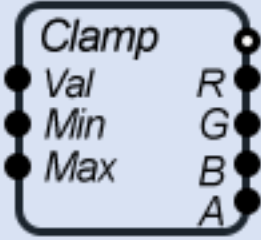
<b>Nombre:</b>	<b>Min</b>
<b>Diseño</b>	
<b>Descripción:</b>	Compara dos valores y devuelve el menor.
<b>Inputs:</b>	X (vec4 o float) y Y (vec4 o float): Valores a comparar.
<b>Outputs:</b>	<p><b>O</b> - Devuelve el vector4 que componen los valores RGBA de la operación.</p> <p>R: Devuelve el componente de rojo del vector (float).</p> <p>G: Devuelve el componente de verde del vector (float)</p> <p>B: Devuelve el componente azul del vector (float).</p> <p>A: Devuelve el componente alpha del vector (float).</p>

<b>Nombre:</b>	<b>Max</b>
<b>Diseño</b>	
<b>Descripción:</b>	Compara dos valores y devuelve el mayor.
<b>Inputs:</b>	X (vec4 o float) y Y (vec4 o float): valores a comparar.
<b>Outputs:</b>	<p><b>O</b> - Devuelve el vector4 que componen los valores RGBA de la operación.</p> <p>R: Devuelve el componente de rojo del vector (float).</p> <p>G: Devuelve el componente de verde del vector (float)</p> <p>B: Devuelve el componente azul del vector (float).</p> <p>A: Devuelve el componente alpha del vector (float).</p>

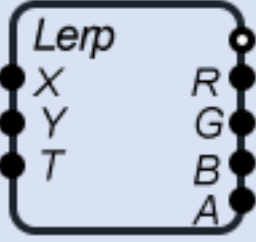
<b>Nombre:</b>	Sign
<b>Diseño</b>	
<b>Descripción:</b>	Devuelve el signo del valor entrante, -1 si es negativo, 1 si es positivo, y 0 si es 0.
<b>Inputs:</b>	Val (vec4 o float): Valor a calcular el signo.
<b>Outputs:</b>	<p><b>O</b> - Devuelve el vector4 que componen los valores RGBA de la operación.</p> <p>R: Devuelve el componente de rojo del vector (float).</p> <p>G: Devuelve el componente de verde del vector (float)</p> <p>B: Devuelve el componente azul del vector (float).</p> <p>A: Devuelve el componente alpha del vector (float).</p>

<b>Nombre:</b>	Abs
<b>Diseño</b>	
<b>Descripción:</b>	Devuelve el valor absoluto de un valor.
<b>Inputs:</b>	Val (vec4 o float): valor a calcular su valor absoluto.
<b>Outputs:</b>	<p><b>O</b> - Devuelve el vector4 que componen los valores RGBA de la operación.</p> <p>R: Devuelve el componente de rojo del vector (float).</p> <p>G: Devuelve el componente de verde del vector (float)</p> <p>B: Devuelve el componente azul del vector (float).</p> <p>A: Devuelve el componente alpha del vector (float).</p>

<b>Nombre:</b>	<b>Mod</b>
<b>Diseño</b>	
<b>Descripción:</b>	Calcula el modulo (resto)
<b>Inputs:</b>	X (vec4 o float): especifica el valor a evaluar Y (vec4 o float): dividendo.
<b>Outputs:</b>	<b>O</b> - Devuelve el vector4 que componen los valores RGBA de la operación. R: Devuelve el componente de rojo del vector (float). G: Devuelve el componente de verde del vector (float) B: Devuelve el componente azul del vector (float). A: Devuelve el componente alpha del vector (float).

<b>Nombre:</b>	<b>Clamp</b>
<b>Diseño</b>	
<b>Descripción:</b>	Acota un valor a un rango. Si el valor es inferior al mínimo, éste devuelve el mínimo, si es mayor que el máximo, éste devuelve el máximo, de lo contrario devuelve el valor original.
<b>Inputs:</b>	Val (vec4 o float): valor a restringir. Min (vec4 o float): valor mínimo. Max (vec4 o float): valor máximo.
<b>Outputs:</b>	<b>O</b> - Devuelve el vector4 que componen los valores RGBA de la operación.

	R: Devuelve el componente de rojo del vector (float). G:Devuelve el componente de verde del vector (float) B: Devuelve el componente azul del vector (float). A: Devuelve el componente alpha del vector (float).
--	--

<b>Nombre:</b>	<b>Lerp</b>
<i>Diseño</i>	
<i>Descripción:</i>	Realiza una interpolación lineal entre dos valores basándose es un porcentaje.
<i>Inputs:</i>	X (vec4 o float): valor inicial Y (vec4 o float): valor final T: (float) porcentaje.
<i>Outputs:</i>	<b>O</b> - Devuelve el vector4 que componen los valores RGBA de la operación. R: Devuelve el componente de rojo del vector (float). G:Devuelve el componente de verde del vector (float) B: Devuelve el componente azul del vector (float). A: Devuelve el componente alpha del vector (float).

## Conexiones

Las conexiones se realizaran haciendo clic sobre un input u output y manteniendo pulsado el botón del ratón mientras se arrastra hasta el puerto donde se quiera conectar estableciendo así una nueva conexión. Para realizar una conexión se ha de tener en cuenta ciertas restricciones de conexión entre nodos y puertos.



## 6.2 Restricciones de conexión:

- Un input solo puede tener una conexión asignada. Si se intenta conectar un nodo hacia un input que ya estaba previamente establecido, la conexión no se realizará.
- Un output puede proveer más de una conexión, lo que facilita la reutilización de nodos, y evita redundancias.
- Se ha de poder eliminar un nodo, en tal caso, todas las conexiones adyacentes a este nodo se deben eliminar.
- Un nodo no puede ser fuente de alimentación de sí mismo, por lo tanto, si se intenta usar un miembro de un mismo nodo para conectar un puerto de dicho nodo, la conexión no se realizará.
- Se han de poder eliminar conexiones más no inputs ni outputs.

### 6.2.1 Motor de dibujado y controles del visor 3D.

Además de la implementación del editor de nodos, el sistema debe contar con un sistema para dibujar el resultado del Shader que está siendo generado desde el grafo de nodos.

El sistema de dibujado será incluido en una parte de la interfaz de usuario. Desde allí se dibujara una malla que será cargada desde un fichero. Además se dispondrá una interfaz para cambiar la malla que está siendo dibujada por una nueva malla elegida de una selección. También habrá controles para establecer las propiedades de la luz, así como para girar la malla que se visualiza.

#### **Modelo de iluminación**

La aplicación debe contar con un modelo de iluminación. Este modelo es importante no solo porque sirve para dibujar, evidentemente, sino porque a partir de él se modelará el sistema entero y se generan los Shader desde el editor de nodos. Por esto la elección del modelo de iluminación afecta en la forma y el comportamiento al resto de la aplicación de alguna manera.

## Modelo de Blinn-Phong.

El modelo a implementar es el modelo de Blinn-Phong. Esto es debido a que es un modelo altamente usado, rápido, fácil de entender y también de implementar. Además con él se pueden obtener resultados bastante realistas o competitivos (depende de factores como el arte y el modelado).

Como bien se ha mencionado en los objetivos del proyecto, uno de las metas es la de dar a entender de una forma más intuitiva como se crean los gráficos y por computador (en especial en la etapa del Fragment Shader) y el modelo de Blinn-Phong es perfecto por su sencillez para tal objetivo.

Además es un modelo clásico y se ha considerado que se merece estar incluido en este proyecto por el valor histórico.

## Interfaz de usuario y uso.

Presentación y descripción de interfaz de usuario:

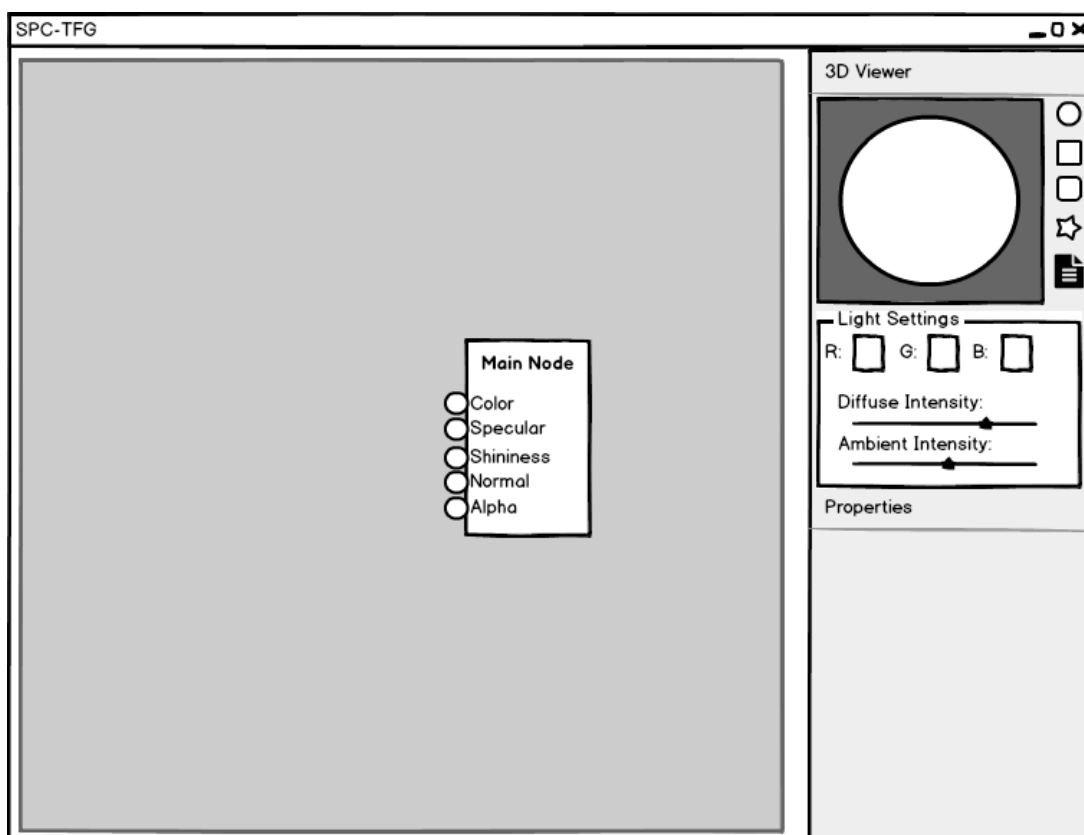
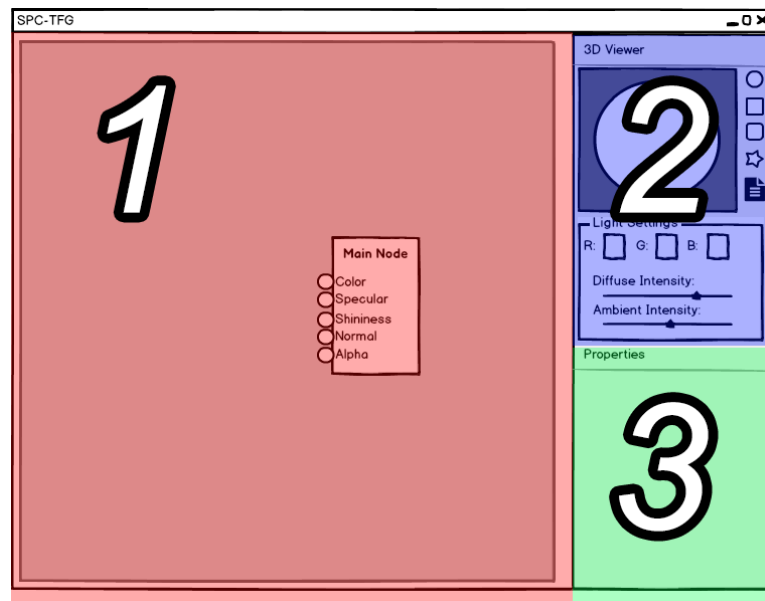


Ilustración 6.1. Interfaz gráfica del programa. (Fuente: Elaboración propia)

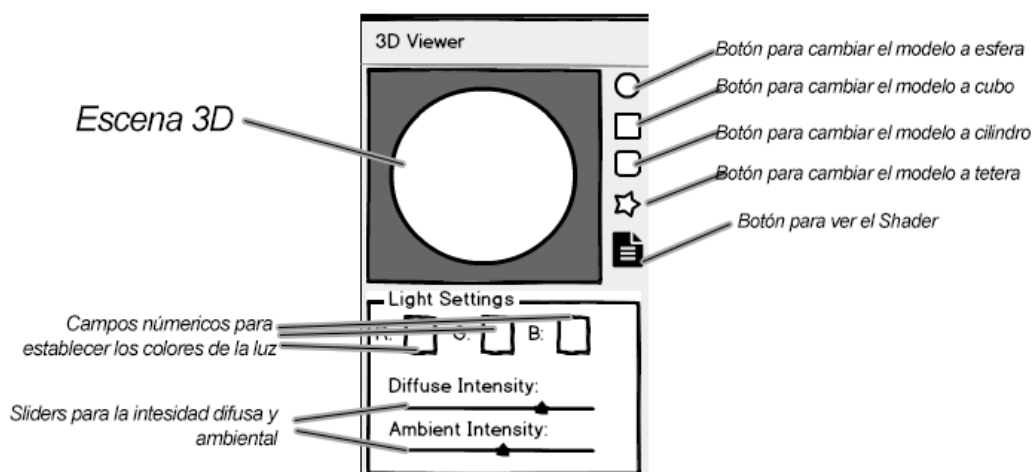
La interfaz de usuario se presenta en forma de ventana de aplicación de escritorio. Está constituida por 3 secciones principales:



**Ilustración 6.2.** Secciones de la interfaz de usuario. (Fuente: Elaboración propia)

La **primera sección** es donde se haya el editor de nodos, en ella aparecen inicialmente el nodo principal. En este espacio es donde se trabajará con los nodos.

