TESINA DE LICENCIATURA

Título: Teardrop: Rendering 3D en tiempo real considerando el contexto lumínico

Autores: Nicolás Belazaras

Director: Dra. Challiol, Cecilia

Codirector: Dra. Gordillo, Silvia

Asesor profesional: -

Resumen

A lo largo de la presente tesina se explora y detalla el desarrollo de un motor (o engine) de renderizado de gráficos en 3D (tres dimensiones) en tiempo real, utilizando una arquitectura en particular, con el cual se exploran técnicas de iluminación o shading y sombreado de alta calidad, como *Screen Space Ambient Occlusion* y técnicas avanzadas de rendering como *Deferred Shading*. Se estudió en detalle el modelo de iluminación *Phong*, con sus tres componentes lumínicas: ambiental, difusa y especular. El motor posee un diseño basado en una arquitectura orientada a componentes, que permite la alta reusabilidad de las partes, con módulos de código específicos que dan cierta característica o funcionalidad al objeto que los contenga. La implementación del motor contiene parte de su código en el lenguaje *C++*, el cual se ejecuta en el CPU y parte escrita en *shaders GLSL de OpenGL* que se ejecutan en la GPU. El motor permite cargar y renderizar diferentes formatos de modelos 3D y texturas. El objetivo de este desarrollo permitió la investigación de técnicas de cómputo visuales utilizadas en motores 3D comerciales de la actualidad.

Palabras Claves

Motor 3D – Contexto lumínico – Rendering diferido – Screen Space Ambient Oclussion – Modelo Phong – Arquitectura basada en componentes – Técnicas de iluminación – OpenGL – GLSL shaders.

Trabajos Realizados

- Diseño y modelado de un Motor Gráfico el cual se denomino Teardrop.

- Exploración de técnicas de renderizado, para comprender mejor la problemática a resolver en dicho motor.

- Investigación, diseño e implementación de técnicas de iluminación actuales, para ser incorporadas al motor. Integración además de distintas técnicas visuales.

Conclusiones

Teardrop es el resultado de un largo camino de aprendizaje de un área muy poco documentada y difundida. En esta tesis se presentó el modelo definido para Teardrop, como así también la implementación llevada a cabo. Se presentó un ejemplo de uso, donde se aplicaron todas las técnicas implementadas mostrando así cómo el mismo puede ser utilizado. Teardrop es altamente extensible con infinitas posibilidades de crecimiento. Utiliza las mismas técnicas que varios estudios profesionales de videojuegos AAA, pero facilitando su extensibilidad.

Trabajos Futuros

Algunos trabajos futuros que se podrían realizar son:

- Incorporar nuevas técnicas de renderizado.

- Explorar algoritmos de iluminación global como por ejemplo, *Voxel Cone Tracing*.

- Crear un instalador multiplataforma del motor.

- Definir un editor gráfico de escenas.

- Carga automática de *Uniforms* en shaders. Además, recarga de shaders globales en tiempo real.

Fecha de la presentación: **AGOSTO 2015**

Tabla de Contenidos

Capítulo 1: Introducción 4
1.1 Motivación 4
1.2 Objetivo 4
1.3 Estructura de la Tesis 5
Capítulo 2: Estado del arte 6
2.1 Arquitectura basada en componentes 6
2.2 Conceptos básicos de la renderización 6
2.2.1 Algebra Lineal aplicada al 3D 6
2.2.2 Near y Far Clipping Planes 10
2.2.3 FPS y real-time 10
2.2.4 Pipeline gráfico de OpenGL 11
2.2.5 Buffers 12
2.2.6 Shaders y GLSL 13
2.3 Modelos de iluminación 14
2.3.1 Modelo Phong 14
2.3.2 Iluminación Rim 16
2.4 Otras técnicas de fidelidad visual 17
2.4.1 Rendering diferido 17
2.4.2 Ambient Occlusion 18
2.4.3 Normal Mapping 19
2.5 Librerías de OpenGL 20
2.5.1 GLM 20
2.5.2 GLEW 21
2.5.3 GLFW 21
Capítulo 3: Modelo Propuesto 22
3.1 Diseño de las clases de Teardrop 22
3.1.1 Engine 22
3.1.2 GameObject 22

3.1.3 Componentes 24
- *Renderer* 25

3.1.4 MainRenderer 30
3.1.5 MainDeferredRenderer 31
3.1.6 MainSSAORenderer 31
3.1.7 Input 32
3.1.8 FPSController 32
3.1.9 Diagrama de clases completo 33
3.2 Extendiendo el modelo 34
3.2.1 Subclasificando Engine 34
3.2.2 Subclasificando Script 35
3.2.3 Creación de Shaders 35
3.3 Ciclo de ejecución de Teardrop 35
Capítulo 4: Desarrollo Realizado (Teardrop) 37
4.1 Implementación de las clases de Teardrop 37
- Engine 37
- GameObject 38
- Component 39
- Renderer 39
- Mesh 40
- Camera 41
- Transform 42
- Material 42
- Shader 43
- Texture 43
- MainRenderer 44
- MainDeferredRenderer 44
- MainSSAORenderer 46
- Input 46
- FPSController 46

4.2 Implementación de los shaders de Teardrop 48

- Shader básico 48
- Shader Phong 50
- Spherical Environment Mapping 53
- Deferred Rendering 55

4.3 Problemas encontrados durante el desarrollo 62

4.4 Instalación 63

4.5 Demostraciones de uso general 64

- ➢ Creando una escena 64
- ➢ Nuestro primer GameObject 65
- ➢ Agregando componentes 66
- ➢ Agregando movimiento 67

Capítulo 5: Ejemplo de Uso de Teardrop 70

- Generando el modelo 70
- Código de la simulación 70
- Distintos métodos de renderizado 72

Capítulo 6: Conclusiones y Trabajos Futuros 79

- Instalador multiplataforma 79
- Reloader de shaders globales 79
- Scripts en GameObjects 80
- Editor gráfico de escenas 81
- Sistema de carga automática de uniforms en shaders 81
- Soporte de distintos formatos 82
- Dependencias de headers automáticas 82
- Otras features 82

Referencias 83

Capítulo 1: Introducción

1.1 Motivación

Las aplicaciones que reaccionan acorde al contexto se denominan sensibles al contexto [6], y los videojuegos [4] pueden plantearse como un caso particular de este tipo de aplicaciones. Donde el contexto del juego o del jugador puede influir en la información que reciba el usuario. Un contexto [7] particular es la luz, esta puede influir en la información que se brinde, por ejemplo, para mejorar la imagen que recibe el usuario, para darle foto-realismo, etc.

Es motivación de este proyecto el estudio y comprensión de las diferentes maneras de proveer foto-realismo a las imágenes generadas por computadora, así como la exploración de las técnicas más usadas en el mercado los videojuegos, no solo para computadoras sino también para consolas. Si bien en la tarea de crear un videojuego se pueden utilizar motores gráficos existentes como Unity[1] o Unreal Engine[2], en el diseño y la programación de un proyecto comercial a gran escala[3] por parte de grandes estudios como DICE con su motor Frostbite[4] o Massive con el motor Snowdrop[5], se requiere un trabajo más profundo y complejo, es decir la creación de motores y no solo su utilización como usuario. Se contempla el desarrollo de técnicas de rendering y el uso de shaders que completan el pipeline gráfico [16] utilizado para producir, generalmente 60 veces por segundo, la imagen final.

Forma parte de la motivación crear un motor de renderización 3D que sea fácilmente extensible acorde a los aspectos que cada aplicación requiera, en particular, se hará hincapié en el contexto de la iluminación para llevar a cabo el prototipo de uso del motor. Para lograr la extensibilidad de este motor el mismo se realizará con una arquitectura basada en componentes [2], [3], [5].

1.2 Objetivo

Desarrollar un motor (o engine) de renderizado de gráficos en 3D (tres dimensiones) en tiempo real, utilizando una arquitectura en particular, con el cual explorar técnicas de iluminación o shading y sombreado de alta calidad, como *Screen Space Ambient Oclussion* y técnicas avanzadas de rendering como *Deferred Shading*. Utilizando cálculos de algebra linear se determina [63], teniendo una fuente de luz y un modelo a iluminar, cómo las caras del mismo se aclaran u oscurecen dependiendo del ángulo en el que la luz incida sobre ellas, si se encuentran obstáculos en el medio, etc. Es decir, acorde a la cantidad de luz que recibe

[1] https://unity3d.com
[2] https://unrealengine.com
[3] Productos llamados AAA
[4] http://frostbite.com
[5] http://massive.se

un determinado objeto, se renderiza la imagen 3D adecuada mejorando la estética de dicho objeto [64].

La arquitectura basada en componentes permitirá la alta reusabilidad de las partes, teniendo módulos de código específicos que le dan cierta característica o funcionalidad al objeto que los contenga.

El objetivo del motor es facilitar tareas en el desarrollo de simulaciones y videojuegos, implementando las partes más genéricas del flujo de trabajo para que el desarrollador usuario no deba hacerlo en cada nuevo proyecto que realice. Por ejemplo, videojuegos con escenas en donde la luz ambiente puede modificarse, podrían hacer uso de este motor, agilizando en gran medida varios aspectos del desarrollo.

1.3 Estructura de la Tesis

La tesis consta de seis capítulos. A continuación se describe cada uno de estos.

En el Capítulo 2 se describe el estado del arte, un proceso de investigación de tecnologías actuales donde se detallan las arquitecturas usadas por Teardrop, cada librería utilizada en el mismo; se da una introducción teórica a cada técnica visual implementada en el motor y se habla de conceptos técnicos que deben conocerse para entender mejor el desarrollo de la tesis.

En el Capítulo 3 se describe al modelo propuesto, se explica el diseño con diagramas de clase y de secuencia detallando qué rol juega cada clase en el engine, se explican los puntos de extensión del modelo y el ciclo de ejecución del motor.

En el Capítulo 4 se detalla el desarrollo realizado, es decir, la implementación del modelo que conforma Teardrop. Se muestra la implementación de cada clase del motor, describiendo los métodos más importantes de cada una. Se detalla luego la implementación de los shaders del motor. Se habla sobre ciertos problemas encontrados a la hora de implementar el motor. Se indica cómo debe ser instalado Teadrop para funcionar en su totalidad. Finalmente se describen demostraciones de uso general, a modo tutorial.

En el Capítulo 5 se muestra un gran ejemplo de uso de Teardrop donde se construye una escena que utiliza todas las técnicas gráficas implementadas en el motor, se muestra el código de la misma y luego variaciones de este que producen diferentes tipos de resultados visuales.

En el Capítulo 6 la tesis llega a su fin con conclusiones y trabajos por realizar a futuro para hacer crecer al motor.

Finalmente, se presentan las referencias utilizadas a lo largo de la tesis.

Capítulo 2: Estado del arte

2.1 Arquitectura basada en componentes

Un paradigma de programación define el estilo en que un programa de computadora es desarrollado. Estos difieren en conceptos y abstracciones utilizadas para representar elementos (así como objetos, funciones, variables, y restricciones).

El paradigma basado en componentes define a cada objeto como un conjunto de componentes, donde cada uno encapsula una propiedad del mismo (variables y métodos). La composición dinámica de objetos permite crear nuevos objetos sin recompilar el código, con la posibilidad de ajustar propiedades de manera sencilla. El objetivo consiste en la alta reusabilidad de las partes, con el menor acoplamiento posible de las mismas. Se debe tener en cuenta que mientras más general sea el componente, mayor será su reusabilidad, pero más complejo su desarrollo y por lo tanto menor su usabilidad.

En un diseño óptimo, dada una entidad compuesta, se podría remover un componente, sin perjudicar el funcionamiento de los otros. A su vez, la experiencia nos demuestra que no es posible la total independencia de las partes, ya que cierta comunicación debe existir entre ellas.

La comunicación directa se refiere a tener referencias (punteros) a otros componentes de un tipo en particular dentro del mismo objeto o entidad. Esta manera de comunicarse es fácil y rápida de implementar (parecida a las prácticas utilizadas en OOP tradicional), pero requiere que los componentes tengan un grado de conocimiento entre sí, con la consecuente pérdida de independencia. En lo posible se deben evitar este tipo de prácticas, y dejarlas para cuando no se encuentra una mejor opción.

La comunicación indirecta consiste en enviar mensajes con información recibida por los componentes suscriptos al mismo tipo de mensaje. Un mensaje posee un emisor, una lista de receptores suscriptos, información opcional y un rango (enviar el mensaje a todos los suscriptores dentro del mismo objeto o entidad, enviarlo a todos los suscriptores del sistema, o de otra entidad). Este método es más flexible que el directo, aunque pueda ser más incómodo de implementar.

Un componente es una entidad ejecutable independiente, donde los servicios que ofrece están disponibles a través de su interfaz, y todas las interacciones ocurren a través de ella. Al desarrollar y e implementar componentes la elección del lenguaje debería ser irrelevante [1], ésta no debería afectar la habilidad de interactuar entre sí de los mismos.

2.2 Conceptos básicos de la renderización

2.2.1 Algebra Lineal aplicada al 3D

El álgebra lineal es la principal disciplina de la programación gráfica en tres dimensiones, por ser el lenguaje matemático para describir geometría espacial. Es el estudio de los

vectores. Un vector sirve para representar tanto una posición en el espacio, con coordenadas (x, y, z) como en (2,-3,1), direcciones y velocidad. Cada modelo 3D está compuesto por un conjunto de vértices, cada uno con su propio vector de posición.

Estos vectores de posición, se dice que están en su propio espacio, o coordenadas relativas a sí mismo. Podemos aplicar operaciones para manejar al modelo, como rotarlo, escalarlo o moverlo. Se puede aplicar de a una operación por vez, pero resulta una tarea muy lenta, más aun teniendo en cuenta que los modelos pueden contar con miles de vértices. Otra forma es agrupando estas transformaciones lineales (transformación que conserva las dos operaciones fundamentales que definen la estructura del espacio vectorial) en una matriz y luego multiplicando cada vértice por esta matriz.

Definimos a la matriz del modelo o *model matrix* a la matriz utilizada para escalar, rotar o mover dicho modelo y ubicarlo en *world space* (espacio del mundo), donde convive con los demás objetos de la escena y los vértices del mismo se definen relativos al centro del mundo, en vez de al centro del modelo.

Otra matriz, es la llamada *view matrix* [10] o matriz de vista, que transforma las coordenadas de world space a *camera space*, es decir, nos muestra cómo se vería el mundo desde el punto de vista de una cámara que observa la escena.

Finalmente, la matriz de proyección o *projection matrix* [11], única para toda la escena, transforma las coordenadas desde camera space al espacio homogéneo, con todos los vértices definidos en un cubo. Este cubo se denomina *viewing frustum* [12] y se forma a partir de una pirámide recortada. Describe la región tri-dimensional que será visible en la pantalla, incluyendo todo lo que se encuentre dentro de este.

Antes de la proyección, el frustum posee su forma original de pirámide recortada y los objetos de la escena vistos por la cámara no poseen perspectiva, es decir, dos objetos iguales, uno a centímetros de la cámara y otro a varios metros, se observan igual de grandes, cuando el más alejado debería parecer más pequeño. Luego de aplicar la proyección, el frustum pasa a tener forma de cubo perfecto, y los objetos de la escena quedan deformados para apreciar el efecto de la perspectiva, siendo los que queden dentro del frustum, los finalmente visibles.

Veamos un ejemplo concreto para graficar los conceptos descriptos anteriormente. En la Figura 1 se pueden apreciar los objetos en azul y el frustum en rojo.

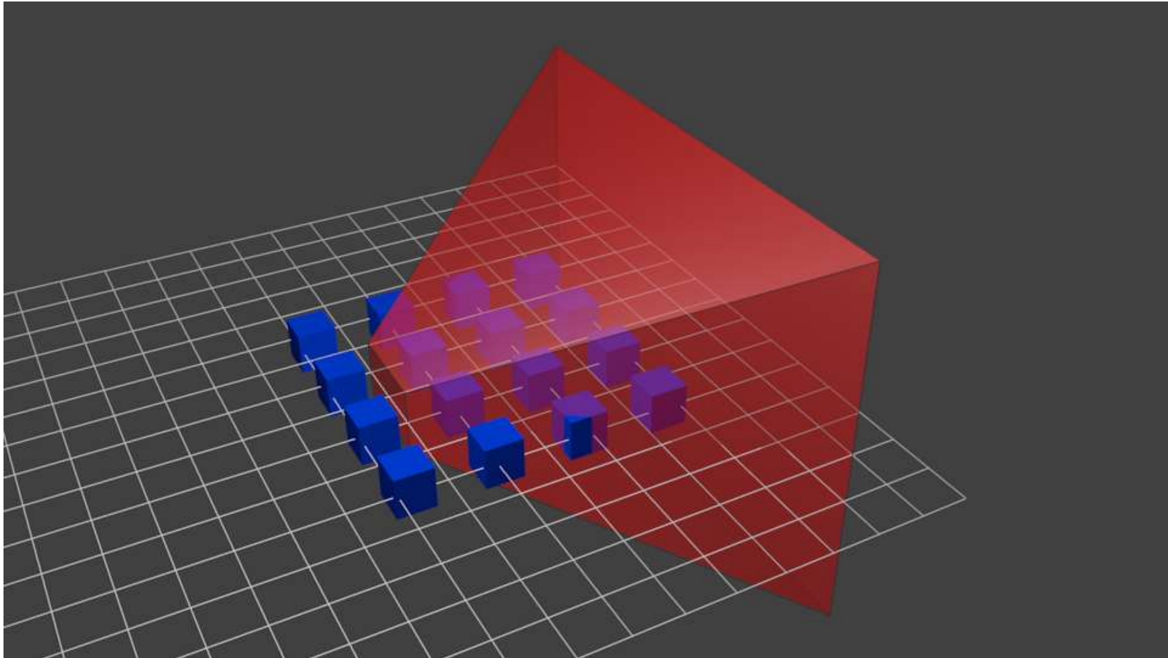

Figura 1 Cubos antes de la proyección y frustum [13]

En el caso de multiplicar la representación presentada en la Figura 1 por la matriz de proyección se logra el efecto mostrado en la Figura 2.

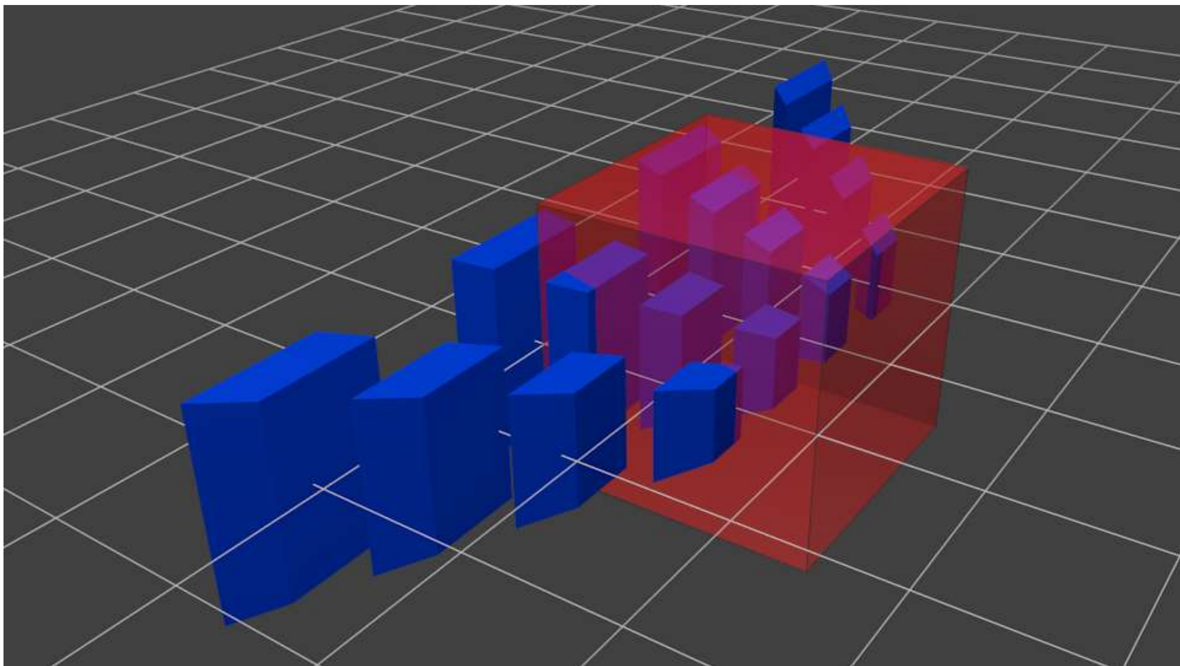

Figura 2 Escena en espacio homogéneo [13]

En la Figura 2, ahora el frustum ahora es un cubo perfecto (con coordenadas entre -1 y 1 en todos sus ejes) y los objetos azules se han deformado de la misma manera. Así los objetos que están cerca de la cámara (cerca de la cara del cubo que no podemos ver) aparecen más grandes, y los otros más chicos, como en la vida real (visto en la Figura 3).

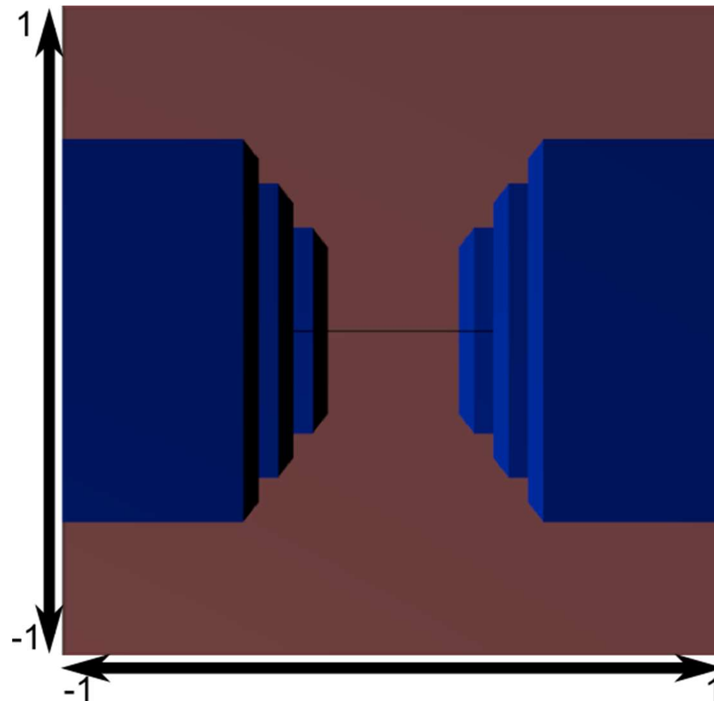

Figura 3 Escena vista desde "detrás" del frustum [13]

El grupo de estas tres matrices puede combinarse en una multiplicándolas entre sí, acelerando el proceso de conversión entre los diferentes espacios, y recibe el nombre de modelo MVP (model, view, projection) [13].

Se necesita una última matriz utilizada para proyectar los puntos tri-dimensionales en un plano de dos dimensiones, esta matriz se suele llamar *3D projection* (proyección 3D) o *graphical projection* (proyección gráfica) [14].

Gracias al uso de OpenGL [25] (librería que se detalla en la Sección 2.3.1), el programador no necesita manipular esta matriz porque se realiza automáticamente. Para diferenciarla de la matriz de proyección que sí manipula el usuario, a ésta se la llama *OpenGL projection matrix* [11]. Esta matriz genera la imagen 2D que luego será mostrada en pantalla. Dicha matriz, se encarga de cortar los vértices que no entraron en el frustum (frustum culling) y de transformar los vértices una última vez a coordenadas normalizadas de dispositivo o NDC.

2.2.2 Near y Far Clipping Planes

Los planos *near clipping plane* y *far clipping plane*, son planos que delimitan la región renderizable de una escena [38], en coordenadas del mundo virtual. Se refieren a las caras que forman al frustum, el cubo transformado con el cálculo de perspectiva. Como la escena se proyecta sobre el near clipping plane, si este es 0 toda la escena se proyectaría sobre un solo punto. El far clipping plane ayuda a mantener *frame rates* interactivos al indicar hasta que distancia desde la cámara se tomarán objetos para renderizar, ayudando a disminuir la cantidad total de vértices, como se ve en la Figura 4 en donde la columna queda fuera de éste y por ende no será renderizada.

