UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE INFORMÁTICA
VALPARAISO - CHILE



"IMPLEMENTACIÓN Y ANÁLISIS DE SOMBRAS DIFUSAS EN COMPUTACIÓN GRÁFICA EN TIEMPO REAL UTILIZANDO VARIADOS MÉTODOS DE PROYECCIÓN Y VOLÚMENES DE SOMBRA"

MICHEL JOFRÉ GODOY

MEMORIA DE TITULACIÓN PARA OPTAR AL TÍTULO DE INGENIERO CIVIL INFORMÁTICO

PROFESOR GUÍA: **HUBERT HOFFMANN**

PROFESOR CORREFERENTE: **SVEN VON BRAND**

DICIEMBRE - 2018

Agradecimientos

Al ver terminada esta memoria, recuerdo a todas las personas que me han apoyado en este camino, no ha sido una tarea fácil pero pude salir adelante. Mis padres siempre me brindaron su apoyo y comprensión durante toda la carrera, tiempo en que crecí y me desarrolle como persona, ahora que soy mayor comprendo todo el esfuerzo que ellos hicieron y se los agradezco desde el fondo de mi corazón. Mi familia fue un soporte que estuvo ahí cuando mas los necesitaba, esa unión me daba fuerza en cada paso que di. La universidad me dio las herramientas para enfrentar al mundo y agradezco a los profesores por el tiempo que me dedicaron, gracias a ellos aprendí bastante en todos estos años y en especial agradezco a mi profesor guiá por toda la paciencia que tuvo para llevar este trabajo hasta el final.

Resumen

El objetivo de este trabajo es revisar la implementación de las sombras difusas en tiempo real, y analizar su calidad y rendimiento. Las sombras difusas toman cada vez más protagonismo en la computación gráfica, todo esto gracias a las nuevas técnicas y al nuevo hardware que se va creando. La implementación se revisará a nivel de código, documentándose cada paso necesario para aquello. El rendimiento de la sombra difusa se revisará a través de un benchmark que se realizará a las implementaciones vigentes. Por otra parte, la calidad de las sombras se extraerá de una encuesta que se aplicará sobre un grupo de personas. Finalmente, los datos obtenidos se utilizarán para crear la conclusión final sobre las sombras difusas y las implementaciones que se pueden hacer para obtenerlas.

Abstract

The aim of this work is the implementation of algorithms for the creation of soft shadows and an analysis of the quality and the performance of the different algorithms. Soft shadows are always a challenge in computer graphics, because in the last time, they evolved thank new techniques and new graphics hardware. The analysis is made at code level with a step by step documentation of the implementation. The performance of the newest algorithms of soft shadows will be analysed with a benchmark. The quality of the shadows will be judged by a group of persons. Finally, the data obtained will be used for the conclusions drawn about the soft shadows and the algorithms used.

Glosario

Aliasing: Son los bordes dentados de una curva o una recta inclinada que se observan en una imagen, este efecto se produce debido a la gran diferencia de color entre una serie de pixeles.

Color Buffer: Es el buffer que almacena los colores de una escena teniendo un color para cada pixel, éste utiliza la representación rojo, verde y azul (RGB, por sus nombres en inglés), en donde se le asigna a cada color primario un valor que va desde 0 hasta 1, la combinación de la fracción de cada color (rojo, verde y azul), dará como resultado un color en especial. Además, este buffer puede contener un valor más, el cual almacena el grado de transparencia del color resultante.

Depth Test: Es la prueba que se hace con el stencil buffer, se lanza una línea recta que entra y sale de todos los polígonos existentes, sumando uno si entra y restando uno si sale, el valor del test dependerá de la suma y resta de esos valores.

Oclusor: Cuando un objeto proyecta una sombra, el oclusor es la superficie que se forma a partir de los bordes de la sombra y su interacción con el objeto.

Penumbra: Es la zona de la sombra en que una fuente de luz es parcialmente obstruida, lo que ocurre solo en una fuente de luz con un volumen considerable.

Pixel Depth: Es la profundidad a la que se encuentra un pixel de una escena con los objetos visibles con respecto a la cámara.

Sombra: Es una zona de oscuridad donde la luz es obstruida.

Sombra Auto Proyectada (Self Shadow): Es un tipo de sombra que se produce cuando un objeto proyecta una sombra sobre sí mismo, pero esta superficie no puede estar dentro de la zona de la sombra propia.

Sombra Difusa (Soft Shadow): Es un tipo de sombra que se produce por una fuente de luz que tiene un volumen considerable, por lo cual los límites de la sombra no

son claros. En este límite existe una zona en que se degrada la sombra, pasando lentamente de una zona de sombra a otra de no sombra.

Sombra Dura (Hard Shadow): Es un tipo de sombra que se produce por una fuente de luz puntual, esto se refiere a que el volumen de la fuente de luz es muy pequeño, por lo cual los límites de este tipo de sombras están bien definidos.

Sombra Propia (Attached Shadow): Es un tipo de sombra que se forma en un objeto debido a la posición de su superficie con respecto a la dirección a la fuente de luz. Toda superficie cuya normal tenga un ángulo mayor a 90° con respecto a la luz incidente pertenecerá a este tipo de sombra.

Sombra Proyectada (Cast Shadow): Es un tipo de sombra que se forma en una superficie debido a que existe otro objeto que se interpone en el paso de la luz bloqueando el paso de ésta.

Stencil Buffer: Es un buffer de enteros que almacena el estado de cada pixel. Este buffer puede ser usado de varias maneras: El método más sencillo para utilizarlo es asignándole el valor 0 o 1 con el cual se puede saber si se dibujará un pixel o no. El stencil buffer también es capaz de efectuar un test en el cual puede incrementar o restar el valor del buffer dependiendo si el pixel pasa el test o no.

Umbra: Es la zona de la sombra en que una fuente de luz es completamente obstruida.

Z-Buffer: Es una textura que almacena la profundidad de la escena mostrando la distancia a la cámara de los objetos visibles.

Z-Depth: Es la profundidad de un vértice con respecto a la cámara, no importa si este vértice esté oculto o no.

Índice general

1.. Alcance de la memoria 3

1.1. Motivación 3

1.2. Objetivo General 4

1.3. Objetivos específicos 4

1.4. Estructura del documento 4

2.. Base conceptual 6

2.1. Explicación física de una sombra 6

2.2. Tipos de sombras 7

2.2.1. Sombra propia 7

2.2.2. Sombra proyectada 7

2.2.3. Sombra auto proyectada 7

2.3. Formas de sombras 7

2.3.1. Sombras duras 7

2.3.2. Sombras difusas 8

3.. Estado del arte 9

3.1. Inicios de las sombras en computación gráfica 9

3.1.1. Shadow computation during scanout 9

3.1.2. Two-Pass Approach 10

3.1.3. Projected Shadow Polygons 11

3.2. Implementación de las sombras duras 11

3.2.1. Mapping Shadows 11

3.2.2. Shadow Volume 14
3.3. Sombras Difusas 17
3.3.1. Mapping Shadows 17
3.3.2. Volúmenes de sombra 24
3.3.3. Técnicas actuales 28

4.. Implementación de las sombras 32
4.1. Plataforma a usar 32
4.1.1. API Gráfica 32
4.1.2. Librerías extras 33
4.1.3. Shaders 35
4.2. Implementación 38
4.2.1. Básico 38
4.2.2. Shader 42
4.2.3. Sombras Duras 47
4.2.4. Técnicas actuales de sombra difusa 72

5.. Análisis de las técnicas actuales de sombra difusa 77
5.1. Metodología 77
5.2. Rendimiento 78
5.3. Calidad 81
5.4. Comparación 90

6.. Conclusiones 92
6.1. Sombras Difusas 92
6.2. Trabajo a futuro 94

1. ALCANCE DE LA MEMORIA

1.1. Motivación

Las sombras han sido un importante tema de estudio en la computación gráfica, ya que desde el inicio de esta disciplina se han investigado variadas formas de implementarlas. Durante bastante tiempo se teorizó su implementación hasta que el hardware se desarrolló lo suficiente para que se crearan soluciones prácticas. Sin embargo, Tomaron casi dos decadas en que se desarrollaran verdaderas soluciones en tiempo real. La evolución de las sombras en computación gráfica ha ido muy de la mano con el desarrollo de las tarjetas gráficas, debido a la gran cantidad de recursos que utilizan, por lo tanto para mejorar la calidad de las sombras es necesario un aumento en la potencia del hardware.

Las sombras en computación gráfica han avanzado bastante, pero aún le queda mucho para mejorar, en especial la tonalidad de la misma sombra que cambia dependiendo de donde se encuentre. En la medida en que se desarrolle nuevo hardware gráfico, se podrán realizar aún más mejoras a las sombras, más aun cuando se intente llevar a cabo ambientes más realistas en tiempo real, ya que se deberá combinar un gran resultado con un rendimiento que permita ser visto a una buena cantidad de fotogramas por segundos.

1.2. Objetivo General

Esta memoria trata sobre las sombras, su implementación y análisis, centrándose especialmente en las sombras difusas. También intenta explicar su papel en las escenas 3D y la importancia que tienen éstas en la computación gráfica.

1.3. Objetivos específicos

- Conocer qué son exactamente las sombras difusas.
- Investigar los primeros pasos de las sombras en computación gráfica.
- Analizar qué técnicas se utilizan hoy en día en computación gráfica para la presentación de sombras en tiempo real.
- Mostrar las implementaciones de varios tipos de sombras en OpenGL en tiempo real
- Analizar las diferentes formas de implementación de sombras difusas que se usan en la actualidad.

1.4. Estructura del documento

- En el capítulo 1 se entregarán las bases con las cuales se realizará la memoria, las motivaciones, los objetivos y la estructura de la misma.
- En el capítulo 2 se explicarán las bases conceptuales para entender las sombras y se desarrollará una explicación del fenómeno físico que está detrás de ellas, entendiéndose los retos que se tendrán que enfrentar para lograr los mejores resultados.
- En el capítulo 3 se hará un recorrido por la historia de las sombras en computación gráfica, profundizándose en las diferentes técnicas que se han propuesto desde los inicios teóricos hasta las técnicas actuales.

- En el capítulo 4 se revisarán las implementaciones de sombras en computación gráfica, por medio de ejemplos prácticos de implementación a través de líneas de código.
- En el capítulo 5 se analizarán las técnicas actuales de sombras difusas en tiempo real, comparando su rendimiento y su calidad.
- En el capítulo 6 se hará una conclusión sobre el papel y el impacto de las sombras difusas en computación gráfica y el futuro del uso de la luz.