La **segunda sección** es el visor 3d y sus controles. Aquí es donde se encuentra la escena en 3D y alrededor los controles para las propiedades de la luz y el cambio de malla, además del botón para ver el Shader generado:



**Ilustración 6.3** Descripción de los controles de la Escena 3D. (**Fuente:** Elaboración propia)

Descripción de los controles:


- **Botones para cambiar el modelo:** Según se pulse un botón de este grupo, se pasará a dibujar el modelo seleccionado.
- **Botón para ver el Shader:** Al darle clic abrirá una ventana modal donde se podrá ver el Shader con el que se está dibujando la escena actual.
- **Componentes de color de la luz:** Estos controles solo aceptan números, sirven para almacenar la información del color de la luz.
- **Sliders para el control de las intensidades:** Controlan la intensidad de la luz para el componente difuso y especular, el hecho de tener dos componentes en lugar de uno (como típicamente se realiza) da pie a realizar ajustes más diversos y obtener resultados mucho más completos (ya que el componente ambiental y el difuso no comparten la misma intensidad de luz).

**La tercera sección** es la sección de las propiedades de los nodos. En ella aparecerán diferentes controles visuales en función del nodo seleccionado. Algunos nodos no contendrán más que un campo para una descripción o una nota, mientras que otros como

los nodos Const Float, Vector2, Vector3, Vector4 y Textura tendrán campos para establecer sus correspondientes valores (Ilustración 6.4 e Ilustración 6.5).

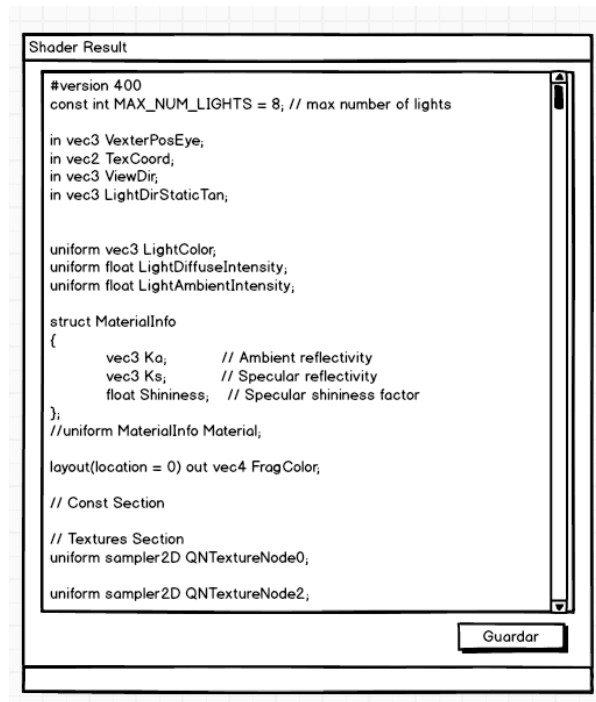
Properties	Properties	Properties	Properties
Desc: <input type="text"/>	Desc: <input type="text"/>	Desc: <input type="text"/>	Desc: <input type="text"/>
R: <input type="text"/>	R: <input type="text"/>	R: <input type="text"/>	V: <input type="text"/>
G: <input type="text"/>	G: <input type="text"/>	G: <input type="text"/>	
B: <input type="text"/>	B: <input type="text"/>		
A: <input type="text"/>			

**Ilustración 6.4.** Diseño de las propiedades de los nodos Vector4, Vector3, Vector2 y Float respectivamente. (Fuente: Elaboración propia)

Properties
Desc: <input type="text"/>
Path: <input type="text" value="C:\files\ir"/> <input type="button" value="Browse"/>


**Ilustración 6.5.** Propiedades del nodo Texture (Fuente: Elaboración propia)

Las propiedades del nodo Texture contienen un botón que al ser pulsado abre una ventana modal para buscar archivos de tipo imagen. Al seleccionar una imagen pasará a mostrarse en miniatura justo debajo.



**Ilustración 6.6.** Ventana modal donde se enseña el Shader resultado. (**Fuente:** Elaboración propia)

La ventana modal que se despliega cuando se pulsa el botón de “Ver Shader” tiene como finalidad el poder mostrar el código generado por el programa y posibilitar la opción de ser guardado en disco (ver Ilustración 6.6).

### 6.2.2 Uso e interacción

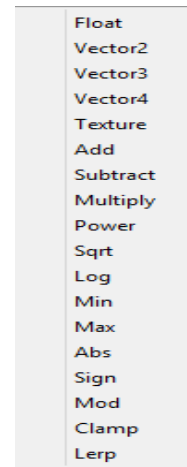
A continuación se hará una descripción de las posibles acciones que se pueden realizar en el programa.

#### **Interacción con el editor de materiales:**

Las interacciones con el editor de materiales son abundantes y cabe listarlas para su mejor documentación:

- **Seleccionar nodos:** Para seleccionar un nodo se debe realizar clic sobre el mencionado nodo. Al seleccionar un nodo se mostraran sus propiedades la sección de “Propiedades” del panel derecho de la interfaz gráfica.

- **Manipular nodos:** Los nodos se puede mover de un sitio a otro. Para realizar esta acción se deberá pulsar clic sobre un nodo y mientras se mantiene presionado arrastrar el ratón hasta la posición donde se quiere dejar.
- **Crear nodos:** Para crear nodos se debe pulsar clic derecho sobre el editor de nodos, en ese momento un menú desplegable aparecerá (ver Ilustración 6.7). En el menú se encuentran opciones para cada uno de los nodos disponibles, para crear un nodo se ha de elegir una de las opciones y el nodo aparecerá en lugar donde se encuentra el cursor.



**Ilustración 6.7.** Menú desplegable para la creación de nuevos nodos. (Fuente: Elaboración propia)

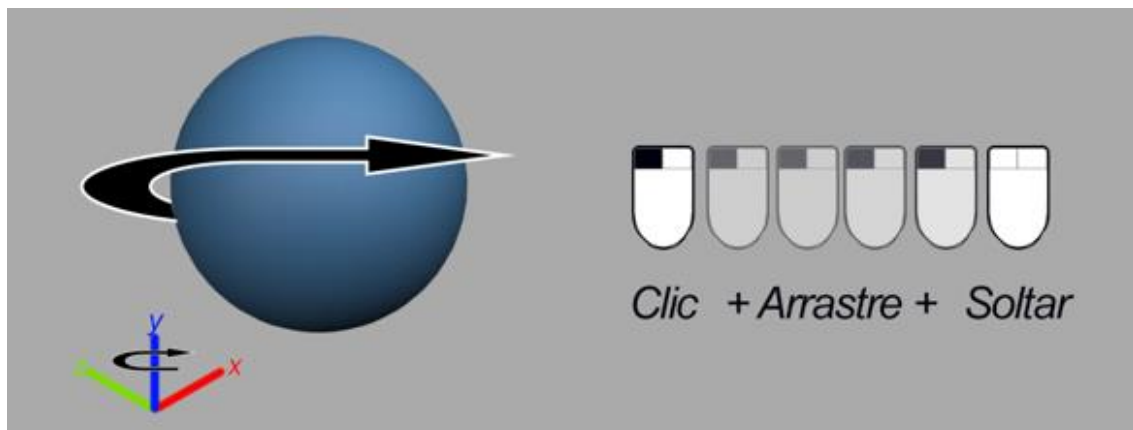
- **Eliminar nodos:** A excepción del nodo principal, el resto de nodos se pueden eliminar. Para eliminar un nodo primero se deberá pulsar la tecla Ctrl, mientras se mantiene pulsada se hará clic derecho sobre el nodo a eliminar.
- **Crear conexiones:** Para crear conexiones se debe hacer clic sobre un input o un output y mientras se mantiene pulsado arrastrar hasta un nuevo input u output. Se creara la conexión si es una conexión legal (ver restricciones de conexión).
- **Eliminar conexiones:** Para eliminar una conexión se debe pulsar la tecla Ctrl y mientras se mantiene, hacer clic derecho sobre la conexión que se desea eliminar.

### **Interacción con el editor el viewport 3D:**

Las acciones que se pueden realizar desde el *viewport* de dibujar de OpenGL son principalmente para manipular y transformar el modelo que se dibujar:

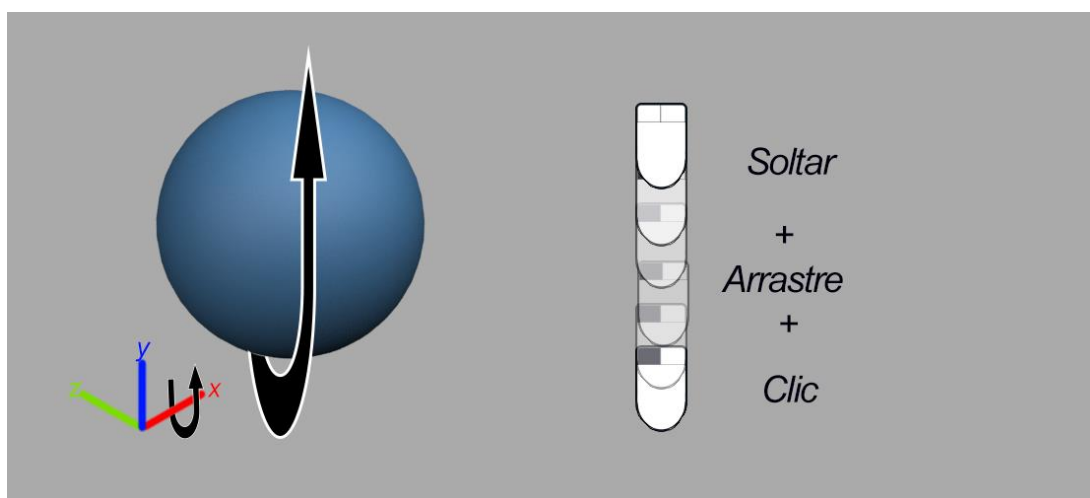
#### **Rotar modelo:**

Para realizar la rotación del modelo en el eje Y (vertical) se deberá hacer clic sobre el viewport y arrastrar de forma horizontal. El movimiento de arrastre puede ser de izquierda a derecha o viceversa. Si el movimiento va hacia la derecha entonces el giro será negativo mientras que si el giro es hacia la izquierda el giro será positivo (Ilustración 6.8).



**Ilustración 6.8.** Descripción de la acción de rotar en el eje Y sobre el visor 3D. (**Fuente:** Elaboración propia)

Complementariamente el giro en el eje X es posible de realizar haciendo clic y arrastrando el ratón verticalmente. El movimiento de arrastre puede realizarse hacia arriba o hacia abajo. Si el movimiento es hacia arriba, entonces el giro será positivo, mientras que si el arrastre es hacia abajo, el giro será negativo (Ilustración 6.9).

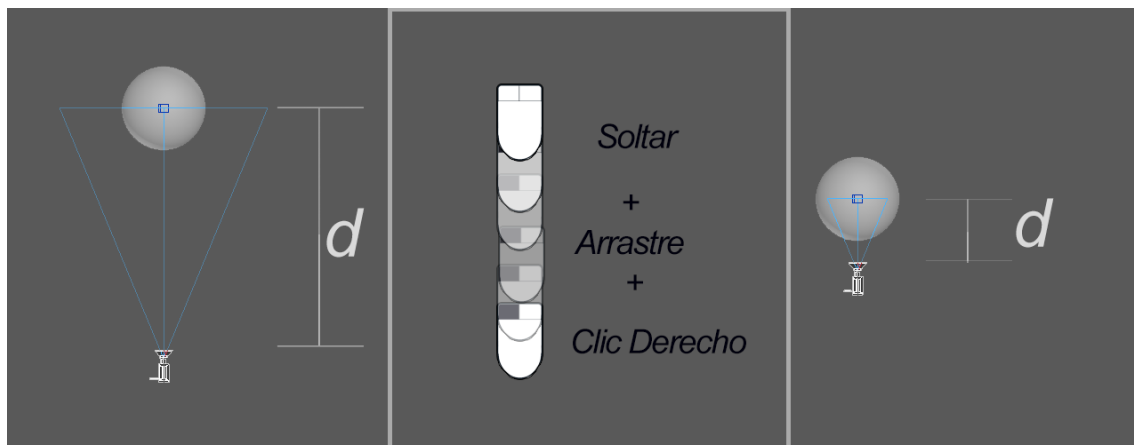


**Ilustración 6.9.** Descripción de la acción de rotar en el eje X sobre el visor 3D. (**Fuente:** Elaboración propia)

La combinación de los dos movimientos da como resultado un movimiento de rotación en los ejes X y Y simultáneamente.

Además de los movimientos de rotación también será posible realizar movimientos para alejar la cámara o acercarla, también para abrir o cerrar el FOV (Ilustración 6.10).