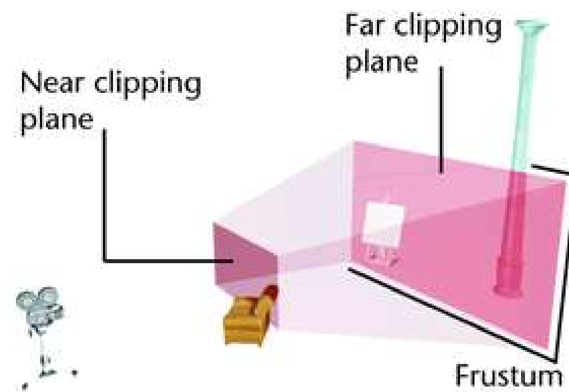

Figura 4 Near y Far Clipping Planes del frustum [39]

2.2.3 FPS y real-time

Siglas en inglés de frames per second, cuadros por segundo; se refiere a la frecuencia de imágenes por segundo enviadas a la pantalla. El ojo humano percibe entre 10 y 12 imágenes por segundo como individuales, pasando este límite se crea la ilusión de movimiento [15].Mientras lo observado normalmente en películas es un frame rate de 24 fps, en los videojuegos el estándar fue de 30 y luego de 60 para percibir mayor fluidez. Mientras mayor sea el poder de cómputo en donde se ejecute la simulación mayor podrá ser el frame rate de la misma, pero dado que la mayoría de los monitores poseen tazas de refresco de 60 Hz, las tazas mayores a 60 fps no son apreciadas.

Con real-time o tiempo real nos referimos a que cada cuadro de la simulación se procesa y genera en el momento, y la entrada del usuario puede alterar la misma, al contrario de una grabación o render estático en donde los cuadros son pre-procesados y almacenados para posterior reproducción.

2.2.4 Pipeline gráfico de OpenGL

El pipeline gráfico [16] consiste en una secuencia de pasos para crear una rasterización 2D de una escena 3D. Lo que solía ser un proceso fijo, con el avance del hardware se transformó en un proceso programable a través de los distintos tipos de shader. En la Figura 5 se observan las diferentes etapas del proceso.

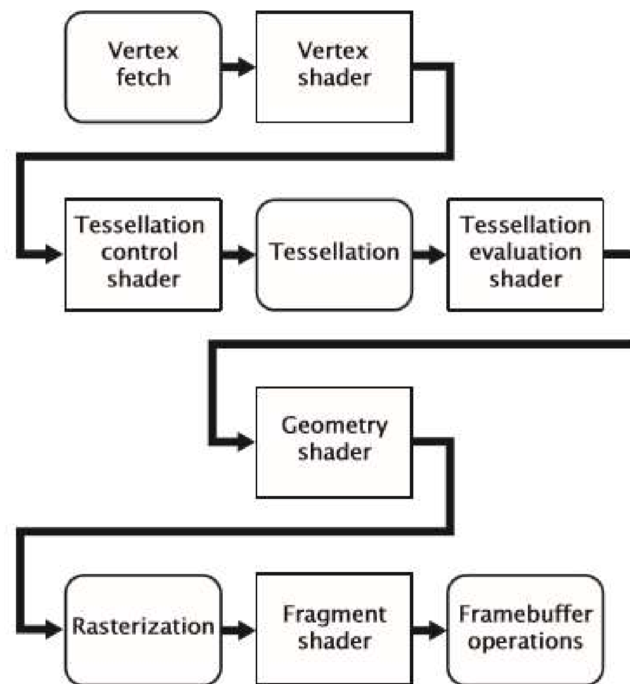

Figura 5 Etapas del pipeline gráfico [8]

El trabajo comienza su camino en un set de vertex buffers [19], llenos de arrays con atributos de vértices [20]. Estos atributos son entrada *del vertex shader*, observado en la Figura 5, entre ellos se encuentra la posición del vértice en el espacio 3D y coordenadas de texturas para mapear el vértice a una o más texturas (imágenes en 2D).

Al ejecutar el vertex shader por cada vértice con sus atributos como input (referido como datos *varying*), también se cuenta con datos compartidos entre todas las instancias del mismo, atributos de solo lectura denominados *uniform*, información que no cambia entre los distintos vértices, ya sean texturas o tablas de datos. Como mínimo, el vertex shader computa como salida, la posición proyectada del vértice en *screen space* (como se lo vería desde la cámara). También puede generar otra salida como información del color o coordenadas de texturas para que el rasterizador mezcle sobre la superficie de los triángulos formados por los vértices.

Luego, teniendo los vértices proyectados indizados y en orden, se toman de a tres para formar triángulos, la figura primitiva más importante. El rasterizador toma cada triángulo,

lo recorta y descarta las partes que queden fuera de lo visible en pantalla; luego rompe las partes visibles sobrantes en lo que se llaman fragmentos del tamaño de un pixel.

Los fragmentos generados pasan a ser input de otro programa llamado *fragment* o *pixel shader*, quien procesa cada uno y dibuja valores de color y profundidad en un contenedor llamado framebuffer. En este shader se suelen realizar las operaciones de iluminación, sombreado y mapeado de texturas, y es una de las partes más sensibles a la performance del pipeline gráfico.

El *framebuffer* es el destino final del output del proceso de renderizado. Además de enviar esta información directamente a la pantalla para su visualización, OpenGL permite guardarla en forma de texturas, que a su vez se puede reutilizar como entrada de otra tanda o *batch* de renderizado. Aquí también ocurre el proceso de *depth testing* o testeo de profundidad en donde fragmentos que se encuentran detrás de otros ya dibujados en pantalla son descartados. El *stencil buffer* se utiliza para definir figuras que determinen qué parte del framebuffer será dibujada, recortando parte del trabajo de renderizado.

2.2.5 Buffers

Para renderizar modelos en tres dimensiones se necesita la información de cada vértice de los mismos, en vez de guardar dicha información en la memoria cliente (RAM del sistema) y enviarla a la GPU en cada cuadro a rasterizar, ésta se envía a la memoria del dispositivo (GPU) una vez sola y se guarda en un buffer denominado VBO (Vertex Buffer Object) [17], en donde por cada vértice de almacena la posición del mismo, el vector normal, las coordenadas de una textura, información de color, etc.

Ahora, si dos triángulos formados por tres vértices cada uno, comparten un borde, tendremos dos vértices repetidos. Gracias a la técnica de indizado o VBO Indexing podemos optimizar dicho caso para reusar los mismos vértices una y otra vez. Esto se realiza con un buffer de índices, quien para cada triángulo almacena tres enteros, cada uno referenciando a un vértice verdadero y su respectiva información, almacenado en el VBO. Estos índices también son utilizados en las coordenadas UV de las texturas. Esto se puede apreciar en la Figura 6.

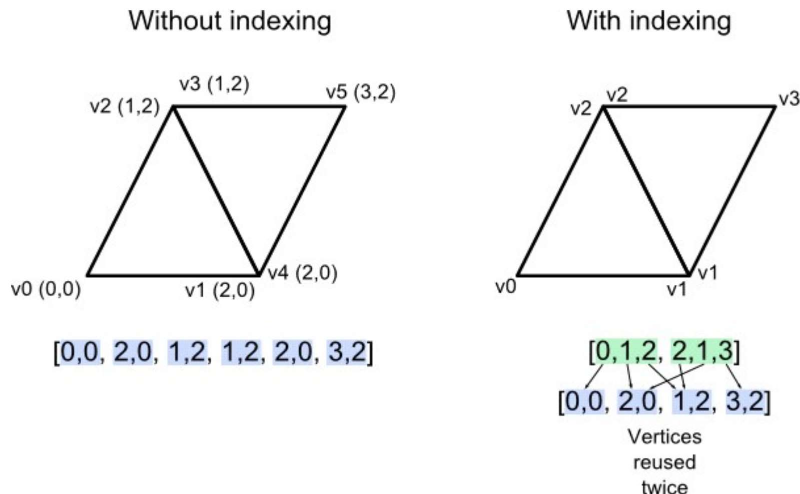

Figura 6 Indexado de vértices [18]

Otro buffer a tener en cuenta es el Vertex Array Object o VAO. Un VAO guarda una configuración de buffers (punteros a éstos), es decir, para utilizar cualquier buffer primero debo activarlo y configurarlo, y si por cada objeto tengo más de un buffer esta tarea resulta tediosa. En cambio, el VAO permite asignar o *bindear* los buffers que se deseen, configurarlos, y al necesitarlos solo se debe setear como activo o *current* dicho VAO. Por lo tanto *bindear* un vertex array object es equivalente al código requerido para configurar todos los atributos de los buffers.

OpenGL además permite la creación de Framebuffer Objects (FBO), buffers definidos por usuarios con los que se puede renderizar fuera de la pantalla u *off-screen*, incluyendo renderizado sobre texturas. Capturando imágenes que normalmente serían dibujadas en la pantalla, pueden utilizarse para implementar una gran variedad de filtros y efectos de pos-procesamiento enviando la información contenida en éstos a fragment shaders (vistos en la sección 2.2.6 Shaders y GLSL). También se pueden usar para crear vistas de otras escenas, por ejemplo una TV en una casa mostrando la vista de una segunda cámara. Una escena se puede renderizar a través de un FBO a una textura, y luego esa textura se puede aplicar a la superficie de la televisión [51].

2.2.6 Shaders y GLSL

Shader se refiere a un programa de computadora utilizado para realizar sombreado o shading, logrando la percepción de profundidad en modelos 3D al variar los niveles de oscuridad [21]. También se utilizan para la creación de efectos especiales y post-producción de video. Utilizados casi completamente en la placa de video, se dividen dos categorías principales dependiendo de la etapa de renderizado del *pipeline* gráfico, visto en la sección 2.2.3, para modificarlo y personalizarlo, a diferencia de pipelines antiguos estáticos en los que el programador no tenía tanta libertad.

Expandiendo lo visto en el punto 2.2.3 sobre el vertex shader, éste consiste en un script ejecutado una vez por vértice del modelo 3D enviado al procesador gráfico, con el objetivo de transformarlo en una imagen en dos dimensiones contribuyendo a la imagen final mostrada en pantalla. Éstos pueden manipular atributos como la posición de un vértice, el color asociado, las coordenadas del mismo sobre una textura, pero no puede crear nuevos vértices. La salida de este programa continúa como entrada en la próxima etapa del pipeline, ya sea un shader geométrico si lo hay, o el rasterizador.

El pixel shader*,* también introducido brevemente en el punto 2.2.3, computa el color final y otros atributos de un fragmento, refiriéndose a un solo pixel, por lo tanto si la imagen final producida es de resolución full hd, este programa se ejecutará 1920x1080 veces. Hay que tener muy en cuenta la eficiencia en el código, tratando de evitar bucles y operaciones costosas. En gráficos 3D, un pixel shader de por sí no puede producir efectos muy complejos, ya que no posee información de la geometría de la escena. Sí conoce su posición final en la pantalla, y puede obtener la información de pixeles cercanos, si el contenido entero de la pantalla es provisto al shader en forma de textura. Esta técnica posibilita una gran cantidad de efectos bidimensionales como el blur, la detección de bordes, o la oclusión ambiental.

GLSL es el lenguaje de alto nivel de programación de shaders de OpenGL, tiene sus orígenes en C. El código fuente se ubica en un *objeto shader* y se compila, luego múltiples objetos shader (vertex, pixel, tessellation, geometry, compute) son *linkeados* juntos para formar un *objeto programa*.

2.3 Modelos de iluminación

El trabajo de una aplicación de renderizado de gráficos es la simulación de la luz. Existen modelos extremadamente avanzados que son tan físicamente precisos como lo entendido por la ciencia sobre las propiedades de la luz. Con esto nos referimos a los algoritmos de *ray tracing* o trazador de rayos, técnica para generar una imagen trazando la ruta de la luz a través de los pixeles en una imagen, simulando los efectos de su encuentro con los diferentes objetos de la escena. Incluye una gran variedad de efectos ópticos como reflexión, refracción, dispersión y aberración cromática [22]. Estos modelos son imprácticos para aplicaciones de tiempo real y por eso se deben utilizar aproximaciones, que aunque no físicamente precisas, produzcan un resultado plausible. A continuación se describen algunas de estas técnicas útiles para renderizado en tiempo real.

2.3.1 Modelo Phong

Es uno de los modelos más simples [40]. Trabaja bajo el principio de que todos los objetos tienen un material con tres propiedades, la componente ambiental, la difusa y la reflectividad especular [41]. Se puede observar el efecto de cada una en la Figura 7. A cada

una se le asigna un valor de color, con colores más brillantes representando mayor reflectividad. Cada fuente de luz también tiene estas propiedades y el resultado final de color es la suma de las interacciones de estas tres propiedades entre las luces y los materiales.

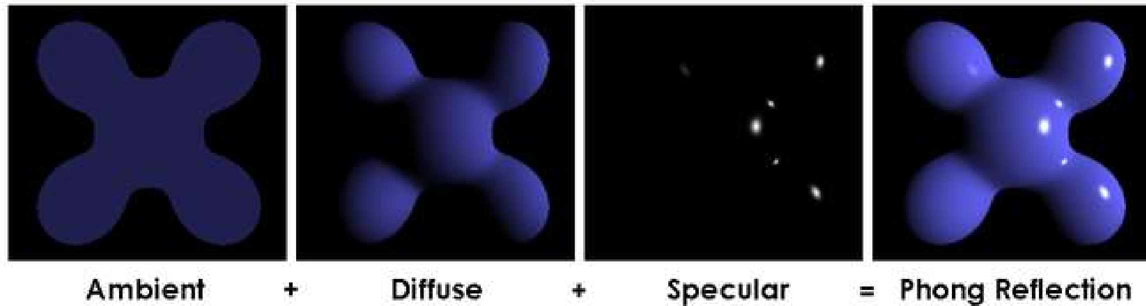

Figura 7 Distintas componentes de luz en el modelo Phong [40]

La luz ambiental no proviene de ninguna dirección en particular. Tiene una fuente originada en algún punto, pero los rayos de luz han rebotado por el espacio y terminaron sin dirección. Los objetos iluminados por luz ambiental están iluminados parejamente en todas las superficies y en toda dirección. Se puede denominar a este tipo de luz, un "abrillantamiento" global, aplicado por fuente de luz. Aproxima la luz dispersa en el ambiente generada por una fuente de iluminación.

La luz difusa es la componente direccional de una fuente de luz.

La componente especular también es direccional, pero interactúa más pronunciadamente con la superficie y en una dirección particular. Causa un punto brillante en la superficie sobre la que brilla, denominado *highlight* especular. Por ser altamente direccional es posible que desde ciertos ángulos esta componente no sea visible. El sol y un reflector son ejemplos de luces que producen fuertes *highlights* especulares, pero solo al brillar sobre un objeto que es "brillante". El highlight especular se puede observar en forma de punto blanco en la esfera de la Figura 8.

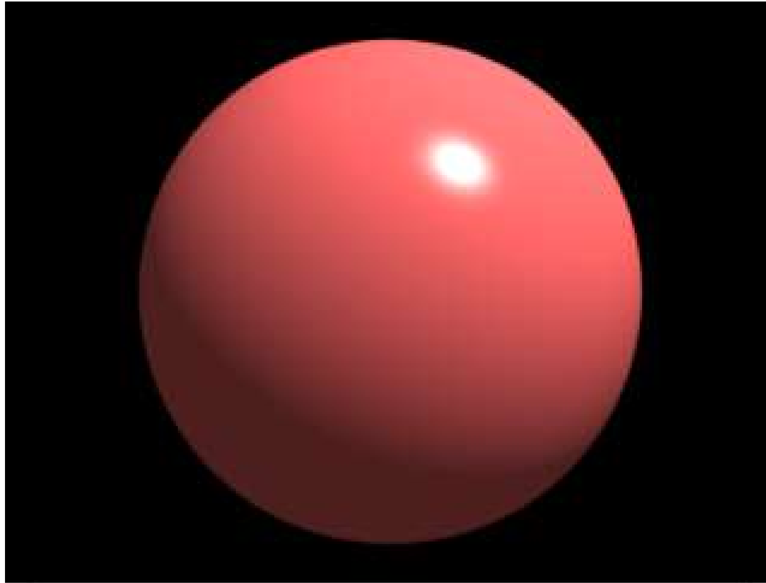

Figura 8 Ejemplo de iluminación Phong [57]

2.3.2 Iluminación Rim

También conocida como iluminación de fondo, es un efecto que simula la iluminación de los bordes de una figura, cuando la fuente de luz se encuentra detrás de ésta, es decir, que no tiene efecto sobre la superficie visible de la figura. En fotografía, esto se logra colocando una fuente de luz detrás del sujeto, así el objeto de interés queda entre la cámara y la fuente de luz. En la Figura 9 se ejemplifica el uso de solo iluminación rim, que ilumina desde atrás, los bordes del sujeto.

Figura 9 Ejemplo de iluminación Rim [58]

2.4 Otras técnicas de fidelidad visual

2.4.1 Rendering diferido

Esta técnica consiste en dividir la etapa del fragment shader del pipeline gráfico visto en el punto 2.2.3 en dos pasadas de renderizado, es decir, la ejecución de dos fragment shaders distintos, en vez de uno solo.

Por lo general, y como se vio en el punto 2.2.5, el fragment shader es quien calcula el color final del fragmento a renderizar, y se ejecutará uno de estos shaders por cada objeto de la escena. Ahora, si después de renderizar un primer objeto, se debe renderizar otro que está por delante de algo ya dibujado en la pantalla, los resultados de los cálculos anteriores son reemplazados por el nuevo renderizado, desperdiciando el trabajo del fragment shader del primer objeto. Esto se denomina *overdraw*. Si el fragment shader es computacionalmente caro o se produce mucho overdraw, esto puede sumarse en una gran pérdida de performance. Para evitarlo se puede usar deferred shading o rendering diferido, una técnica para retrasar el trabajo pesado hasta el último momento.

Para esto, primero renderizamos la escena con un fragment shader bien básico que guarde en un framebuffer cualquier parámetro de cada fragmento que luego necesitemos para calcular la iluminación o *shading*. En la mayoría de los casos se necesitarán varios buffers. El buffer utilizado para guardar esta información intermedia recibe el nombre de G-Buffer, donde G proviene de geometría.

Una vez generado el G-Buffer, se puede iluminar cada punto de la pantalla utilizando un quad (figura de cuatro vértices) del tamaño de la pantalla completa. Esta pasada final utilizará los algoritmos de iluminación de total complejidad, pero en vez de ser aplicado a cada pixel de cada triángulo, es aplicado a cada pixel del framebuffer, exactamente una vez, así reduciendo sustancialmente el costo de los cálculos, especialmente si se utilizan muchas luces o un algoritmo de shading complejo.

En la Figura 10 se muestran en pantalla las diferentes componentes de información guardada en el G-Buffer, como es, de arriba a la izquierda a abajo a la derecha, la componente de color difuso, las normales, la profundidad, y finalmente la imagen final con sombras aplicadas.

Figura 10 Componentes del G-buffer [59]

2.4.2 Ambient Occlusion

Ambient occlusion u oclusión ambiental es una técnica para simular una componente de iluminación global. Iluminación global es el efecto observado de la luz rebotando de objeto en objeto en una escena, tal que las superficies son iluminadas indirectamente por la luz reflejada en las superficies vecinas. La luz ambiental es una aproximación a esta luz dispersa y es una cantidad pequeña y fija agregada a los cálculos de iluminación. Sin embargo, en pliegues profundos o espacios entre objetos, menos luz llegará debido a las superficies cercanas ocluyendo las fuentes de luz, de aquí el término oclusión ambiental.

La luz ambiental podría ser considerada como la cantidad de luz que llegue a un punto en una superficie, si esta estuviese rodeada de un número arbitrariamente grande de fuentes de iluminación. En una superficie perfectamente plana, cualquier punto es visible por cada una de las luces sobre la superficie. Sin embargo, en una superficie con bultos, no todas las luces serán visibles desde todos los puntos. Los bultos en la superficie ocluyen la luz que llega a los puntos en los valles, y por lo tanto éstos recibirán menores cantidades de luz, oscureciendo esa parte del modelo.

En la Figura 11 se observa cómo, desde el punto dado, se puede trazar una línea a ciertas fuentes de iluminación, pero no a todas ya que las cuestas del modelo rayado obstaculizan el paso de la luz. Por eso, tiene sentido que la zona del modelo en ese punto se vea más oscura que, por ejemplo, un punto en la cima de una cuesta, de donde se puede trazar una línea a todas las fuentes de luz.

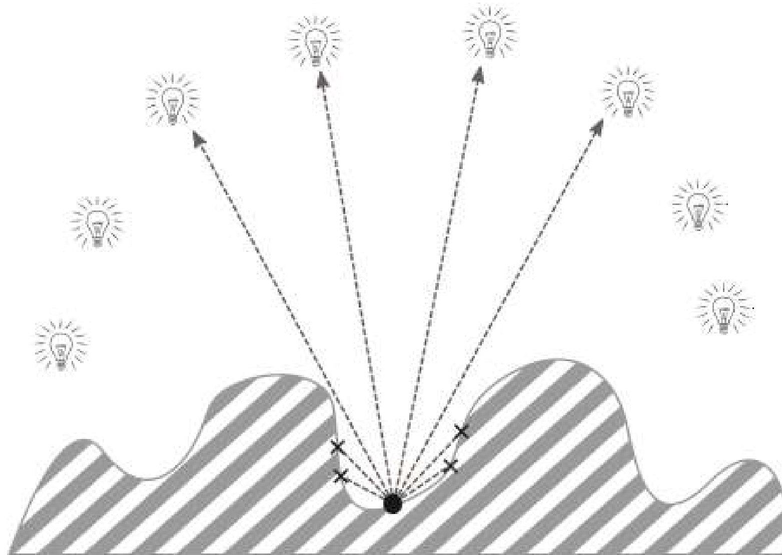

Figura 11 Técnica de oclusión ambiental en 2D [8]

Esta técnica hace uso de la técnica de rendering diferido porque necesita guardar información en el G-Buffer y luego, en una segunda pasada, computar el nivel de oclusión de cada pixel, esto hace que sea una técnica de *screen-space* o espacio de la pantalla. La información es luego mostrada en un quad, al igual que en la técnica de rendering diferido.

En la Figura 12 queda demostrado el efecto de aplicar ambient oclussion a un modelo, el antes (izquierda) y el después (derecha) con distintas configuraciones.

Figura 12 Ejemplo de oclusión ambiental [23]

2.4.3 Normal Mapping

Al realizar cálculos de iluminación se suelen utilizar las normales de los vértices de un modelo, interpoladas sobre los triángulos formados. Éstas indican en qué dirección apunta su cara (la parte visible del triángulo). Si quisiéramos un nivel de detalle mayor, deberíamos

aumentar la cantidad de vértices, para que éstos cubran un menor número de pixeles, pero este método lleva a una cantidad inadmisible de geometría.

Una manera de incrementar el nivel percibido de detalles sin aumentar el número de vértices es el normal o bump mapping. Esta técnica consiste en el uso de texturas que guarden una normal por cada texel (unidad mínima de la textura). Ésta es luego aplicada al modelo y usada en el pixel shader para calcular una normal local por cada fragmento (en vez de cada vértice). Finalmente se invoca el modelo de iluminación a elección para calcular iluminación por fragmento. En la Figura 13 se ve cómo es aplicado un normal map sobre un cubo y el efecto final producido sobre el mismo, un mayor nivel de detalles.

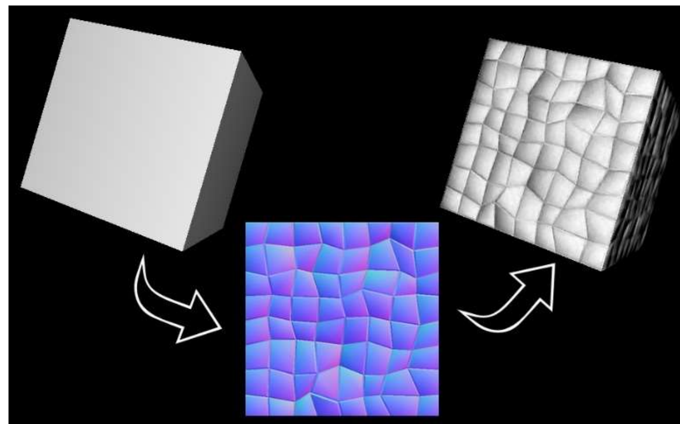

Figura 13 Ejemplo de normal mapping sobre un cubo [24]

2.5 Librerías de OpenGL

OpenGL es una interfaz de programación de aplicaciones (API) multi-plataforma y multi-lenguaje creada para renderizar imágenes en dos y tres dimensiones. Se utiliza para interactuar con la placa de video o GPU para lograr renderizado acelerado por hardware. Lanzado en 1992, OpenGL se utiliza para todo tipo de simulaciones, videojuegos y visualizaciones científicas. Consiste en un conjunto de funciones a ser llamadas por el programa cliente y un número de constantes enteras nombradas a pasar como parámetro. Posee *bindings* a distintos lenguajes, siendo uno de los más innovadores el de JavaScript, WebGL, para renderizado 3D en navegadores de internet; el binding a C para iOs y a Java y C para Android.

A continuación se describen algunas librerías *helper* que ofrecen las utilidades necesarias para realizar renderizado en OpenGL.