2. BASE CONCEPTUAL

2.1. Explicación física de una sombra

Una sombra es un fenómeno físico en el cual un objeto que se llama oclusor se interpone en la trayectoria de la luz de una cierta fuente, bloqueándola parcial o totalmente. Este bloqueo permite que esta sección de luz no llegue al objetivo que se encuentra en la dirección en que los rayos se propagan. El oclusor es la superficie que se forma a partir de los últimos bordes del volumen que entran en contacto con la luz. Para conocer los bordes que pertenecen al oclusor es necesario hacer un producto punto entre la luz y los dos polígonos que tienen en común una arista. El producto punto puede ir desde -1 hasta 1, indicando que la arista forma parte del oclusor si el producto en un polígono es positivo y el otro es negativo, o que dicha arista no pertenece al borde si ambos son positivos o negativos. La forma de esta superficie es similar a la forma de la sombra que proyecta: si los rayos de luz fueran paralelos entre sí y perpendiculares al oclusor, la sombra tendría la misma forma y el mismo tamaño de la superficie del oclusor, aunque esto podría variar si la superficie no se encontrara en forma paralela a la superficie del oclusor. Por esta razón la forma y el tamaño de la sombra pueden ser calculados con transformaciones simples, ya sea para cambiar el tamaño de la sombra en forma general o cambiando el tamaño según sus ejes si la superficie está inclinada con respecto a la superficie del oclusor.

2.2. Tipos de sombras

2.2.1. Sombra propia

La sombra propia es provocada por el mismo objeto debido a su geometría, ya que el porcentaje de luz reflejada es igual al producto punto entre la normal del polígono y el vector de la dirección de la luz. Cuando el ángulo que forman ambos es de $0°$, el producto punto es 1 dando el mayor valor posible, pero si el ángulo es de $90°$ o más, el producto punto obtenido es 0 o negativo; en este caso al valor de reflexión se le da un valor 0, lo que significa que no se refleja luz en la superficie.

2.2.2. Sombra proyectada

La sombra proyectada se produce cuando un objeto bloquea la luz a otro y dado que es un objeto opaco, no deja pasar los rayos de luz, por lo tanto, la sombra proyectada tiene la forma del oclusor que está bloqueando la luz.

2.2.3. Sombra auto proyectada

Este tipo de sombra se produce porque hay sectores que no son iluminados por la fuente de luz a pesar de que el ángulo de la luz incidente y la normal sea menor a $90°$, debido a que existe una sección del objeto que bloquea la luz a otra parte del mismo, provocando una sombra.

2.3. Formas de sombras

2.3.1. Sombras duras

Cuando una fuente de luz es puntual (lo cual significa que casi no tiene volumen, reduciéndose la fuente a un único punto) produce una sombra dura, debido a que existe una única dirección desde la cual provienen los rayos de luz y estos inciden en un único punto, entonces el borde de la sombra coincide con un único rayo que pasa

por el borde del oclusor, por lo tanto existe un límite claro entre sombra y no sombra.

2.3.2. Sombras difusas

Las sombras difusas se producen debido a que la fuente de luz que debiera iluminarlas tiene volumen, por lo cual dicha fuente de luz tiene una amplia superficie que produce luz. Debido a esto, los múltiples rayos que parten de la superficie de la fuente pueden incidir en un mismo punto desde diferentes direcciones, por esta razón los rayos que pasen directamente cerca del borde del oclusor llegarán a diferentes posiciones de la superficie, por lo cual no existe un borde exacto de sombra y no sombra, denominándose a esta área difusa penumbra. La penumbra está limitada, ya que empieza desde el punto en que toda la luz proveniente de la fuente logra iluminarlo por completo hasta otro punto en que toda la luz que es emitida por la fuente es bloqueada; así, toda el área intermedia entre estos dos puntos recibirá una cantidad no exacta de luz de la superficie emisora de luz, ya que varios de sus rayos son bloqueados por el oclusor. La intensidad de la sombra en la penumbra depende del porcentaje de luz que logra llegar hasta ese punto. Las sombras tienen otra zona que se le llamará umbra, la cual es de sombra total, ya que no recibe luz. Las sombras duras solo están compuestas de esta zona, pero las sombras difusas pueden no tenerla, debido a que la fuente de luz es más grande que el oclusor, por lo cual no existen puntos dentro de la sombra en que no llegue al menos un cierto porcentaje de la luz.

3. ESTADO DEL ARTE

3.1. Inicios de las sombras en computación gráfica

En los inicios de la computación gráfica se analizaban varias formas para enfrentar cada uno de los retos que conlleva recrear un modelo en 3 dimensiones dentro de un computador. Uno de estos retos eran las sombras, ya que las sombras juegan un rol muy importante en toda representación gráfica, puesto a que aporta información extra de un objeto dentro de un escenario complejo. Es por esto que se intentaba dilucidar soluciones a este problema gráfico, pero las soluciones creadas solo se basaban en la teoría y las soluciones reales no aparecieron hasta mas de una decada después, siendo que la teoría comenzó a mitad de la década de los 70 y los primeros usos prácticos estarían apareciendo a principio de los 90.

3.1.1. Shadow computation during scanout

Shadow computation during scanout utiliza el cálculo geométrico, ya que esta sombra se va creando a partir de las superficies y su relación con la luz. Durante el análisis de la escena, se generan líneas que van desde la posición original de la luz hasta los vértices de la superficie oclusora; los vectores obtenidos son proyectados a las demás superficies, entonces se calcula qué superficies puedan estar detrás de estos vectores y se les esconde, mientras que las superficies que no estén dentro de las sombras calculadas por estos vectores se muestran, entonces en pantalla solo se muestran las superficies que no tengan ningún tipo de sombra sobre ellas, tal como se ve en la figura 3.1.

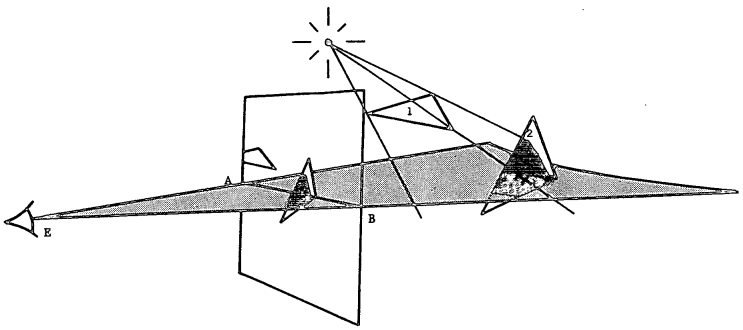


Fig. 3.1: Fuente: Shadow algorithms for computer graphics [1]

Existen otras formas de aplicar este método, una de éstas es considerando coordenadas esféricas cuyo centro sería la luz, entonces se podrán definir las rectas y los segmentos que van cortando de una manera más simple, sin embargo, el problema de este método es realizar la transformación entre las diferentes formas de coordenadas.

3.1.2. Two-Pass Approach

Two-Pass Approach sirve para encontrar superficies escondidas, las que pueden estar escondidas de la fuente de luz o del observador de la escena, conociendo esos datos se pueden mejorar los tiempos de procesamiento de la escena, ya que también ayuda en otros pasos como preparar los objetos que se mostrarán en la escena, todo comienza buscando las siluetas de los objetos en escena para generar las superficies, siendo estas siluetas las que separan la parte visible de un objeto con la parte sombreada. Luego se clasifican las superficies en orden de importancia, ya que cada una tiene un cierta importancia en la escena y si el objeto no tiene mucha importancia, este se encierra en un cubo delimitador, este cubo utiliza los puntos más altos y bajos, como los extremos que se encuentran en cada lado, para fijar estas aristas; esta caja simplifica el cálculo, ya que se coloca una caja en vez de procesar toda la superficie para encontrar el límite de la misma, dejando fuera en este proceso las superficies que no son visibles para luego calcular las sombras.

3.1.3. Projected Shadow Polygons

Projected Shadow Polygons utiliza polígonos creados a partir de la proyección de la zona sombreada con respecto a la fuente de luz. Estos polígonos contienen a todos los objetos que serán sombreados, por lo cual estas secciones serán invisibles, es decir, no serán considerados a la hora de crear la imagen final, solo se considerarán su intercepción con los polígonos que se encuentren en el camino. Este método es el más complejo de los tres, ya que los cálculos y la cantidad de memoria requerida es mucho mayor que los dos anteriores, debido a que es necesario crear los polígonos a partir de una superficie y calcular su proyección, sin considerar que se necesita encontrar, además, la intercepción de estos con los demás objetos.

3.2. Implementación de las sombras duras

3.2.1. Mapping Shadows

La primera vez que se habló sobre el método de mapping shadows para su utilización en sombras fue en el año 1978 por Williams [2], quien en ese entonces propuso que se podían crear arbitraria y rápidamente sombras usando algoritmos basados en z-buffer, la idea básica era usar la escena desde el punto de vista de la luz, lo que se realizaría comparando el z-buffer desde el punto de vista de la luz y el z-depth. El z-buffer es lo que se llama shadow map, el cual se obtiene al calcular la profundidad de los diferentes puntos de una escena, pintando en una imagen la profundidad en vez del color tal como se ve en la figura 3.2 y este puede ser utilizado como textura aplicándose sobre la superficie de cualquier objeto, pero solo es usado para el cálculo de la sombra; las sombras se reconocen cuando el z-depth, el cual es la distancia real entre la fuente de luz y el punto que se quiere comparar, es mayor que el z-buffer de ese mismo punto, ya que estos puntos tienen valores diferentes cuando existe un objeto que se interpone, el z-buffer del shadow map muestra la distancia de la luz al objeto que se interpone, mientras que el z-depth muestra la distancia de la luz al punto

que se quiere averiguar si es luz o sombra.

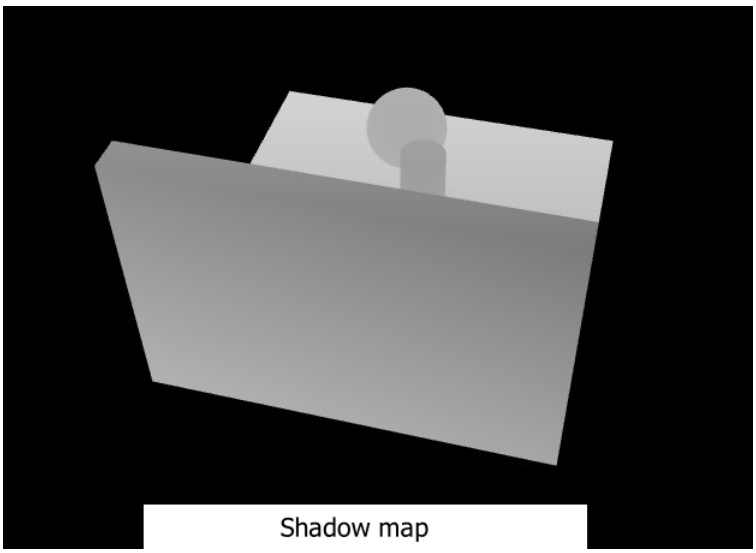


Fig. 3.2: Fuente: Elaboración propia

La primera implementación real se dio en el año 1992 por Segal et al. [3], quienes utilizaron la tecnología de texture mapping, la cual consistía en renderizar la escena desde el punto de vista de la luz, para luego cargar los valores de la profundidad en un shadow map. Lo siguiente consistía en proyectar la textura sobre la escena que es vista desde el punto de vista de la cámara, con lo que se procede a comparar la distancia del pixel depth con la obtenida del shadow map, como se observa en la figura 3.3. Esto muestra qué zonas son sombras y cuáles no, para coordinar el sistema de luz y adjudicarle un alpha a cada pixel: al pixel se le dará valor 1 si no existe una sombra en él y 0 si es que la hay, resultando ser el color de un pixel la suma del color del objeto más la luz ambiental más el efecto de la luz particular sobre el objeto multiplicado por alpha.