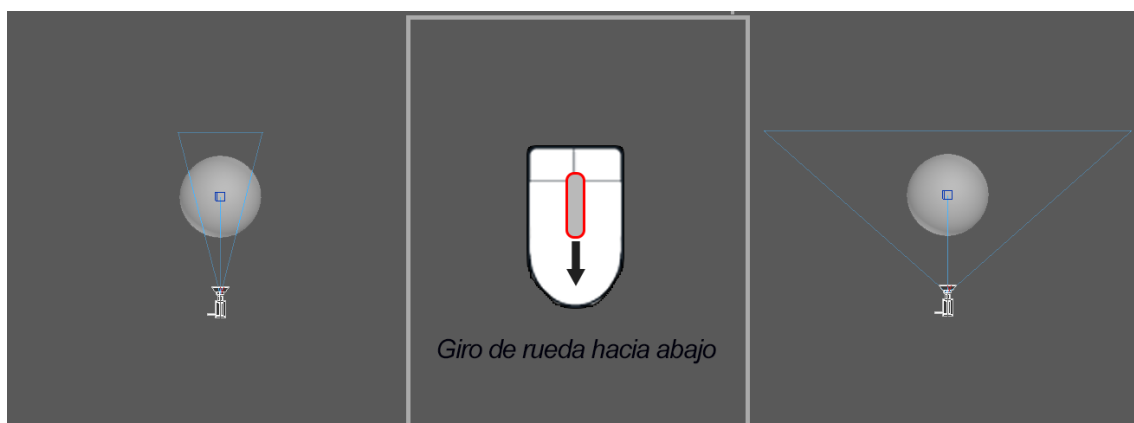




**Ilustración 6.10.** Acción de acercamiento y alejamiento de la cámara. (**Fuente:** Elaboración propia)

Para realizar el movimiento de acercamiento de la cámara se debe pulsar clic derecho sobre el viewport y arrastrar hacia arriba el ratón. Para realizar el movimiento de alejamiento de la cámara se ha de pulsar clic derecho y arrastra hacia abajo el ratón.

El movimiento se hará sobre el eje Z.



**Ilustración 6.11.** Acción de incrementar el FOV. (**Fuente:** Elaboración propia)

La acción de modificar el FOV de la cámara se hará con la rueda del ratón (Ilustración 6.11). Para incrementar el FOV se debe girar la rueda del ratón hacia abajo y para decrementarlo se debe girar hacia arriba. El incremento del FOV producirá una distorsión en la cámara que producirá un efecto de zoom.

### 6.2.3 Requisitos.

A continuación se realiza un análisis y enunciado de los requisitos funcionales y no funcionales del sistema.

## Requisitos funcionales

Identificador	RF-1
Título	Iniciar la aplicación
Necesidad	Esencial
Descripción	El sistema deberá iniciarse sin errores y desplegarse con normalidad.

Identificador	RF-2
Título	Mostrar interfaz de usuario
Necesidad	Esencial
Descripción	El sistema deberá mostrar en la pantalla la interfaz de usuario.

Identificador	RF-3
Título	Interactuar con nodos
Necesidad	Esencial
Descripción	El sistema deberá permitir al usuario manipular los nodos del editor de nodos.

Identificador	RF-4
Título	Eliminar nodos
Necesidad	Esencial
Descripción	El sistema deberá permitir al usuario el poder eliminar los nodos que desee nodos. Para eliminar un nodo se deberá mantener pulsada la tecla Ctrl y pulsar clic derecho sobre el nodo en cuestión.

Identificador	RF-5
Título	Crear nodos

Necesidad	Esencial
Descripción	El sistema deberá permitir al usuario el crear nuevos nodos. Para crear un nuevo nodo el usuario deberá pulsar clic derecho sobre el área de edición de nodos y seleccionar una opción de un menú desplegable que contiene el nombre de todos los nodos.

Identificador	RF-6
Título	Eliminar conexiones
Necesidad	Esencial
Descripción	El sistema deberá permitir al usuario eliminar conexiones entre nodos. Para eliminar una conexión se deberá mantener pulsada la tecla Ctrl y pulsar clic derecho sobre la conexión en cuestión.

Identificador	RF-7
Título	Crear conexiones
Necesidad	Esencial
Descripción	El sistema deberá permitir al usuario crear nuevas conexiones entre nodos. Para crear una conexión se deberá hacer clic sobre un input o un output de algún nodo y se arrastrar hasta llegar a la posición de un input/output.

Identificador	RF-8
Título	Dibujar una escena 3D
Necesidad	Esencial
Descripción	El sistema debe dibujar una escena en 3D.

Identificador	RF-9
---------------	------

Titulo	Modificar los parámetros de luz
Necesidad	Esencial
Descripción	El sistema deberá permitir mediante controles visuales cambiar los parámetros de la luz: Color (tres campos numéricos para valores R, G y B), intensidad (un campo numérico), intensidad difusa e intensidad ambiental (un campo numérico).

Identificador	RF-10
Titulo	Cambiar de modelo a dibujar
Necesidad	Deseable
Descripción	El sistema deberá permitir cambiar el modelo que se está dibujando por uno del sistema, en ningún caso un modelo elegido por el usuario.

Identificador	RF-11
Titulo	Refrescar escena 3D
Necesidad	Esencial
Descripción	El sistema deberá redibujar la escena 3D cada vez que se genere un Shader nuevo o que se interactúe con la misma.

Identificador	RF-12
Titulo	Interactuar con el visor 3D
Necesidad	Escencial
Descripción	<p>El visor 3D debe permitir al usuario el poder girar el modelo en los ejes Y (vertical) y X (horizontal) mediante una acción de clic y arrastre del ratón.</p> <p>Además se debe permitir alejar o acercar la cámara con una acción de clic derecho y arrastre vertical del ratón y de hacer</p>

	zoom in o zoom out con el movimiento de la rueda del ratón. Todas las acciones deben producirse dentro del visor 3D.
--	---

Identificador	RF-13
Título	Generar Shader
Necesidad	Esencial
Descripción	El sistema deberá permitir generar un Shader a partir de las conexiones hechas sobre el nodo principal. En caso de no tener conexiones se asignaran valores por defecto. Los Shader nuevos se generarán cada vez que surja un cambio en el grafo de nodos del editor de nodos o en las propiedades de los nodos.

Identificador	RF-14
Título	Visualizar el Shader
Necesidad	Esencial
Descripción	El sistema deberá permitir ver el código fuente del Shader en una ventana modal.

Identificador	RF-15
Título	Guardar el Shader
Necesidad	Esencial
Descripción	El sistema deberá permitir guardar el código del Shader como fichero de texto.

Identificador	RF-16
Título	Salir del programa
Necesidad	Esencial
Descripción	El sistema deberá permitir salir de la aplicación.

Identificador	RF-17
Título	Encapsular nodos para crear un nodo personalizado.
Necesidad	Deseable
Descripción	El sistema deberá permitir encapsular funcionalidades agrupando nodos que el usuario seleccione y creando un nodo personalizado que recoja los inputs y outputs de los nodos seleccionados. Este nodo podrá ser usado como cualquier otro nodo para operar en el editor.

### Requisitos no funcionales

Descripción de los requisitos no funcionales.

Identificador	RNF-1
Título	Escalabilidad
Necesidad	Esencial
Descripción	El sistema debe permanecer abierto a posibles mejoras en aspectos como el número de operaciones o el modelo de Shading empleado.

Identificador	RNF-2
Título	Desempeño
Necesidad	Deseable
Descripción	El sistema no deberá en ningún momento superar un tiempo estimado de espera de más de 2 segundos. Este es un tiempo aproximado de lo que puede tardar en resolverse la operación más costosa de la aplicación (Crear el Shader)

Identificador	RNF-3
Título	Facilidad de uso
Necesidad	Esencial
Descripción	El sistema debe ser de fácil uso y no hará falta entrenamiento básico para su utilización. (No incluye entendimiento del proceso de Shading y de los Shaders)

#### 6.2.4 Planificación:

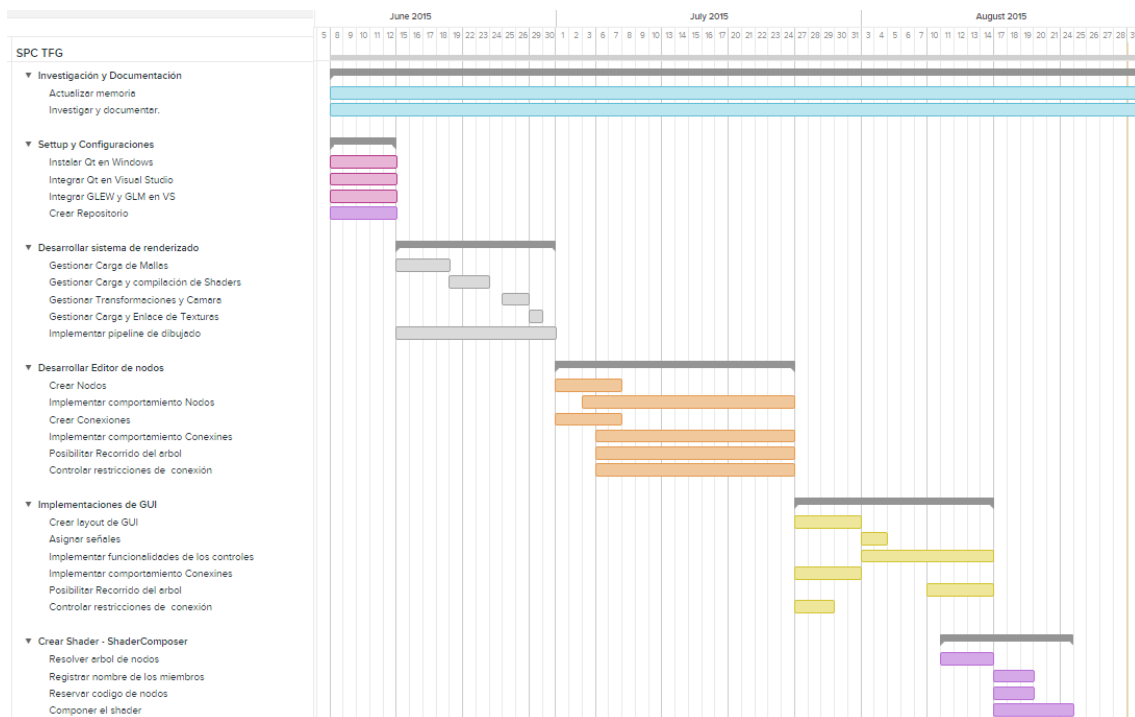
El tiempo de planificación que se ha establecido para el desarrollo del todo el proyecto ha sido desde el mes de Junio hasta el mes de Septiembre. Por lo tanto el tiempo objetivo para terminar el desarrollo y la documentación será de aproximadamente 3 meses.

A continuación se presenta un diagrama de Gantt que refleja de una manera más detallada las tareas (Ilustración 6.12) y los plazos a cumplir para el proyecto. Por problemas de visibilidad primero se presentará la lista de tareas y posteriormente el diagrama de Gantt (Ilustración 6.13).

## SPC TFG

- ▼ Investigación y Documentación
  - Actualizar memoria
  - Investigar y documentar.
- ▼ Setup y Configuraciones
  - Instalar Qt en Windows
  - Integrar Qt en Visual Studio
  - Integrar GLEW y GLM en VS
  - Crear Repositorio
- ▼ Desarrollar sistema de renderizado
  - Gestionar Carga de Mallas
  - Gestionar Carga y compilación de Shaders
  - Gestionar Transformaciones y Cámara
  - Gestionar Carga y Enlace de Texturas
  - Implementar pipeline de dibujo
- ▼ Desarrollar Editor de nodos
  - Crear Nodos
  - Implementar comportamiento Nodos
  - Crear Conexiones
  - Implementar comportamiento Conexiones
  - Posibilitar Recorrido del arbol
  - Controlar restricciones de conexión
- ▼ Implementaciones de GUI
  - Crear layout de GUI
  - Asignar señales
  - Implementar funcionalidades de los controles
  - Implementar comportamiento Conexiones
  - Posibilitar Recorrido del arbol
  - Controlar restricciones de conexión
- ▼ Crear Shader - ShaderComposer
  - Resolver arbol de nodos
  - Registrar nombre de los miembros
  - Reservar código de nodos
  - Componer el shader

**Ilustración 6.12.** Lista de tareas planificadas. (Fuente: <https://teamgantt.com>)



**Ilustración 6.13.** Diagrama Gantt. (Fuente: <https://teamgantt.com>)



El diagrama presenta las tareas planificadas y su duración, así como también su fecha estimada de inicio y de finalización. Las tareas de documentación e investigación están pensadas para que abarquen todo el proceso de desarrollo. También al ser el desarrollo realizado por una persona, la mayoría de tareas no son dependientes de otras dentro de un mismo grupo de tareas.

Todos los tiempos son estimados y han sido calculados mediante la experiencia obtenida en anteriores proyectos y teniendo en cuenta las tareas donde se requiere más trabajo basándose en el conocimiento que se tiene de los diferentes campos de los problemas que se intentan resolver. Por ejemplo, las tareas relacionadas con desarrollar el editor de nodos son más costosas que otros ya que la experiencia con Qt y sus QGraphics es poca.