2.5.1 GLM

OpenGL Mathematics [26] es una librería header-only en C++ para software gráfico basada en la especificación del lenguaje de shaders de OpenGL, GLSL. Provee clases y funciones diseñadas e implementadas con mismas convenciones y funcionalidades que las que ofrece

GLSL, así el programador que ya conoce dicho lenguaje de shaders, conoce esta librería también, haciéndola de fácil uso. Incluye transformaciones de matrices, quaternions, números aleatorios, ruido, etc.

2.5.2 GLEW

Siglas de OpenGL Extension Wrangler Library [27], es una librería multi-plataforma open-source para C y C++, para la carga de extensiones de OpenGL. Provee mecanismos eficientes para determinar en tiempo de ejecución qué extensiones son soportadas por la plataforma *target* para saber a qué funciones GL llamar, en vez de llamar directamente a una que cierta plataforma puede no soportar. [28]

2.5.3 GLFW

GLFW [29] es una librería open-source multi-plataforma para la creación y administración de ventanas con contexto de OpenGL y manejo de *input* y eventos. Funciona como capa de abstracción y soporta el uso de múltiples monitores, teclado, mouse, joysticks, eventos de tiempo y de ventana, por polling o *callbacks*, elementos que OpenGL solo no provee. [30]

Capítulo 3: Modelo Propuesto

Teardrop Engine es un motor de renderizado 3D en tiempo real, orientado a componentes. Su objetivo es el de facilitar tareas realizadas en C++ y OpenGL, acortando los períodos de desarrollo y disminuyendo la dificultad de las mismas.

3.1 Diseño de las clases de Teardrop

En esta sección se detallan los detalles de diseño de las distintas clases que conforman a Teardrop como motor gráfico. El diseño será presentado de manera incremental para posteriormente mostrar el modelo completo.

3.1.1 Engine

La clase principal del motor es *Engine* (ver Figura 14), ésta se encarga de dirigir el flujo de ejecución de la aplicación.

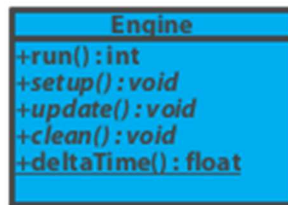

Figura 14 Clase Engine en UML

La ejecución empieza con el método `run`(), éste conforma un patrón de comportamiento Template Method [9] [31] ya que define el esqueleto de un algoritmo, siendo un método plantilla, y se encarga de llamar a los distintos métodos abstractos denominados pasos. Éstos métodos abstractos son `setup`(), `update`() y `clean`(). Se deben redefinir en la subclase, siendo ésta un hotspot para desarrollar por el usuario. Cabe destacar que el método `run`() retorna un entero que varía según se haya producido un error en ejecución o no.

El método `deltaTime`() retorna el tiempo que se tarda en renderizar un frame entero en segundos y es muy importante ya que se utiliza para hacer a los cálculos de movimiento de un objeto independientes del frame rate. Si se quiere sumar o restar a un valor en cada frame es posible que se deba multiplicar por el deltaTime. Cuando se multiplica por deltaTime lo que se hace, por ejemplo, es indicarle esencialmente al motor que en vez de mover un objeto 10 metros por frame, se lo haga 10 metros por segundo [32], logrando que el movimiento sea el mismo independientemente del hardware donde se corra el programa (el frame rate varía).

3.1.2 GameObject

El *GameObject*, abreviado como GO, es la estructura fundamental de Teardrop, cada objeto en la escena está constituido por instancias de esta clase. Aunque no logran mucho por sí

solos, actúan como contenedores de *Components* (clase que se presentará más adelante), teniendo referencias a éstos, quienes implementan la verdadera funcionalidad. La definición de esta clase se puede apreciar en la Figura 15.

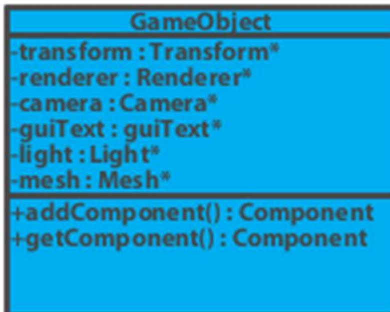

Figura 15 Clase GameObject en UML

Posee referencias a cada uno de los posibles componentes que pueda tener (puede tener uno solo de cada tipo, salvo en primera instancia de componentes de tipo script), esta relación se mostrará más adelante. Con el método `addComponent`() se los agrega al *GameObject* y con `getComponent`() se obtiene una referencia a los mismos.

Todo *GameObject* viene asociado por defecto con el componente *Transform* (para representar posición, escala y orientación) y no es posible removerlo, ya que una entidad sin ubicación en el mundo virtual carece de sentido. Esta asociación se da en el momento en que se crea un *GameObject*.

Dependiendo de qué tipo de objeto se quiera crear, se le agregan diferentes combinaciones de componentes. Por ejemplo, en la Figura 16 se observa un *GameObject* con el *Transform* obligatorio, una *Mesh* con un modelo 3D, un *Renderer* que lo dibuje en pantalla, y un *Script* para programar su comportamiento. Más detalles sobre los componentes serán vistos en la sección específica de cada uno.

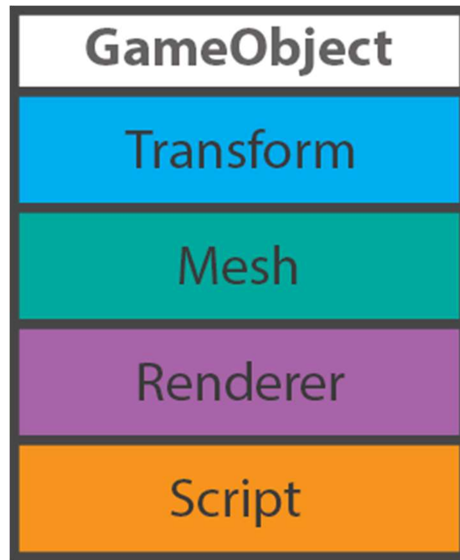

Figura 16 Ejemplo de GameObject con cuatro componentes asociados

3.1.3 Componentes

Los componentes son clases que representan el comportamiento de todo *GameObject*. Todos heredan de la clase *Component* vista en la Figura 17. Se puede apreciar que el GO conoce a sus componentes.

Figura 17 Clase Component en UML

En cuanto a la elección de qué tipos de componentes (funcionalidad) brindar en el motor, Teardrop implementa los siete tipos de componentes como se puede apreciar en la Figura 18 enfocándose en materias de iluminación y no de un motor de renderizado 3D completo; por eso no incluye componentes que añadan física a los objetos, ni sonido. Esta elección está basada en la investigación de motores existentes en la actualidad como son Unreal Engine y Unity. A su vez, este diseño propuesto permite la fácil extensión a futuro (solo por parte de desarrolladores del motor, no del usuario) de diferentes tipos de funcionalidad, solo basta con subclasificar de *Component*.

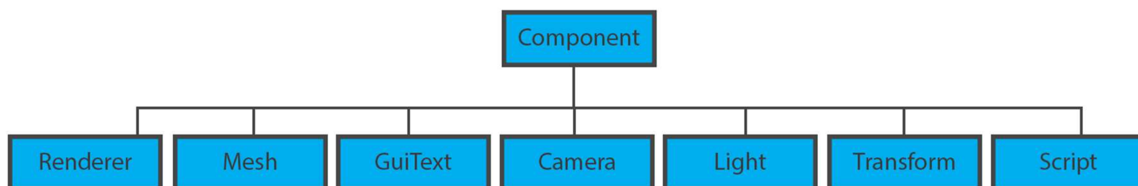

Figura 18 Los diferentes tipos de componentes

A continuación se brindarán más detalles de cada uno de los componentes presentados en la Figura 18:

- *Renderer*

  El componente *Renderer* le da la propiedad, al *GameObject* que lo posea, de renderizarse en pantalla. Para ello necesita un material, el cual le indica cómo. En la Figura 19 se aprecia cómo la clase *Material* a su vez está compuesta, es decir, posee referencias, a un objeto de tipo *Shader* y una *Texture* a ser aplicada en el modelo a renderizar. Este componente necesita la presencia, en el GO que lo contiene, de un componente *Mesh* (visto más adelante) con los vértices del modelo a renderizar. El *Renderer* se encarga de pedirle las matrices de posición, rotación y escala al componente *Transform* y prepararlas como entrada a los shaders (a los programas vertex y pixel shader que corren en la GPU, no se debe confundir con el objeto de tipo *Shader* del framework, que solo actúa como *wrapper* de éstos).

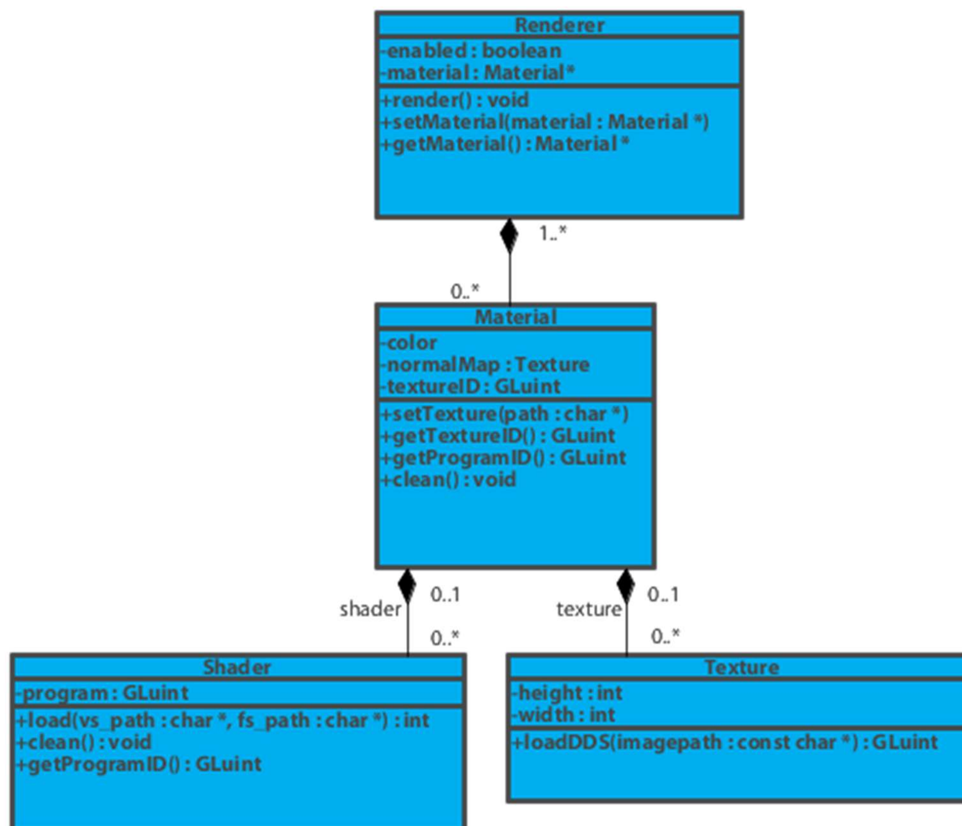

Figura 19 Clase de Componente Renderer con Material asociado en UML

El *Material* no agrega funcionalidad sino que actúa como contenedor. Su utilidad es la de proveer en el framework varios materiales ya configurados para la obtención de distintos tipos de efectos lumínicos. Entre sus diferentes parámetros de configuración se encuentran el color, la posibilidad de agregar texturas y normal maps, y los shaders que utilice.

La clase *Shader* representa los dos tipos de shaders utilizados en el motor, se encarga de cargar los scripts de los mismos desde el disco duro, los valida y compila en un solo objeto denominado *shader program*, un objeto binario alojado en la placa de video [21], manteniendo referencias al mismo (variable *program* de la clase *Shader*) para posterior uso mediante el getter `getProgramID`().

La clase *Texture* provee un método para cargar el formato de texturas DDS, propietario de Microsoft, directamente descomprimible en el hardware de la GPU y consolas como la PlayStation 3 y la Xbox 360. Este formato es importante no solo por poder guardar la información descomprimida, sino por permitir mipmapping, una técnica para guardar varias veces la misma imagen en distintos tamaños [33]. Así al ver un modelo desde distintas distancias, se elige automáticamente el tamaño acorde y se evitan imperfectos de blurring o morié patterns (al renderizar una textura grande en un pequeño subconjunto de vértices).

- *Mesh*

  El componente *Mesh* se encarga de manejar modelos 3D, provee métodos para cargarlos desde distintos formatos como el estándar OBJ que almacena los vértices, mapeo de texturas y normales del mismo en texto plano, y un formato binario propietario de mucha mayor velocidad por no requerir parseo (diferencias entre 30 segundos y 11 milisegundos). Una vez cargado el modelo a memoria RAM se trasmite la información del modelo a buffers alocados en la GPU, y quedan allí por el resto del tiempo de vida de la aplicación o hasta que el usuario los desee eliminar. En las primeras épocas de renderizado se solía enviar la geometría a la GPU una vez por frame, lo que resultaba muy ineficiente.

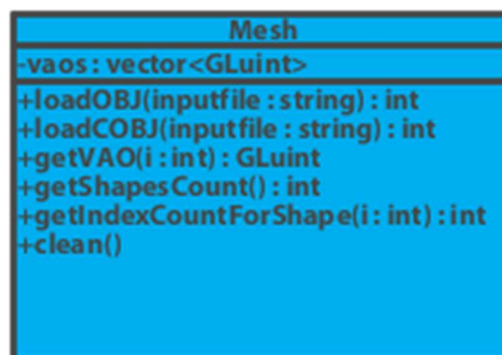

Figura 20 Clase Mesh en UML

Posee además métodos getter para indicarle al *Renderer* del su mismo GO en qué buffer se encuentra el modelo a renderizar.

- *GuiText*

  GuiText está diseñado para mostrar texto en 2D en pantalla, es decir, elementos de GUI (Graphical User Interface). Este se puede apreciar en la Figura 21.

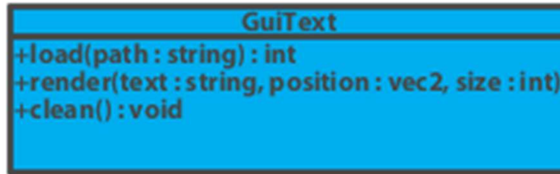

Figura 21 Clase GuiText en UML

OpenGL no posee por defecto manera de procesar texto ni fuentes tipográficas. Una manera es la de generar un mapa de caracteres como el de la Figura 22 en una textura, y renderizar en quads cada letra. Esta técnica se denomina Texture Mapped Text.

! " # $ % & ' ( ) * + , - .
/ 0 1 2 3 4 5 6 7 8 9 : ;
< = > ? @ A B C D E F
G H I J K L M N O P Q
R S T U V W X Y Z [\ ]
^ _ ' a b c d e f g h i j k
l m n o p q r s t u v w
x y z { | } ~

Figura 22 Ejemplo de mapa de caracteres [35]

El problema con este método es el de tener que generar el mapa de caracteres en una aplicación externa, además de tener que hacerlo por cada tipografía y tamaño deseado.

La técnica utilizada en Teardrop involucra la librería FontStash [36] que crea este mapa o atlas de caracteres bajo demanda, es decir en ejecución, permitiendo la carga dinámica de diferentes fuentes en formato TTF, y su uso en diferentes tamaños sin pérdida de calidad.

El método `load`() recibe como parámetro la ruta del archivo de fuentes para que FontStash cree el atlas a utilizar.

El método `render`() renderiza el texto que recibe como parámetro en *text*, en la posición y tamaño indicados en *position* y *size* respectivamente.

- *Camera*

  El componente *Camera* (ver Figura 23) es esencial ya que sin este no podríamos observar la escena del mundo virtual. Se encarga de computar la matriz de proyección, que luego el *Renderer* toma para enviarla a los shaders. Pueden existir varias cámaras en la escena, pero como solo se puede ver la misma desde una a la vez, se utiliza la primera habilitada en una cola de cámaras.

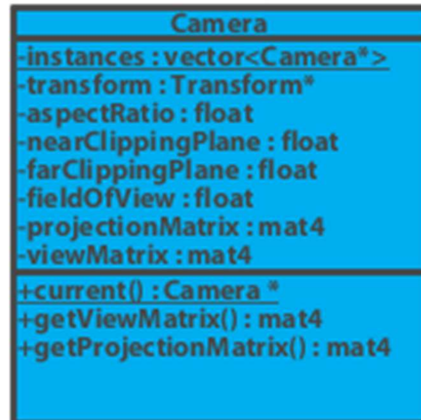

Figura 23 Clase Camera en UML

  Para configurar los parámetros de la matriz de proyección, la cámara maneja el campo de visión o *field of view* [37], que es el ángulo de visión o porción observable del mundo. En el humano este suele ser de casi 180 grados, pero el parámetro se suele setear entre 45 y 60 grados.

  El *aspect ratio* o relación de aspecto indica la proporción entre el ancho y el largo de la imagen generada, calculado dividiendo el ancho por la altura. Este valor depende de la resolución de pantalla del monitor target, siendo las más comunes 4:3 para resoluciones como 800x600 y 16:9 para la resolución full HD (1929x1080).

  Finalmente, permite la posibilidad de configurar los parámetros *near clipping plane* y *far clipping plane* con los que se manejará la cámara.

- *Light*

  El componente de iluminación permite indicarle a un *GameObject* que éste será una fuente de luz, indicando propiedades del material de la misma, para tener en cuenta por los shaders al renderizar la escena. La clase *Light* se puede apreciar en la Figura 24.

Figura 24 Clase Light en UML

Las propiedades de la luz varían según el modelo de iluminación a utilizar, aunque por defecto se utiliza el modelo Phong, teniendo en cuenta los tres aspectos básicos vistos en la Sección 2.3.1, no solo de la luz sino también del material, la componente ambiental, la difusa y la especular. Estos parámetros se multiplican entre sí entre el material y las fuentes de luz para computar el valor final de color que tomará el modelo o escena a renderizar.

- *Transform*

  Transform es un componente obligatorio en cada GO, el mismo se puede apreciar en más detalle en la Figura 25. Indica la posición en el mundo virtual en la que se encuentra el *GameObject* en todo momento, su rotación y escala. Sirve para multiplicar cada vértice del modelo por estos valores y así moverlo por el sistema de coordenadas 3D, utilizar los valores en shaders para determinar cómo incide la luz sobre ellos, computar colisiones entre cuerpos si se incluyera física en la simulación, etc.

Figura 25 Clase Transform en UML

La clase *Transform* posee métodos para obtener la *model matrix*, setters para los diferentes parámetros, realizar rotaciones con métodos específicos para cada eje (pitch sobre el x, yaw sobre el z y roll sobre el eje y) y realizar traslaciones de *x* unidades en cada eje por separado. Además, posee setters y getters para el lookAt, un vector de tres

dimensiones que indica hacia donde mira el GO que posea dicho *Transform*, dentro del mundo virtual.

- *Script*

  El componente *Script* (ver Figura 26) presenta el principal punto de personalización de los *GameObjects*, donde al programador usuario del motor se le permite programar el comportamiento del mismo, no solo la configuración de los demás componentes del mismo GO, sino también la interacción con los otros GOs de la escena. También se puede manejar el *input* de usuario, o la respuesta ante determinados eventos externos.

Figura 26 Clase Script en UML

La clase Script posee los métodos abstractos `setup`(), `update`() y `clean`() para ser subclasificados por el usuario. Se verán en más detalle en la Sección 3.2, como punto de extensión del framework.

3.1.4 MainRenderer

La clase *MainRenderer* se encarga de mantener una referencia a todos los componentes *Renderer* de la escena, para poder de manera centralizada, pedirle a cada uno de cada *GameObject* que renderice lo que le corresponda.

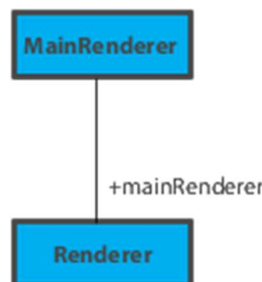

Figura 27 Clase MainRenderer en UML

Para lograr este comportamiento, se utiliza una ligera modificación del patrón de diseño de comportamiento Observer [9], en el cual observadores (en este caso los componentes *Renderer*), se subscriben al sujeto (*MainRenderer*) ni bien son creados. Cuando el sujeto necesita renderizar, en vez de indicarles a los observadores que cambió su estado interno (como la versión original del patrón sugiere), les indica, llamando al método `render`() en cada frame, en orden de suscripción y solo a los que se encuentran activos, que rendericen.

En la Figura 28 se ilustra la secuencia de métodos llamados así como el loop en donde el *MainRenderer* notifica a todos los *Renderers* que rendericen.

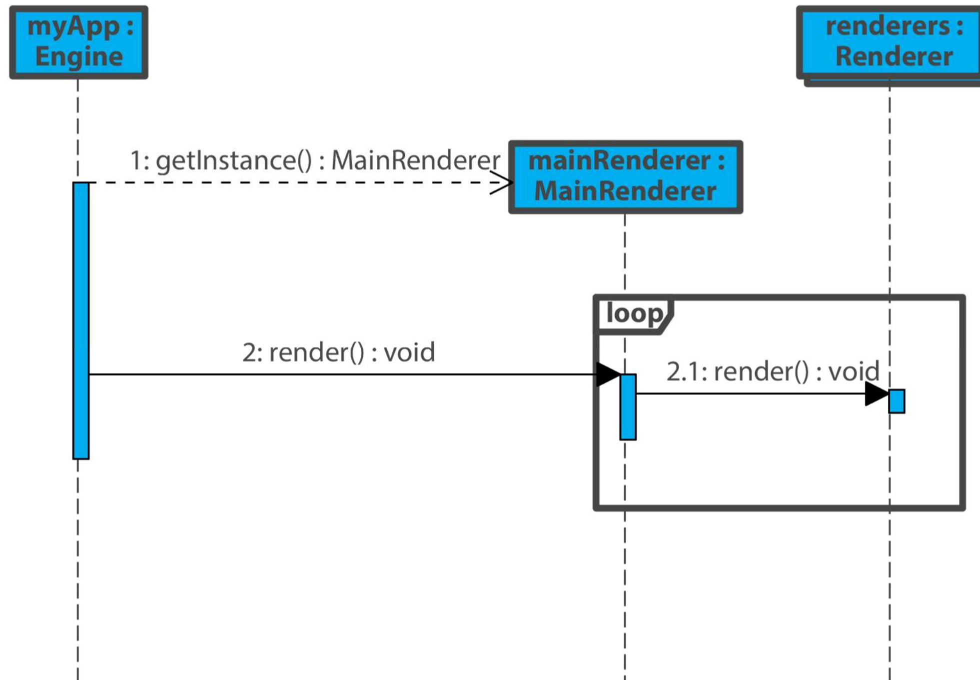

Figura 28 Diagrama de secuencias del renderizado

Se presentaron dos opciones a la hora de diseñar la clase *MainRenderer*. Al saber que solo se tendría una instancia en la ejecución del programa, se podía optar por hacer estáticos a los métodos pero se eligió la opción del patrón de diseño creacional Singleton [9] [42], en el cual se tiene una sola instancia de la clase y se la obtiene a través de un método estático o de clase. Este método instancia un objeto de su misma clase si todavía no se lo ha hecho, y luego retorna su referencia. Esto provee acceso global a la instancia única.

3.1.5 MainDeferredRenderer

Luego de la incorporación de la técnica de rendering diferido, se debió subclasificar de esta clase para crear la clase *MainDeferredRenderer*. Esta clase en vez de solo delegar la responsabilidad de renderizar a cada componente *Renderer*, primero se encarga de configurar el G-Buffer, consistente de tres texturas. Además, carga los shaders necesarios para la primera y segunda pasada de rendering, en donde la primera consiste en llenar el G-Buffer, para consumirlo luego en la segunda. Al renderizar, antes de llamar a cada *Renderer*, activa el G-Buffer para que escriban sobre él, y luego de notificar a los *Renderers* llama a la segunda pasada de la técnica, que dibuja el contenido del buffer en un quad del tamaño de la pantalla target (la designada a mostrar la imagen).

3.1.6 MainSSAORenderer

La clase *MainSSAORenderer* es similar a la clase *MainDeferredRenderer* en el sentido en que ambas configuran el G-Buffer con tres texturas, y cargan dos shader programs, uno para

cada pasada. La principal diferencia está en que estos shaders son otro conjunto de programas distintos para implementar la técnica de Screen Space Ambient Occlusion, descripta en la Sección 2.4.2 Ambient Occlusion y por lo tanto necesitan hacer uso de distintos uniforms.