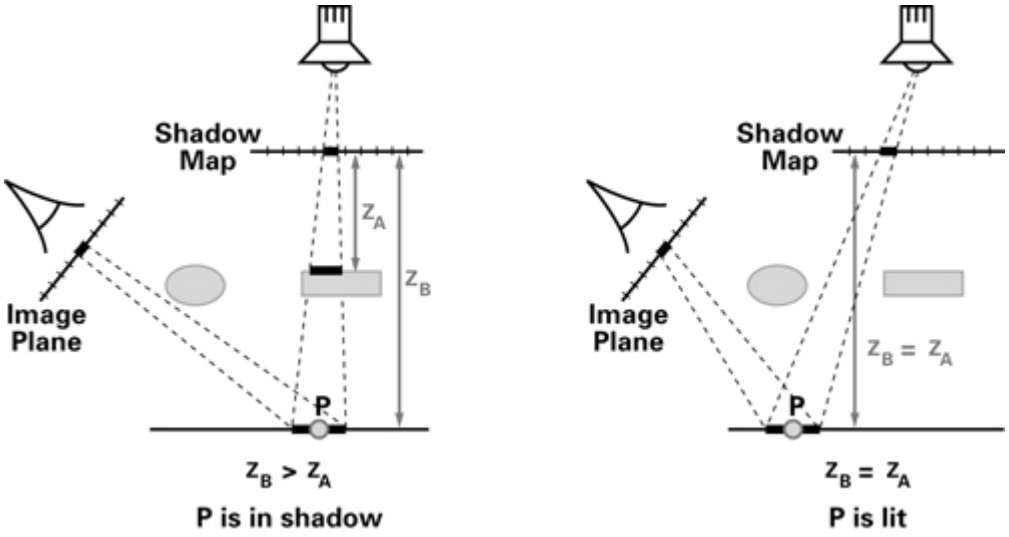


Fig. 3.3: Fuente: The Cg Tutorial, Chapter 9: Advanced Topics [4]

La calidad de las sombras puede ajustarse con la resolución del shadow map, sin embargo, el gran problema de esta técnica es el aliasing, el cual se puede notar con bloques de sombras o sombras como rayas, tal como si fuera una área achurada. Una de las soluciones es crear una pequeña región de lo que se considera plano cercano y no un punto como se suele utilizar, lo que le da más libertad al algoritmo para encontrar las sombras; además, esto hace más efectiva la resolución del z-buffer, pero limita la posición de la fuente de la luz. Este método sufre 3 tipos de aliasing, siendo el primero el aliasing de perspectiva, éste se produce por un shadow map de baja resolución. Al estar la superficie en un gran ángulo respecto a la fuente de luz principal, la comparación realizada entre la distancia de la posición del punto en el shadow map crece demasiado con respecto a la posición real del pixel, al aplicarse el pixel depth de un punto con el shadow map, no se puede encontrar una sombra, porque no se encuentra donde deberia estar. En segundo lugar se encuentra el aliasing de proyección, el que ocurre cuando la luz es paralela al oclusor, provocando que se creen sombras paralelas sobre la superficie. Finalmente, cuenta con el aliasing de la falta de profundidad, el cual se debe a la discontinuidad del muestreo o a falsas selfshadows.

3.2.2. Shadow Volume

Shadow Volume es otro de los métodos utilizados para la elaboración de sombras de tiempo real. estas sombras son formadas a partir de una superficie que se genera detrás de un oclusor, creándose un espacio por la relación del oclusor y la fuente de luz. La primera vez que se presentó este algoritmo fue en el año 1977[1], sin embargo, fue en el año 1991 que se implementó por primera vez por Heidmann [5], contando con dos pasos. El primero consiste en construir varios polígonos que se forman a partir de la extensión de la silueta del oclusor, tomando a la fuente de luz como punto de referencia. La partida de la proyección de la superficie sería en los bordes del oclusor y se continúa en dirección opuesta a la fuente de luz; luego de que todos los polígonos sean calculados, se localizan los objetos que están dentro de estos volúmenes. El segundo paso es la implementación del uso del stencil buffer; primero se verifican qué objetos son visibles en la escena y luego se sigue el camino hasta el pixel. Se le suma 1 si el rayo lanzado entra en una de estas superficies y se le resta 1 si el rayo sale de la superficie. Si al llegar al pixel el número de buffer es mayor o igual a 1, la zona en que se encuentra dicho pixel es una sombra; por el contrario, si el buffer es igual a 0, la zona se encuentra iluminada.

El algoritmo de shadow volume está basado en la geometría, ya que las superficies que se crean están formadas de polígonos, ya que aunque estos sean invisibles para la cámara, igualmente son manejados por el hardware gráfico, por lo cual este hardware es el que se lleva la mayor parte del trabajo en este algoritmo.

Existe un costo por crear los polígonos para utilizarlos en el algoritmo, pues son bastantes los recursos utilizados tanto en memoria como en poder de procesamiento, por lo cual es necesario bajar lo mayor posible la cantidad de superficies que se utilizarán para ganar rendimiento. El número de shadow volume puede ser reducido usando una estructura de datos llamada BSP (binary-space-partitioning) o árboles de

pre procesamiento. El árbol BSP fue propuesto por primera vez por Fuchs, Kedem y Naylor en el año 1980 [6] y consiste en la separación recursiva del espacio a través de híper planos que van almacenando objetos en su interior mientras van dividiendo el espacio a la mitad, lo que sirve para tener una posición relativa entre los diferentes objetos. En el año 1989 Chin y Feiner [7] proponen un nuevo método que se llamará árbol SVBSP (shadow volumen BSP), el cual sirve para construir polígonos a través de BSP tree, de esta manera es fácil saber cuál es el orden de los polígonos que se encuentran en participación con la luz y se puede determinar la participación de cada uno en la sombra para así crear su shadow volume. En el año 1999, Batagelo y Junior[8] minimizaron el el número de polígonos a crear, pre calculando el BSP y ganando así velocidad. Finalmente Liu y Wu en el año 2000 [9] presentan un algoritmo en tiempo real para la generación de sombras en 3D para fuentes de luz puntuales basado en el algoritmo SVBSP. Ellos pre procesaron la escena con el tiling method y determinaron la unión de los polígonos relevantes de los shadow volumes, creando las sombras.

McCool [10] en el año 2000 propuso que las siluetas de los shadow volumes puedan ser calculadas a partir de un mapa de profundidad para escenas más complejas. Este método logra reducir la cantidad de shadow volumes que se generarán, pero requiere que la GPU lea la memoria del CPU, lo cual es bastante costoso. Seidel [11] reporta en el año 2003 una forma para implementar ese método, pero dependiendo más de la aceleración por hardware, por lo cual su algoritmo buscaba calcular la silueta de los oclusores a través de hardware gráfico, pero debiendo leer aún el buffer de la CPU.

Después de que se definen todos los shadow volumes, es necesario revisar qué puntos se encuentran dentro de estos polígonos, sin embargo, definir esto puede resultar tedioso y toma mucho tiempo en realizarse, pero el stencil buffer, junto con el depth testing, pueden ayudar a solucionar este problema. Heidmann en el año 1991 [5] propone un método para aplicar las sombras, al cual llamará z-pass. Lo primero

que realiza es limpiar el stencil buffer y activar el depth test, y luego se renderiza toda la escena solo con la luz ambiental y las luces puntuales que haya en el escenario, obteniendo los objetos iluminados para ingresarlos al color buffer y además tener la información de profundidad del z-buffer. En seguida, es necesario actualizar el z-buffer y calcular los polígonos de los shadow volumes; durante este proceso, el valor del stencil buffer es incrementado cada vez que la cara frontal de un polígono pasa el test depthing y se disminuye cada vez que la cara trasera del polígono falla el test depthing, dibujándose finalmente solo con la luz direccional todos aquellos puntos donde el stencil buffer tenga valor igual a 0, ya que el 0 indica que el z-buffer nunca atravesó un polígono, o que bien entró y salió las suficientes veces como para que no se encuentre dentro de uno.

El algoritmo z-pass tenía dos errores; el primero es que no era capaz de encontrar una sombra cuando la cámara se encuentra dentro de una, ya que el conteo del stencil buffer es 0 cuando se encuentra con una sombra, puesto que se produce un conteo irregular, debido a que contará mal las superficies. Normalmente el buffer no debería dar números negativos, pero en este caso sí los da, perdiendo la noción de cuáles pixeles son realmente una sombra. El segundo error es que si hay objetos demasiado cerca el algoritmo no logra vislumbrar que estos participen en la sombra. Para resolver estos errores se propone un nuevo método que se llamará z-fail, el cual fue propuesto por Bilodeau [12] y Carmack[13]. Este método funciona extendiendo la lectura de los tests, ya que el nuevo test del stencil buffer comenzará desde un punto ubicado en el infinito, aunque el vector será el mismo de la cámara, solo que su punto de inicio se proyectará para comenzar desde otro punto en el espacio. Hacer la lectura desde el infinito permite no estar dentro de una sombra al comenzarlo, evitando cometer el primer error; además, por la misma razón de que el test comience desde el infinito, no habría problemas con los planos cercanos, y por lo tanto el segundo error tampoco se comete.

3.3. Sombras Difusas

En el desarrollo de las sombras difusas se han propuestos implementaciones utilizando los métodos de mapping shadow y shadow volume, los cuales se revisarán en esta sección. Del mismo modo, se verán las técnicas que se utilizan actualmente para la representación de estas sombras.

3.3.1. Mapping Shadows

Combinación de muchas fuentes puntuales de luz

Este es el método más simple de todos, ya que es muy similar al algoritmo base de mapping shadow, pero con la diferencia de que utiliza más de un shadow map. Estos shadow maps extras provienen de una serie de puntos repartidos por la fuente de luz como se ve en la figura 3.4, debido a que la fuente de luz tiene un volumen, por lo cual se pueden repartir estos puntos por toda la superficie, creandose una sombra para cada uno de estos shadow maps, para finalmente sumarlos en una gran sombra difusa. Cada imagen se creará con una fracción de la intensidad de la luz original, la cual será igual a la intensidad de luz total dividida por la cantidad de puntos que se utilizará; por ejemplo, si todas las fuente de luz puntuales que están repartidas por la superficie de la fuente de luz están iluminando una sección de la superficie, la intensidad de la luz que recibe es igual a la intensidad total de la fuente de luz. Para aplicar este método se asume lo siguiente: la distribución de la intensidad de luz en toda la superficie será constante tanto en el área como en el tiempo, por lo cual no debería haber problemas repartiendo los puntos que se utilizarán para los shadow maps, ya que estos están distribuidos homogéneamente en toda la superficie.