## 6.3 Diseño e implementación del sistema

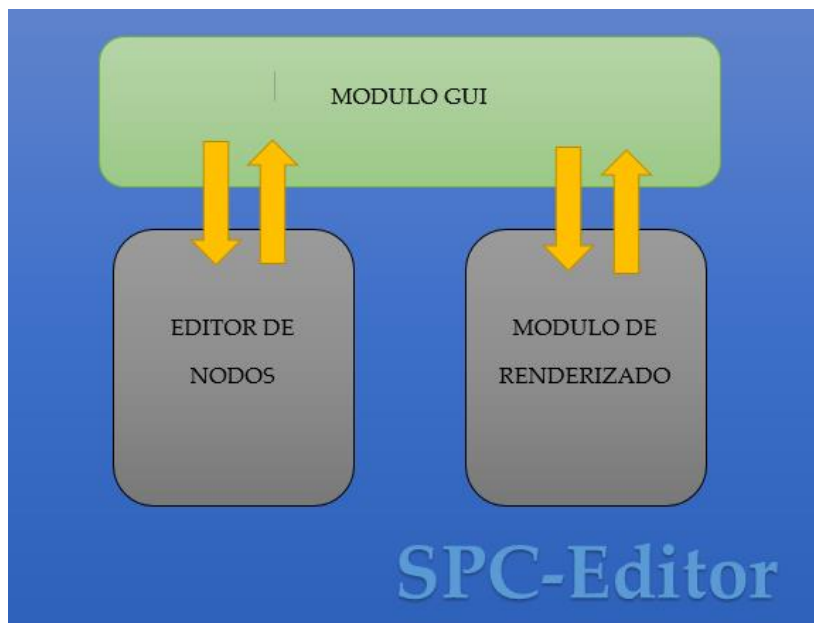
En el siguiente apartado se describirá con detalle todo lo relevante con el diseño y la implementación del sistema, de cómo se ha desarrollado y dado solución a los problemas planteados. Además se hará énfasis en la justificación de las decisiones tomadas a partir de los requisitos y especificaciones expuestos anteriormente.

Arquitectura del programa.

El desarrollo software se basó en el lenguaje de programación C++ bajo un paradigma de programación orientada a objetos.

En términos de patrones de diseño no se siguió uno en específico, aunque se trató siempre de mantener en todo momento una buena clasificación de las clases procurando siempre mantener la cohesión ente objetos y de dividir las problemáticas más complejas en módulos de desarrollo.

Dados los requisitos de la aplicación y la especificación arrojados en la fase de análisis, se deben de resolver tres aspectos funcionalidades importantes. Estas funcionalidades se derivaron en tres módulos de desarrollo principal:

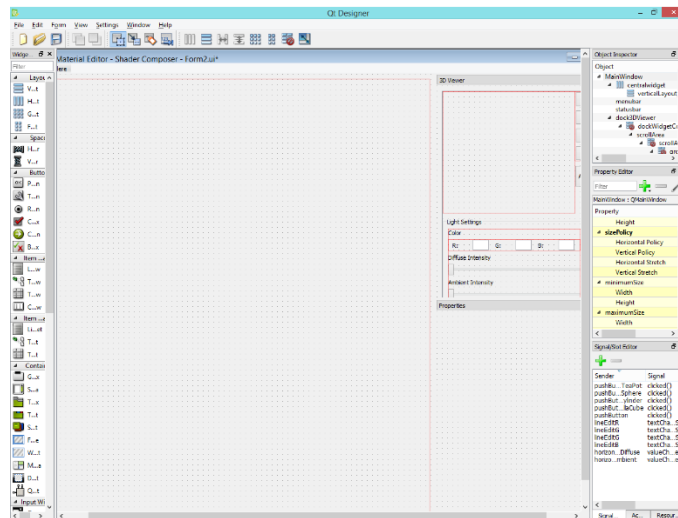


**Ilustración 6.14.** Módulos software de la aplicación. (Fuente: Elaboración propia)

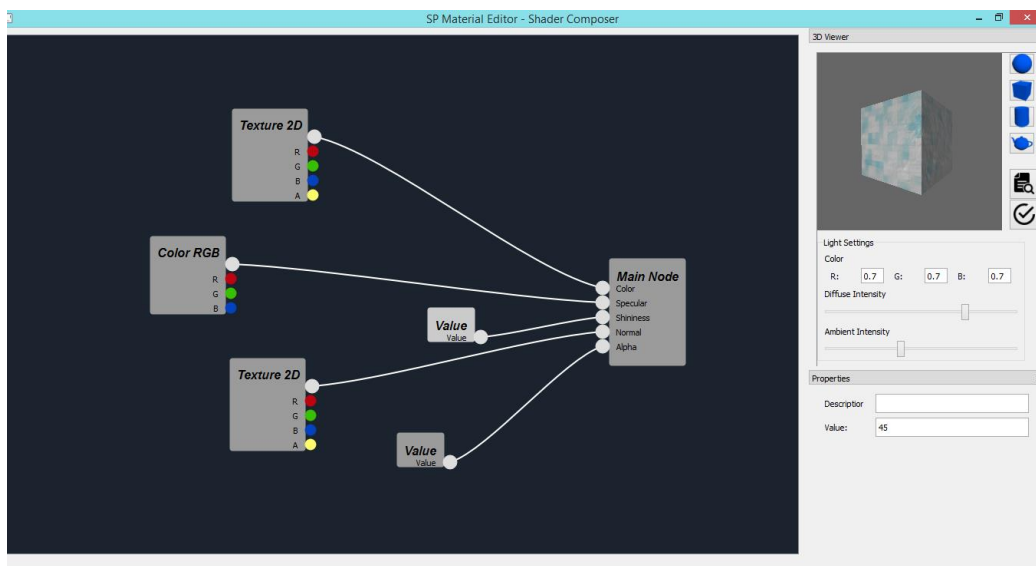
### 6.3.1 Modulo GUI:

Este módulo se encarga de recibir las entradas del usuario y procesarlas. El desarrollo se llevó a cabo principalmente mediante Qt.

Desde el diseñador de Qt se diseñó y creó un archivo .ui (Qt user interface file). En este archivo se especificó la maquetación de la ventana y los distintos espacios de trabajo y secciones del programa.











**Ilustración 6.15.** Diseño de la ventana en Qt Designer. (Fuente: Elaboración propia)



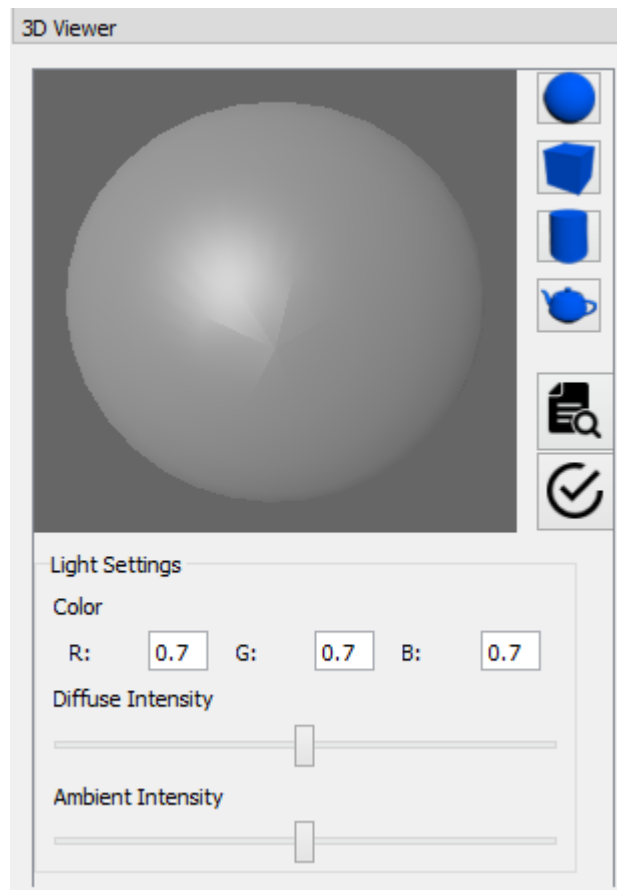
**Ilustración 6.16.** Diseño de la interfaz de usuario. (Fuente: Elaboración propia)

En este fichero se definieron las secciones del programa que son: espacio para el editor de nodos, sección del visor 3D (Ilustración 6.17) y panel de propiedades.

A continuación se describirán todos los controles visuales que posee la interfaz gráfica.

	Botón para cambiar el modelo a esfera.
	Botón para cambiar el modelo a Cubo.
	Botón para cambiar el modelo a cilindro.
	Botón para cambiar el modelo a tetera.
	Botón para ver Shader generado. Abre una ventana modal donde se puede ver el Shader.
	Botón para aplicar los cambios realizados. Se usa para refrescar el visor 3D. Cada vez que se pulsa se genera un nuevo Shader.
<p>Color</p> <p>R: <input type="text" value="0.7"/> G: <input type="text" value="0.7"/> B: <input type="text" value="0.7"/></p>	Estos tres inputs establecen el color de la luz de la escena. El valor es modificado al introducir datos, es decir, no hacer falta pulsar otro botón para hacer efectivo el cambio.
<p>Diffuse Intensity</p> 	Slider para graduar la intensidad de la luz para la componente difusa.
<p>Ambient Intensity</p> 	Slider para graduar la intensidad de la luz para la componente ambiental.

**Tabla 6.1.** Descripción de los controles visuales. (Fuente: Elaboración propia)



**Ilustración 6.17.** Interfaz de usuario de la sección “Visor 3D”. (Fuente: Elaboración propia)

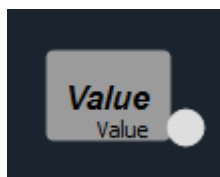
### **Capturando eventos con Qt.**

Qt permite de forma sencilla capturar todos los eventos a disposición un objeto. El método más cómodo desde el punto de vista de desarrollo es asignar lo que se denomina signals/slots para definir la captura de los eventos. Un signal de Qt es un acontecimiento que emite una señal. Cuando una signal emite su señal, todos aquellos que estén suscritos a esta recibirán la notificación. Cuando una signal es asignada a un slot, también se le asigna un código fuente que es el que finalmente se ejecutará cuando la signal emita su señal. Este enfoque fue utilizado mayormente desde el diseñador Qt ya que éste posibilita la asignación de signals y slots de una forma realmente sencilla.

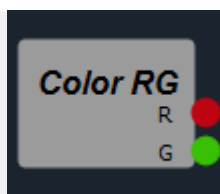
Otra posibilidad para capturar eventos con Qt es mediante la generalización de las clases que implementan los controles y sobrescribiendo las funciones miembro que se llaman cuando se invoca un evento. Para determinados objetos esta opción fue la utilizada en ausencia de la posibilidad de incluir el método de signals/slots.

## Diseño gráfico de del editor de nodos.

Si bien los nodos tienen una especificación muy clara, el diseño visual final es ligeramente diferente al de los mockups del apartado análisis y especificación. A continuación se listarán los diseños resultantes de los nodos para algunos tipos de nodos clasificando por número de entras y de salidas:



Diseño de un nodo con un output.



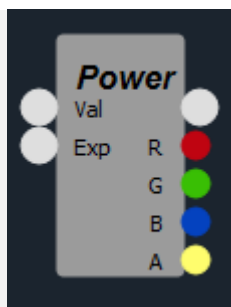
Diseño de un nodo con dos outputs.

El color de los inputs determina la componente que devolverá (R y G).



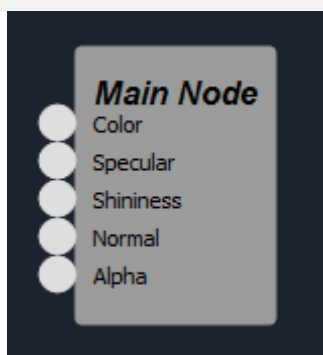
Diseño de un nodo con dos outputs.

El color de los inputs determina la componente que devolverá (R, G y B).



Diseño de un nodo con dos outputs y dos inputs.

El color de los inputs determina la componente que devolverá (R, G, B y A).



Diseño del main node. Este posee 5 inputs y ningún output.

**Tabla 6.2.** Diseño final de nodos. (Fuente: Elaboración propia)

### 6.3.2 Módulo de renderizado.

El módulo de renderizado es el encargado de dibujar los modelos 3D en el visor. Para dibujar se ha usado OpenGL bajo un contexto de dibujado provisto por Qt. El módulo de renderizado cubre las opciones básicas de dibujado como transformaciones, carga de modelos etc.

Una funcionalidad importante de este módulo es que debe ser capaz de cargar shaders para el dibujado no solo al iniciar la aplicación sino que también debe ser capaz de cargarlos durante su ejecución, creando nuevos Shaders y destruyendo antiguos. En caso de que un Shader no sea compilado correctamente, bien sea porque tienen errores o por problemas con la tarjeta gráfica, el sistema debe dibujar un Shader provisional y no dejar de dibujar en ningún momento.

A este punto cabe describir las pautas, la forma y la estructura de los Shaders aquí implementados. Además de describir la información que se le debe suministrar si se desea incorporar en algún otro sistema.

#### **Estructura de Shaders**

Los Shaders usados e implementados por el programa no son muy diferentes del estándar. Es realmente importante explicar cómo están diseñados, no para exponer el cómo están implementados sino para que todo aquel que quiera integrarlos sepa que información suministrarles de forma sencilla, lo que constituye uno de los objetivos primordiales de este proyecto.

#### **Comunicando información a los shaders.**

Para comunicar información con los shaders que se usan en el pipeline de OpenGL, se le deben pasar ciertos parámetros de funcionamiento básicos primero. Comunicar esta información es sencillo mediante las funciones que OpenGL deja a disposición. Los tipos de variables que se le envían a los shaders son llamadas **uniforms** y se declaran en los ficheros de los shaders (.vs y .fs para las convenciones de este proyecto), y se establecen en el programa principal. Se sobreentiende que todo aquel interesado en integrar los shaders generados por la aplicación tengan un método propio para gestionar los envíos; no obstante, si no se tiene un método, en el repositorio del proyecto se encuentran todo el

código fuente donde se incorporan colecciones de funciones para enviar información a los Shaders de todo tipo (vec3, vec4, float, etc.).