Para implementar esta técnica necesitamos trazas líneas desde cada posición en screen space sobre una dirección al azar y determinar la cantidad de oclusión obtenida en cada punto a lo largo de dicha línea. [8] El pixel shader de la 2da pasada leerá valores de profundidad de cada posición, guardados por nosotros en la 1ra pasada, elegirá una dirección aleatoria y tomará varios pasos sobre dicha dirección

Para elegir una dirección al azar, esta clase se encarga de guardar en el G-Buffer un conjunto de 256 vectores aleatorios en un buffer uniform, todos contenidos dentro de una esfera de radio unitario (normalizados). [8]

3.1.7 Input

La clase Input ofrece métodos para la manipulación de eventos de entrada de teclado y mouse. Está pensada para actuar de wrapper o contenedor, sobre métodos y constantes del manejo de input de otras librerías, haciendo más fácil su uso. Esta clase se puede apreciar en la Figura 29.

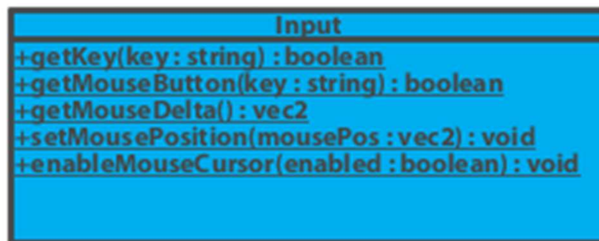

Figura 29 Clase Input en UML

Sirve para saber si se presionó determinada tecla con el método getKey() o botón del mouse con getMouseButton(). Se pueden obtener las coordenadas del mouse con getMousePosition(), setear su posición con setMousePositioN(), obtener el delta (desviación en cierto tiempo de las coordenadas con respecto al centro de la pantalla) con getMouseDelta() y habilitar o deshabilitar el cursor con enableMouseCursor().

3.1.8 FPSController

Esta clase FPSController ayuda al usuario del motor a manejar la cámara, y de por sí no es indispensable en el framework pero facilita varias operaciones, por eso se la denomina clase helper.

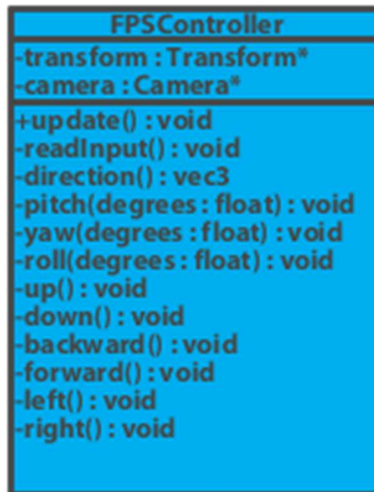

Figura 30 Clase FPSController en UML

Recibe en su constructor el *Transform* y la *Camera* de un *GameObject* y mapea la entrada del usuario obtenida por la clase *Input* a movimientos. Posee métodos para moverse en todos los ejes del espacio tridimensional así como para rotar sobre los mismos. Además aplica rotational y velocity damping a los movimientos, esto consiste en no decrementar la velocidad del movimiento bruscamente al detenerse, sino de hacerlo gradualmente (pudiéndose configurar qué tan gradual), produciendo un efecto de suavidad agradable a la vista.

Para su uso se debe instanciar y luego llamar a `update`() en cada frame, para que, en resumen lea la entrada del usuario, calcule y aplique las traslaciones y rotaciones y luego aplique el damping.

3.1.9 Diagrama de clases completo

Luego de detallar en partes cada clase con sus métodos, referencias y patrones utilizados, en la Figura 31 se observa el diagrama de clases completo del motor Teardrop, con las relaciones entre las distintas clases, y en dónde se utiliza cada patrón dado. Por simplicidad se sacó el comportamiento de cada clase, ya que las mismas fueron detalladas anteriormente. Se puede apreciar cómo en el diagrama se define *myApp*, esta clase se observa como subclase de *Engine* y conoce a sus *GameObjects*. De esta manera, se aprecia uno de los puntos de extensión del framework propuesto.

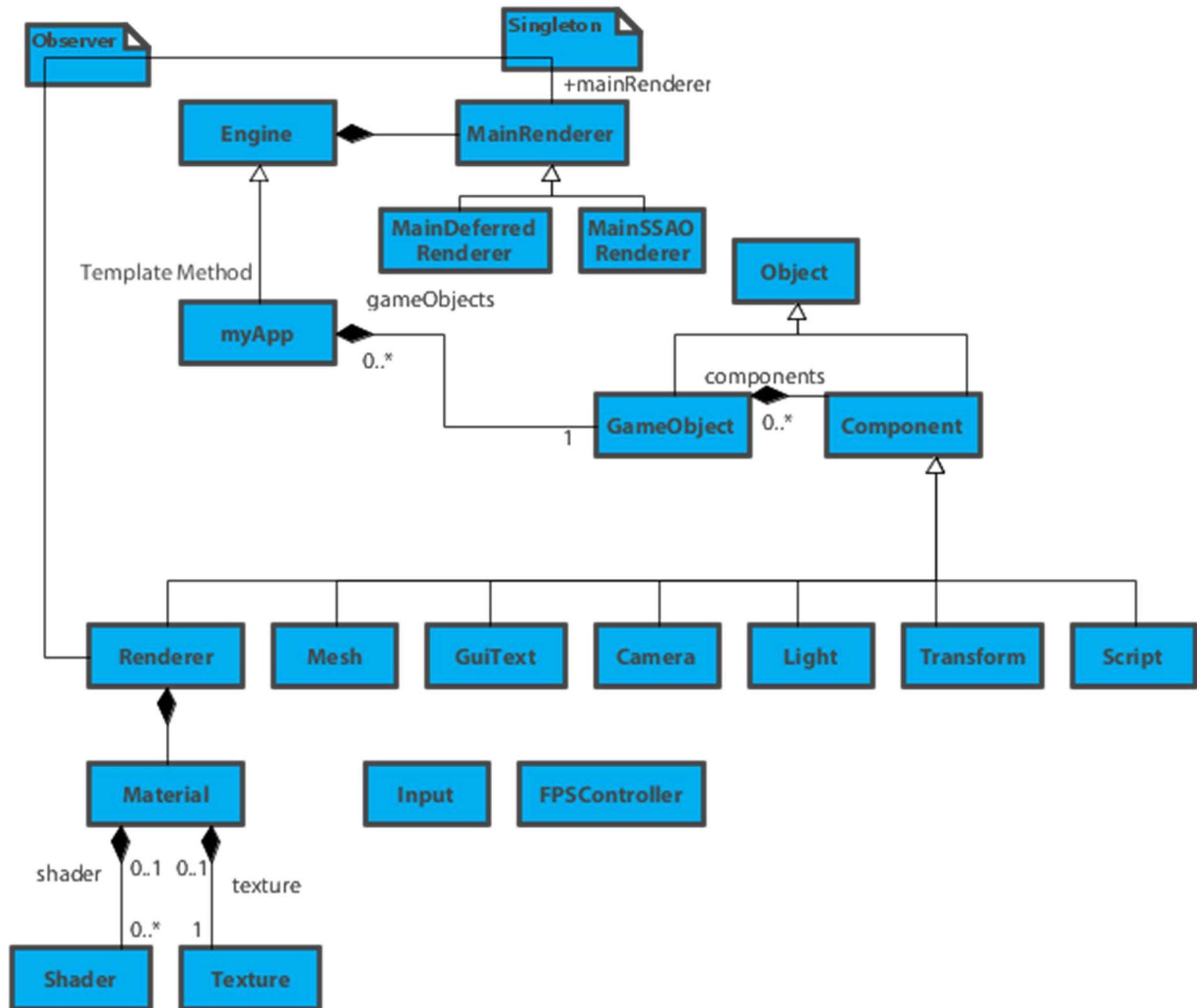

Figura 31 Diagrama de clases completo en UML

3.2 Extendiendo el modelo

Una vez entendido el funcionamiento del framework veremos cuáles son los puntos de extensión del mismo, en los que el usuario puede agregar comportamiento propio y así, crear aplicaciones.

3.2.1 Subclasificando Engine

La principal manera de extender el motor es creando una subclase de *Engine*, la clase central del motor. Esta clase, en futuras versiones del motor (como se describirá en más detalles en la Sección 6), sería manejada por un editor gráfico en vez de con código ya que la misma sirve para configurar la escena, instanciar *GameObjects*, agregarles componentes, configurarlos indicándoles qué archivos cargar (texturas, modelos, shaders, etc.).

De la clase *Engine* se heredan tres métodos abstractos a redefinir, éstos son `setup`(), `update`() y `clean`().

En el método `setup`() el usuario inicializa variables y configura la escena. Deja los objetos a utilizar preparados para el loop principal.

En el método `update`() se actualizan propiedades dinámicas de los objetos, como por ejemplo, la posición de los modelos en pantalla. También se capturan eventos de entrada producidos por el usuario de la aplicación.

El método `clean`() permite realizar las últimas operaciones antes de finalizar la ejecución del programa, como borrar objetos de memoria.

3.2.2 Subclasificando Script

Otra manera de agregar comportamiento propio es a través de sublclasificaciones de *Script*, componentes que también poseen los métodos `setup`(), `update`() y `clean`(). Éstos, una vez agregados a un GO, se encargan de manejar los demás componentes del mismo, obteniendo así mayor organización y separación de código, en vez de ubicar el comportamiento de todos los GOs en el archivo de la escena (subclase de Engine).

Cada *GameObject* podría tener la cantidad de componentes *Script* que desee, por ejemplo, uno para manejar el componente *GuiText* y otro para actualizar el color de la luz del componente *Light*.

3.2.3 Creación de Shaders

Una forma de modificar particularmente las propiedades visuales de los objetos es mediante la creación de distintos tipos de *Shader* scripts. El motor ya incluye un conjunto de shaders, pero se pueden agregar nuevos cargándolos en un objeto de clase *Shader* y luego asignando éste a componentes de tipo *Material*.

3.3 Ciclo de ejecución de Teardrop

Gracias al patrón Template Method, la clase *Engine*, cómo se mostró en la Sección 3.1.1, ejecutará mediante el método `run`() los métodos creados por el usuario en una subclase. Cómo se observa en la Figura 32, el ciclo de ejecución comienza en el método `setup`() que se ejecuta una sola vez.

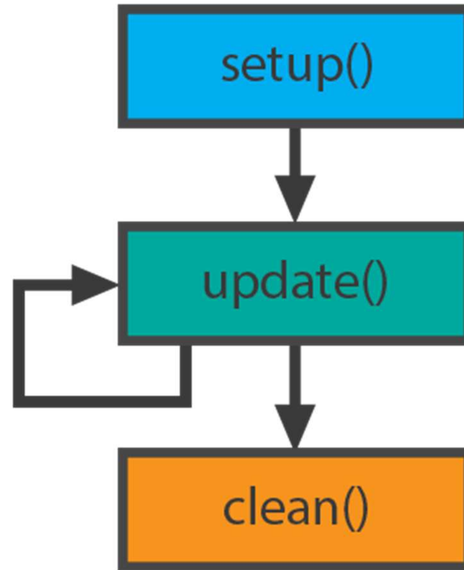

Figura 32 Ciclo de ejecución de Teardrop

Luego, se llama al loop principal del motor en `update`(), este método se ejecutará indefinidamente hasta que un evento externo le indique lo contrario.

Cuando se produce un evento que termina la ejecución del loop principal o *main loop*, como por ejemplo, si el usuario presiona la tecla Escape, se llama al método `clean`() y luego de éste finaliza la ejecución de la aplicación.

Capítulo 4: Desarrollo Realizado (Teardrop)

A lo largo de este capítulo se describe la implementación de todas las clases del motor gráfico Teardrop (acorde al modelo presentado en el Capítulo 3), así como se explican ciertos problemas encontrados al momento de desarrollarlas. Luego se muestran ejemplos generales de su uso.

Como se da a suponer a partir de las definiciones descriptas en el Capítulo 2, se utilizó la *API gráfica OpenGL* para el control de la placa de video, y no la contraparte propietaria *DirectX de Microsoft* [34]. En cuanto a la elección, ésta se basó principalmente en la cantidad de fuentes bibliográficas, tutoriales y abundancia de material encontrados sobre OpenGL. El material troncal de investigación fue el libro OpenGL SuperBible en su 5ta y 6ta edición.

Por utilizar C++ las clases se dividen en dos archivos, los *headers* o cabeceras con extensión ".h" en donde se declara la interfaz de la clase, es decir, las variables y los métodos pero sin su implementación. Luego, en los archivos de implementación con extensión ".cpp" se encuentra la implementación de los métodos declarados en la interfaz.

4.1 Implementación de las clases de Teardrop

A continuación se tomará cada clase presentada en el Capítulo 3 y se describirá cómo la misma fue implementada.

- Engine

  La clase *Engine* es la principal por manejar el ciclo de ejecución con el método run(). Éste inicializa la librería GLFW finalizando la ejecución si la misma falla, luego crea una ventana gráfica del sistema operativo de 1280x720 pixeles, aunque puede configurarse a elección y se la setea como contexto actual, es decir, le pasa la ventana al thread activo. Sacarla de contexto forzaría a vaciar el pipeline gráfico [62]. Luego se le pasa por parámetro una función de callback a GLFW indicándole para la ventana utilizada a qué función llamar cuando se presiona una tecla y allí solo se indica que al presionar Escape se cierre la aplicación.

  Se inicializa GLEW y luego se procede a configurar OpenGL indicándole que se quiere realizar testeo de profundidad (si un vértice está más lejos que otro de la cámara se descarta) y cómo hacerlo. Luego, se le indica a la clase Input cuál es la ventana, la necesita para saber el tamaño de la misma.

  Se llama al primer método abstracto, setup() y luego se entra en un condicional while hasta que GLFW determine que se debe cerrar la ventana; se determina chequeando un flag que se setea en el número uno cuando el usuario aprieta la el botón de cerrar de la ventana o la combinación de teclas Alt+F4. La ventana no se cerrará automáticamente sino que se debe destruir la ventana manualmente. Dentro del mismo se vacían los buffers de OpenGL de color y de profundidad y se le indica a *MainRenderer*

que mande a renderizar a todos los *Renderers*. Si hay una cámara en la escena se le indica que actualice sus matrices. Se llama al segundo método hook-up o de enganche del Template Method, update(). Como GLFW usa una técnica de *buffering* doble[43], se swapean (intercambian) los buffers de la ventana, esto se hace porque en realidad todo se escribe sobre un buffer trasero fuera de pantalla mientras se muestra el ya escrito previamente, el buffer frontal. Cuando el frame entero se renderizó, los buffers deben intercambiarse entre sí, por lo que el buffer trasero se transforma en el frontal y viceversa; esto es para evitar efectos visuales no deseados en los que se podría ver la imagen procesada parcialmente. También, si los buffers se swapean instantáneamente se puede producir *screen tearing* [44], un desperfecto en donde la pantalla muestra parte de dos frames en simultáneo, en un solo refresco de pantalla, produciendo un efecto desagradable de corte entre las dos imágenes.

Como último paso del bucle condicional, se llama a una función de GLFW que procesa eventos, ésta es necesaria para que GLFW se comunique regularmente con el sistema de ventanas para recibir eventos y saber que la aplicación no se ha trabado. Se suele hacer en cada frame luego de haber swapeado los buffers. Se usa glfwPollEvents() que solo procesa los eventos que ya se han recibido y retorna inmediatamente, la mejor opción para sistemas que renderizan continuamente.

Finalmente, antes de terminar la ejecución se llama al último método hook-up clean() y se le indica a GLFW que destruya la ventana y finalice la ejecución. Se retorna el número cero para indicar que no se produjeron errores.

El método deltaTime() obtiene el tiempo de ejecución desde que GLFW se inicializó, lo hace dos veces. Luego resta estos tiempos entre sí para calcular cuántos milisegundos tardó en renderizarse el frame y retorna dicho número (parámetro de solo lectura).

Esta clase no se instancia, sino que se subclasifica de ella para utilizar el motor. Con una macro de C++ se crea una función llamada DECLARE_MAIN que recibe como parámetro un tipo genérico A. Dentro de esta función se encuentra el método main de toda aplicación C o C++. Allí se instancia ese tipo A y se llama al método run() del mismo. Entonces, la subclase de *Engine* que quiera utilizarse como clase principal deberá llamar a DECLARE_MAIN pasándose como parámetro a sí misma.

- GameObject

  La clase *GameObject* en su constructor, que no recibe ningún parámetro, inicializa un nuevo componente *Transform* por ser obligatorio que posea uno. Tiene punteros nulos a cada tipo de componente existente en el framework y además posee setters y getters para éstos. Estos métodos utilizan plantillas de funciones o *templates* [47], es decir, se define una función genérica de tipo T y luego se especializa este template con los distintos tipos de componentes. Así, si uno quiere agregar un componente de tipo *Camera* al GO utiliza la nomenclatura GO.addComponent<Camera>() y se llama

específicamente a la función que instancia un componente en el puntero de tipo Camera. Sin esta funcionalidad tendrían que crearse las funciones addCamera, addLight, etc., por cada componente.

Se presentaron problemas a la hora de implementar el setter y getter para el componente de tipo *Script*, por ser este el único que podría según diseño, agregarse más de una vez a un GO. Se podía optar por determinar que se pueda agregar un solo componente o ninguno y que se escriba la lógica del programa en la subclase de *Engine*. Para esta versión del prototipo se optó por la segunda opción.

- Component

  La clase *Component* por el momento no posee ningún tipo de implementación, sino que solo actúa como clase abstracta de los componentes. Posee una referencia al GO que lo contiene, pero en vez de asignarse en el constructor de esta clase, esta referencia se asigna en el constructor de cada componente individual, ya que por ser una sola línea de código era prácticamente lo mismo llamarla en cada componente o llamar al constructor de la superclase. De agregarse más líneas en común entre los componentes, se subirá el comportamiento a la superclase (es decir, a *Component*).

- Renderer

  La clase *Renderer* en su constructor se encarga de añadirse a sí mismo a la cola de renderers del *MainRenderer* llamando a *MainRenderer*::getInstance()->attach(this).

La clase *Renderer* posee getters y setters para añadir el *Material* con el cual renderizar, además de un setter especial que recibe un arreglo de materiales en vez de uno solo, esto se debe a que los modelos 3D pueden contener más de una figura por archivo y se desea renderizar cada una con un material diferente.

Luego vienen los métodos más importantes, `render`() y `deferred_render`(). El método `render`() comienza obteniendo el componente *Transform* del GO que lo contiene y que tienen en común, además de la *Mesh*, y pide a la clase *Camera* la cámara actual. Todos estos componentes se necesitan para calcular matrices y luego renderizar la escena.

Se entra en un condicional if en donde se pregunta si de hecho hay mesh y cámara (podrían estar en null). Si se cumplen estas condiciones se computa la modelViewMatrix, multiplicando la ViewMatrix de la cámara por la ModelMatrix del *Transform*. Se computa a su vez, la matriz MVP o ModelViewProjection multiplicando la ProjectionMatrix de la cámara por la modelViewMatrix recién creada.

Se sigue con otro condicional que pregunta si la cantidad de materiales que posee el Renderer es igual a 1 o más. Solo cambia en que si es un solo material se aplica el mismo a todas las *shapes* o figuras que posea la *Mesh* y si no, uno distinto por cada shape (la cantidad de materiales y shapes deben ser iguales).

Dentro de este bloque de código, se *bindea* con `glBindTexture()` la textura del material a utilizar, es decir, se le indica a OpenGL qué textura utilizar a continuación, y se indica que ésta es del tipo GL_TEXTURE_2D, o sea, una textura de dos dimensiones. Luego se obtiene el identificador del shader ya alocado en la placa de video, correspondiente al material a utilizar, y se le indica a OpenGL que se desea utilizar dicho shader con la instrucción `glUseProgram()`. Con `glGetUniformLocation()` se obtiene la ubicación del uniform (variable que no cambia para los diferentes vértices o pixeles del shader) "MVP" dentro del shader y se envía la información computada previamente de MVP, con glUniformMatrix4fv(). De la misma manera se envía a la placa de video la información de la modelViewMatrix y de la projectionMatrix. Finalmente, para cada shape del modelo, se bindea el VAO correspondiente con `glBindVertexArray()` y se le pide a OpenGL que dibuje los triángulos (denotado por GL_TRIANGLES) de cada figura con `glDrawElements()`, indicando la cantidad de vértices indizados en el índice de la figura.

El método `deferred_render()` es en su mayoría idéntico a `render()`, con la diferencia de que en vez de utilizar el shader del *Material* que tenga asociado, utiliza el shader de rendering diferido para "dibujar" las figuras en el G-Buffer. Recibe el identificador del shader de rendering diferido por parámetro.

- Mesh

  En la clase *Mesh* los métodos más importantes son los de cargar modelos 3D del disco a la memoria de la placa de video, pasando primero por la memoria principal o RAM. Por el momento Teardrop soporta dos tipos de formato, el formato ".obj" y el *custom* ".cobj".

  El método `loadOBJ()` recibe como parámetro un String indicando la ruta del modelo a cargar. Luego deriva el trabajo de cargar el formato ".obj" a una pequeña librería de terceros llamada TinyOBJLoader [45] creada por Syoyo Fujita [46], que sube el modelo a RAM. Luego se llama al método `setUp()` de *Mesh*, descrito más adelante.

  El método `loadCOBJ()` carga el formato binario ".cobj" y fue creado por Roald Fernandez, de SwarmingLogic [62], Teardrop solamente lo utiliza. Luego de subirlo a RAM también llama al método `setUp()`.

  En ambos métodos se cuenta el tiempo tardado en cargar el modelo para medir la performance de los mismos y observar las diferencias entre un modelo guardado en texto plano y en formato binario. El método custom siempre es más rápido porque carga el modelo a RAM a partir de un binario que representa al modelo tal cual como es guardado en memoria. En cambio el de texto plano debe leer cada línea del archivo y parsear si lo que está leyendo es un vértice, una normal, una coordenada de textura, etc, y eso por cada uno de los miles de vértices que un modelo suele tener.

El método `setUp()` contiene un bucle for que llama a `setUpShape()` por cada shape que se encuentre en el modelo. Luego desbindea el último VAO para que no quede activo y desde otro lado modifiquen su estado. El método `setUpShape()` se encarga de cargar cada figura del modelo a la placa de video. Para eso, primero genera un VAO con `glGenVertexArrays()` para cada una, guarda su referencia y lo pone como activo. Luego se genera un VBO en donde cargar los vértices con `glGenBuffers()`, se bindea y luego se suben los vértices a la memoria de la GPU con `glBufferData()` donde se indica el tipo de buffer a alimentar, el tamaño en bytes a subir, el puntero a la primera posición de la figura en RAM, y finalmente la constante GL_STATIC_DRAW que indica el patrón de uso que se le dará al buffer (frecuencia de acceso y la naturaleza del mismo), permitiendo que OpenGL maneje de diferentes maneras el acceso al mismo, mejorando su performance [46].

Pasando como parámetro un número a la instrucción `glEnableVertexAttribArray()` se le indica a OpenGL que a continuación se describirá qué tipo de información se usará como input al shader en la ubicación con dicha numeración (En GLSL a cada variable de entrada se le puede asignar un número de ubicación o *location*). Se describe dicha información con `glVertexAttribPointer()` indicándole nuevamente que location se está describiendo, cuántos elementos primitivos tiene ese tipo de variable (un vector 3D tiene las 3 coordenadas), de qué tipo de datos son (por ejemplo GL_FLOAT), si los vectores están normalizados o no, si la información se almacena con stride (si en un mismo buffer se almacenan, por ejemplo, tanto vértices como otro tipo de información, aquí se indica cada cuantos bytes vuelve a encontrarse la información de otro vértice [48]) y si posee un offset dentro del buffer.

Se repite el proceso de generar un buffer, bindearlo, cargar la información y definir los datos de entrada al shader, pero esta vez con las coordenadas de texturas o UV, otro para las normales, y otro para los índices de los vértices (a este no se le indica información de input al shader ya que OpenGL los maneja automáticamente).

Finalmente, en la clase *Mesh* solo quedan setters, como `getVAO()` que devuelve el identificador del VAO de la figura que se pase por parámetro, `getShapesCount()` que retorna la cantidad de figuras del modelo, `getIndexCountForShape()` que devuelve la cantidad de vértices indizados para la figura que se pase por parámetro y por último, el método `clean()`, que para cada figura se encarga de borrar los VBOs de índices, coordenadas, vértices y normales y el VAO.

- Camera

La clase *Camera* realiza un poco más de trabajo en su constructor ya que se añade a sí misma a la colección estática de cámaras. Además se guarda una referencia al *Transform* del GO que lo contiene para posterior uso e inicializa las variables de enabled en

verdadero, fieldOfView en 45 grados, aspectRatio en 16/9 (depende de la ventana creada, 1280x720 pertenece a este aspecto), nearClippingPlane en 0.1, farClippingPlane en 1000 y luego llama a `setUpProjMatrix()` que configura con la ayuda de la librería GLM y estos datos proporcionados, la matriz de proyección.