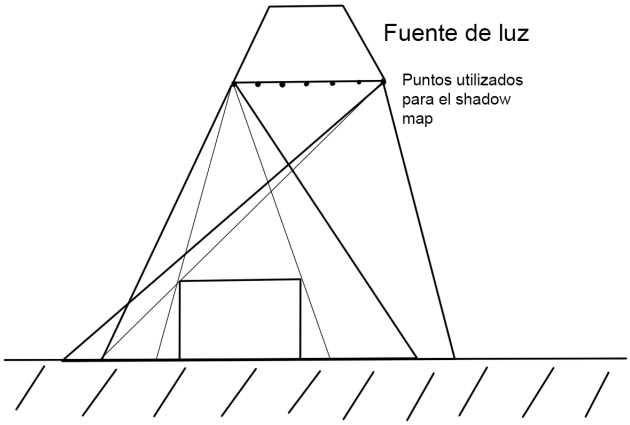


Fig. 3.4: Fuente: Elaboración propia

Todo comienza con un mapa binario para cada shadow map creado, el que solo contendrá los valores 1 y 0, con lo cual se irá trabajando cada pixel dándole un cierto valor con la información que dé la suma de los buffers auxiliares dividido por la cantidad de puntos creados. Cabe destacar que entre mayor sea el número de shadow maps, mejor será la sombra difusa, ya que los puntos tomados de la fuente de luz estarán mejor repartidos. Durante el proceso se perderá la noción de una sola sombra e irá apareciendo el patrón de la sombra difusa. Por otra parte. el aumento en el cálculo de datos será significativo dependiendo de la cantidad de shadow maps extras creados ya que será necesario calcular individualmente cada shadow map; si su número crece demasiado, será necesario mayor procesamiento y más memoria gráfica para almacenar los shadow maps. Finalmente, cada vez que se revise un pixel, el algoritmo calculará el porcentaje de sombra utilizando todas las muestras creadas.

Mapas de atenuación por capas

Mapas de atenuación por capas está basado en el método modified layered depth image [14], el cual es una extensión del método anterior, y consiste en crear mapas de sombras para cada polígono en la escena a partir de las muestras tomadas.

El método consiste en utilizar varios shadow maps provenientes de puntos arbitrarios ubicados en la superficie de la fuente de luz, luego se calcula la dirección de la normal de la superficie con respecto a la dirección de la luz, se toman los shadow maps y se unifican tomando como referencia el punto central de la fuente de luz. El resto del algoritmo es similar al de mapping shadow, con la diferencia de que se crean varios shadow maps. Finalmente, al revisar si el objeto es visto desde cada uno de ellos, se suman los resultados de los shadow maps, calculando el porcentaje de luz que recibe. Se debe considerar, al mismo tiempo, la distancia de la muestra de luz y la dirección de la normal, ya que cada muestra tiene una importancia dentro de la escena. Una vez listo este mapa con las sombras de cada polígono y con los porcentajes de luz que tiene cada punto, se aplica en primera instancia la luz ambiental y luego la luz direccional multiplicado por el porcentaje de luz que recibe dicho punto.

Las ventajas de este método son, entre otras, el ahorro de memoria, ya que no almacena todos los mapas creados, sino que va almacenando la información final de los porcentajes incrementalmente a medida que los va creando, y luego los desecha. También mejora el aspecto de las sombras, ya que la luz no funciona como un punto uniforme, sino que tienen más características con respecto a la naturaleza de interacción de la luz con la superficie que la recibe.

Uso avanzado de shadow map

Este método es una extensión de la técnica de shadow mapping, ya que a partir de los mapas de profundidad se puede obtener más información de las sombras, dando como resultado los porcentajes de luz con que se ilumina cada zona. Sin embargo, este método solo trabaja con luces lineales, que es la forma simple de llamar a una fuente de luz cuya forma es extremadamente delgada y la luz se distribuye en lo que parece una línea, aunque es posible implementarlo en casos de luces poligonales.

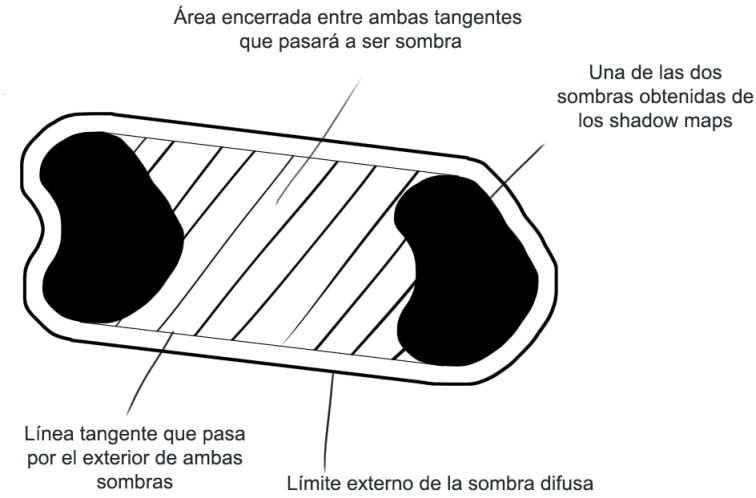


Fig. 3.5: Fuente: Elaboración propia

Para su ejecución se necesitan pocas shadow maps para generar las sombras difusas, ya que al limitarse a luces de forma lineal, solo se necesitan dos shadow maps para llevarla a cabo, uno para cada vértice de la fuente de luz lineal. Para cada shadow map obtenido de las fuentes de luz, se buscarán las discontinuidades de las sombras; en un principio deberían haber dos sombras generadas de los dos shadow maps, las que pueden unirse, si se crean dos líneas que sean tangentes a sus bordes sin que estas pasen por dentro de las sombras. Todo lo que esté al interior de las dos líneas se considerará sombra, tal como se ve en la figura 3.5. A continuación, se le asignará valor 0 a todo lo que esté dentro de la sombras y de las tangentes, mientras que se le asignará valor 1 a toda la región que se encuentre alejada a una distancia mayor del tamaño del ancho de la sombra difusa. La distancia que se tome será el largo de la normal de la superficie que se haya considerado sombra anteriormente, mientras que a los puntos que no se encuentren dentro de la zona de sombra ni de la zona de luz a la que se le ha asignado el valor de 1, se le dará un porcentaje que representará la intensidad de luz que recibe,

lo que se calculará por la distancia en la que se encuentre alejado de la zona de sombra dividido por el ancho de la sombra difusa. Como se mencionó anteriormente, solo se consideran las fuentes de luces lineales, porque la atenuación de la sombra es lineal, lo que ocurre debido a que la cantidad de luz que pasa a ser vista al otro lado aumenta gradualmente a medida que se va mostrando más, resultando en una fórmula lineal, tal como se ve en la figura 3.6. una vez obtenidos los porcentajes de la luz, se dibuja la sombra, mientras que para la zona de sombra difusa se utilizará el porcentaje obtenido para aplicarle la intensidad de luz en la fórmula que se esté utilizando para darle color al pixel, por ejemplo en P_1 el porcentaje de luz es de 100 %, mientras que en P_2 el porcentaje de luz es de 0 %.

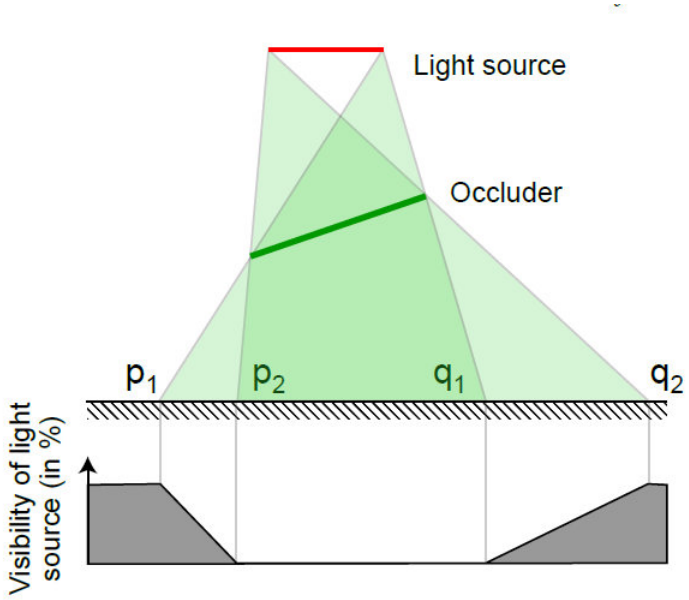


Fig. 3.6: Fuente: A Survey of Real-time Soft Shadows Algorithms [15]

Gracias a que solo utiliza dos shadow maps, este algoritmo es bastante liviano, ya que en los dos métodos anteriores son necesarias mas shadow maps para lograr un buen resultado. Sin embargo, la desventaja es que se limita a fuentes de luz lineales y no poligonales, por lo que se podría hacer una modificación al algoritmo, pero éste se volvería más complejo.

Sombra difusa con un solo shadow map

Sombra difusa con un solo shadow map es introducido por Parker et al. [16] y funciona a partir de un shadow map con el cual generará una sombra difusa, a través del análisis de los pixeles a medida que se renderiza la escena. La imagen obtenida es la escena con sombras duras, por lo que, para poder agregarle sombras difusas, es necesario trabajar a nivel de pixeles sobre ella. Esta imagen tendrá información extra necesaria para modificarla, como por ejemplo si el pixel en cuestión es sombra, la información del oclusor y la profundidad de cada pixel.

En un principio el funcionamiento es muy parecido al de shadow mapping, pues al tener la imagen creada se analizará pixel por pixel; si el pixel analizado da como información que no se encuentra en la sombra, se procederá a buscar en la imagen al pixel más cercano cuya información diga que se encuentra en una sombra y si se encuentra en la sombra, será necesario saber a qué distancia se encuentra ese punto de la luz, esto sirve para verificar primero si es parte de la penumbra y si lo es, se calculará el factor de atenuación sobre la superficie utilizando en la fórmula la distancia del oclusor y de la fuente de luz.

$$f = \frac{dist(Pixel_{oclusor}, Pixel_{receptor})}{RSZ_{receptor}|Z_{receptor} - Z_{oclusor}|}$$

- $Pixel_{oclusor}$: Es el punto mas cercano al borde de la sombra del pixel observado.
- $Pixel_{receptor}$: Es punto que actualmente está observando el pixel.
- RS: Constantes arbitrarias.
- $Z_{receptor}$: Es la profundidad en que se encuentra el pixel observado por la luz.
- $Z_{oclusor}$: Es la profundidad en que se encuentra el objeto que produce la sombra mas cercana al punto que observa el pixel.

En el dividendo de la fórmula aparece la distancia entre el pixel que se está revisando ($Pixel_{receptor}$) y el borde de la sombra, que sería el pixel más cercano que pertenezca a la que pertenezca a ella ($Pixel_{oclusor}$), permitiendo esta distancia saber si el pixel que se está revisando pertenece a la sombra difusa. En el divisor existe una resta a la cual se le aplica un absoluto, considerando la profundidad del pixel que se está revisando ($Z_{receptor}$) y la del ($Z_{oclusor}$). Diferencia que es importante para conocer el tamaño de la sombra difusa, lo cual también se multiplica por la profundidad del punto receptor, R y S, que son valores dados al sistema y pueden corresponder al tamaño de la fuente de luz. Lo más difícil de calcular es el pixel más cercano que corresponda al borde de una sombra, por lo cual se comenzará desde el pixel analizado y se revisará el área que cubra la distancia máxima a la que pueda estar una sombra con la posibilidad de que haya sombra difusa. La fórmula para conocer esto es $R * Z_{receptor}$: todo lo que esté dentro del dominio de esa distancia es analizado y en el caso de que no haya nada, ese pixel se deja como iluminado o sombra, dependiendo si está dentro o fuera. Para finalizar, una vez que se encuentre un pixel del límite de la sombra, se calcula f.