#### **Nota sobre texturas:**

Las texturas que sean usadas para dibujar en la aplicación deberán ser sincronizadas para su funcionamiento en entornos donde se vaya a integrar algún Shader generado por esta aplicación. Así si una textura (sampler2D) está siendo usada por un Shader, esta misma deberá ser sincronizada como tal en el entorno a integrar. De la misma forma que pasa con el resto de uniforms.

### **6.4 Vertex Shader**

El Vertex Shader usado por el programa es prácticamente inalterable. Este Shader solo se ocupa calcular la posición de los vértices a *Clip Space* y llevar los vectores de dirección de la luz, dirección de la vista y normal al sistema de coordenadas tangencial para enviarlas al Fragment Shader. El motivo para el cual se usar un sistema de coordenadas tangencial es que es necesario para aplicar la técnica de normal mapping (ver apartado Estado del arte y contexto).

Las uniforms necesarias para integrar el Vertex Shader son:

Nombre de la uniform	tipo	Descripción
<b>ModelViewMatrix;</b>	mat4	Esta matriz es la encargada de llevar el vértices coordenadas de la view es decir, la “cámara”
<b>NormalMatrix</b>	mat3	Esta matriz es la encargada de transformar las normales del vértice y llevarlas a coordenadas del objeto. La matriz Normal suele ser calculada como la inversa transpuesta de la matriz Model (transformaciones del modelo)
<b>ProjectionMatrix</b>	mat4	Esta matriz se encarga de proyectar los vértices sobre un plano imaginario. Para calcular esta matriz se debe componer una con los datos de FOV, de los planos z-near y z-far, y el aspect ratio. La librería GLM proporciona



		herramientas para construir fácilmente una matriz de proyección tanto de perspectiva como ortogonal.
<b>MVP</b>	mat4	Esta matriz es la encargada de calcular la transformación final que se le pasará al FragmentShader. Para calcular la matriz MVP se debe concatenar la multiplicando las matrices M, V y P. Por lo tanto $MVP = P * V * M$
<b>LightDirStatic</b>	vec3	Dirección de la luz en coordenadas del mundo.

**Tabla 6.3.** Información necesaria para el funcionamiento del Vertex Shader de la aplicación. (**Fuente:** Elaboración propia)

Además de las uniforms que este Shader requiere, también hace falta que los atributos de vértice estén establecidos de forma correcta. Los atributos para vértices que requiere este Shader son:

Nombre del atributo	tipo	Descripción
<b>VertexPosition</b>	vec3	Guarda la posición del vértice como tal.
<b>VertexTexCoord</b>	vec2	Guarda las coordenadas UV del vértice.
<b>VertexNormal</b>	vec3	Guarda la normal del vértice.
<b>VertexTangent</b>	vec3	Guarda la tangente del vértice.

**Tabla 6.4.** Atributos de los vértices empleados por el VertexShader. (**Fuente:** Elaboración propia)

Todos estos valores son típicamente suministrados por programas de edición 3D a la hora de exportar los modelos.

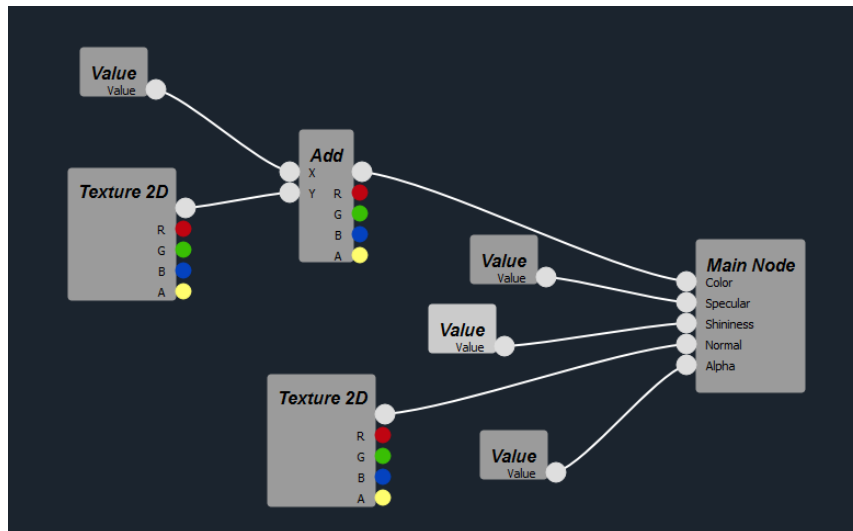
## 6.5 Fragment Shader

Los Fragment Shader son los Shaders que la aplicación va a generar a causa de la interacción con los nodos. La es código fuente generado en este Shader sigue el siguiente esquema:

Bloque	Descripción
<b>1. Declaraciones estáticas</b>	Son partes del código estáticas que describen elementos necesarios para la ejecución del Shader.
<b>2. Declaración de Constantes</b>	Aquí se ubican las representaciones en código de los nodos de tipo constantes (float, vec2, vec3 y vec4). La decisión de declarar estos miembros como globales constantes del Shader es que de esta forma se pueden reutilizar desde cualquier parte del código.
<b>3. Declaración de Texturas</b>	En esta sección se declaran las variables de tipo sampler2D que representan nodos de tipo textura y que se encuentran en el editor de nodos.
<b>4. Declaración de Funciones</b>	Los nodos que representan operaciones se escriben en esta sección. Las operaciones son creadas como funciones personalizadas y la gran mayoría hace llamadas a funciones integradas de GLSL.
<b>5. Función de Blinn-Phong</b>	Se trata de la definición de la función que realiza el Shading de Blinn-Phong.
<b>6. Main</b>	La declaración del main es realizada en esta sección. El main solo recoge los parámetros que son los nodos conectados al nodo Main, y posteriormente llama a la función de Blinn-Phong.

**Tabla 6.5.** Bloques que estructuran el Fragment Shader. (Fuente: Elaboración propia)

Para ilustrar de una forma más grafica la creación de los nodos a continuación se situara un ejemplo de material hecho con el editor de nodos (Ilustración 6.18) y el Fragment Shader generado (Tabla 6.6).



**Ilustración 6.18.** Ejemplo de material creado con el editor de nodos. (Fuente: Elaboración propia)

```
#version 400

in vec3 VexterPosEye;
in vec2 TexCoord;
in vec3 ViewDir;
in vec3 LightDirStaticTan;

uniform vec3 LightColor;
uniform float LightDiffuseIntensity;
uniform float LightAmbientIntensity;
layout(location = 0) out vec4 FragColor;

// Const Section
const float QNConstFloatNode0 = 0.3;

const float QNConstFloatNode1 = 1;

const float QNConstFloatNode2 = 1;

const float QNConstFloatNode3 = 40;

// Textures Section
uniform sampler2D QNTextureNode0;

uniform sampler2D QNTextureNode1;

// Functions Section
vec4 QNAddNode0 ()
{
    vec4 A = vec4(QNConstFloatNode0);
    vec4 B = vec4(texture(QNTextureNode0, TexCoord));
    return A + B;
}

vec3 BlinnPhong(vec3 diffR, vec3 norm, vec3 specularLvl, float Shininess)
{
    // this is blinn phong
    vec3 h = normalize(LightDirStaticTan + ViewDir);
```

```

    vec3 ambient = (LightAmbientIntensity * LightColor);

    float sDotN = max(dot(LightDirStaticTan, norm), 0.0);

    vec3 diffuse = (LightDiffuseIntensity * LightColor) * diffR * sDotN;

    vec3 spec = vec3(0.0);

    if (sDotN > 0.0)
        spec = LightColor * specularLvl * pow(max(dot(h, norm), 0.0),
Shininess);

    return ambient + diffuse + spec;
}
void main()
{

    vec4 colorBase = vec4(QNAddNode0());
    vec4 specularLvl = vec4(QNConstFloatNode1);
    vec4 normal = 2 * vec4(texture(QNTextureNode1, TexCoord))-1;
    vec4 alpha = vec4(QNConstFloatNode2);
    vec4 lightIntensity = vec4(0.0);
    float shininess = QNConstFloatNode3;

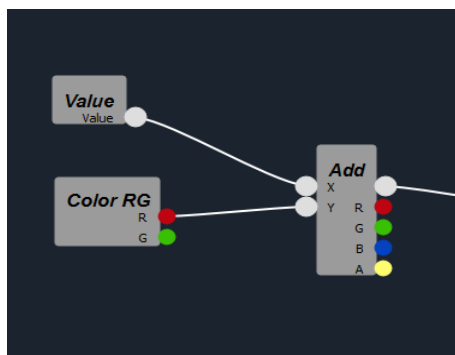
    lightIntensity += BlinnPhong(colorBase.rgb, normal.rgb,
specularLvl.rgb, shininess);
    FragColor = lightIntensity;
    FragColor.a = alpha.r;
}

```

**Tabla 6.6.** Ejemplo de Shader Generado. Resaltadas aparecen cada uno de los bloques que componen el Shader. (Fuente: Elaboración propia)

### 6.5.1 Resolviendo el grafo de nodos.

Probablemente se trate de la tarea más importante de todo el proyecto ya que aquí es donde se vuelca toda la información relacionada con los nodos en un FragmentShader. El flujo de las conexiones realizadas en el editor representa el flujo del código del programa. Por ejemplo una operación como la de Suma (nodo Add) dispuesta de la siguiente manera reflejara una función donde sus operandos serán los correspondientes miembros que representan los nodos conectados:



El nodo Add creará una función en el Shader, los parámetros para calcular su operación son los provisto por los nodos Value (float) y ColorRG(vec2):

```
// Functions Section
vec4 SPC_AddNode0 ()
{
    vec4 A = vec4 (SPC_ConstFloatNode0);
    vec4 B = vec4 (SPC_Vector2DNode0.r);
    return A + B;
}
```

La tarea de resolver el grafo y crear el Fragment Shader representa es el foco principal del proyecto y por ello cabe describir su desarrollo y forma de implementación.

#### Implementación.

Cada nodo representa una clase en términos de programación. Todos los nodos son una especialización de una clase general donde se definen de forma virtual comportamientos básicos; de esta forma el programa puede llamar cualquier nodo que sea una especialización del padre sin tener que saber a quién se refiere (principio de polimorfismo). Esto permite resolver el problema del grafo de una forma más sencilla.

#### Shader Composer.

El Shader Composer es una clase estática que permite almacenar las declaraciones de forma

global que se están llevando a cabo por los nodos en el momento de resolver el grafo. Es importante porque se encarga de administrar las inclusiones de código y de almacenar el nombre de cada miembro que genera el nodo para el programa GLSL y relacionarlo con el nodo al que corresponde. La tupla `IdNodo`, `NombreMiembro` son almacenados por el Shader Composer.

Finalmente el Shader Composer recoge en orden todas las declaraciones de código que generan los nodos y posteriormente genera el código fuente del Shader concatenando en orden cada bloque dinámico junto con los estáticos que son inalterables.

## 6.6 Explicación del algoritmo.

Todos los nodos poseen una función llamada ***Resolve***. El programa principal inicia el proceso llamando la función *Resolve* del nodo principal.

La función *Resolve* de cada nodo realiza las siguientes operaciones:

- 1) Registrar el nodo en el ShaderComposer, una vez hecho esto, el nodo ya tiene un nombre dentro del código fuente asignado. De esta forma si el nodo es solicitado para operar dentro de otras operaciones de otros nodos, el nombre que ha obtenido es usado para hacer referencia a sí mismo.
- 2) Por cada input que posea el nodo:
  - Se deberá obtener el nombre del miembro que representa el determinado input. Para obtener este nombre se llama a la función *Resolve* del nodo conectado a dicho input haciendo uso de la recursividad. Todas las funciones *Resolve* de los nodos retornan el nombre en del miembro en el código fuente.
- 3) Cuando ya se tienen los nombres de todos los miembros relacionados con los inputs conectados, se pasa a generar el código fuente del miembro en sí. Por ejemplo si se trata de una función este genera el código de la función y donde se opera usará los nombres obtenidos en los pasos anteriores para concatenar las asignaciones pertinentes dentro de la cadena del código fuente.

- 4) Registrar el código generado dentro del ShaderComposer. El ShaderComposer almacena el Id del nodo y el código que ha generado dicho nodo.
- 5) Al finalizar el proceso se devuelve su propio nombre para que así se cierre el bucle de la recursividad y los demás nodos que solicitan este nodo tengan la información del nombre de este nodo dentro del código.

De esta manera es como se realiza un bucle para recorrer el grafo mientras se van registrando en orden todos los nombres y códigos de los nodos encontrados. El orden lo establecen las conexiones y de esta forma cada nodo sabe a qué miembro operar cuando debe componer sus operaciones.

## 6.7 Acerca de los tipos compatibles.

Uno de los mayores quebraderos de cabeza durante la etapa de diseño y que finalmente se resolvió de forma bastante natural durante la etapa de implementación, fue el hecho de que todos los nodos representan un tipo de datos dentro del GLSL que deben de coincidir para evitar problemas de compilación. Esto quiere decir que existen operaciones dentro de GLSL que no se puede realizar, y si se intentan, generaran un error de sintaxis en el momento de compilar el Shader. Por ejemplo:

```
// Const Section
const float SPC_ConstFloatNode0 = 0.3;

const vec2 SPC_Vector2DNode0 = vec2(0, 0);

// Functions Section
vec4 SPC_AddNode0 ()
{
    vec4 A = SPC_ConstFloatNode0;
    vec4 B = SPC_Vector2DNode0.r;
    return A + B;
}
```

El anterior código fuente da un error de compilación ya que se intenta asignar a dos miembros –A y B- de tipo vec4, dos variables de tipo float y vec2 correspondientemente.