El método `current()` recorre la colección de cámaras en busca de la primera cámara que esté habilitada (propiedad enabled) y la retorna.

Luego quedan los getters de la matriz viewMatrix y projectionMatrix y finalmente el método `update()` que actualiza la matriz viewMatrix con ayuda del método `lookAt()` de GLM, pasándole la posición del *Transform* del GO, su lookAt (hacia dónde mira el mismo) y su Up (en qué dirección es arriba para el *Transform*).

- Transform

  La clase *Transform* inicializa en la matriz identidad a su modelMatrix dentro de su constructor, así como también setea en 0 la posición, el look_at en (0,0,-1), es decir, mirando hacia el lado negativo del eje Z; también setea la dirección arriba como (0,1,0) y la rotación en la identidad de un quaternion.

  Posee mayoritariamente métodos setter y getter, para la posición recibiendo como parámetro un vector de tres dimensiones (tipo vec3), o las componentes X, Y y Z individualmente; para la dirección de arriba, el lookAt y la modelMatrix.

  Luego tiene métodos para aplicar movimiento, como `applyRotation()` que recibe un ángulo y un vector de orientación, con los que calcula un quaternion y multiplica a la modelMatrix por la matriz generada a partir de este quarternion. El método `yaw()` aplica una rotación en ángulos pasados por parámetro sobre el eje Y, `pitch()` sobre el eje X y `roll()` sobre el eje Z.

  El método `translate()` actualiza la posición sumándole el vec3 de traslación que recibe como parámetro, el look_at sumando la posición y dirección actual del Transform, y la modelMatrix se actualiza con ayuda de `translate()` de GLM. También existe `translate()` con parámetros individuales para los ejes X, Y y Z.

  El método `rotate()` recibe los ejes por separado, y realiza un producto cruzado entre ciertos quaternions para computar una matriz de rotación la cual finalmente es aplicada a la modelMatrix.

  Finalmente se tienen los métodos `scale()`, uno con ejes separados, otro con un escalar uniforme a los tres ejes, éste método escala la modelMatrix con ayuda de `scale()` de GLM.

- Material

  Como se vio previamente en el Capítulo 3, la clase *Material* actúa de wrapper de los shaders y texturas por lo que no posee mucho comportamiento. En su constructor

recibe la ruta del vertex y pixel shader a cargar, crea un nuevo shader y llama al método `load()` del mismo con dichas rutas como parámetro. Luego inicializa una nueva textura.

Finalmente posee un setter para la textura, en el cual se pasa la ruta de la misma y con ella se llama al método `loadDDS()` de la clase *Texture*; y dos getters, uno para el identificador de la textura en la placa de video y el otro para el identificador del shader program.

- Shader

  La clase *Shader* se encarga de cargar los dos tipos de shaders del disco a RAM y luego a la memoria de la placa de video. Para esto posee un método `load()` que recibe dos strings con las rutas del vertex y pixel shader. Dentro declara dos variables de tipo GLuint en donde almacenará los identificadores de cada shader y en ellos asigna el valor retornado por una sobrecarga del método `load()` que devuelve este tipo de dato. Al nuevo `load()` se le pasa la ruta del shader, el tipo (GL_VERTEX_SHADER o GL_FRAGMENT_SHADER) y un boolean que indica si se quieren verificar errores o no.

  Luego se crea un shader program con la instrucción `glCreateProgram()` asignando su identificador a otra variable, a la cual se le adjuntan los dos tipos de shader con la instrucción `glAttachShader()`. Finalmente se linkea el shader program en un solo objeto con `glLinkProgram()` creando un programa con un ejecutable que correrá en el procesador programable de vértices y otro en el procesador programable de fragmentos. Luego se eliminan los shaders individuales con `glDeleteProgram()`.

  El método sobrecargado `load()` lee el archivo del shader desde disco con `fopen()` cargando la información de este en un arreglo. Crea un shader individual con `glCreateShader()`, se le pasa el contenido a OpenGL con `glShaderSource()` y luego lo compila con `glCompileShader()`[49] quien procesara el código fuente y almacenará el estado de la compilación como parte del objeto. Este estado se puede consultar pasando como parámetro el objeto shader y GL_COMPILE_STATUS a la función `glGetShader()`. Si el mismo es GL_FALSE significa que se produjo un error e información del mismo se puede obtener con la función `glGetShaderInfoLog()` a la que se le pasa por parámetro un buffer en donde copiar los errores.

  Continuando con la implementación del método `load()`, si encuentra errores de compilación los imprime y elimina el objeto shader creado, si no lo retorna terminando su ejecución.

- Texture

  En la clase *Texture* solo se tienen los dos métodos para cargar imágenes, en formato “.bmp” y “.dds”. El método `loadBMP_custom()` carga la imagen del disco a un array, determinando a su vez propiedades como ancho, largo, tamaño (ancho x largo x 3 bytes, uno por cada color RGB) y un header de 54 bytes. Luego genera una textura con

`glGenTextures()` y se guarda su identificador, la bindea y copia la información hacia la placa de video. Finalmente configura el trilinear filtering (forma de interpolación lineal entre los diferentes mipmaps generados) y genera un mipmap.

El método `loadDDS()` carga el archivo de disco con la ayuda de una sencilla librería llamada GLI [50], genera la textura en OpenGL, la configura y luego dependiendo de si la imagen está comprimida o no, para cada nivel o mipmap de la misma, lo sube a la placa de video con `glCompressedTexSubImage2D()` o `glTexSubImage2D()`.

- MainRenderer

  La clase *MainRenderer* hace uso del patrón Singleton, por lo que tiene una variable de clase de tipo *MainRenderer* llamada "instance" donde mantendrá la instancia que los *Renderers* pedirán para subscribirse. Para hacer uso de esta variable, el método `getInstance()` pregunta, cada vez que lo llaman, si la instancia es nula para crear una nueva, y luego la retorna.

  El método `setDeferredRendering()` setea el boolean de clase "deferred" y en la variable "instance" instancia un objeto de tipo *MainDeferredRenderer* (subclase de *MainRenderer*). Así, luego de llamar este método al comienzo de la ejecución, cuando se llama a `getInstance()` se observa que ya hay una instancia y se retorna ese objeto para render diferido.

  El método `attach()` recibe como parámetro un objeto de tipo *Renderer* y lo guarda en una cola, es decir, lo subscribe.

  El método `render()` simplemente se encarga, con un bucle, de avisarle a cada Renderer habilitado de la cola que renderice.

- MainDeferredRenderer

  Esta clase es subclase de *MainRenderer*. El constructor se dedica a llamar al método `setupGBuffer()`, luego genera un VAO para referenciar al quad a pantalla completa en donde se mostrará la imagen final, luego de la segunda pasada del rendering diferido. Finalmente, crea dos objetos de tipo *Shader*, la etapa de llenado del GBuffer y la segunda pasada en donde éste se consume mostrándolo en el quad.

  El método `setupGBuffer()` genera un FBO (vistos en la sección 2.2.5 Buffers) y lo bindea para hacerlo activo y que las siguientes operaciones se produzcan sobre éste y no sobre la pantalla principal. Nos interesa guardar sobre este buffer información de color, normales, y coordenadas.

  OpenGL soporta adjuntar hasta ocho texturas por cada FBO y cada uno puede tener hasta cuatro canales de 32 bits. Sin embargo, cada canal de cada anexo consume ancho de banda de memoria y si no se presta atención a la cantidad de información que se escribe sobre el framebuffer se empieza a perder el cómputo ahorrado por el rendering

diferido con el costo agregado de ancho de banda de memoria requerido para guardar toda esta información. [8]

Usaremos tres componentes de 16 bits para almacenar la normal en cada fragmento, tres componentes de 16 bits para el albedo del fragmento (color plano), tres componentes de punto flotante de 32 bits para las coordenadas en world-space del fragmento, un entero de 32 bits para el identificador de material por pixel, y una componente de 32 bits para almacenar el factor especular por pixel. [8]

Para los seis componentes de 16 bits, usamos los primeros tres canales del formato GL_RGBA32UI. El cuarto componente sirve para el identificador de materiales. Los cuatro componentes de 32 bits restantes pueden almacenarse en un tipo de dato GL_RGBA32F. [8]

Continuando con el código de `setupGBuffer`(), se generan las tres texturas bidimensionales necesarias para almacenar lo discutido previamente, todas del mismo tamaño de la ventana (1280x720), la textura con formato GL_RGBA32UI almacena los cuatro canales de colores RGBA de enteros sin signo de 32 bits cada uno, la del tipo RGBA32F almacena los cuatro colores pero en floats de 32 bits, y la tercera con formato GL_DEPTH_ATTACHMENT almacena una componente de profundidad en floats de 32 bits.

Luego de configurar cada textura, se adjuntan al framebuffer con la operación `glFramebufferTexture`() y finalmente se desbindea el mismo para volver a dejar activa la pantalla principal.

El último método a discutir en esta clase es `render`(), el mismo bindea al GBuffer para escribir sobre éste. La operación `glDrawBuffers`() recibe un arreglo de buffers adjuntos (texturas) al FBO sobre los cuales la información de salida del fragment shader será escrita. Si un fragment shader escribe un valor en una o más variables de salida definidas por un usuario, entonces el valor de cada variable será escrito sobre el buffer especificado en el arreglo correspondiente a la ubicación asignada a esa variable en el shader [52]

A continuación se inicializan los buffers del FBO (los de color en 0 y el de profundidad en 1). Se obtiene el identificador del shader que escribe al GBuffer y se notifica a todos los componentes *Renderer* que rendericen de manera diferida pasándoles este identificador por parámetro.

Se desbindea el GBuffer y se continúa con la segunda pasada de la técnica de rendering diferido, en la cual se bindean las texturas producidas en la primera pasada, para actuar de información de entrada al fragment shader, se le indica a OpenGL de usar el shader de la segunda pasada, se bindea el VAO del quad y se le pide a OpenGL que dibuje cuatro

vértices (las cuatro esquinas del quad). Finalmente se desbindean las texturas del GBuffer para que no sean afectadas externamente y termina la ejecución.

- MainSSAORenderer
  Por ser de similar implementación a *MainDeferredRenderer*, solo vale mencionar la implementación del método `generateRandomVectors`() quien en un bucle for se encarga de generar 256 vectores normalizados, que indican una dirección en el espacio, de 4 dimensiones y otro bucle crea 256 vectores desnormalizados. Finalmente, el método se encarga de enviar esta información a la placa de video cargándola en un buffer uniform llamado points_buffer.

- Input
  La clase *Input* posee una variable de clase de tipo diccionario que almacena para cada letra que se pueda presionar la constante correspondiente en GLFW como por ejemplo, para la letra B, la constante GLFW_KEY_B. También tiene un diccionario para los botones del mouse. Además del setter de la ventana gráfica (necesaria para calcular el centro de la pantalla y posicionar el mouse allí), tiene el método `getKey`() que devuelve verdadero si se presionó la tecla pasada por parámetro y lo mismo para el mouse con `getMouseButton`(). El método `getMouseDelta`() retorna un vector de dos dimensiones (vec2) que indica cuánto se movió el mouse a partir del centro de la pantalla en un tiempo dado llamado delta (un frame). Esto sirve para realizar cálculos de movimiento, desplazar la cámara, etc. El método `windowSize`() como su nombre lo indica retorna un dato vec2 con el ancho y largo de la ventana en donde se renderiza.

  Se puede setear la posición del mouse pasándola como parámetro al método `setMousePosition`(). O se puede optar por centrarlo con `centerMouse`(). El último método es `enableMouseCursor`(), recibe un boolean y sirve para habilitar o deshabilitar el cursor o puntero del mouse. Todos estos métodos se mapean casi directamente a funciones de GLFW. Por ejemplo, para deshabilitar el cursor se llama a `glfwSetInputMode`() pasando por parámetro la ventana, qué se quiere modificar (en este caso GLFW_CURSOR) y que se lo quiere deshabilitar con GLFW_CURSOR_DISABLED.

- FPSController
  La clase helper *FPSController* recibe en su constructor un componente de tipo *Camera* y otro de tipo *Transform*. No se los puede pedir a un *GameObject* ya que esta clase no es un componente en sí. Inicializa los vectores de rotación y velocidad en cero, el de damping rotacional en (0.6, 0.6, 0), el damping de velocidad en 0.85, la aceleración en 0.05 y la máxima velocidad alcanzable en 5.

  Posee un getter de la dirección hacia la que apunta, y métodos para moverse en las distintas direcciones (adelante y atrás, derecha e izquierda, y arriba y abajo), donde cada

uno modifica el vector de velocidad actual incrementándolo según la aceleración y la dirección que corresponda.

El método update() debe llamarse una vez por frame y se encarga de llamar a readInput() para leer la entrada de teclado y mouse, aplica el movimiento trasladando al Transform según el vector de velocidad, aplica las rotaciones necesarias nuevamente al Transform, y procede a aplicar damping tanto a la rotación como a la velocidad.

Los métodos pitch() y yaw() aplican rotaciones (en ángulos que llegan por parámetro) en los ejes X e Y respectivamente, teniendo en cuenta no pasarse de la máxima velocidad de rotación y de no pasarse de ciertos ángulos como 360° en el pitch.

El último método, readInput(), aplica rotaciones según el movimiento del mouse y mapea el tecleo de letras a movimientos, por ejemplo, que cuando se apriete la tecla A, se aplique un movimiento a la izquierda. Además hace que mientras permanezca presionada la tecla Shift izquierda, la aceleración a aplicar sea mayor.

4.2 Implementación de los shaders de Teardrop

En esta sección se describirá la implementación de los siguientes shaders que integran Teardrop:

- Shader básico
- Shader Phong
- Spherical Environment Mapping
- Deferred Rendering

A continuación se describe cada uno en detalle.

- Shader básico

  Este es un shader bien sencillo, en el que solo se transforma la posición de un espacio a otro y se pasan estos datos al Fragment Shader. Como se puede apreciar en el Código 1, la línea 1 indica al compilador con qué versión de GLSL trabajar. En las líneas 3 y 4 se declaran las variables de entrada (que llegan del código de C++ ejecutado en CPU), y se mapean entre sí por el número entero *location*. En la línea 7 se declara el único dato de salida custom; también exportamos la posición, pero esta se guarda en una variable especial que OpenGL nos provee, gl_Position. En la línea 9 declaramos el único uniform del shader (recordemos que son valores que no cambiar entre distintas ejecuciones del mismo). En la línea 11 comienza el método main(), allí arranca la ejecución de cualquier shader al igual que un programa escrito en C. Sencillamente multiplicamos el vértice de entrada por el uniforme MVP asignándolo a la variable de salida especial, y asignamos las coordenadas UV de entrada a su correspondiente variable de salida.

```
1. #version 420 core
2. // Datos por vértice de entrada, diferentes por cada ejecución.
3. layout(location = 0) in vec3 vertexPosition_modelspace;
4. layout(location = 1) in vec2 vertexUV;
5.
6. // Datos de salida, serán interpolados entre los fragmentos.
7. out vec2 UV;
8.
9. uniform mat4 MVP;
10.
11. void main() {
12.        // Posición de salida del vértice, en clip space (cubo desde -1
    a 1) : MVP * position
13.     gl_Position =  MVP * vec4(vertexPosition_modelspace,1);
14.
15.     // Coordenadas UV del vértice. No hay un espacio especial para
    éstas.
16.     UV = vertexUV;
17. }
```

Código 1 Vertex Shader básico.

En el Código 2, se especifica un fragment shader en el que se declara una variable de entrada y una de salida con el color final del fragmento. Este programa es muy sencillo, solo pinta todos los fragmentos con el mismo color encontrado en el uniform *input_color*.

```
1. #version 420 core
2.
3. // Valores interpolados desde el vertex shader.
4. in vec2 UV;
5.
6. // Datos de salida.
7. out vec3 color;
8.
9. // Valores que permanecen constantes en todas las ejecuciones.
10.      uniform vec3 input_color = vec3(1.0,0.5,1.0);
11.
12.      void main(){
13.          color = input_color;
14.      }
```

Código 2 Fragment Shader básico V1.

Esta nueva versión que se presenta en el Código 3, en vez de utilizar un mismo color para todos los fragmentos que el shader procese, el mismo se encarga de buscar el color correspondiente a cada fragmento en la textura que entra por el uniform *myTextureSampler* en base a las coordenadas *UV* (que en el Código 2 están quedaban sin uso). El método `rgb` extrae los tres colores del vec4 devuelto por el método `texture`() siendo el cuarto elemento la componente de transparencia *alpha*.

```
1. #version 420 core
2.
3. // Valores interpolados desde el vertex shader.
4. in vec2 UV;
5.
6. // Datos de salida.
7. out vec3 color;
8.
9. // Valores interpolados desde el vertex shader.
10.      uniform sampler2D myTextureSampler;
11.
12.      void main(){
13.          // Color de salida = Color de la textura en la
   coordenada especificada por UV.
```

```
14.            color = texture( myTextureSampler, UV ).rgb;
15.        }
```

Código 3 Fragment Shader básico V2.

- Shader Phong

  Como en todos los shaders, tenemos variables de entrada, variables uniformes y de salida. En el caso particular del Código 4, las variables de salida están agrupadas en un struct de tipo VS_OUT llamada o instanciada en *vs_out* (declarado y utilizado en el mismo lugar). Dentro se observa que exportamos un vector para la normal, la posición de la luz y de la vista o cámara. En el método `main` calculamos la posición en view-space en vez de en clip para poder utilizarla en los cálculos siguientes. Hacemos lo mismo con la normal. Luego, restando la posición del vector a la posición de la luz, computamos un vector con dirección hacia la posición de la luz. También computamos un vector que apunta hacia la cámara simplemente negando el vector de posición. Finalmente llevamos el vector de posición de view-space a clip-space y lo asignamos en *gl_Position* para exportarlo al fragment shader.

```
1. #version 420 core
2. // Entrada por vértice
3. layout (location = 0) in vec3 position;
4. layout (location = 2) in vec3 normal;
5.
6. // Matrices que usaremos
7. uniform mat4 mv_matrix;
8. uniform       mat4 proj_matrix;
9.
10.       // Posición de la luz
11.       uniform vec3 light_pos = vec3(0.0, 0.0, 0.0);
12.
13.       // Salida del vertex shader
14.       out VS_OUT
15.       {
16.               vec3 N;
17.               vec3 L;
18.               vec3 V;
19.       } vs_out;
20.
21.       void main(void)
22.       {
23.               // Calcula coordenadas en espacio view
24.               vec4 P = mv_matrix * vec4(position,1);
25.               // Calcula la normal en view-space
26.               vs_out.N = mat3(mv_matrix) * normal;
```

```
27.             // Calcula el vector de la luz
28.             vs_out.L = light_pos - P.xyz;
29.             // Calcula el vector de la vista
30.             vs_out.V = -P.xyz;
31.             // Calcula la posición en clip-space de cada
   vértice
32.             gl_Position = proj_matrix * P;
33.     }
```

Código 4 Vertex Shader Phong V1.

Como se puede observar en el Código 5, luego de declarar el output y el input del mismo tipo de struct VS_OUT que sale del vertex shader, declaramos uniforms para configurar los colores con los que se iluminará cada fragmento. Entre ellos tenemos los tres componentes Phong, el ambiental, difuso y especular, y luego valor de tipo float llamado specular_power, el cual determina la intensidad de la componente especular. En el método main se normalizan las entradas (por si la interpolación entre fragmentos que corresponden al mismo vértice las sacó de rango), y se calcula el color final utilizando la fórmula de iluminación Phong.

```
1. #version 420 core
2. // Salida
3. layout (location = 0) out vec4 color;
4.
5. // Entrada desde el vertex shader
6. in VS_OUT
7. {
8.      vec3 N;
9.      vec3 L;
10.     vec3 V;
11.     } fs_in;
12.
13.     // Propiedades del material
14.     uniform vec3 diffuse_albedo = vec3(1.0, 0.078, 0.576);
15.     uniform vec3 specular_albedo = vec3(1.0);
16.     uniform float specular_power = 30.0;
17.     uniform vec3 ambient = vec3(0.105, 0.105, 0.105);
18.
19.     void main(void)
20.     {
21.             // Normalizar los vectores N, L y V entrantes
22.             vec3 N = normalize(fs_in.N);
23.             vec3 L = normalize(fs_in.L);
24.             vec3 V = normalize(fs_in.V);
```

```
25.
26.             // Calcular R localmente
27.             vec3 R = reflect(-L, N);
28.
29.             // Computar las componentes difusa y especular
   para cada fragmento
30.             vec3 diffuse = max(dot(N, L), 0.0) *
   diffuse_albedo;
31.             vec3 specular = pow(max(dot(R, V), 0.0),
   specular_power) * specular_albedo;
32.
33.             // Escribir color final al framebuffer
34.             color = vec4(ambient + diffuse + specular, 1.0);
35.     }
```

Código 5 Fragment Shader Phong V1.

Como se mencionó anteriormente, el specular_power determina la intensidad de la componente especular, mientras mayor sea este factor, más chico y brillante será este efecto, como observamos en la Figura 33.

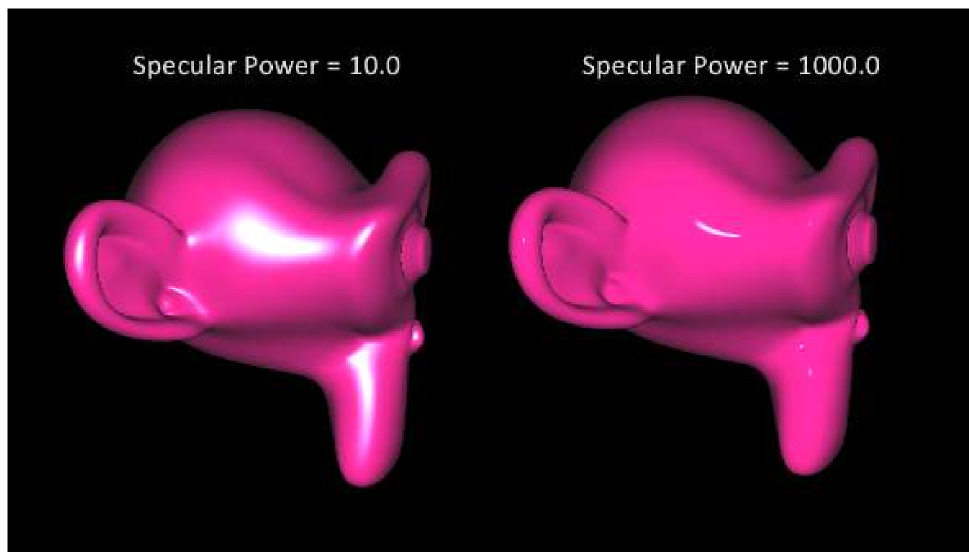

Figura 33 Distintos valores del factor especular.

Veamos ahora otra versión de Vertex Shader, esta versión solo cambia en que el color difuso del fragment shader se obtiene de una textura en vez de ser un color fijo. Para esto necesitamos las pasar las coordenadas UV interpoladas que es lo que hace este shader, simplemente las recibe y las asigna al struct de salida. Para samplear la textura recibimos las coordenadas UV en el struct de input, y la textura como uniform, luego de la línea 27 del Código 5 insertamos el fragmento del Código

6 en donde luego de samplear, utiliza los valores obtenidos en la formula Phong y finaliza juntando las componentes en la variable de output *color*.

```
1. // Obtener el color difuso de la textura en base a las
2. // coordenadas UV de fs_in
3. vec3 diffuse_texture = texture( textureInput, fs_in.UV ).rgb;
4.
5. // Computar las componentes difusa y especular para cada
   fragmento
6. vec3 diffuse = max(dot(N, L), 0.0) * diffuse_texture;
7. vec3 specular = pow(max(dot(R, V), 0.0), specular_power) *
   specular_albedo;
8.
9. // Escribir color final al framebuffer
10.     color = vec4(ambient + diffuse + specular, 1.0);
```

Código 6 Fragmento del Fragment Shader Phong V2 (texturado).

- Spherical Environment Mapping

  Este conjunto de shaders implementa una técnica que no ha sido descrita en detalle en el Capítulo 2, su implementación no es tan importante pero se incluye porque produce efectos visuales agradables con poco poder de cómputo asociado. Lo que hace es samplear texturas como ya vimos en ejemplos previos, pero en vez de utilizar las coordenadas UV, genera unas nuevas en el vertex o fragment shader (depende de la versión) en base a las normales del modelo. Luego, el tipo de texturas que utiliza se denominan “esferas iluminadas” o matcaps, como se puede observar en la Figura 34, y contienen la información completa de iluminación, no solo una componente de color (generalmente difusa).