Si se está en una zona iluminada, se calcula el valor de $0{,}5(1 + f)$, el que se encontrará entre los números [0.5 , 1]. Su valor maximo es 1, ya que 1 corresponde a una zona completamente iluminada, y, por lo tanto, no puede haber más luz que la emitida por las fuentes. Por el contrario, si se está en una zona de sombra, se calcula el valor de $0{,}5(1 - f)$, el cual fluctuará entre [0 , 0.5]. Como resultado, el pixel se ilumina dependiendo del valor obtenido en la operación, con valores entre [0 , 1].

Los pixeles se deben encontrar a una distancia 0,5L (donde L es el valor del ancho de la sombra difusa) del pixel que se indicó que pertenecía al oclusor, ya que ese pixel se encuentra al medio de la penumbra, entonces los valores se van asignando de manera lineal, siendo la parte más lenta del proceso encontrar los pixeles vecinos.

3.3.2. Volúmenes de sombra

Combinación de sombras duras

El método de combinación de sombras duras es muy parecido al primer método mencionado en la sección de shadow mapping, pues se necesitan varias muestras para llevarlo a cabo. En este caso se podría traducir en varios polígonos de sombras que correspondan a puntos aleatorios escogidos de la fuente de luz, ya que mientras más puntos se escojan, la luz estará mejor distribuida y es por eso se pueden aplicar diferentes distribuciones para simular distintas formas en que la luz es emitida, por ejemplo, el que la luz no esté uniformemente distribuida sobre la superficie de la fuente de luz, sino que ciertos puntos tengan más intensidad que otros, lo que podría resultar, incluso, en diferentes tonalidades.

Sombras difusas planas usando mesetas

Sombras difusas planas usando mesetas es implementado por Haines [17] donde las sombras difusas se relacionan con una meseta, debido a que se puede aislar a la umbra del resto de la sombra difusa, la cual podrá ser relacionada a una sección completamente plana y se crearían bordes a su alrededor que identificarían a la penumbra. El shadow volume se proyecta sobre la superficie que debe generar la sombra, la cual es creada a partir de un plano con vértices claramente reconocibles; en cada vértice del plano se crean otros polígonos con forma de conos, cuyos anchos dependerán de la distancia de la luz del oclusor y los cuales viajarán junto al volumen de la sombra de la umbra como se ve en la figura 3.7. Una vez que ambos polígonos de sombra se encuentren con la superficie del objeto se generará una umbra por el polígono de sombra y la superficie que intercepta el cono de la penumbra, sin contar al área atrapada por el polígono de la umbra, será parte de la sombra difusa y se unirá a los conos adyacentes a través de una arista que sea tangente a su superficie. Esta unión creará un área, la cual formará la sección exterior de la meseta, produciendo la penumbra de la sombra.

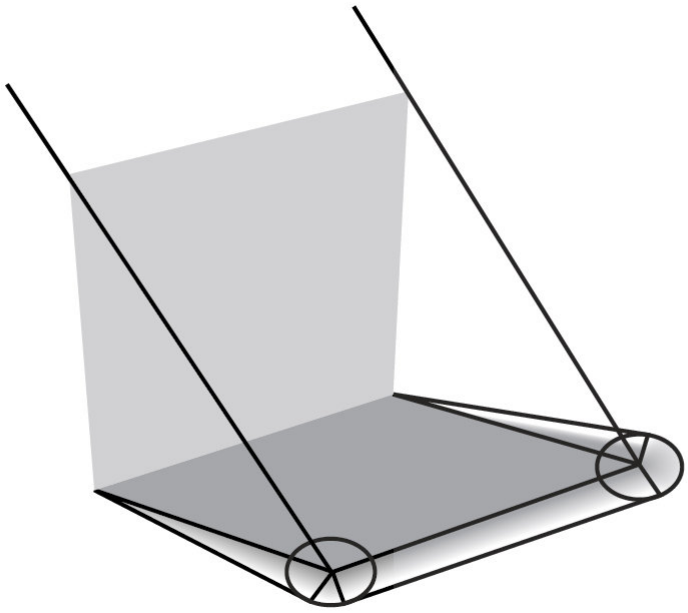


Fig. 3.7: Fuente: A Survey of Real-time Soft Shadows Algorithms[15]

La desventaja de este método es que solo se puede utilizar en planos cuyos vértices sean claramente reconocibles, ya que dificultaría el trabajo si se utilizase en figuras cuyos bordes y vértices sean difíciles de individualizar.

Smoothies

Chan y Durand [18] presentaron este método que trabaja usando a las Smoothies, utilizando shadow volume y mapping shadow. El método de mapping shadow servirá para dibujar la sombra dura sobre el escenario, mientras que el shadow volume usará a las smoothies, que son las superficies planas que están unidas a los bordes del oclusor y son perpendiculares a la superficie. Estas smoothies están para aplicar una sombra difusa usando la distancia de la superficie y el oclusor.

La principal desventaja de este método es que solo considera la penumbra externa[1],

[1] La penumbra se puede dividir en dos partes iguales, llamadas penumbra interna y penumbra externa. Esta división es hecha por los bordes originales de la sombra dura, por lo tanto la penumbra externa corresponde a la mitad que está alejada de la umbra.

por lo que la penumbra interna también pasaría a ser parte de la umbra, aunque la fuente de luz sea muy grande, incluso para eliminar la umbra, por lo tanto la umbra seguirá existiendo.

Soft Shadows Volume

Akenine-Möller y Assarsson [19] desarrollaron Soft Shadows Volume, el cual obtiene la sombra difusa a partir de los bordes de la fuente de luz, debido a que utiliza la proyección de estos para generar la sombra difusa. Primero se crea una sombra dura cuyos bordes se llamarán edge, luego se calculan los bordes con la silueta de la luz creando un segundo borde, llamado wedge, como se puede observar en el figura 3.8.

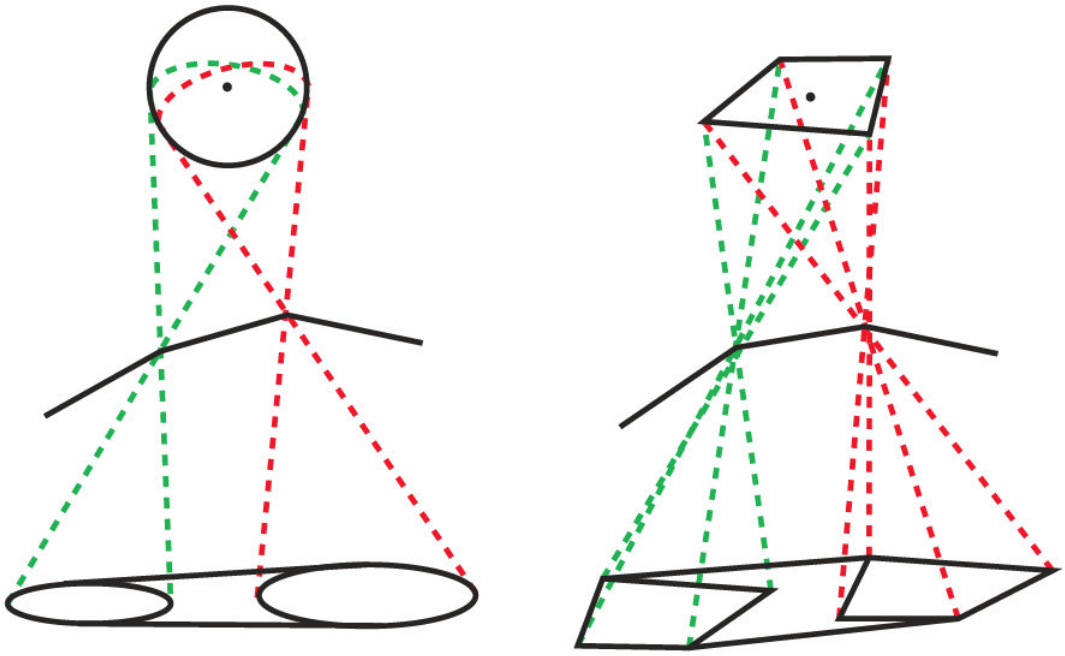


Fig. 3.8: Fuente: A Survey of Real-time Soft Shadows Algorithms[15]

La sombra dura es creada utilizando el método de shadow volume, teniendo un buffer aparte que almacene el porcentaje de sombra para cada pixel, al mismo tiempo que se deben crear algunos algunos triángulos que identifiquen la sombra dura que

se encuentra dentro del edge. Para la sección del wedge es necesario revisar cada pixel caso por caso; para cada uno se calculará el porcentaje de sombra que contiene, sacando una muestra del área que lo rodea, la cual podrá contener una parte de los triángulos creados del edge. el porcentaje del área que estén usando los triángulos se calcula usando la técnica que aparece en la figura 3.9, la que sirve para obtener el porcentaje total de sombra que contiene dicho pixel. Este porcentaje dependerá de si el pixel se encuentra o no dentro del edge; luego será almacenado en el buffer para ser utilizado posteriormente. Es necesario calcular el visibility buffer de los pixeles, los que se encuentran divididos en dos áreas, los pixeles que están dentro del edge y dentro del wedge, a los que se van sumando las áreas de los triángulos visibles y los pixeles que están dentro del wedge pero fuera del edge, para los cuales se calculará su visibilidad realizando la misma suma, pero el resultado se obtendrá restando este valor a 1. La información sobre el porcentaje de sombra que contiene el área revisada se ingresa en el buffer para luego dibujar la escena.

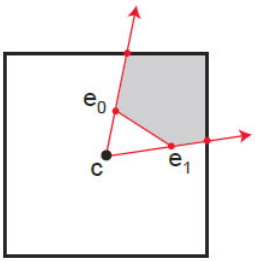


Fig. 3.9: Fuente: A Survey of Real-time Soft Shadows Algorithms[15]

Este método es bastante barato en recursos, ya que utiliza el hardware gráfico para realizar los cálculos de las áreas, lo cual lo hace bastante eficiente comparado con otros métodos basados en CPU. Otra ventaja que posee es su facilidad de optimizar, previniendo varios tipos de escenarios posibles, aunque falla en calcular bien las siluetas de objetos que estén demasiado cerca entre sí.

3.3.3. Técnicas actuales

Las técnicas utilizadas actualmente provienen del método Percentage Closer Filtering (PCF) y poseen, entre otras, la ventaja de permitir crear sombras difusas a partir de una sola shadow map, lo que de partida lo hace bastante más liviano que otras técnicas que utilizan más de un shadow map. Otra ventaja es que la mayor parte del algoritmo funciona en la tarjeta gráfica y en un código ya hecho con shadow mapping no es difícil implementar estos algoritmos, ya que solo hay que modificar el shader que crea la sombra.