La solución a este problema radica en la forma en que GLSL gestiona los tipos de datos.

GLSL permite crear datos usando constructores sobrecargados que permite que diferentes datos sean usados para crear otros tipos de datos. La cuestión fue encontrar el tipo de dato que más información pueda almacenar y que mejor se adapte para las necesidades del programa, y a partir de ahí usar este tipo de dato para globalizar toda la aplicación. Es por este motivo por el que todas las funciones y los datos del programa son almacenados y creados como datos de tipo vec4. El tipo vec4 almacena todas las componentes de color y además sus constructores sobrecargados permiten crear otros vec4 a partir de otros vec4. De esta forma todos los datos del programa pueden ser compatibles si se meten en un constructor de tipo vec4. De la misma, los datos de valores de componentes también pueden ser compatibles con el resto de la aplicación ya que el tipo vec4 posee un constructor que se basa en un parámetro de tipo float para componer un vector en escala de grises. Este método es el usado cuando el programa tiene conexiones con componentes de color como las de los inputs R, G, B A que hay por la aplicación o bien por los tipos de nodo float.

De esta forma el anterior ejemplo se puede resolver usando los constructores sobrecargados de los vec4:

```
// Const Section
const float SPC_ConstFloatNode0 = 0.3;

const vec2 SPC_Vector2DNode0 = vec2(0, 0);

// Functions Section
vec4 SPC_AddNode0 ()
{
    vec4 A = vec4(SPC_ConstFloatNode0);
    vec4 B = vec4(SPC_Vector2DNode0.r);
    return A + B;
}
```



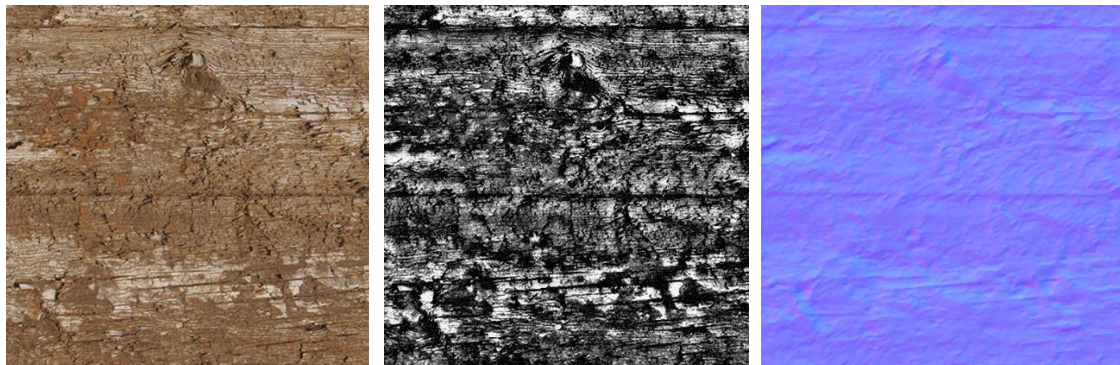
## 7 Experimentación

A este punto es conveniente poner a prueba el conjunto de aplicación, por lo tanto, a continuación se realizarán algunos ejercicios de experimentación basados en la realización de algunos ejemplos con la aplicación. Los resultados serán analizados y descritos y también se hará una comparación para algunos ejemplos, guardando las distancias con algunos de los productos existentes en el mercado.

### 7.1 Creando un material básico.

El siguiente ejemplo ilustra la creación básica de un material usando mapas de texturas para los parámetros de Color, Normal y Specular.

Las texturas usadas son las siguientes:



Mapa difuso

Mapa Especular

Mapa Normal

**Ilustración 7.1.** Mapas de textura empleados en ejemplo1. (**Fuentes:** difuso: <http://www.cgtextures.com/>, especular y normal generados usando CrazyBump a partir del difuso)

#### 7.1.1 Implementación:

La implementación consta básicamente en la creación de tres nodos de tipo Texture donde se eligen las imágenes que se van a usar. La disposición del grafo quedaría de la siguiente forma:

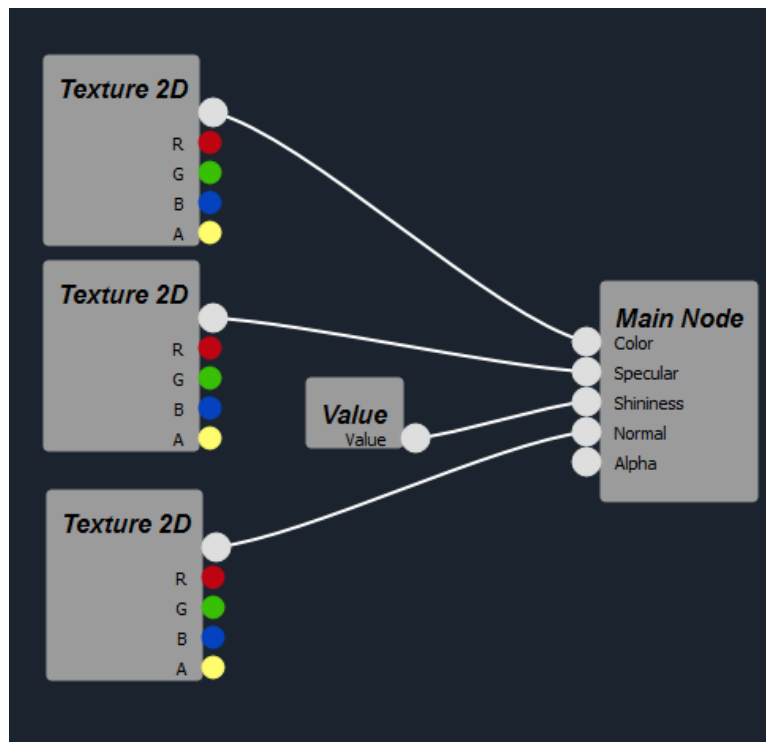
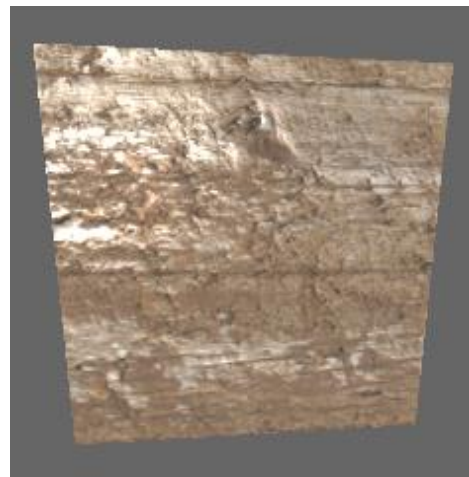
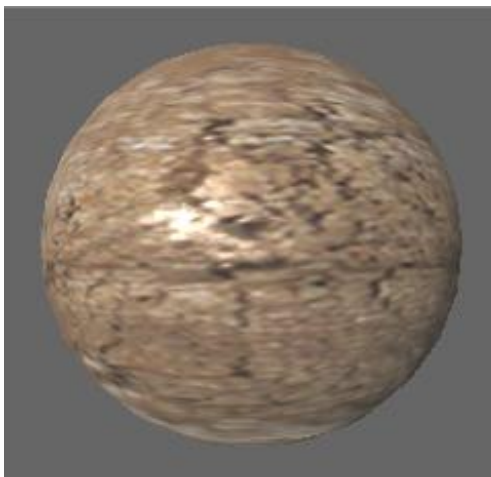


Ilustración 7.2. Implementación del ejemplo 1. (Fuente: Elaboración propia)

El resultado arrojado es un material donde actúan los tres mapas suministrando los parámetros correspondientes.



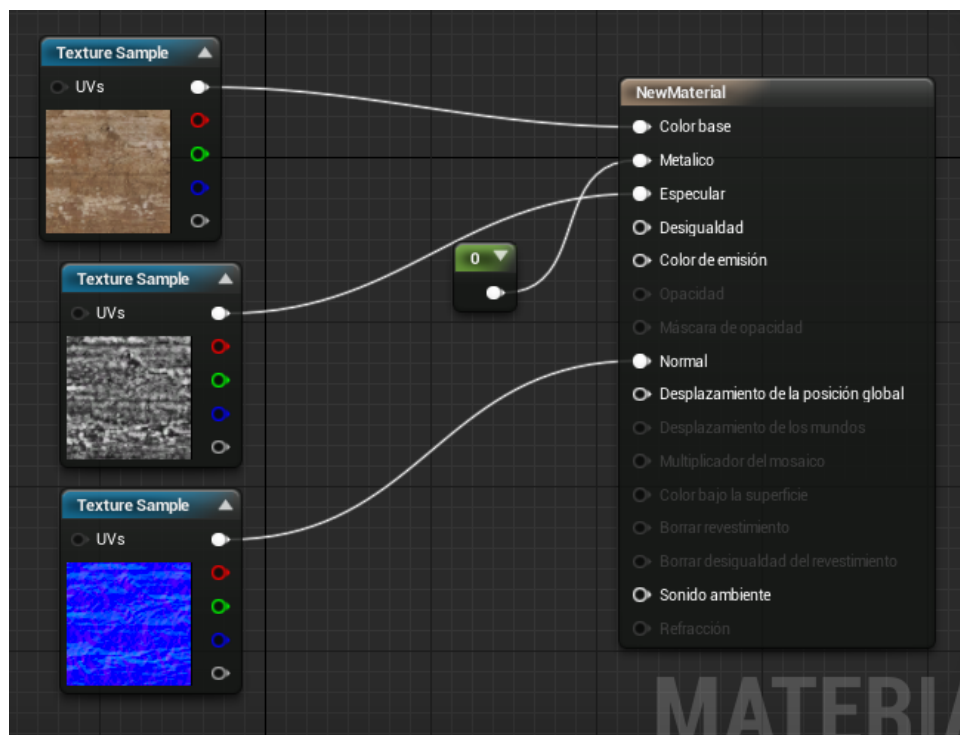


**Ilustración 7.3.** Resultados obtenidos del ejemplo 1 dibujados sobre los modelos de prueba. (Fuente: Elaboración propia)

Como punto a destacar cabe mencionar el realismo que se puede conseguir con el sencillo modelo de Blinn-Phong simplemente contando con piezas de arte bien elaboradas como las texturas elegidas.

Un aspecto clave para conseguir realismo es el contar con un buen mapa de normales. Este mapa incrementa de una manera considerable un aspecto mucho más realista.

### 7.1.2 Implementado en UE4.



**Ilustración 7.4.** Implementación del ejemplo 1 en UE4. (Fuente: Capturado desde UE4)

La implementación análoga en UE4 del material (Ilustración 7.4) creado con la aplicación es en esencia muy parecida. Se crean 3 nodos para los mapas y se asignan a los inputs correspondientes.



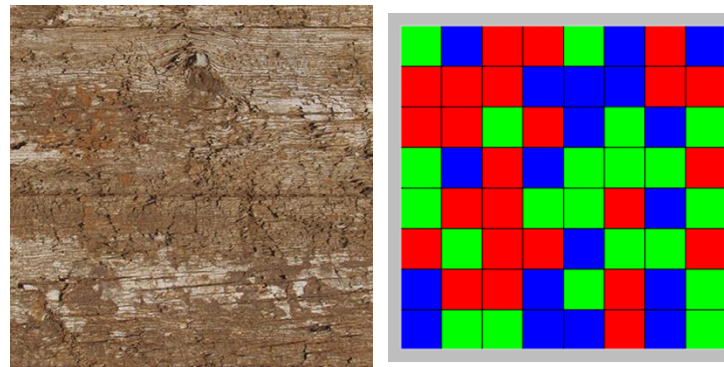
**Ilustración 7.5.** Resultados obtenidos del ejemplo 1 con UE4. (Fuente: Capturado desde UE4)

Los resultados (Ilustración 7.5) a primera vista son muy similares. Como cabe esperar la calidad visual que posee UE4 es muy buena comparada con la aquí desarrollada.

## 7.2 Creando materiales usando mascarar

La técnica de usar mascarar para crear diversos resultados es muy común en el campo del Shading. Las máscaras permiten elegir partes determinadas de las imágenes para realizar operaciones que a su vez permiten alterar determinadas secciones de los materiales.