Figura 34 Distintos ejemplos de matcaps [55]

Aplicando estas texturas sobre un modelo con sus nuevas coordenadas interpoladas obtenemos un efecto como el observado en la Figura 35, un material parecido al plástico en este ejemplo, y así se suelen tener distintos tipos de materiales, de distintos colores, y bajo distintas condiciones de iluminación. Se facilita el trabajo de programación (shaders más sencillos), pero se incrementa el trabajo de los diseñadores (muchos tipos de materiales a generar).

Figura 35 Renderizando con Spherical Environment Mapping y distintos Matcaps.

En el Código 7 se muestra el vertex shader que calcula las nuevas coordenadas UV llamadas *vN*, el fragment shader simplemente samplea la textura con éstas coordenadas.

```
1. #version 420 core
2. // Entrada por vértice
3. layout (location = 0) in vec3 position;
4. layout (location = 2) in vec3 normal;
5.
6. // Matrices que usaremos
7. uniform mat4 MVP;
8. uniform mat4 mv_matrix;
9. uniform       mat4 proj_matrix;
10.
11.       // Salida del vertex shader
12.       out vec2 vN;
13.
14.       void main()
15.       {
```

```
16.
17.              // Posición y normal normalizadas y en espacio
   view.
18.              vec3 e = normalize( vec3( mv_matrix * vec4(
   position, 1.0 ) ) );
19.              vec3 n = normalize( vec3(mv_matrix * vec4( normal,
   0.0) ) );
20.
21.              // Fórmula SEM.
22.              vec3 r = reflect( e, n );
23.              float m = 2. * sqrt( pow( r.x, 2. ) + pow( r.y, 2.
   ) + pow( r.z + 1., 2. ) );
24.              vN = r.xy / m + .5;
25.
26.              gl_Position = MVP * vec4( position, 1);
27.       }
```

Código 7 Vertex Shader de Spherical Environment Mapping

- Deferred Rendering

  El primer vertex shader de este programa de dos pasadas simplemente computa *gl_Position* y asigna un conjunto de valores directo de la entrada al struct de salida. Los valores, como se observa en el Código 8 son la posición, la normal y las coordenadas UV (texcoord0). La variable material_id actualmente no se utiliza, pero sirve para diferenciar distintos tipos de materiales (al ser este un shader global, y no haber uno por cada objeto de la escena) y según el mismo, renderizar el objeto en cuestión de manera diferente.

```
1. // Valores de entrada.
2. layout (location = 0) in vec4 position;
3. layout (location = 2) in vec3 normal;
4. layout (location = 1) in vec2 texcoord0;
5.
6. // Struct de salida.
7. out VS_OUT
8. {
9.      vec3    ws_coords;
10.          vec3    normal;
11.          vec3    tangent;
12.          vec2    texcoord0;
13.          flat uint    material_id;
14.       } vs_out;
```

Código 8 Variables del Vertex Shader de Deferred Rendering – First Pass

El shader presentado en el Código 9 se encarga únicamente de comprimir o codificar diferentes tipos de datos empacándolos en dos variables vector de 4 dimensiones de salida, *color0* (de tipo entero sin signo) y *color1* (de tipo float). Lo que se hace es llenar el GBuffer conformado por dichas variables de output, y el concepto detrás de esto es que el GBuffer debe ser lo más pequeño posible para consumir menos ancho de banda, este puede actuar como cuello de botella del proceso de rendering si no se tiene cuidado. Así es como en cada componente de las variables de salida almacenamos distintas componentes de los valores de entrada, no importa el orden siempre y cuando se descompriman de la misma manera para mantener la integridad de los datos. Lo que se observa, por ejemplo en la línea 24 es que en una componente de outvec0 se almacenan dos de color, x e y. Esto es posible gracias a la función `packHalf2x16()` que convierte dos números de punto flotante de 32 bits en números de 16 bits y luego los empaca en un solo entero de 32 bits [56].

```
1. #version 420 core
2.
3. layout (location = 0) out uvec4 color0;
4. layout (location = 1) out vec4 color1;
5.
6. in VS_OUT
7. {
8.       vec3    ws_coords;
9.       vec3    normal;
10.          vec3    tangent;
11.          vec2    texcoord0;
12.          flat uint    material_id;
13.      } fs_in;
14.
15.      layout (binding = 0) uniform sampler2D tex_diffuse;
16.
17.      void main(void)
18.      {
19.          uvec4 outvec0 = uvec4(0);
20.          vec4 outvec1 = vec4(0);
21.
22.          vec3 color = texture(tex_diffuse,
   fs_in.texcoord0).rgb;
23.
24.          outvec0.x = packHalf2x16(color.xy);
25.          outvec0.y = packHalf2x16(vec2(color.z,
   fs_in.normal.x));
26.          outvec0.z = packHalf2x16(fs_in.normal.yz);
27.          outvec0.w = fs_in.material_id;
```

```
28.
29.             outvec1.xyz = fs_in.ws_coords;
30.             outvec1.w = 30.0;
31.
32.             color0 = outvec0;
33.             color1 = outvec1;
34.         }
```

Código 9 Fragment Shader de Deferred Rendering – First Pass

El shader que se observa en el Código 10 es muy sencillo, solo define el quad de pantalla completa del que tanto hablamos en la Sección 2.4.1 Rendering diferido sobre el cual renderizaremos la escena final. Dependiendo del valor del índice `gl_VertexID` asignamos una de las cuatro esquinas a `gl_Position`.

```
1. #version 420 core
2.
3. void main(void)
4. {
5.     const vec4 verts[4] = vec4[4](vec4(-1.0, -1.0, 0.5, 1.0),
6.                                   vec4( 1.0, -1.0, 0.5, 1.0),
7.                                   vec4(-1.0,  1.0, 0.5, 1.0),
8.                                   vec4( 1.0,  1.0, 0.5, 1.0));
9.
10.             gl_Position = verts[gl_VertexID];
11.         }
```

Código 10 Vertex Shader de Deferred Rendering – Second Pass

En el shader presentado en el Código 11 es donde ocurre la magia del rendering diferido, este es el encargado de consumir el GBuffer y renderizar la escena en base al contenido del mismo. Debido a la extensión del programa iremos explicando su código en partes. En la línea 3 del Código 11 declaramos la variable de salida para el color, luego las entradas del GBuffer *gbuf_tex0* y *gbuf_tex1*, de tipo *usampler2D* (sampler de enteros sin signo) y *sampler2D* (sampler de floats) respectivamente por tratarse el GBuffer de texturas que deben ser sampleadas. Luego en la línea 8 declaramos un struct de lo que conformaría una fuente de luz, compuestas por una posición y un color, el pad es para rellenar espacio, no es un detalle importante. La variable *num_lights* indica la cantidad de luces que utilizaremos al iluminar, *vis_mode* qué modo de renderizado y light es un arreglo de 64 structs *light_t*, por lo que tenemos espacio para iluminar la escena hasta con 64 fuentes de luz. El struct `fragment_info_t` representa un solo pixel o fragmento de la escena a iluminar, y está compuesto de todo el material necesario para ello, es decir, el color, la normal, el poder especular, las coordenadas de la textura y un identificador de material.

```
1. #version 420
2.
3. layout (location = 0) out vec4 color_out;
4.
5. layout (binding = 0) uniform usampler2D gbuf_tex0;
6. layout (binding = 1) uniform sampler2D gbuf_tex1;
7.
8. struct light_t
9. {
10.           vec3    position;
11.           uint    pad0;
12.           vec3    color;
13.           uint    pad1;
14.       };
15.
16.       uniform int num_lights = 4;
17.       uniform int vis_mode = 5;
18.
19.       light_t     light[64];
20.
21.       struct fragment_info_t
22.       {
23.           vec3 color;
24.           vec3 normal;
25.           float specular_power;
26.           vec3 ws_coord;
27.           uint material_id;
28.       };
```

Código 11 Input y Output Fragment Shader de Deferred Rendering – Second Pass

La función unpackGBuffer de la línea 30 del Código 12 (continuación del Código 11) recibe coordenadas sobre qué pixel de la imagen final renderizar (uno de los 1920x1080 en una pantalla Full HD) y un struct vacío de tipo `fragment_info_t` donde desempacar la información desde el GBuffer. Obtiene la información desde las dos texturas del buffer con la función `texelFetch()`, y luego la convierte nuevamente a su tipo original con `unpackHalf2x16()`.

```
30.       void unpackGBuffer(ivec2 coord,
31.                          out fragment_info_t fragment)
32.       {
33.           uvec4 data0 = texelFetch(gbuf_tex0, ivec2(coord), 0);
34.           vec4 data1 = texelFetch(gbuf_tex1, ivec2(coord), 0);
35.           vec2 temp;
```

```
36.
37.             temp = unpackHalf2x16(data0.y);
38.             fragment.color = vec3(unpackHalf2x16(data0.x),
   temp.x);
39.             fragment.normal = normalize(vec3(temp.y,
   unpackHalf2x16(data0.z)));
40.             fragment.material_id = data0.w;
41.
42.             fragment.ws_coord = data1.xyz;
43.             fragment.specular_power = data1.w;
44.         }
```

Código 12 Función unpackGBuffer(), Fragment Shader de Deferred Rendering – Second Pass

La función `vis_fragment()` observada en el Código 13 recibe un struct `fragment_info_t` con la información necesaria para renderizar un fragmento, la procesa y devuelve el color final para ese fragmento.

El quinto modo de renderizado es el efecto final de renderizar la escena con varias fuentes de iluminación, como se observan declaradas cuatro luces desde la línea 70 a la 80 del Código 13. Luego para cada fuente de luz (con un bucle for) se va iluminando el pixel con iluminación Phong para el cual se calcula una mezcla entre los colores difuso, tanto de la textura del objeto como de la fuente de luz, y también así de la componente especular. Notar que en la línea 94 se encuentra comentada la multiplicación del color especular por el color de la luz, esto hace que el brillo observado sea blanco en vez de colorido, se puede optar por ambas opciones. Finalmente esta función retorna el color final del pixel.

```
46.         vec4 vis_fragment(fragment_info_t fragment)
47.         {
48.             int i;
49.             vec4 result = vec4(0.0, 0.0, 0.0, 1.0);
50.
51.             switch (vis_mode)
52.             {
53.                 case 1:
54.                 default:
55.                     result = vec4(fragment.normal * 0.5 +
   vec3(0.5), 1.0);
56.                     break;
57.                 case 2:
58.                     result = vec4(fragment.ws_coord * 0.02 +
   vec3(0.5, 0.5, 0.0), 1.0);
59.                     break;
60.                 case 3:
```

```
61.                     result = vec4(fragment.color, 1.0);
62.                     break;
63.                 case 4:
64.                     result = vec4(fragment.specular_power,
65.                                   float(fragment.material_id & 15)
   / 15.0,
66.                                   float(fragment.material_id / 16)
   / 15.0,
67.                                   1.0);
68.                     break;
69.                 case 5:
70.                     light[0].position = vec3(0,30,60);
71.                     light[0].color = vec3(1,1,1);
72.
73.                     light[1].position = vec3(0,30,120);
74.                     light[1].color = vec3(0,0,1);
75.
76.                     light[2].position = vec3(60,30,60);
77.                     light[2].color = vec3(1,0,0);
78.
79.                     light[3].position = vec3(0,30,0);
80.                     light[3].color = vec3(0,1,0);
81.
82.                     for (i = 0; i < num_lights; i++)
83.                     {
84.                         vec3 L = light[i].position -
   fragment.ws_coord;
85.                         float dist = length(L);
86.                         L = normalize(L);
87.                         vec3 N = normalize(fragment.normal);
88.                         vec3 R = reflect(-L, N);
89.                         float NdotR = max(0.0, dot(N, R));
90.                         float NdotL = max(0.0, dot(N, L));
91.                         float attenuation = 500.0 / (pow(dist,
   2.0) + 1.0);
92.
93.                         vec3 diffuse_color  = 1.0 * light[i].color
   * fragment.color * NdotL * attenuation;
94.                         vec3 specular_color = vec3(1.0) /* *
   light[i].color */ * pow(NdotR, fragment.specular_power) *
   attenuation;
95.
96.                         result += vec4(diffuse_color +
   specular_color, 0.0);
```

```
97.                     }
98.
99.                     break;
100.            }
101.
102.            return result;
103.    }
```

Código 13 Función vis_fragment(), Fragment Shader de Deferred Rendering – Second Pass

En base al modo de renderizado elegido previamente con la variable `vis_mode` se entra a la rama adecuada del bloque switch para renderizar el fragmento y poner el valor final en la variable `result`. Los primeros cuatro modos son de debuggeo, como el vis_mode número uno que simplemente dibuja una representación de las normales de la escena (la corrección de multiplicarlo por 0.5 y sumarle 0.5 es para cambiarlo de rango, desde [-1,1] a [0,1]). El segundo modo de renderizado muestra una representación visual de las coordenadas con las cuales acceder a las texturas del buffer. El tercer modo muestra el color difuso del fragmento y el cuarto elige un color por identificador de material para visualizar qué material posee cada objeto de la escena. En la Figura 36 se observa el efecto producido por cada uno, como por el momento todos los objetos poseen el mismo material_id toda la escena se ve roja en el *vis_mode* 4.

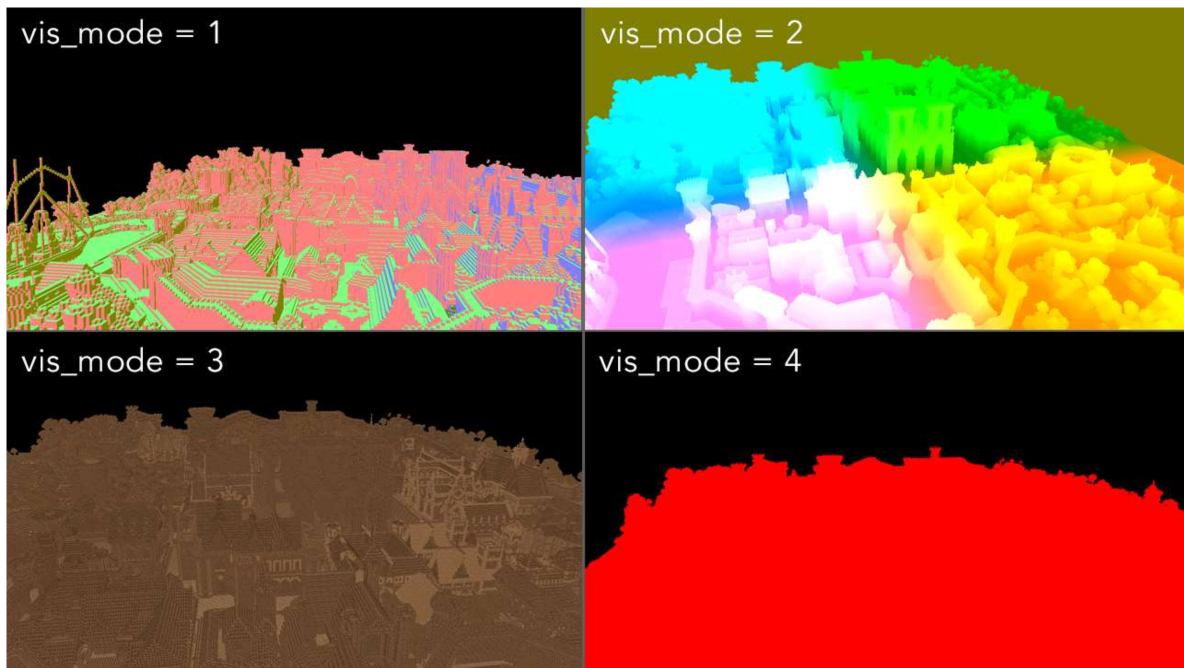

Figura 36 Distintos modos de renderizado.

La función `main`() vista en el Código 14 simplemente declara un fragmento a colorear, desempaca la información del GBuffer sobre él y luego asigna el resultado de colorearlo en la variable de salida *color_out*.

```
1. void main(void)
2. {
3.     fragment_info_t fragment;
4.
5.     unpackGBuffer(ivec2(gl_FragCoord.xy), fragment);
6.
7.     color_out = vis_fragment(fragment);
8. }
```

Código 14 Función main(),Fragment Shader de Deferred Rendering – Second Pass

4.3 Problemas encontrados durante el desarrollo

En esta sección se describirán los problemas identificados cuando se desarrolló Teardrop y cómo fueron solucionados.

El mayor problema se tuvo con las dependencias circulares como se describe a continuación.

El problema se presentó durante la implementación de ciertas clases que directa o indirectamente dependen entre sí, módulos conocidos como mutuamente recursivos. Es decir, una situación como la observada en la Figura 37 en donde la clase *A* referencia a la clase *B* y a su vez, *B* intenta referenciar a la clase *A*. En este tipo de casos el compilador de C++ (etapa del linker) comienza a loopear incluyendo el header de *A*, que incluye a *B*, que incluye a *A*, y así sucesivamente, produciendo un error de compilación (muy poco descriptivo, las primeras veces siendo difícil deducir de qué problema se trataba).

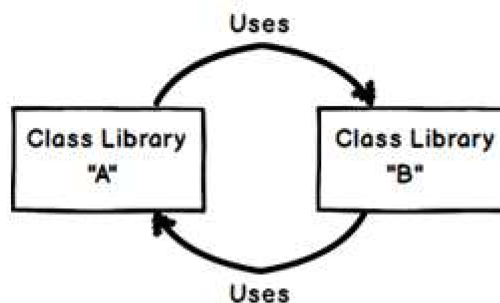

Figura 37 Demostración gráfica de una dependencia circular [60]

Se suele decir que este error aparece por un mal diseño, y se lo considera un anti-patrón [53], pero en Teardrop este tipo de conocimiento mutuo se considera necesario, por lo que en vez de modificar el diseño se buscó cómo solucionar el problema en la implementación.

Particularmente, las ocurrencias de este problema se dieron entre la clase *GameObject* y los componentes (recordemos que cada componente conoce al *GameObject* que lo contiene), como por ejemplo *Camera*. La clase *GameObject* necesita conocer a *Camera* para

mantener referencia a dicho componente y declarar variables de este tipo, y para eso, en su header debe incluir al header de la clase *Camera*(#include <Camera.h>). Ahora, *Camera* debe conocer a *GameObject* para referenciar al GO que lo contiene y por eso "Camera.h" incluye a "GameObject.h", provocando error en compilación.

Para solucionar el problema, en vez de que *Camera* incluya a *GameObject* en su header, se mueve la inclusión al archivo de implementación "Camera.cpp". Pero así toda variable o parámetro de tipo *GameObject* declarada en Camera.h produciría un error de tipo no encontrado. Lo que falta es realizar una *forward declaration* [54] del tipo *GameObject*, ésta consiste en la declaración temprana de un identificador al que todavía no se le ha dado una definición completa. Con declarar una clase vacía (class *GameObject*;) previo a la declaración de variables de este tipo el compilador no produce errores, solucionando finalmente el problema de dependencias circulares.

4.4 Instalación

La instalación de Teardrop es sencilla, solo se deben seguir la siguiente lista de pasos para tener una copia local del repositorio de código y ejecutarlo para seguir desarrollando el motor o alguna aplicación que lo utilice.

1. Descargar e instalar Microsoft Visual Studio 2013 Express for Desktop desde la web de Microsoft, por ejemplo desde la dirección https://www.microsoft.com/en-us/download/details.aspx?id=44914.
2. Descargar el repositorio de código de Teardrop en GitHub desde la dirección https://github.com/belazaras/teardrop. Guardarlo en cualquier directorio apretando en "Download ZIP" como se muestra en la Figura 38.

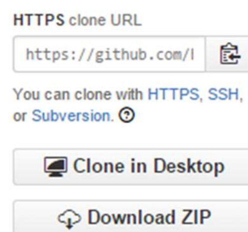

Figura 38 Descargando Teardrop de GitHub.

3. Descargar los modelos desde el link https://drive.google.com/file/d/0B5_qj1VB-rhUMlM2MGM1TFBhOE0/view haciendo click en el botón de descarga visto en rojo en la Figura 39. Éstos están separados por ser muy pesados como para subirlos al repositorio. Descomprimir la carpeta "models" dentro del repositorio, en la carpeta /Debug/media/ de modo que quede RAIZ_REPOSITORIO/Debug/media/models.

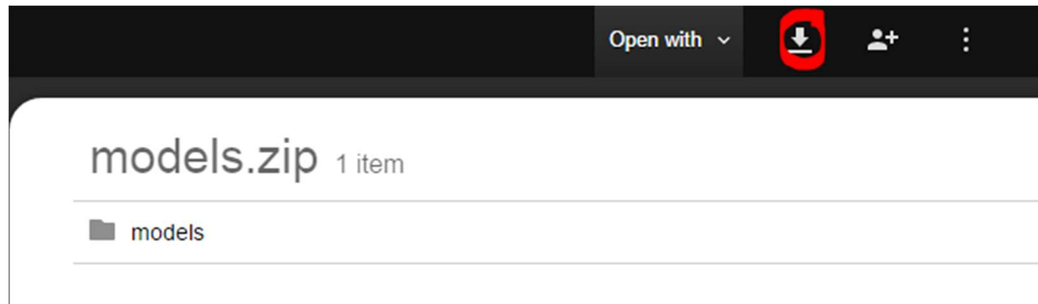

Figura 39 Descargando modelos de Google Drive

4. Abrir el proyecto ejecutando Teardrop.sln con Visual Studio, hacer caso omiso de la advertencia sobre que la fuente del proyecto no es confiable haciendo clic en OK. Hacer click en el botón Play o “Local Windows Debugger” que se muestra en la Figura 40. Teardrop compilará y luego se ejecutará. Tener en cuenta que consume varios recursos por lo que requiere el uso de una computadora de medio a alto rango (que soporte OpenGL 4.2).

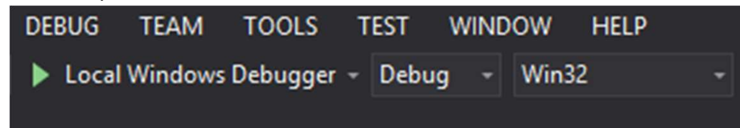

Figura 40 Ejecutar Teardrop desde Visual Studio

4.5 Demostraciones de uso general

En esta sección se presentará de manera general cómo usar Teardrop, detallando distintos ejemplos a nivel de código para mostrar su uso.

- Creando una escena

  Para empezar con nuestra simulación primero debemos crear una clase que extienda a *Engine*, llamémosla *ourScene*. En el archivo ourScene.h declaramos la clase con los métodos a redefinir como se ve en el Código 15. Al final del archivo llamamos a la función DECLARE_MAIN() con nuestra nueva clase como parámetro para que ésta sea el punto de entrada principal de ejecución.

```
#pragma once
#include <Engine.h>

class ourScene : public Engine
{
	public:
		void setup();
		void update();
};

DECLARE_MAIN(ourScene)
```

Código 15 Archivo ourScene.h

Luego creamos el archivo de implementación ourScene.cpp en donde implementaremos el código de los métodos declarados en el header como se puede apreciar en el Código 16.

```
#include "ourScene.h"

void ourScene::setup()
{

}

void ourScene::update()
{

}
```

Código 16 Archivo ourScene.cpp

Si ejecutamos lo creado hasta ahora, veremos una pantalla totalmente en negro como se puede apreciar en la Figura 41. Esto significa que todo salió bien, ahora debemos empezar a agregar objetos a la escena vacía.