Percentage Closer Filtering

La técnica de PCF fue creada por Reeves en 1987 [20]. Esta técnica no buscaba crear sombras difusas, su meta era buscar una solución al aliasing de las sombras, por lo cual necesitaba difuminar las sombras de forma uniforme, por esto es que todas las sombras a las cuales se le haya aplicado este método tienen el mismo tamaño de penumbra, a diferencia de una sombra difusa real cuya penumbra depende de varios factores.

Esta técnica consiste en tomar una muestra de un conjunto de pixeles cuyo centro es el pixel que se observa y a la que se llamará kernel, como se ve en la figura 3.10. En la muestra se observa a los pixeles como 1 y 0, lo que significa si tiene luz o no, entonces se toma el kernel con todos estos pixeles y se saca el promedio con lo cual se podrá obtener un porcentaje de luz. La principal característica de esta técnica es que el kernel tiene tamaño fijo, por ejemplo podría ser de tamaño 5x5 pixeles, por esta razón la penumbra tiene tamaño constante.

0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	1	1	1
0	0	0	0	0	1	1	1	1
0	0	0	0	0	1	1	1	1
0	0	0	0	1	1	1	1	1
0	0	0	0	1	1	1	1	1
1	1	1	1	1	1	1	1	1

Fig. 3.10: Fuente: GPU Gems, capitulo 11. Shadow Map Antialiasing [21]

Percentage Closer Soft Shadow (PCSS)

La técnica PCSS es creada por Fernando [22] y es un derivado de la técnica de PCF, con la diferencia de que el tamaño del kernel es variable, ya que dependerá de la distancia de la fuente de luz al oclusor y del receptor, como también del tamaño de la fuente de luz. La forma en que se aplica este método se observa en la figura 3.11.

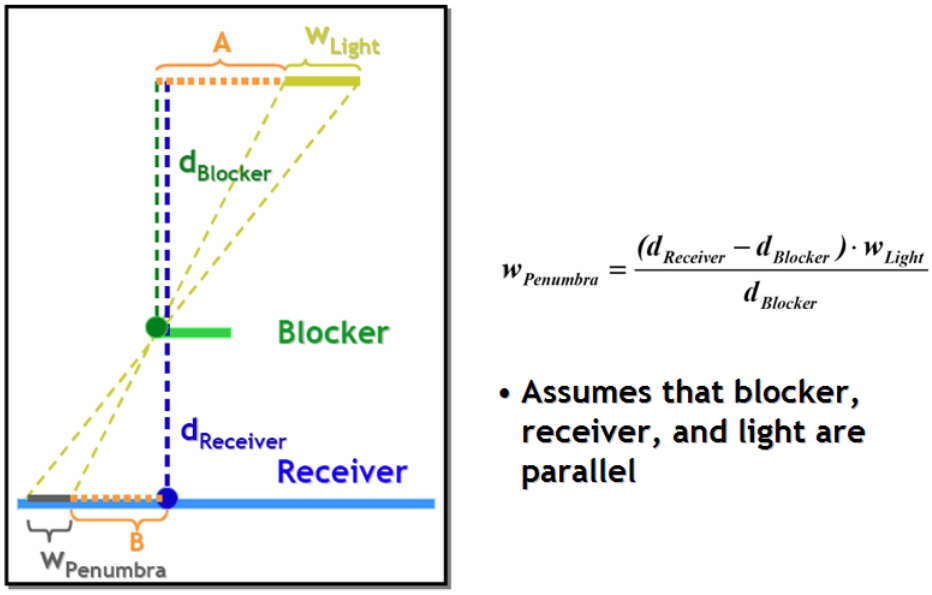


Fig. 3.11: Fuente: Percentage-Closer Soft Shadows. ACM SIGGRAPH 2005 Sketches[22]

En sí el algoritmo asume que las superficies son paralelas entre sí, como una manera de mantener al algoritmo liviano, ya que trabajar con más variables aumentaría considerablemente el tiempo que tomaría procesarlo.

Contact Hardening Shadow (CHS)

El método Contact Hardening Shadow nace a partir de la investigación de Wyman y Hansen en el año 2003 [23], en la cual se implementaron sombras difusas a través de un shadow map, el que es creado a partir un shadow map inicial al cual se calcula la penumbra desde la geometría. Para finalizar, se genera un nuevo buffer en el cual se indica dónde se encuentra la penumbra que se obtiene a partir del cálculo sobre la imagen.

El método es desarrollado finalmente por Klein, Nitschwitz y Obermeier [24] y deriva de los métodos de PCF y PCSS utilizando como base central los mapas de penumbra, aunque su forma de generarlo es diferente, ya que aprovecha de utilizar los shaders que hacen un mejor trabajo para esto. El primer paso consiste en generar un shadow map, el cual servirá para crear sombras duras que se utilizarán para llenar un buffer binario con 0 y 1, donde 0 es la sombra y 1 las zonas con luz. a este mapa se le multiplica la siguiente matriz:

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

Esta matriz sirve para encontrar los bordes de las sombras, es decir, en la matriz resultante, los bordes serán los 0. Luego, se aplicará la técnica de erosión a la imagen resultante, la cual quita 1 de una imagen binaria pero lo hace de manera uniforme por los bordes, como se observa en la imagen 3.12 y puesto que los bordes son 0, el borde de la penumbra debería crecer de manera uniforme con la erosión. Finalizando el proceso se aplica el PCF, utilizando el mapa de penumbra creado y es en este momento en donde se debería utilizar un algoritmo similar al PCSS, aunque solo limitándose a los sectores donde exista penumbra según el mapa de penumbra.

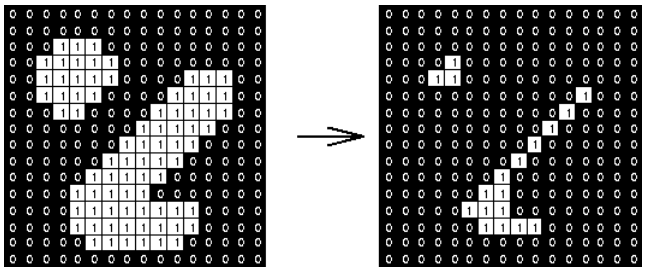


Fig. 3.12: Fuente: 20th International Conferenceon Computer Graphics,Visualization and Computer Vision 2012[24]

4. IMPLEMENTACIÓN DE LAS SOMBRAS

4.1. Plataforma a usar

4.1.1. API Gráfica

Las APIs gráficas son librerías que contienen una serie de funciones que son capaces de trabajar con un ambiente gráfico que puede ser visto posteriormente y pueden funcionar tanto en ambientes 2D como en 3D. Estas librerías nacen de la necesidad de contar con herramientas que faciliten el trabajo con funciones gráficas, ya que antiguamente solo se podía trabajar con estas funciones si se programaba directamente sobre el hardware, teniendo que hacer una implementación diferente para cada hardware existente, lo que aumentaba considerablemente la dificultad para realizar un programa. Como solución a esta dificultad aparecen las APIs, que hacen transparente la comunicación del programa con el hardware gráfico, obviando el hardware específico con que se cuenta, ya que una API traduce el lenguaje del programa al lenguaje del hardware sobre el cual se está corriendo, haciendo mucho más fácil programar aplicaciones con funciones gráficas.

Hoy en día, existen dos grandes APIs gráficas que han sobrevivido y evolucionado, siendo posible trabajar con gráficos en tiempo real con buenos resultados, estas APIs son:

OpenGL[25]: Esta API gráfica está presente desde 1997, año en que apareció su primera versión. OpenGL hoy en día se encuentra en su versión 4.5, la cual es capaz de trabajar gráficos 3D a varios niveles, es capaz de generar polígonos,

darle texturas, trabajar con la luz, entre otras cosas, pero solo se remite al área gráfica, ya que si se desea darle más características como audio o entradas físicas como mouse o joystick, es necesario utilizar librerías externas a ésta para que cumpla esos roles. Una de sus ventajas es que es multiplataforma, lo que significa que está presente en varias plataformas, desde sistemas operativos de escritorio hasta los smartphones.

Direct3D: Esta API gráfica es capaz de crear y manejar ambientes gráficos y se considera ser mucho más completa que la anterior, ya que de base viene acompañada de otras librerías que son capaces de manejar interfaces humanas, audio y redes. Esta API no tiene soporte para multiplataforma, ya que fue creada y es mantenida por Microsoft, para su sistema operativo Windows.

Por lo tanto, dadas las características señaladas sobre las dos APIs, la elección realizada para esta memoria será OpenGL. En primer lugar, por su característica multiplataforma, ya que las soluciones creadas en OpenGL pueden ser llevadas a cualquier plataforma que éste soporte. En segundo lugar, debido a la cantidad de ejemplos existentes, ya que esta API cuenta con una buena cantidad de documentación, entre la que se encuentra un código de sombras duras, lo cual facilita bastante la tarea.

4.1.2. Librerías extras

OpenGL con el tiempo ha ido adquiriendo librerías externas, lo cual ha ayudado a ser una herramienta cada vez más potente, sin embargo aún le faltan ciertas funciones, carencias que debe suplir con librerías capaces de realizar dichas funciones.

GLEW

OpenGL Extension Wranger Library (GLEW) [26] es una librería open source y multiplataforma de extensión de carga en C/C++, la cual provee de un mecanismo eficiente para la rutina de OpenGL y está hecha para dar eficiencia dependiendo de la plataforma en la cual se le indique debe funcionar. Para usar esta librería, solo hay que

cambiar el header, es decir, no se utiliza

#include <GL/gl.h>

Sino

#include <GL/glew.h>

GLFW

GLFW es una librería [27] open source y multiplataforma, que tiene como función la creación de ventanas para OpenGL y el manejo de los inputs, y en la cual los eventos son fáciles de integrar. El programa no corre dentro de un bucle general, y dentro de sus principales características se encuentran el manejo de múltiples pantallas, el soporte de teclado, mouse, gamepad, y que se puede utilizar en varios lenguajes de programación. Finalmente, una de sus principales ventajas es que vienen ejemplos, tutoriales y guías para su uso dentro de la página web de GLFW[27].

GLM

OpenGL Mathematics (GLM) [28] es una librería para OpenGL encargada del manejo de matrices. Las sombras no vienen incluidas en OpenGL, por lo que para poder implementarlas es necesario programar las funciones que sean capaces de pintarlas, siendo necesario para este proceso el uso de muchos cálculos sobre matrices. Esta librería tiene la capacidad de trabajar con cuaterniones, vectores de 4 elementos, con los cuales se pueden hacer rotaciones, transformaciones y traslaciones, entre otras operaciones sobre vectores.