A continuación se muestran las texturas a emplear en este ejemplo:

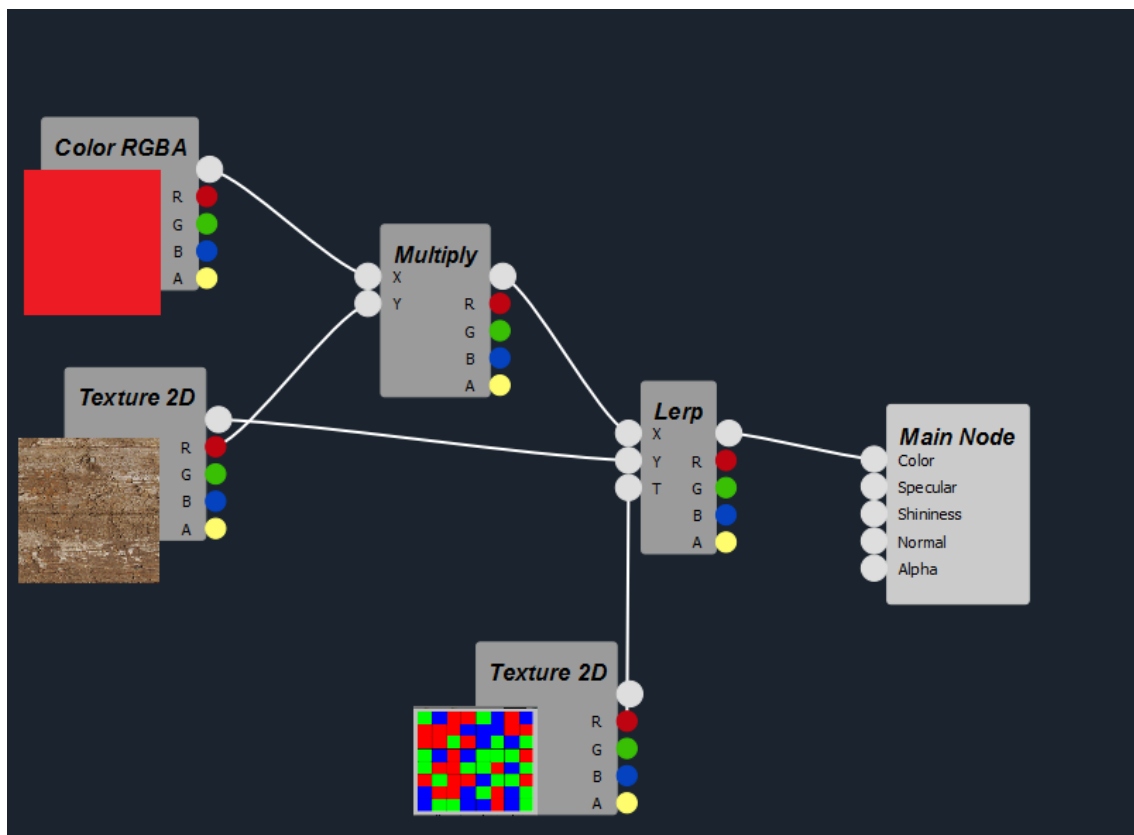


Mapa difuso

Mapa de mascaras

**Ilustración 7.6.** Mapas de texturas empleados en el ejemplo 2. (Fuente: Mapa difuso: <http://www.cgtextures.com/> | Mapa de mascaras: <http://www.cs.cmu.edu/~tcortina/15110sp12/lab11/> )

## 7.2.1 Implementación



**Ilustración 7.7.** Implementación del ejemplo 2. Disposición de los nodos para crear un material usando mascaras. Para mayor comodidad se ha puesto la imagen correspondiente a la textura al lado de los nodos que corresponden para ilustrar mejor el ejemplo, así como también del vector 4 usado (imagen roja). (Fuente: Elaboración propia)

En este ejemplo se emplea enmascaramiento. Para explicar la implementación hay que situarse al inicio de la imagen. Primeramente se sitúan un color rojo, con una imagen de

madera. A continuación se realiza una multiplicación entre el color rojo y la componente R de la textura. Esta componente devuelve una imagen de la textura en escala de grises donde la presencia de rojo constituye la intensidad del gris. Como resultado de la multiplicación se obtiene una imagen mezclada entre rojo y los detalles de la madera:

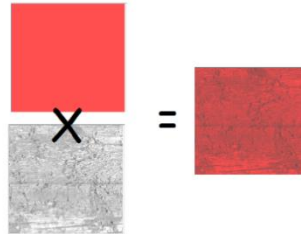
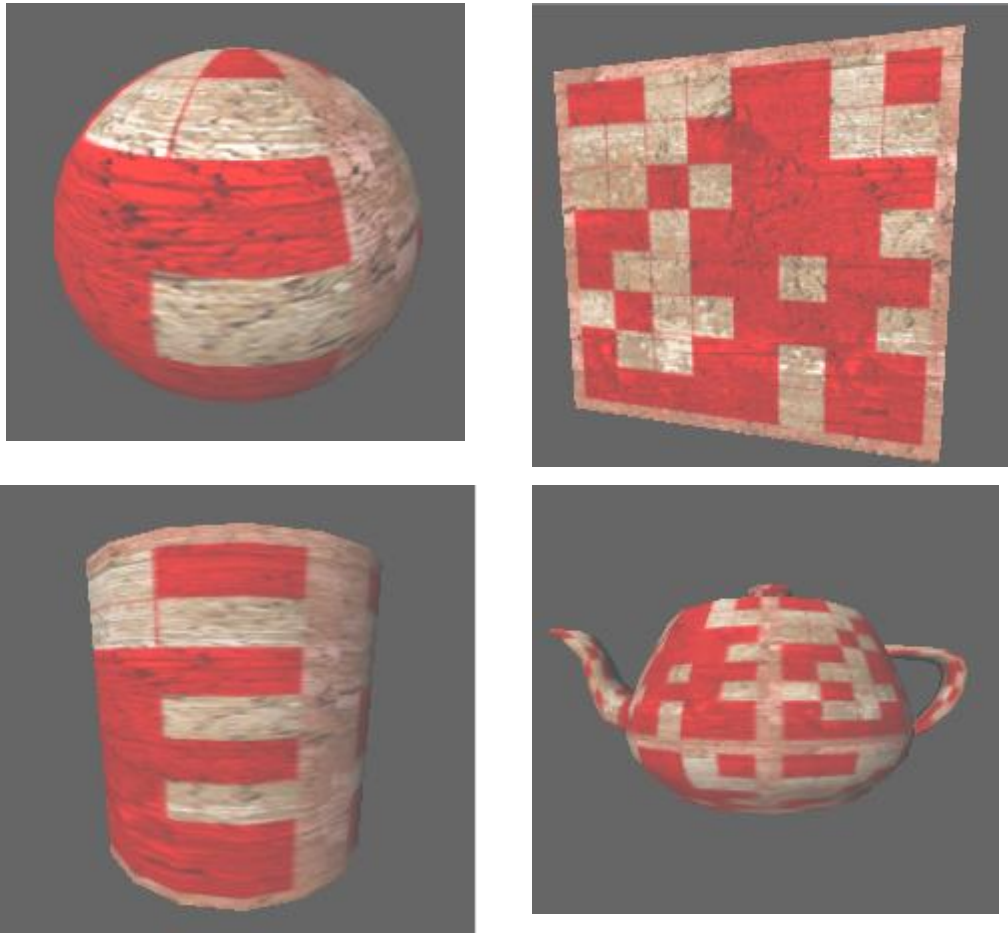


Ilustración 7.8 Resultado de la multiplicación entre el componente R de la textura y el color (Color RGBA).  
(Fuente: Elaboración propia)

Una vez obtenida esta imagen se pasa a crear una interpolación mediante el nodo de interpolación Lerp. Este nodo cuenta con 3 parámetros, en el primero se especifica el valor inicial, en el segundo el final y el tercero es el porcentaje. Cuando el porcentaje valga 1, el resultado arrojado será la textura resultado de la anterior multiplicación (una madera rojiza), mientras que si el valor porcentaje es 0, el valor a tomar es el de la madera. El enmascaramiento llega cuando para el valor de porcentaje le pasamos como parámetro una componente de color de la imagen que estamos usando como máscara. Esta imagen en particular posee niveles altos por partes en las componentes RGB por lo que si devolvemos una de sus componentes, la imagen a escala de grises que dará será una con tonos muy claros (casi blanco) en los sitios de la imagen donde se encuentra tal componente. Así si cogemos la componente roja (R), la imagen resultado será una imagen donde todos los valores son blancos cuando hay rojo y negros cuando hay ausencia de rojo. Con estas permutaciones de 1 y 0 podemos generar un buen valor de porcentaje para la operación Lerp.

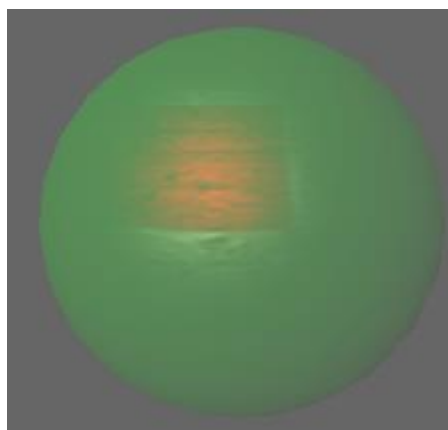
El resultado final dará un aspecto cuadriculado con “parches” de madera de color rojo. Ya que es “madera roja” el 0% y madera original el 100%.





**Ilustración 7.9.** Resultados obtenidos del ejemplo 2. Enmascarando texturas y colores. (**Fuente:** Elaboración propia)

En este ejercicio se trabaja solo sobre el canal difuso, no obstante la misma idea se puede aplicar para crear efectos similares para todos los demás canales:



**Ilustración 7.10.** Misma técnica aplicada para el canal Specular. Solo se ve el resultado del enmascaramiento cuando hay brillos especulares. *Nota:* Para incrementar el efecto se estableció un color verde plano para la componente difusa. (**Fuente:** Elaboración propia).

## 7.2.2 Implementación con UE4.

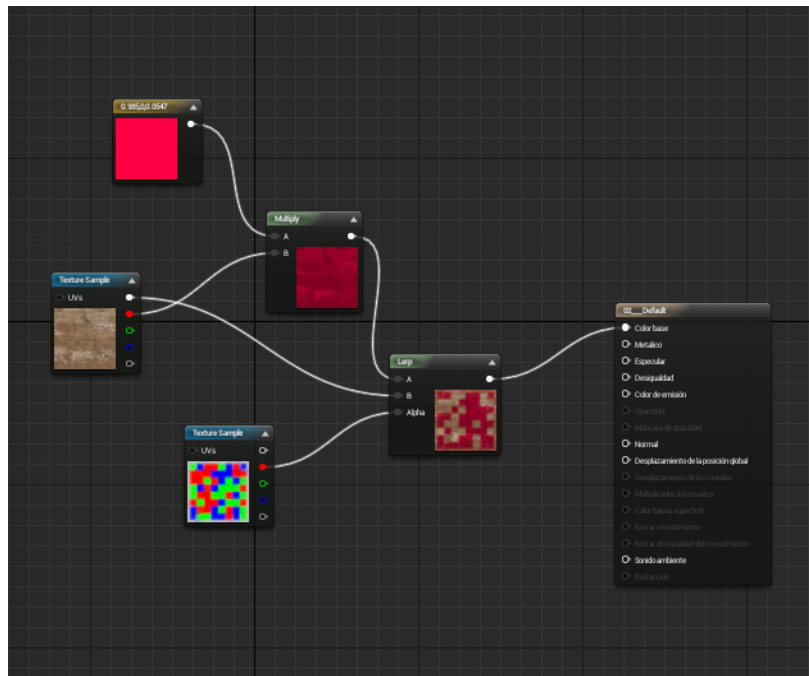


Ilustración 7.11. Implementación del ejemplo 2 en UE4. (Fuente: Capturado desde UE4).

Como se puede observar, la implementación con UE4 no guarda muchas diferencias en cuando a estructura. En ejercicios como este queda en evidencia la utilidad de tener pre visualizaciones de las operaciones mientras se trabaja.

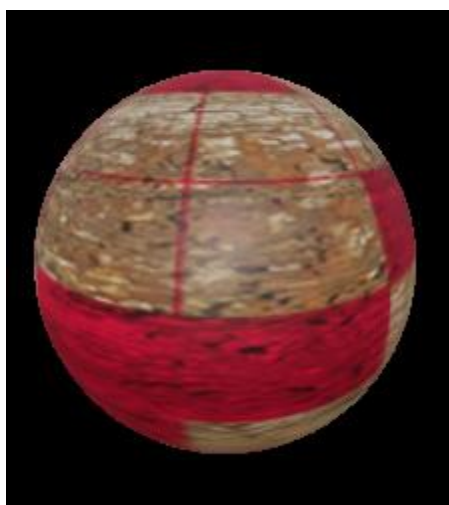


Ilustración 7.12. Resultado obtenido del ejemplo2 con UE4. (Fuente: Capturado desde UE4)

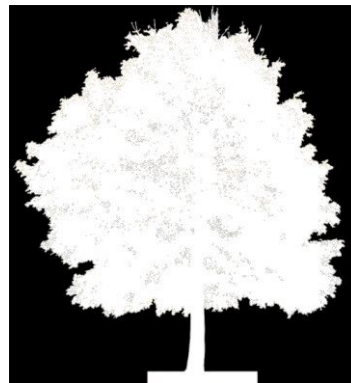


## 7.3 Crear materiales con transparencia.

El ejemplo a continuación descrito resulta útil cuando se intentan simular objetos complejos bajo un plano estático. El efecto de dibujar sobre un plano que siempre está orientado hacia la cámara se denomina Billboarding. Un mapa de textura de alpha en combinación con el Billboarding, son elementos frecuentes a la hora de aplicar efectos como bosques, explosiones, efectos de partículas etc. A continuación se mostrará cómo crear un material con áreas transparentes usando clipping en los valores del canal alpha.