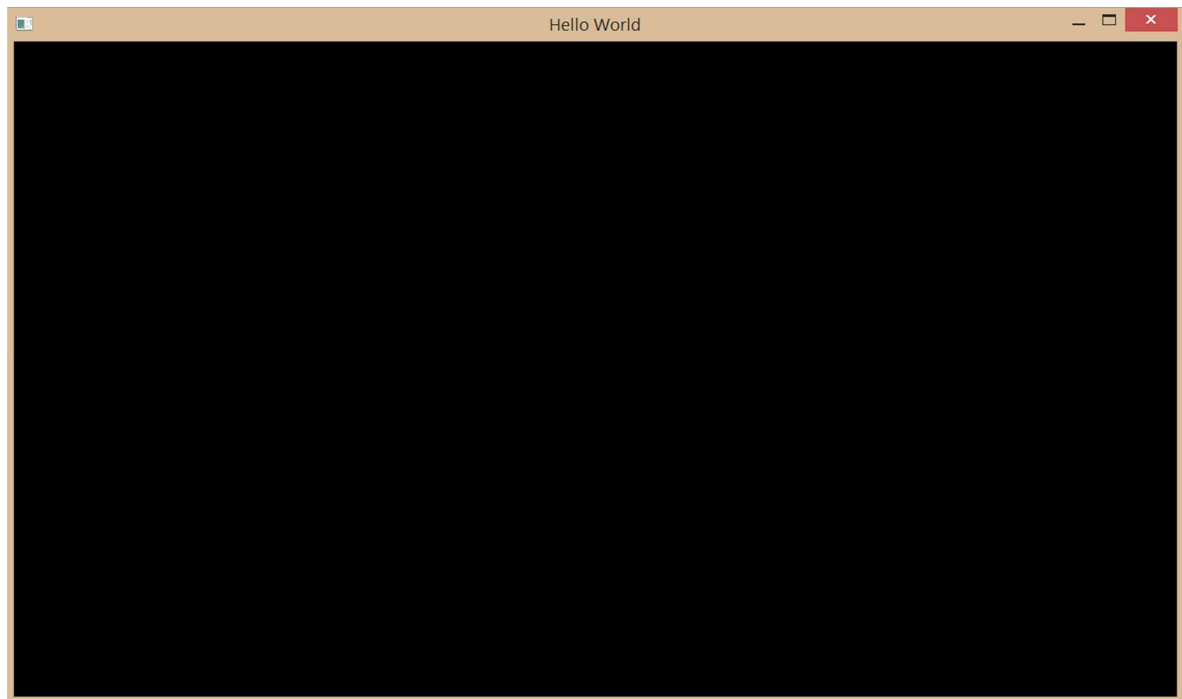

Figura 41 Ejecución de una simulación vacía

Nuestro primer GameObject

Para agregar un *GameObject* primero debemos incluir su header. Logramos esto agregando la línea #include <GameObject.h> en el header de nuestra escena (ourScene.h),

luego de la inclusión del header de *Engine*. Luego declaramos una variable de tipo *GameObject* en el mismo archivo, como se ve en el Código 17.

```
private:
    GameObject monkey;
```

Código 17 Variable de tipo GameObject en ourScene.h

- Agregando componentes

A continuación se mostrará cómo se pueden agregar componentes al *GameObject.*

- *Cargando una Mesh*

Hasta ahora el GO monkey solo tiene un componente *Transform*, que como vimos previamente, es agregado automáticamente. Queremos cargarle un modelo 3D por lo que en el método setup() de nuestra escena (ourScene.cpp) agregamos un componente de tipo *Mesh* al GO y le indicamos cargar el modelo 3D en formato OBJ llamado `suzanne.obj`. Esto se puede apreciar en el Código 18.

```
Mesh *monkeyMesh = monkey.addComponent<Mesh>();
monkeyMesh->loadOBJ("models/suzanne.obj");
```

Código 18 Añadiendo una Mesh y cargando un modelo 3D en ourScene.cpp

Cuando añadimos el componente, este método devuelve una referencia al mismo, que guardamos en la variable monkeyMesh para más tarde indicarle que cargue el modelo.

- *Añadiendo un Renderer*

En el archivo de implementación de nuestra escena ya cargamos el modelo al GO, pero para renderizarlo y que se vea en pantalla necesitamos agregar al mismo GO el componente *Renderer*. Este necesita un *Material* que le indica al *Renderer* cómo renderizar la *Mesh*.

Continuamos en el Código 19 declarando la variable monkeyMaterial en donde cargamos el vertex y pixel shader, agregamos el *Renderer* y le asignamos el material.

```
Material *monkeyMaterial = new Material("shaders/phong.vs", "shaders/phong.fs");

Renderer *monkeyRenderer = monkey.addComponent<Renderer>();
monkeyRenderer->setMaterial(monkeyMaterial);
```

Código 19 Agregando el Renderer a ourScene.cpp

- *Agregando una cámara*

Por el momento, nuestro modelo está listo para ser renderizado, pero necesitamos una cámara desde donde observarlo. Por eso debemos agregar un nuevo GO llamado camera en ourScene.h. A este mismo, en el archivo de implementación de la escena le añadiremos un componente de tipo *Camera* con la línea camera.addComponent<Camera>();. Esto seteará automáticamente las matrices

necesarias, una vista en modo perspectiva y con el aspect ratio y clipping planes por defecto.

Si ejecutamos la escena ahora, seguiremos viendo la pantalla en negro. Esto se debe a que tanto la cámara como nuestro objeto están en la misma posición. Procedemos a obtener el componente *Transform* de nuestro objeto monkey para pedirle que traslade al mismo 5 unidades sobre el eje Z, y lo hacemos como indica el Código 20.

```
monkey.getComponent<Transform>()->translate(0, 0, -5);
```

Código 20 Trasladando un Transform en ourScene.cpp

Como resultado producido al mover el objeto lejos de la cámara obtenemos lo apreciado en la Figura 42 (mostrando la ejecución).

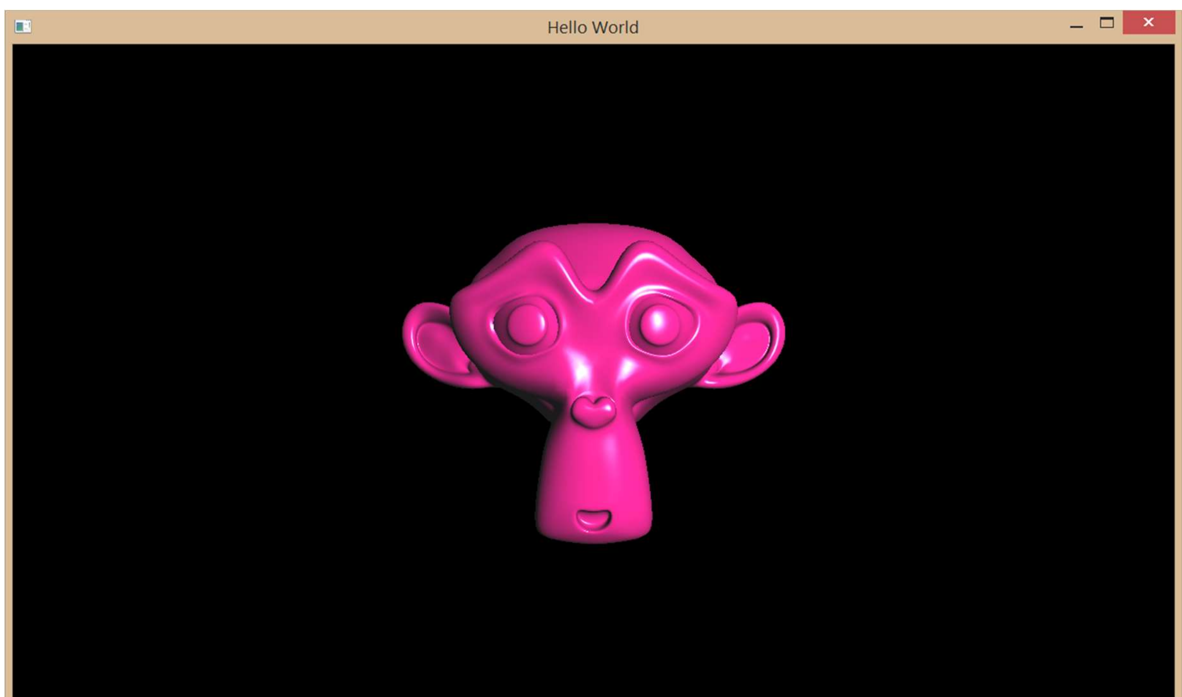

Figura 42 Objeto renderizado en pantalla

- Agregando movimiento

Por ahora la escena es muy estática, pero cambiar eso es tan fácil como rotar el *Transform* del mono cierta cantidad de ángulos por cada frame.

Para ello, en el método `update()` de la escena escribimos lo observado en el Código 21.

```
double delta = Engine::deltaTime();
monkey.getComponent<Transform>()->rotate(0, 50 * delta, 0);
```

Código 21 Rotando el Transform

Hacemos rotar al *Transform* sobre el eje Y, y multiplicamos los 50 grados por el tiempo delta obtenido previamente para hacer la rotación independiente del frame rate.

Como resultado, el mono comienza a rotar sobre sí mismo como se aprecia en la Figura 43.

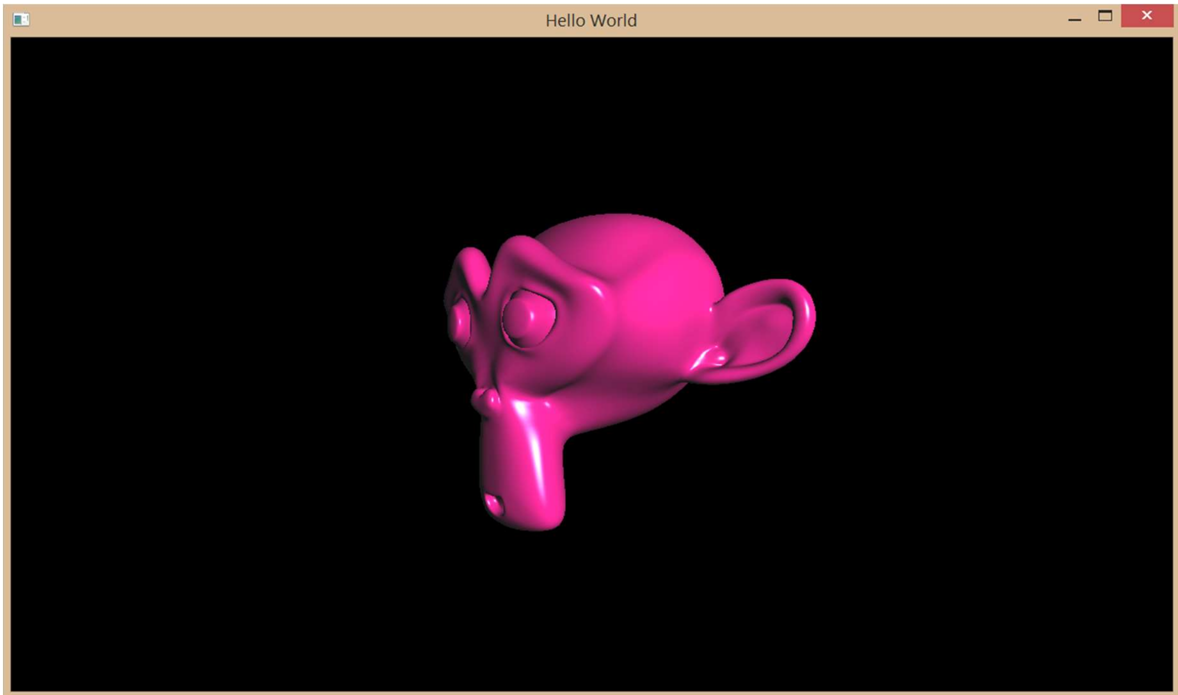

Figura 43 Rotando el Transform

El código completo de la escena, tanto la interface como la implementación, se observan en el Código 22 y en el Código 23 respectivamente.

```
#pragma once
#include <Engine.h>
#include <GameObject.h>

class ourScene : public Engine
{
private:
	GameObject monkey;
	GameObject camera;
public:
	void setup();
	void update();
};

DECLARE_MAIN(ourScene)
```

Código 22 Código completo de ourScene.h

```
#include "ourScene.h"

void ourScene::setup()
{
	Mesh *monkeyMesh = monkey.addComponent<Mesh>();
	monkeyMesh->loadOBJ("media/models/suzanne/suzanne.obj");
```

```
	Material *monkeyMaterial = new Material("media/shaders/phong/phong.vs",
"media/shaders/phong/phong.fs");
	Renderer *monkeyRenderer = monkey.addComponent<Renderer>();
	monkeyRenderer->setMaterial(monkeyMaterial);

	camera.addComponent<Camera>();

	monkey.getComponent<Transform>()->translate(0, 0, -5);

}

void ourScene::update()
{
	double delta = Engine::deltaTime();
	monkey.getComponent<Transform>()->rotate(0, 50 * delta, 0);
}
```

Código 23 Código completo de ourScene.cpp

Capítulo 5: Ejemplo de Uso de Teardrop

A lo largo de este capítulo se presentará una simulación 3D de ejemplo que usa gran mayoría de las técnicas visuales desarrolladas en el motor gráfico, desde la carga de grandes modelos en formato binario, renderizado de los mismos con el modelo de iluminación Phong, la posibilidad de moverse por el mundo virtual utilizando el *FPSController* y las técnicas de rendering diferido y screen space ambient occlusion en conjunto.

El código completo se puede ver en el Código 24 con comentarios explicando cada sentencia, a lo largo del capítulo se explicarán algunas en más detalle mostrando el resultado producido por las mismas.

Se mostrará que la misma escena puede ser renderizada con los tres tipos diferentes de *MainRenderer*, el homónimo que utiliza forward rendering, el *MainDeferredRenderer*, que usa rendering diferido y *MainSSAORenderer* que usa rendering diferido con SSAO.

- ## Generando el modelo

  Para generar el modelo binario a partir de uno en formato .obj debemos utilizar una herramienta, también creada por Roald Fernandez [61], que parsea este formato de texto buscando cada vértice, normal, coordenada de textura, y hasta valores de color ambiental, difuso y especular (gracias a una modificación propia) y luego escribe bloques de memoria tal cual están en la memoria RAM, lo que incrementa mucho la velocidad de carga.

  En nuestro ejemplo el modelo es una escena extraída del famoso juego Minecraft donde todo está formado por cubos, y consiste en una ciudad silvestre rodeada de agua y algunos barcos, esta posee más de 3.28 millones de vértices (5.8 millones de triángulos) y su versión .obj pesa 262.9MB y tarda 1187036ms en cargar (19.78 minutos!). Su contraparte binaria .cobj pesa bastante más por estar en cierta forma descomprimida, unos 485MB, pero a cambio ganamos un increíble speedup de hasta 247X, con una carga promedio de 4806ms (4.8 segundos), y al no ser hoy en día el espacio en disco un problema, esto es una mejora importante.

- ## Código de la simulación

  El Código 24 no presenta muchas sentencias desconocidas por lo que es auto explicativo y además posee comentarios para guiar al lector por lo que no se consideró necesario explicarlo línea por línea. Ciertas sentencias involucradas en los métodos de renderizado se verán luego.

```
1. #include "rungholtApp.h"
2.
3. #include <iostream>
4.
5. #include <Uniform.h>
```

```
6.  #include <UniformDerived.h>
7.
8.  #include <glm/gtc/type_ptr.hpp>
9.
10.
11. using namespace std;
12.
13. void rungholtApp::setup()
14. {
15.         printf("Starting.\n");
16.
17.         // Setea el color de fondo de la escena en un celeste claro.
18.         glClearColor(68 / 255.0, 169 / 255.0, 255 / 255.0, 1);
19.
20.         // Indicamos a Teardrop que queremos usar rendering diferido
21.         // con Screen Space Ambient Occlusion.
22.         bool ssao = true;
23.         MainRenderer::setDeferredRendering(ssao);
24.
25.         // Deshabilitamos el puntero del mouse.
26.         Input::enableMouseCursor(false);
27.
28.
29.         // Cargamos un material de "tierra", con iluminación phong
30.         // texturada y una textura .dds con una imagen de tierra.
31.         Material dirt =
    Material("media/shaders/phong/phong_textured.vs",
    "media/shaders/phong/phong_textured.fs");
32.         dirt.setTexture("media/images/dirt.dds");
33.
34.         // Inicializamos el GO de nuestra ciudad Rungholt,
35.         // le agregamos un componente Renderer, un Mesh
36.         // y finalmente cargamos el modelo al Mesh.
37.         rungholt_world = new GameObject();
38.         Renderer *myRender = rungholt_world->addComponent<Renderer>();
39.         myRender->setMaterial(dirt);
40.         Mesh *myMesh = rungholt_world->addComponent<Mesh>();
41.         myMesh->loadCOBJ("media/models/rungholt/rungholt.cobj");
42.
43.         // Inicializamos el GO de la cámara,
44.         // pedimos su componente Transform,
45.         // y le seteamos una posición arbitraria en una esquina
46.         // del mundo mirando hacia la otra.
47.         player_camera = new GameObject();
```

```
48.         Transform *myTran = player_camera->getComponent<Transform>();
49.         myTran->setPosition(196.59, 143.11, 239.22);
50.         myTran->setLookAt(vec3(196.072, 142.516, 238.603));
51.
52.         // Pedimos también la cámara y inicializamos
53.         // el FPSController con ésta y el Transform.
54.         Camera *myCam = player_camera->addComponent<Camera>();
55.         fps = new FPSController(myCam, myTran);
56. }
57.
58. void rungholtApp::update()
59. {
60.         // LLamamos al FPSController para que se actualice.
61.         fps->update();
62.
63.         // Simplemente para debuggear, si se aprieta el clic
64.         // derecho del mouse, mostramos la posición del jugador
65.         // y hacia dónde mira.
66.         if (Input::getMouseButton("LEFT"))
67.         {
68.                 Transform *t = player_camera-
    >getComponent<Transform>();
69.                 printf("Estoy en (%f,%f,%f) mirando a (%f,%f,%f).\n",
    t->getPosition().x, t->getPosition().y, t->getPosition().z,
70.                         t->getLookAt().x, t->getLookAt().y, t-
    >getLookAt().z);
71.         }
72. }
73.
74. void rungholtApp::clean()
75. {
76.         printf("Cleaning.\n");
77.
78.         // Limpiamos la memoria antes de terminar.
79.         rungholt_world->getComponent<Mesh>()->clean();
80.         }
```

Código 24 Escena Rungholt entera (rungholt_app.cpp)

- Distintos métodos de renderizado

  Para cambiar la estrategia de renderizado vimos previamente que debemos decirle a la clase *MainRenderer* qué método utilizar. Por defecto se usa forward rendering, donde cada *GameObject* utiliza los shaders de sus propios materiales. También se tiene en cuenta el color del fondo de la escena seteado con `glClearColor()` en la línea 18 del Código 24. Entonces podemos utilizar el material de tierra declarado en la línea 31

(Código 24) con la textura *dirt.dds* y el shader *phong_textured* y la escena se vería como en la Figura 44. La misma escena renderizada con forward rendering pero sin texturas quedaría como en la Figura 45 (con la posibilidad de elegir un color por cada elemento, pero por simplicidad se pintó toda la escena del mismo).

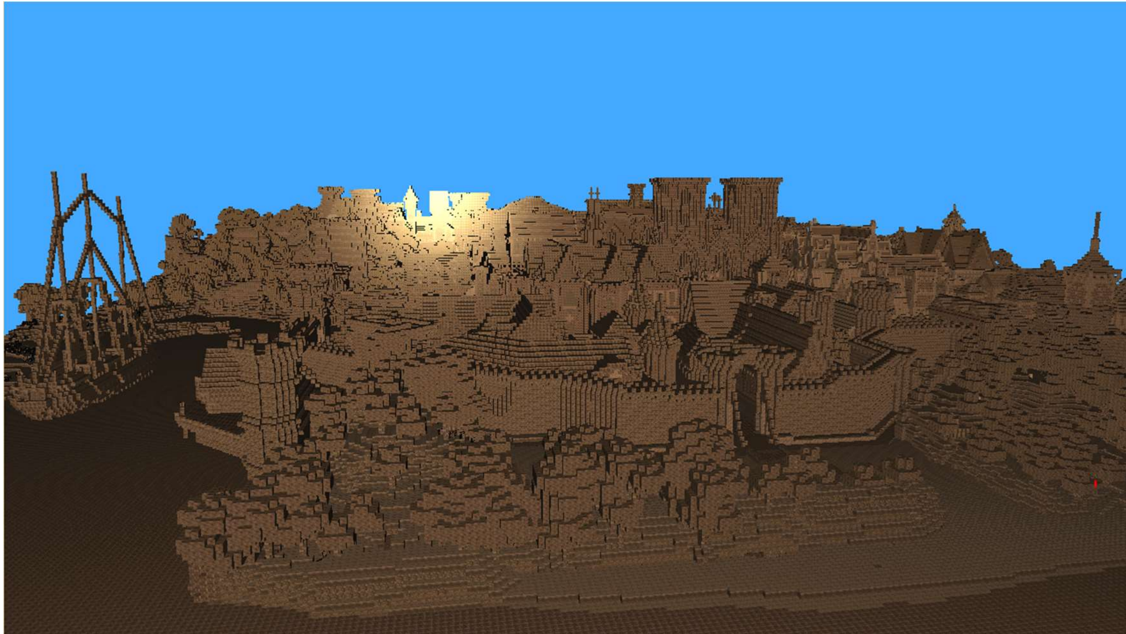

Figura 44 Escena Rungholt con forward rendering, Phong texturado.

Figura 45 Escena Rungholt con forward rendering, Phong sin texturas.

Continuando con el rendering diferido, este se activa con el método `setDeferredRendering`() de la clase *MainRenderer* en la línea 23 (Código 24), en

donde el booleano que recibe indica si queremos utilizar Ambient Occlusion o no. Primero lo seteamos en falso, y el resultado de renderizar la escena con cuatro luces de distintos colores es el de la Figura 46. En este caso, se pasan por alto los shaders individuales de cada objeto en la escena para utilizar el shader global de rendering diferido. Este según un identificador de material de cada objeto podría renderizarlo de distintas maneras pero por simplicidad se renderizó toda la escena con el mismo modelo. También se pasa por alto el cambio de color del fondo de la línea 18 del Código 24 (es negro en vez de celeste). Lo que sí se sigue utilizando de los materiales individuales de cada objeto, son las texturas de los mismos.

Figura 46 Escena Rungholt con deferred rendering, con luces de distintos colores.

Ahora, si en vez de llamar al método `setDeferredRendering`() con el parámetro en *false* lo llamamos con *true*, activaremos la técnica de Screen Space Ambient Occlusion (esta se activa en este método porque también usa rendering diferido). Esta sentencia internamente cambiará la estrategia de renderizado instanciando en *MainRenderer* un objeto de tipo *MainSSAORenderer*, que a su vez se encargará de cargar los shaders adecuados para utilizar la técnica. El efecto producido por la misma es el de oscurecer los bordes de los objetos de la escena, dándoles mayor realismo, y se puede observar aislado en la Figura 47.

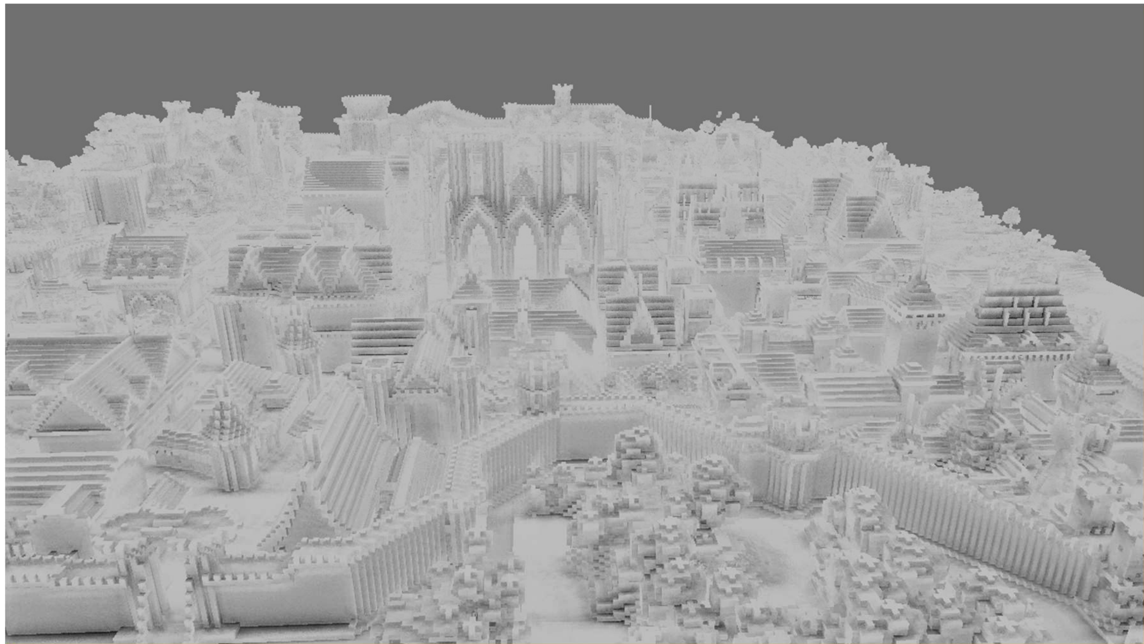

Figura 47 Escena Rungholt con rendering diferido y SSAO, solo componente de oclusión.

Se debe tener en cuenta que esta técnica por usar rendering diferido tampoco hace uso de los shaders individuales de cada material, pero sí de sus texturas (aunque en este ejemplo no sea el caso). El ejemplo demandó modificar el método de carga de los modelos, para tener en cuenta el color difuso y ambiente de cada objeto del mismo, ya que cargar a mano uno por uno sería muy tedioso. Como trabajo a futuro quedará decidir si esta característica debe incorporarse al motor por defecto o si debe formar parte de cada aplicación en particular.

Finalmente, combinando esta técnica con el modelo de iluminación Phong visto previamente es como alcanzamos el mayor nivel de integración de técnicas y fidelidad visual de Teardrop hasta el momento, como se demuestra en algunas imágenes como la Figura 48 y la Figura 49.