Se pueden usar vectores de diferentes dimensiones, por ejemplo, para un vector de 3 dimensiones, se usaría esta iniciación de variable glm::vec3. También existen las matrices, como glm::mat4, por lo que multiplicar un vector con una matriz es tan fácil como hacer esto:

```
glm::vec4 vector4d = glm::vec4(1,0,0,0);
glm::mat4 matriz4d = glm::mat4(1.0);//Es una matriz identidad
glm::vec4 vectormulti = matriz4d * vector4d;
```

ASSIMP

ASSIMP [29] es una librería que ayuda a importar modelos de 3 dimensiones de manera sencilla y ordenada. Los modelos de 3 dimensiones son difíciles de abrir, ya que además de la información de los puntos de los polígonos que traen, también contienen otro tipo de información, como la dirección de las normales y la forma en que se le aplican las texturas. Es debido a eso que ASSIMP es una librería que ayuda a importar estos modelos de una manera sencilla y ordenada.

GLUT

OpenGL Utility Toolkit (GLUT) [30] es una librería que facilita la implementación de ventanas y que cuenta con la posibilidad de utilizar interfaces usuarias, como el uso de mouse, teclado y gamepad, contando también con soporte para multiplataforma y multilenguaje. La gran desventaja es que fue descontinuada en el año 2000, pero existen soluciones libres actualizadas que se le parecen bastante. freeGLUT [31] es una de ellas; la original era de código cerrado y contaba con licencias que no permitía que se siguiera desarrollando a partir de ella, por lo cual se debió realizar esta versión desde cero, pero conservando la forma en que se trabajaba en GLUT.

4.1.3. Shaders

En OpenGL viene una librería llamada OpenGL Shading Language (GLSL), la cual contiene la implementación de los shaders, facilitando la programación de efectos de color y luz en cada pixel y son altamente paralelizables. Los shaders son programas auxiliares que funcionan en comunión con el programa de OpenGL y lo hacen

cada vez que éste lo necesita. Existen diferentes tipos de shaders que se pueden implementar, entre los que se encuentran vertex shader, fragment shader, geometry shader, tesellation shader, etc, pero para el desarrollo de esta memoria solo se necesitarán los vertex, fragment y geometry shaders, ya que son los shaders que participan en la implementación de las sombras.

Durante la ejecución de un ciclo de programa, el flujo del pipeline que genera la escena sigue el camino que se ve en la figura 4.1:

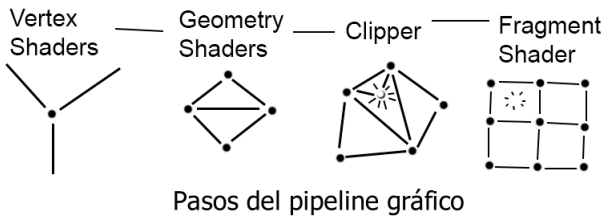


Fig. 4.1: Fuente: Elaboración propia

Es en el vertex processor en donde los vértices de todos los objetos son tratados individualmente por el shader, recibiendo varias transformaciones, para luego salir del proceso. Esta etapa no conoce la forma exacta del objeto y tampoco cambia la cantidad de vértices.

El geometry processor es capaz de comprender formas, ya que se le pueden pasar grupos de vértices para que pueda realizar cambios sobre ellos, teniendo la capacidad de crear nuevos vértices.

El clipper es una etapa automática, que elimina los vértices que no van a ser vistos según la cámara, mientras que aquellos que sí se verán se preparan para ser renderizados.

El fragment shader se encarga de recalcular los colores de cada pixel utilizando

los datos del vértice.

El único paso que por obligación debe existir es el del Clipper, siendo opcional implementar los demás, aunque si se implementa algunos de los shaders, es posible que se necesite implementar el vertex shader, para obtener información de los vértices sobre los cuales se trabajará.

Vertex Shader

Los objetos dentro de una escena están compuestos de polígonos, los que a su vez contienen vértices. Es así como el vertex shader solo conoce las características del vértice con el que está trabajando, a las cuales se pueden agregar datos de entrada que vienen del programa principal, como por ejemplo, información sobre la posición, las normales o diferentes tipos de transformaciones. Dentro del programa se pueden calcular más características para el vértice, las cuales pueden agregarse en el output, para que sean utilizados en algún proceso posterior en el pipeline.

Geometry Shader

El geometry shader puede trabajar con las primitivas geométricas, de hecho puede trabajar sin ellas o incluso con más de una, lo que le permite tener una idea de la realidad local de la figura. también puede crear nuevos vértices, nuevas primitivas, dibujar un vértice, una arista o un triángulo.

Fragment Shader

El fragment shader trabaja sobre los pixeles visibles gracias a la información que se ha obtenido a través del pipeline o información que le ha pasado el programa principal. Su trabajo consiste en entregarle un color a un pixel, incluso asignándole más de un color para obtener un mejor resultado, lo que depende de la cantidad de muestras que se asignen inicialmente, las cuales también sirven para aplicar técnicas de

antialiasing. A medida de que se van ejecutando más muestras, estas trabajan sobre la asignación de un punto exacto en las coordenadas de los pixeles en una pantalla. La coordenada central de un pixel cualquiera sería (x.5,y.5), ya que los números enteros pertenecerían a los bordes del pixel, pero si se aplicaran más muestras, las coordenadas que se usan variarían y se podría encontrar algo como (x.3,y.6). Los diferentes valores de color que se obtengan en las muestras se unirán de alguna forma para generar un color final. Para finalizar, cada iteración de este shader tiene un output de color, el cual se usará para pintar dicho pixel.

4.2. Implementación

4.2.1. Básico

Antes de entrar en la implementación de las sombras difusas en OpenGL, es necesario echar un vistazo a los otros elementos que participan en su elaboración, para así saber qué papel cumplen en ella.

Estas son las bibliotecas necesarias para la implementación:

```
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <GL/glew.h>
#include <glfw3.h>
```

Todo lo que sigue, irá dentro de la función main del programa, a menos que se explicite lo contrario.

Para comenzar se iniciará el GLFW con glfwInit(). Esta función iniciará esa librería que se encarga de las ventanas y las interfaces humanas:

```
if( !glfwInit() )
{
        fprintf( stderr, "Failed to initialize GLFW\n" );
        return -1;
}
glfwWindowHint(GLFW_SAMPLES, 4);
```

Esta característica está relacionada con la cantidad de muestras que tendrá el multisample, es decir, el número de veces que se ejecutan los fragment shaders para cada pixel, lo que resulta útil para resolver el problema de aliasing.

```
glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
```

En este punto se configura la versión de GLFW que se utilizará, las que pueden ser varias, para estar acorde a la versión de OpenGL que tiene el sistema.

```
glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE
    );
```

Se le dirá a GLFW en qué contexto de OpenGL se encuentra, para que se prepare para ello.

```
GLFWwindow* window;
```

```
window = glfwCreateWindow( 1024, 768, "Nombre de la ventana"
    , NULL, NULL);
```

Se debe crear la variable window como una variable global, ya que es posible que se use en funciones fuera de la función main. En esta representación se crea la ventana, se le da el tamaño en pixeles y un nombre, mientras que la siguiente variable sirve para decirle al programa de qué forma abrirá esta ventana, si será fullscreen u otro tipo de ventana. Si dicha variable se deja null, será una ventana de tamaño fijo por la cantidad de pixeles que se le ha dado.

```
glfwMakeContextCurrent(window);
```

Esta función registra el contexto actual, este contexto es para un solo thread y cada thread puede tener un solo contexto; la ejecución del programa se realizará sobre esa ventana.

Ahora es necesario iniciar otra librería, GLEW, a través de la función glewInit(), que gestiona la rutina de OpenGL:

```
if (glewInit() != GLEW_OK) {
        fprintf(stderr, "Failed to initialize GLEW\n");
        return -1;
}
glfwSetInputMode(window, GLFW_STICKY_KEYS, GL_TRUE);
glfwSetCursorPos(window, 1024/2, 768/2);
```

A la ventana se le configuran los dispositivos de entrada, que puede ser mouse o teclado, y dónde se encontrará el centro del mouse en la ventana creada.

```
glClearColor(0.27f, 0.57f, 0.8f, 0.0f);
```

La escena tendrá un color base, el cual está dado a través de esta función. Los valores van desde 0 hasta 1, representando los valores de rojo, verde, azul y alpha.

Durante el inicio del programa es necesario compilar e inicializar los shaders, debido a que los shaders son programas separados y que es más eficiente que el computador que los vaya a utilizar, los compile para que sean compatibles con la tarjeta gráfica que tenga, ya que cada fabricante de tarjetas gráficas trae dentro de los drivers el compilador y cómo se ejecutan los shaders. Una vez compilados los shaders, es necesario linkearlos a las variables para su uso posterior.

```
GLuint VertexShaderID = glCreateShader(GL_VERTEX_SHADER);
GLuint FragmentShaderID = glCreateShader(GL_FRAGMENT_SHADER)
    ;
```

Los vertex y fragment shaders se leen y compilan de forma similar, por lo cual los siguientes pasos pueden ser repetidos para ambos casos y al final se juntan en un

puntero.

```
std::string VertexShaderCode;
        std::ifstream VertexShaderStream(vertex_file_path, std::ios
            ::in);
        if(VertexShaderStream.is_open()){
                std::string Line = "";
                while(getline(VertexShaderStream, Line))
                        VertexShaderCode += "\n" + Line;
                VertexShaderStream.close();
        }
        char const * VertexSourcePointer = VertexShaderCode.c_str();
```

El código fuente de los shaders se carga en un string, el que se compilará en las funciones del GLSL.

```
        glShaderSource(VertexShaderID, 1, &VertexSourcePointer ,
            NULL);
        glCompileShader(VertexShaderID);
```

El código del shader se une a la variable declarada, para luego compilarlo.

```
GLuint ProgramID = glCreateProgram();
        glAttachShader(ProgramID, VertexShaderID);
        glAttachShader(ProgramID, FragmentShaderID);
        glLinkProgram(ProgramID);
```

Una vez lista la compilación de ambos shaders, estos se unen en una sola variable, funcionando desde ahora en el mismo pipeline.

```
glDeleteShader(VertexShaderID);
        glDeleteShader(FragmentShaderID);
```

Finalmente, se eliminan los dos punteros, ya que ambos shaders ya fueron unidos a un puntero general, en la figura 4.2 se gráfica los pasos de como se inicia el programa.