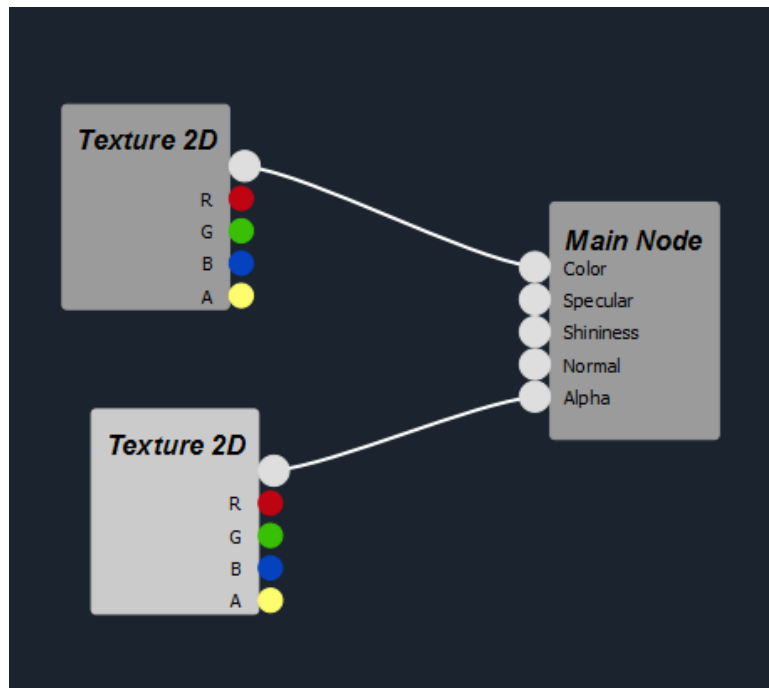
Mapa difuso



Mapa alpha

**Ilustración 7.13.** Mapas usados en este ejemplo. (Fuente: Mapa difuso: <http://www.cgtextures.com/> Mapa alpha: Elaboración propia a partir del difuso)

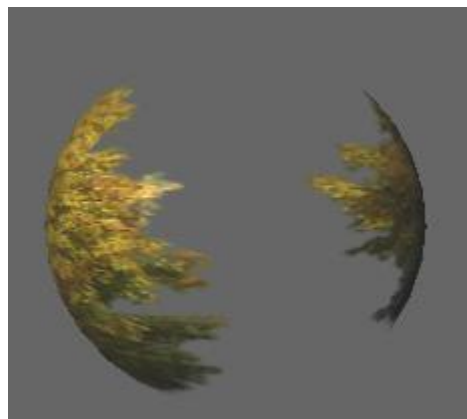
### 7.3.1 Implementación

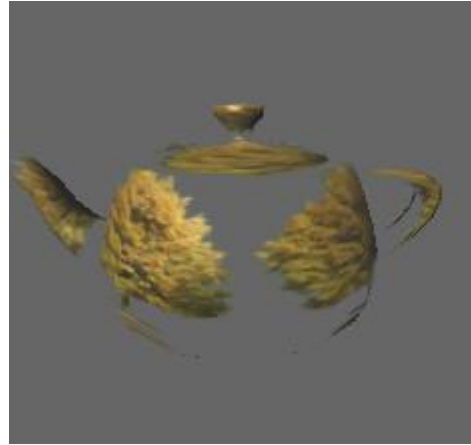
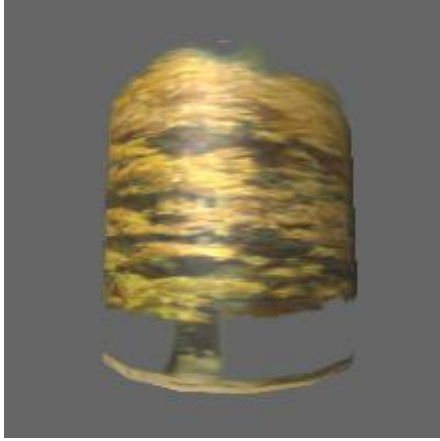


**Ilustración 7.14.** Implementación del ejemplo 3 aplicando alpha clipping. (**Fuente:** Elaboración propia)

La implementación aquí descrita es muy sencilla: Para crear el material se crean dos nodos de tipo textura, cada nodo carga el mapa que le corresponde al canal al que irá conectado. El de color cargará la imagen difusa, y el de alpha cargará el mapa de alpha.

Los valores pasados como parámetros son interpretados por el canal alpha como valores para la transparencia. El canal alpha interpreta el valor 0 como transparencia total, y el valor 1 como opacidad total. Por este motivo los mapas de alpha son en escala de grises.





**Ilustración 7.15.** Resultados obtenidos del ejemplo 3. (**Fuente:** Elaboración propia)



## **8 Conclusiones y trabajos futuros.**

### **8.1 Aportaciones**

El desarrollo ha arrojado una herramienta estable y lo más importante funcional. Queda patente que se ha sabido llevar un lenguaje de programación como lo es GLSL a términos más sencillo y elevarlo a un nivel de menos abstracción mediante el uso de VLP. La implementación basada en nodos es el presente de no solo los editores de materiales y Shaders, sino también de tareas que solían realizarse de formas más a bajo nivel y que ahora se realizan mediante VLP y nuevamente mediante nodos y grafos de nodos.

El producto final es un editor de materiales totalmente funcional y que cumple sobradamente su diseño, resolviendo además problemáticas como las planteadas apartados anteriores (ver apartado 4.12. Discusión) en donde se detectaban carencias o vacíos generales dentro de productos similares. La ausencia de una herramienta donde lo generado fuera pensado para el usuario en vez de para complejos sistemas internos, es la principal aportación que se le puede atribuir a este proyecto. Permitir a los usuarios controlar en todo momento el dibujado de las escenas y más importante aún el proceso de dibujado de los materiales mediante la creación de una herramienta que lo propicia, es la forma en la que desde aquí se ha intentado resolver el problema planteado.

### **8.2 Comparación con los objetivos**

En términos generales y en gran medida, todos los objetivos previstos se han conseguido cumplir. A continuación se enumeraran los objetivos descritos para este proyecto y su justificación.

## 8.3 General.

“El objetivo general de este proyecto es el de realizar una herramienta capaz de generar Shaders que definan el aspecto de objetos en 3D. La herramienta debe funcionar bajo una interfaz basada en nodos”. El objetivo general fue cumplido de manera exitosa, el resultado es una aplicación que cume con lo requerido y que su interfaz se basa de nodos.

## 8.4 Específicos

“Desarrollar un modelo de iluminación parametrizable y capaz de ser alimentado por la información que produzca el usuario usando el grafo de nodos.”. El modelo de iluminación fue desarrollado y parametrizado con éxito. Resultado de esto es la existencia de un nodo principal al cual se le alimenta conectando sus entradas que representan los parámetros mencionados. El modelo de iluminación es el modelo de reflexión de Blinn-Phong.

“Construir un editor de nodos que permita crear y eliminar nodos así como también crear y eliminar conexiones entre los nodos.”. La construcción del editor de nodos fue concluida correctamente, en un principio la implementación de Stanislaw Adaszewski sirvió como punto de partida y como base para continuar trabajando.

“Implementar una aplicación usable e intuitiva que permita al usuario llevar a cabo su tarea de la forma más sencilla posible proporcionándole soluciones para problemas típicos como la redundancia, las repeticiones y en definitiva mejorar el flujo de trabajo.”. Este objetivo fue cumplido en cierta medida, si bien el usuario puede llevar a cabo el trabajo con cierta facilidad, la aplicación carece de más elementos que pueden facilitar de muchas otras maneras el trabajo del usuario.

“Implementar una aplicación capaz de generar un Shader bajo los estándares de GLSL basado en el grafo de nodos definido por el usuario. Dicho Shader debe ser fácil de integrar en sistemas externos, o bien se le ha de dar usuario algún método para hacerlo. “. Este

objetivo ha sido cumplido de forma exitosa. La aplicación crea Shaders cada vez que se genera un grafo en el editor de nodos. Los Shaders son visibles en la ventana de Shaders. Además, para la integración de los Shaders en otros sistemas no se requiere de datos muy ajenos a los típicos usados por cualquier sistema de gráficos amateur, por lo que la facilidad está asegurada.

“Obtener la capacidad para hacer uso de tecnologías nunca antes vistas haciendo uso del sentido de investigación y de aprendizaje que los años de carrearla han aportado como parte de la experiencia educativa.”. Este objetivo fue completado de buena forma. Si bien no hay forma de medir algo tan subjetivo como ese esfuerzo para afrontar un proyecto, el hecho de que se haya finalizado y sea completamente funcional, da fe de que se logró encarar la mayoría los problemas de forma exitosa.

## **8.5 Trabajos futuros**

### **8.5.1 Mejoras en el programa.**

Si llevamos al campo de la comparación la herramienta final, resulta evidente la grandísima cantidad de mejoras y añadidos que se podrían aplicar sobre el proyecto. A continuación se hará un listado con los añadidos y mejoras que más se han creído necesarios a posteriori en el proyecto.

#### **Modelo de iluminación mejorado.**

El modelo implementado aquí no es el peor de todos, incluso es el más extendido y el más usado de entre otros muchos. No obstante, para lograr resultados más competitivos en el aspecto visual, se deben introducir técnicas y procesos que hagan del dibujo final algo más realista y estético. Con esto no se está diciendo que haya que evitar el modelo de Blinn-Phong, más bien, se pide rediseñar de una mejor manera el modelo y si hace falta evitarlo, pues que así sea. Casos como el de añadir procesos de postprocesado incrementa la calidad del dibujado notoriamente, así como también mejores definiciones de la iluminación y la atenuación.

En definitiva, y mientras sea plausible, aquí se pide intentar diseñar un sistema de renderizado basado en físicas (PBR) al alcance de mis posibilidades.

## **Operaciones con nodos**

Si bien se han diseñado unas cuantas operaciones, cabe mencionar que las librerías de funciones dedicadas a Shading cuentan con cientos y cientos de ellas. Rediseñar funciones e implementarlas sería altamente recomendable como trabajo futuro.

## **Mejoras en usabilidad**

Son incontables las mejoras que se pueden realizar en este apartado. Aspectos como eliminar una conexión cuando se sobrepone otra, o como más presencia de elementos interactivos son los principales elementos a rediseñar e implementar como mejoras en usabilidad.

Además, procesos como la compilación, p. ej., ganarían en usabilidad aplicando elementos como animaciones o cambios en la interfaz como soporte para propiciar más feedback al usuario.

## **Asistencia para exportar.**

Una buena mejora sería la de añadir una manera fácil para que gente interesada en integrar los materiales en un sistema externo lo puedan hacer. Generar dlls para gestionar lo relevante con la carga y el dibujado podrían ser buenas alternativas para facilitar la mencionada integración entre otras.

## **8.6 Conclusión personal.**

Todo el proceso de desarrollo de este proyecto ha dejado como resultado en primera instancia un sitio donde ver reflejado todo lo capaz y trabajador que he podido ser. Durante el proceso me he visto obligado a dar cada vez más de mí mismo ante la llegada de nuevos obstáculos. Sin embargo todo ha valido la pena y visto el resultado del proyecto ha dado en mi humilde opinión, muy buenos frutos que me servirán no solo como hoja de presentación sino también como lo dicho: una fuente de satisfacción personal y profesional



y que me animarán a continuar por el camino tomado aquí y que me motivan a seguir esforzándome para llegar a cosas más grandes.



## 9 Bibliografía y referencias

- [1] Akenine-Möler, T., Haines, T., & Hoffman, N. (2008). *Real-Time Rendering* (Tercera ed.).
- [2] Assimp. (s.f.). *Sitio web de Assimp*. Obtenido de <http://assimp.sourceforge.net/index.html>
- [3] Epic Games, Inc. (s.f.). *Physically Based Materials*. Obtenido de <https://docs.unrealengine.com/latest/INT/Engine/Rendering/Materials/PhysicallyBased/index.html>
- [4] Fletcher, D., & Parberry, I. (2011). *3D Math Primer for Graphics and Game Development* (Segunda ed.).
- [5] Freepik. (s.f.). Diseño de los iconos de "Ver Shader" y Aplicar Cambios.
- [6] Joachim, H. (s.f.). *Documentación de Shader Forge*.
- [7] Reed, N. (s.f.). *Radiometry Versus Photometry*. Obtenido de <http://www.reedbeta.com/blog/2014/08/17/radiometry-versus-photometry/>
- [8] Ruh, M. (s.f.). *Cook-Torrance*. Obtenido de <http://ruh.li/GraphicsCookTorrance.html>
- [9] Russell, J. (s.f.). *Basic Theory of Physically-Based Rendering*.
- [10] Stanislaw, A. (s.f.). *QNodesEditor – Qt nodes/ports-based data processing flow editor*. Obtenido de <http://alcoholic.eu/qnodeseditor-qt-nodesports-based-data-processing-flow-editor/>
- [11] Wikipedia. (s.f.). *Blinn-Phong*. Obtenido de [https://en.wikipedia.org/wiki/Blinn%E2%80%93Phong\\_shading\\_model](https://en.wikipedia.org/wiki/Blinn%E2%80%93Phong_shading_model)
- [12] Wikipedia. (s.f.). *BRDF*. Obtenido de [https://en.wikipedia.org/wiki/Bidirectional\\_reflectance\\_distribution\\_function](https://en.wikipedia.org/wiki/Bidirectional_reflectance_distribution_function)
- [13] Wikipedia. (s.f.). *Colorimetría*. Obtenido de <https://es.wikipedia.org/wiki/Colorimetr%C3%ADa>
- [14] Wikipedia. (s.f.). *OpenGL Extension Wrangler Library*. Obtenido de [https://en.wikipedia.org/wiki/OpenGL\\_Extension\\_Wrangler\\_Library](https://en.wikipedia.org/wiki/OpenGL_Extension_Wrangler_Library)
- [15] Wikipedia. (s.f.). *Radiometría*. Obtenido de <https://es.wikipedia.org/wiki/Radiometr%C3%ADa>
- [16] Wikipedia. (s.f.). *The Rendering Equation*. Obtenido de [https://en.wikipedia.org/wiki/Rendering\\_equation](https://en.wikipedia.org/wiki/Rendering_equation)