Figura 48 Barco en escena Rungholt con rendering diferido, iluminación Phong y SSAO.

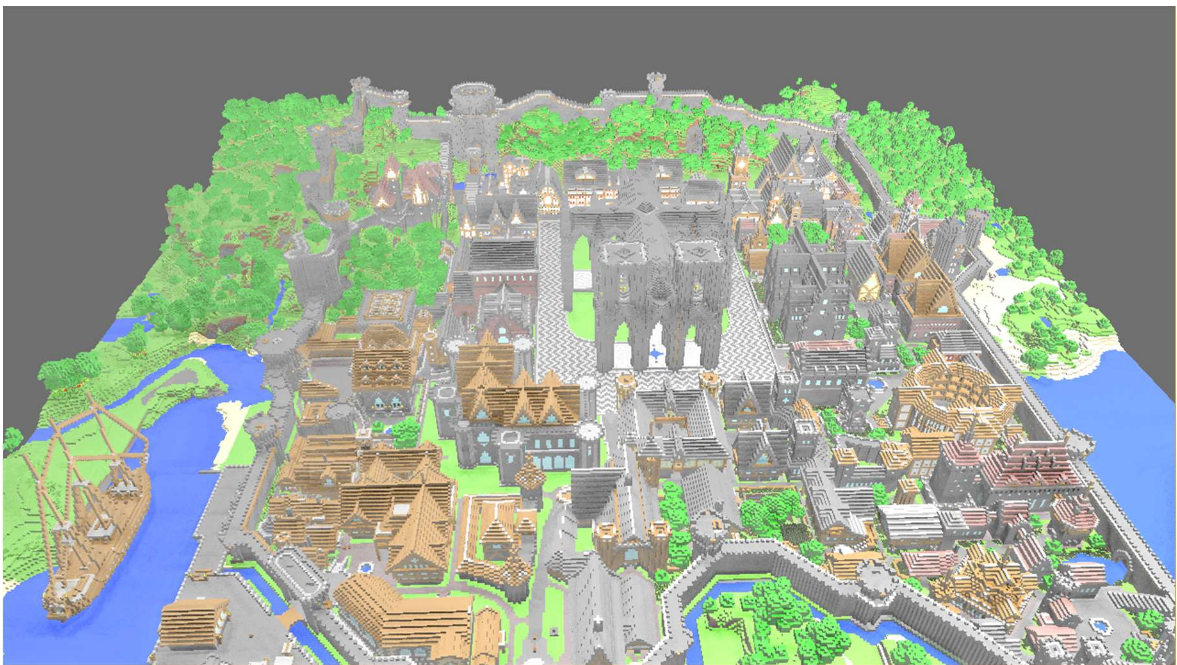

Figura 49 Escena Rungholt vista desde arriba con rendering diferido, iluminación Phong y SSAO.

En la Figura 50 se observa la misma escena renderizada en la Figura 49 pero con distintas intensidades de la componente difusa del modelo Phong, se puede apreciar cómo la parte derecha de la imagen presenta colores más brillantes que la parte izquierda. De acuerdo a este valor es como se puede variar la iluminación de la imagen.

Figura 50 Diferentes niveles de intensidad en la componente difusa.

En la Figura 51 se aplicó una textura sobre el modelo y mezclamos su color con el color ambiente que extrajimos del modelo, obteniendo un efecto extraño al estilo sepia.

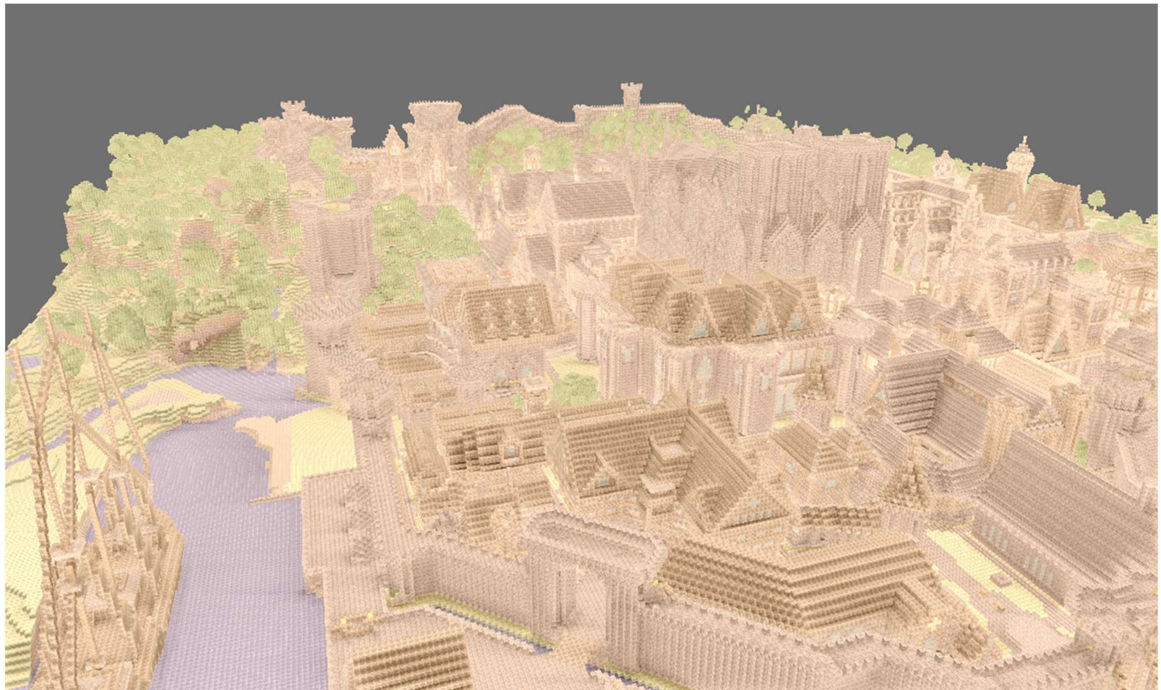

Figura 51 Jugando aplicando una textura sobre el color difuso.

La Figura 52 presenta una modificación en vivo de los colores, tomamos el color difuso del modelo y le modificamos las componentes RGB para experimentar diferentes combinaciones.

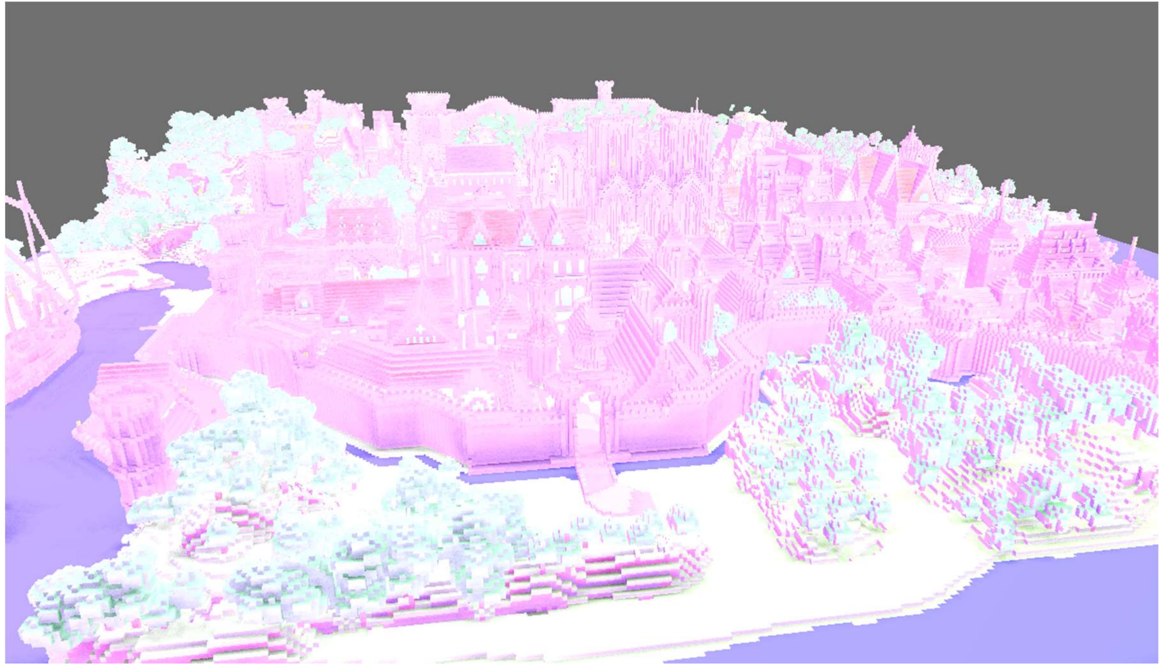

Figura 52 Experimentando con modificaciones del color difuso desde el shader.

Como se puedo apreciar en las Figuras 48, 49, 50, 51 y 52, las posibilidades de renderizar una misma escena son infinitas, en estas figuras las modificaciones se aplicaron sobre la escena entera, pero podrían hacerse individualmente de manera diferente en cada objeto, separándolos de acuerdo a un ID de material, solo es cuestión de tener creatividad y querer diseñar algo agradable a la vista. Se pueden combinar las distintas técnicas implementadas en el motor, agregar nuevas fuentes de luz que modifiquen el contexto lumínico, de distintos colores, distintos tipos de materiales de acuerdo a distintas matcaps, texturas, variaciones de componentes del modelo Phong, etc.

Capítulo 6: Conclusiones y Trabajos Futuros

Teardrop es altamente extensible con infinitas posibilidades de crecimiento. Utiliza los mismos lenguajes y librerías que varios estudios profesionales de videojuegos AAA, y ciertas técnicas implementadas son actuales como para verse en juegos de hoy. Teardrop es el resultado de un largo camino de aprendizaje e investigación de un área de la informática muy poco documentada y difundida, la curva de aprendizaje se empina mucho luego de los primeros tutoriales para entrar en tema, por lo que para implementar técnicas nuevas se debe descifrar código de terceros no documentado o directamente implementar papers de SIGGRAPH, el grupo de computación gráfica, y para esto se necesita un conocimiento muy avanzado en matemática y física.

En esta tesis se presentó el modelo definido para Teardrop, como así también la implementación llevada a cabo. Con un ejemplo de uso se pudo apreciar cómo el mismo puede ser utilizado.

Entre las infinitas formas de mejorar Teardrop y hacerlo crecer para transformarlo en una herramienta competente y capaz de usarse para el desarrollo de videojuegos modernos, se encuentran las siguientes *features*, algunas que pensadas originalmente, no se llegaron a implementar, algunas planteadas a futuro pero empezadas a implementar y otras simplemente pensadas a futuro.

- Instalador multiplataforma

  Por el momento Teardrop no tuvo tiempo de salir del ambiente de desarrollo de Visual Studio, es decir, hace uso de ciertas librerías que vienen instaladas en el mismo y por lo tanto depende de su instalación, esto lo hace no solo dependiente de Visual Studio, sino dependiente de Windows. Se debe crear un instalador *standalone* que instale Teardrop en cualquier sistema operativo, y para ello quizás se necesite cambiar algunas líneas de código encargadas de ejecutar el programa, como la declaración del método main() o la macro DECLARE_MAIN(). El nivel de dificultad de esta tarea o *feature* es bastante bajo.

- Reloader de shaders globales

  Durante el desarrollo de una aplicación, uno debe probar distintas opciones en cuanto a valores de variables en los shaders, flujos de ejecución, etc. El problema que se presenta generalmente es el de tener que cerrar la aplicación principal de C++ y volver a abrirla para que estos cambios se vean reflejados en pantalla, es decir, que el motor utilice la nueva versión de los mismos. Para mejorar este flujo de trabajo en Teardrop se implementó un recargador de shaders que no requiere reiniciar la aplicación sino simplemente apretar una combinación de teclas definidas por el usuario, así se pueden visualizar los cambios realizados con sencillez y velocidad.

  Sin embargo, esta tarea solamente funciona con los shaders individuales que posee cada objeto de tipo Material, como el de iluminación Phong o Spherical Environment Mapping y no con las técnicas globales o también llamadas de screen space como

Deferred Rendering o Screen Space Ambient Occlusion. Esto conforma una buena funcionalidad de baja dificultad para implementar en el futuro que mejoraría bastante la experiencia de desarrollo de shaders.

- Scripts en GameObjects

  Como se ha visto en el diseño de clases del Capítulo 3, específicamente en la descripción de los componentes en la Sección 3.1.3, se diseñó un sistema de componentes de tipo Script donde cada uno manejaría el comportamiento del *GameObject* al que perteneciera, liberando al archivo de implementación de la escena de código específico de *GameObjects*. Si bien se comenzó a implementar dicha funcionalidad, se presentaron problemas, como por ejemplo la dificultad de permitir que un mismo GO posea varios componentes distintos de tipo Script, donde cada uno maneje funcionalidades independientes, o cada uno a un componente particular; y finalmente la falta de tiempo de desarrollo. Si bien este no es un bien esencial donde su falta deje al motor inutilizable, clasifica como una funcionalidad crucial a implementar en el futuro próximo, de prioridad alta por estructurar de manera organizada y limpia el código de las aplicaciones.

  La mayor dificultad recae en que cada script tenga código diferente (este es su propósito elemental). Una opción es que todos subclasifiquen de la superclase abstracta Script que posea métodos como setup(), run() y clean(), los cuales serían llamados por la clase *Engine*. El *GameObject* guardaría punteros inteligentes de cada script en un vector de C++ del tipo de la superclase, para dinámicamente poder referenciar a distintas subclases. En el archivo de la escena se indicaría qué scripts tendría cada GO y para ello debería incluir el header de cada uno de ellos. La clase *Engine*, que solo llamaría a los métodos definidos en la clase abstracta Script, no necesitaría más que referenciar al header de la misma.

  Otra aproximación de mucha mayor complejidad, pero más flexible, poderosa y ventajosa para el usuario del motor, es desarrollar un sistema de scripting, semejante al del motor Unity3D, en donde el usuario desarrolle sus scripts en un lenguaje de scripting de alto nivel a su elección como Javascript, C# o Ruby, y el motor se encargue de compilarlos y traducirlos a C++ para su utilización. El desarrollo de aplicaciones sería mucho más sencillo (lenguajes más fáciles de aprender por usuarios inexpertos), y con implementar el compilador/traductor a C++ ya se integrarían diferentes sintaxis, permitiendo que haya scripts programados en distintos lenguajes interactuando entre sí. Esta gran funcionalidad brindaría al motor la flexibilidad de un lenguaje de alto nivel con la performance de uno de bajo nivel. Para implementarla se requeriría un grupo de desarrolladores más grande ya que llevaría mucho tiempo enfocarse en dicha tarea siendo un equipo de uno, cuando hay muchas otras features a desarrollar de mayor prioridad, como nuevas técnicas visuales.

- Editor gráfico de escenas

  Por el momento, para editar una escena uno debe editar el código de la misma, agregando variables e interfaces en el header e implementaciones en el archivo cpp. Allí se instancian los *GameObjects*, se le agregan y configuran distintos tipos de componentes, etc. Sería muy práctico poder realizar todas estas tareas de manera visual, en donde por un lado se tiene una vista de la escena en 3D y por otro un panel con la lista de *GameObjects* que posee. Entonces clickeando un GO nos aparecería una ventana con sus propiedades, con la posibilidad de agregarle componentes con solo unos clicks. Se podría cambiar la posición, rotación y escala de los Transforms de cada GO desde la vista en 3D, pudiendo así armar una escena de manera sencilla e intuitiva con la posibilidad de ver los resultados de cada cambio en tiempo real. Se debería tener una escena principal que se ejecute al abrir la aplicación y tener la posibilidad de cargar distintas escenas en tiempo de ejecución dependiendo de las acciones del usuario (distintos niveles de un juego). Para esto, habría que automatizar la declaración de qué escena es la principal y crear un sistema de cambio de escenas en ejecución con la limpieza de memoria como principal punto a tener en cuenta.

- Sistema de carga automática de uniforms en shaders

  El hecho de tener que cargar uniforms manualmente para cada shader es una tarea tediosa y muy poco genérica. Esto conforma un problema grave ya que el encargado de cargarlos es la clase *Renderer* y el método debería funcionar para todos los shaders sin importar qué uniforms necesite.

  Un planteo muy interesante que automatiza todo este proceso es el de tener un representante de cada uniform en C++ (con una clase) que mapee los valores del uniform en memoria de la placa de video, con los valores de la instancia en memoria principal. Para utilizarlo se escanearía el código de los shaders en busca de uniforms y una vez encontrados un UniformFactory crearía su contraparte en memoria principal encargándose de que cada uniform sea del tipo correspondiente. Como hay uniforms que se repiten entre distintos shaders (como la matriz MVP), un sistema de suscripción (con patrón Observer) permitiría que el shader se suscriba a un uniform y cuando el valor de este cambie, cada shader suscripto sea notificado.

  Llegado el momento, cada shader por su cuenta pediría el valor actualizado del uniform y lo enviaría a la ubicación correspondiente en la placa de video.

  Actualmente Teardrop tiene una implementación en fase beta de este sistema para tipos de uniform estándar como vec3, vec4, mat4, int o floats. Por ser una funcionalidad sumamente útil que haría a Teardrop un motor escalable, a futuro, además de terminar esta implementación e integrarla al resto del motor, se debería plantear una solución para uniforms de tipos personalizados (quizás si éstos son structs de tipos primitivos, accederlas y realizar el mismo proceso con estos tipos primitivos).

- Soporte de distintos formatos

  Para que el motor sea competente y práctico en su utilización debería soportar la mayor cantidad de formatos de assets. Para las texturas podría incluir el uso de imágenes en formato PNG o JPG, y para modelos 3D formatos propietarios como 3DS del software de modelado 3DStudio, MA de Maya, y el más importante por ser universal y guardar animaciones, el formato FBX. Para lograr esto se debería no solo implementar los métodos de carga de cada formato, sino tener en cuenta la eficiencia de cada uno, su peso en memoria, el tipo de carga, etc. y evaluar si quizás haga falta una conversión interna al formato más eficiente. En cuanto a la carga de modelos, el formato FBX debería ser el primero en incorporar, pero a su vez implementando un sistema de animaciones que Teardrop actualmente no posee.

- Dependencias de headers automáticas

  A la hora de utilizar una clase, una opción es incluir el header de la misma con la directiva del preprocesador #include. Otra opción utilizada para evitar dependencias circulares, como se vio en la Sección 4.3, consiste en declarar una clase sin cuerpo con el mismo nombre en el archivo en el cual la usemos para que el compilador no de errores de tipo desconocido. Este tipo de tareas tediosas no solo las tendría el desarrollador del motor, un usuario del mismo queriendo utilizar cualquier clase que haya implementado lo tendría también. Por eso, una feature de dificultad baja a intermedia sería desarrollar un resolvedor de dependencias que se encargue de importar automáticamente las clases utilizadas por el programador teniendo en cuenta el problema de las dependencias circulares para que todo este proceso sea transparente al usuario.

- Otras features

  Entre otras características a implementar a futuro en Teadrops se presenta la de poder leer las componentes de color difuso y especular directamente desde un modelo 3D. Así el modelador puede asignarle colores en el editor 3D y que Teadrop los cargue y asigne a un material por cada figura dentro del modelo. Esta funcionalidad está parcialmente implementada en el motor pero se debe refactorizar y generalizar para que sea escalable.

  Una funcionalidad muy útil que debería implementarse en próximas versiones del motor es la de permitir una jerarquía de *GameObjects*. Es decir, que un *GameObject* contenga en sí mismo una colección de GOs, entonces, por ejemplo el GO auto tendría distintas partes como motor, puerta y asiento, donde cada uno sería otro GO y tendría sus propios componentes. Si bien no parece presentar mayores dificultades su implementación, agregaría mucha flexibilidad al diseño y desarrollo de aplicaciones de Teardrop. Una posibilidad, como se planteó previamente al hablar de Scripts, es la de que cada *GameObject* tenga un arreglo interno con referencias a otros GOs, formando una estructura de árbol general.

Referencias

1. Juval Löwy. (2005). Programming .NET Components. O'Reilly.
2. Karel Crombecq. (n.d.). Component-based programming. A new programming paradigm.
3. Ian Sommerville. (n.d.). Component-based software engineering.
4. Michael Doherty. (n.d.). A Software Architecture for Games.
5. David Garlan, Robert T. Monroe, David Wile. (n.d.). Acme: Architectural Description of Component-Based Systems.
6. Schilit, W. (1995). System architecture for context-aware mobile computing. PhD thesis, Columbia University
7. Dey A. K. (2000). Providing Architectural Support for Building Context-Aware Applications.PhD Thesis, Georgia Institute of Technology.
8. Wright Jr., Richard S., Nicholas Haemel and Graham Sellers. (2014). OpenGL Superbible, 6th edition, Addison-Wesley, Pearson Education Inc.
9. Gamma, E., Helm, R., Johnson, R., y J. Vlissides. 1994. Design Patterns: Elements of Reusable Object-Oriented Software.
10. http://stackoverflow.com/questions/5550620/the-purpose-of-model-view-projection-matrix
11. http://www.songho.ca/opengl/gl_projectionmatrix.html
12. http://en.wikipedia.org/wiki/Frustum
13. http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/
14. http://en.wikipedia.org/wiki/3D_projection
15. http://en.wikipedia.org/wiki/Frame_rate
16. http://duriansoftware.com/joe/An-intro-to-modern-OpenGL.-Chapter-1:-The-Graphics-Pipeline.html
17. http://en.wikipedia.org/wiki/Vertex_Buffer_Object
18. http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-9-vbo-indexing/
19. https://www.opengl.org/wiki/Vertex_Specification
20. http://stackoverflow.com/questions/11821336/what-are-vertex-array-objects
21. http://en.wikipedia.org/wiki/Shader
22. http://en.wikipedia.org/wiki/Ray_tracing_%28graphics%29
23. http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter14.html
24. http://www.independentdeveloper.com/archive/tag/normal-mapping
25. http://en.wikipedia.org/wiki/OpenGL
26. http://glm.g-truc.net/0.9.6/index.html
27. http://glew.sourceforge.net/
28. http://stackoverflow.com/questions/17809237/what-does-glew-do-and-why-do-i-need-it
29. http://www.glfw.org/
30. http://en.wikipedia.org/wiki/GLFW
31. http://sourcemaking.com/design_patterns/template_method

32. http://docs.unity3d.com/ScriptReference/Time-deltaTime.html
33. http://en.wikipedia.org/wiki/Mipmap
34. http://en.wikipedia.org/wiki/DirectDraw_Surface
35. http://madlevelmanager.madpixelmachine.com/doc/2.0.1/advanced/fonts/texture_and_glyph_list.html
36. https://github.com/memononen/fontstash
37. http://en.wikipedia.org/wiki/Field_of_view
38. http://answers.unity3d.com/questions/129030/why-is-near-clip-plane-necessary.html
39. http://help.autodesk.com/view/MAYAUL/2015/ENU/?guid=Camera_set_up_Clipping_planes
40. http://en.wikipedia.org/wiki/Phong_shading
41. http://en.wikipedia.org/wiki/Phong_reflection_model
42. http://sourcemaking.com/design_patterns/singleton
43. http://www.glfw.org/docs/latest/quick.html
44. https://en.wikipedia.org/wiki/Screen_tearing
45. http://syoyo.github.io/tinyobjloader/
46. https://www.opengl.org/sdk/docs/man3/xhtml/glBufferData.xml
47. http://en.cppreference.com/w/cpp/language/function_template
48. https://www.khronos.org/opengles/sdk/docs/man/xhtml/glVertexAttribPointer.xml
49. https://www.opengl.org/sdk/docs/man/html/glCompileShader.xhtml
50. http://www.g-truc.net/project-0024.html
51. http://en.wikipedia.org/wiki/Framebuffer_Object
52. https://www.opengl.org/sdk/docs/man/html/glDrawBuffers.xhtml
53. http://en.wikipedia.org/wiki/Circular_dependency
54. http://en.wikipedia.org/wiki/Forward_declaration
55. http://www.clicktorelease.com/blog/creating-spherical-environment-mapping-shader
56. https://www.opengl.org/sdk/docs/man/html/packHalf2x16.xhtml
57. http://cse.csu.edu/tongyu/courses/cs420/notes/lighting.php
58. http://www.predatron.co.uk/three-point-lighting/
59. http://www.neuroproductions.be/opengl/making-a-3d-game-with-opengl-deferred-shading-and-stuff/
60. http://www.codeproject.com/Articles/616344/What-is-Circular-dependency-and-how-do-we-resolve#_articleTop
61. http://swarminglogic.com/
62. http://www.glfw.org/docs/latest/group__context.html#ga1c04dc242268f827290fe40aa1c91157
63. http://blog.wolfire.com/2009/07/linear-algebra-for-game-developers-part-1/
64. http://gamedev.stackexchange.com/questions/11398/math-topics-for-3d-graphics-programming