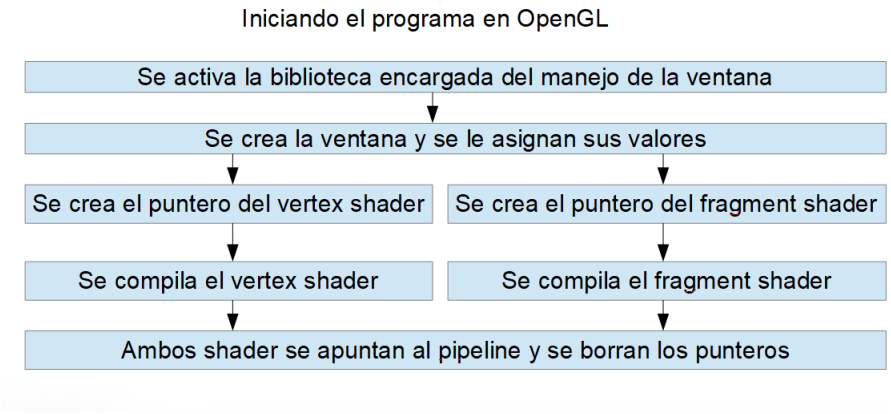


Fig. 4.2: Fuente: Elaboración propia

4.2.2. Shader

Los shaders tiene una función muy importante en OpenGL, la cual será explicada en la siguiente sección. El siguiente código va dentro de los archivos de los shaders; primero se explicará el vertex shader.

```
#version 330 core
```

Todo shader comienza mencionando la versión del GLSL en que funciona, la que coincide con la versión de OpenGL, por ejemplo, 330 equivale a decir que se usa la versión 3.3 de OpenGL.

```
layout(location = 0) in vec3 vertexPosition_modelspace;
```

En la declaración de variables se encuentran estos datos de entrada, los que son únicos para cada vértice. Debido a esto el programa en OpenGL se encarga de enviarle los datos de cada vértice al shader; el dato location = 0 sirve para identificar la entrada de la variable, ya que desde el programa en OpenGL de OpenGL solo se manda información a la location 0, sin ningún nombre; in significa que este valor es de entrada; vec3 significa que es un arreglo de 3 valores y por último se menciona el nombre que tendrá la variable dentro del shader.

```
glEnableVertexAttribArray(0);
glBindBuffer(GL_ARRAY_BUFFER, vertexbuffer);
glVertexAttribPointer(0,3,GL_FLOAT,GL_FALSE,0,(void*)0);
```

En el programa en OpenGL, se envían los datos a través de distintas funciones a los programa de shaders. La función glEnableVertexAttribArray apunta al dato de entrada al cual se quiere vincular, el número se refiere al valor location que se encuentra en la declaración de variables del programa del shader. La función glBindBuffer vincula la variable que tiene el programa en OpenGL a la variable existente en el programa del shader, la cual es un arreglo unidimensional. La función glVertexAttribPointer tiene varios parámetros: la primera variable le vuelve a decir que es parte del location 0; 3 se refiere a la cantidad de datos que se usarán en un shader, ya que la variable en el shader era vec3 (también podría decir que es una variable vec4 si la cantidad de datos que le pasase del arreglo serian 4). Luego se ingresa el tipo de dato de ese arreglo, se le dice si es normalizado, si tiene algún salto entre cada selección de datos y por último si el arreglo tiene un offset. Por cada variable que sale de este arreglo, se ejecuta una instancia de shader en particular, debido a que pertenece a un vértice.

Para poner un ejemplo, si se tiene un triángulo, se debería tener un vertexbuffer que contenga todas las coordenadas de los puntos del triángulo, por lo cual se vería como un arreglo de 9 datos tal como se ve en la figura 4.3, ya que cada vértice está compuesto de 3 componentes que hacen referencia a [x,y,z] y existen 3 vértices, entonces se le dice al shader que cada uno de estos vértices tiene 3 datos, entonces va sacando los datos del arreglo de 3 en 3 y cada uno de ellos tiene su propio programa de shader ejecutándose.

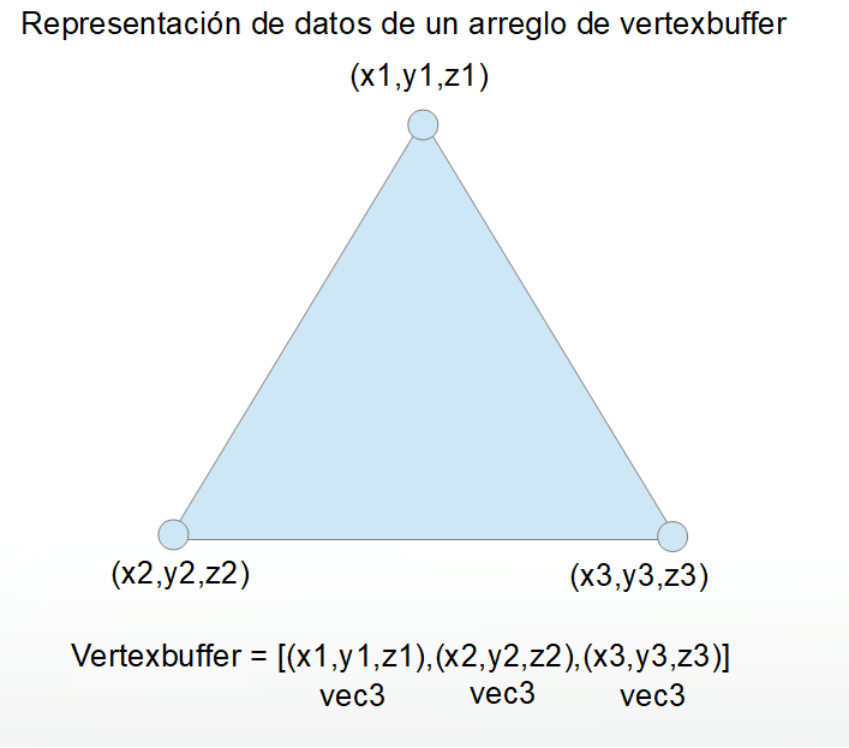


Fig. 4.3: Fuente: Elaboración propia

```
out vec3 Position_worldspace;
```

Los shaders también tienen datos de salida, lo que se define poniendo out al principio, luego se le acompaña con el tipo de dato y el nombre del dato. Como se ha explicado anteriormente, los shaders forman parte de un pipeline, por lo cual estos datos irán a parar a la siguiente etapa, en este caso particular, al fragment shader.

En este punto se le pueden aplicar transformaciones a los vértices y se pueden calcular datos adicionales que provienen de la característica del vértice, por eso se le pueden poner los cálculos sobre las normales, las posiciones bajo diferentes transformaciones, etc.

```
uniform mat4 MVP;
```

También existen constantes, valores que serán iguales para todos los shaders que

se ejecuten y que son unidas a otras variables usando el puntero del shader.

```
GLuint MatrixID = glGetUniformLocation(programID, "MVP");
```

En el programa en OpenGL se puede crear un puntero que apunta a la variable en el shader, en él se debe especificar su nombre.

```
glUniformMatrix4fv(MatrixID, 1, GL_FALSE, &MVP[0][0]);
```

Luego se usa ese puntero para unir una variable usada con la misma estructura de la variable en el shader, ya que la variable es mat4, que sería una matriz de 4x4, entonces se le pasa al shader la dirección del primer valor de esa variable, para que pueda sacar el resto de los valores.

```
void main(){
```

Luego de declarar las variables, se puede empezar el programa del shader, cuyo lenguaje está basado en C, por lo cual cuenta con una función main. Este programa funciona como un proceso aparte del programa en OpenGL, ya que se ejecuta sobre la tarjeta gráfica trabajando en paralelo.

```
gl_Position = MVP * vec4(vertexPosition_modelspace,1);
```

Lo primero que se debe saber es gl_Position, que es la variable que da la posición del vértice que se está trabajando. hay varias formas de calcularlo, en ese ejemplo se usa una matriz de transformación para llevar al vértice, usando la misma matriz que se uso para llevar la cámara al origen.

En el resto del shader se pueden calcular las otras variables de salida que sean necesarias en el próximo paso del pipeline, luego se cierra el paréntesis y se termina el programa del shader.

Los Fragment Shaders se encargan del pintado de los pixeles. Por cada pixel que hay en la pantalla, se ejecuta la cantidad de shaders indicada por la cantidad de samples que se le ordenó al programa.

La forma en que están estructurados los fragment shaders es igual a los vertex shaders, es decir, también se debe poner #version 330 core al inicio, declarar las variables de forma similar y usar una función main para la ejecución del código, así que se explicarán algunas de sus características particulares.

```
in vec3 Position_worldspace;
```

La declaración de la variable in es similar, pero el lugar del origen de los datos es diferente. Esta vez el origen de los datos es el vertex shader, pues proviene de la variable out que tiene dicho programa de shader, debido a que se encuentra a continuación en el camino del pipeline.

```
layout(location = 0) out vec3 color;
```

El resultado del fragment shader es el color del pixel que se está ejecutando en ese momento, por lo cual esta variable tiene como objetivo pintarlo, así que todo lo que se realice en este shader, tendrá que dar como resultado el valor de la variable color.

También se pueden usar las variables de tipo uniform, por si se quiere usar constantes que provengan del programa en OpenGL. Luego se inicia la función main, en cuyo interior se encuentra el algoritmo que le dará un color al pixel, dándole un valor a la variable de salida, la que corresponde a un vec3, porque guarda los valores RGB.

4.2.3. Sombras Duras

Mapping Shadow

La implementación de las sombras duras en OpenGL se divide en 2 fases. La primera fase es generar el mapa de profundidad, el que se encarga de guardar la profundidad de los vértices dentro de una escena vista desde la posición de la fuente de luz, por lo cual se debe crear una imagen cuyos valores solo tengan una escala, similar a una imagen con escala de grises. luego se guardan los valores y se mandan a los shaders.

En el segundo paso, se procesan los datos en los shaders, ya sea la profundidad, el ángulo de la normal del objeto con la luz incidente y otras características, con los cuales es posible saber si el pixel tiene sombra o no y se pinta de acuerdo a ese juicio.

```
GLuint FramebufferName = 0;
glGenFramebuffers(1, &FramebufferName);
glBindFramebuffer(GL_FRAMEBUFFER, FramebufferName);
```

Lo primero que es necesario para empezar a utilizar los mapas de profundidad es crear un FrameBuffer, que se encarga de administrar los diferentes buffers existentes y en los cuales se pueden almacenar las texturas producidas.

```
GLuint depthTexture;
glGenTextures(1, &depthTexture);
glBindTexture(GL_TEXTURE_2D, depthTexture);
```

Una vez iniciado el framebuffer, se puede inicializar la variable que tendrá el mapa de profundidad, creando primero un puntero que lo vinculará a la textura y luego asignándole el tipo de textura.

```
glTexImage2D(GL_TEXTURE_2D, 0,GL_DEPTH_COMPONENT16, 1024, 1024, 0,
    GL_DEPTH_COMPONENT, GL_FLOAT, 0);
```

Ahora se inicializa la textura aplicándole diferentes características, como por ejemplo, que tendrá un solo componente de 16 bits, que su tamaño será de 1024x1024

pixeles, que guardará valores de profundidad y tendrá valores del tipo float. Se debe recordar que entre más grande sea la textura, la sombra tendrá mejor calidad.

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_FUNC, GL_LEQUAL);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_MODE,
    GL_COMPARE_R_TO_TEXTURE);
glFramebufferTexture(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
    depthTexture, 0);
```

Por último se agregan otras características importantes y se le aplican a la textura a través de su puntero.

El siguiente código está dentro del bucle que dibuja la escena en 3 dimensiones y es parte del algoritmo para generar las sombras.

```
glBindFramebuffer(GL_FRAMEBUFFER, FramebufferName);
glViewport(0,0,1024,1024);
```

Esta vez se inicia el framebuffer para esta ejecución del código, pero como se puede usar más de un framebuffer, es necesario saber cuál está funcionando en este momento, así se le puede indicar el tamaño que tendrá el framebuffer a utilizar. La siguiente ejecución deberá ocupar el espacio indicado con un X e Y inicial y final, lo que además sirve para colocar más vistas en un mismo framebuffer que también pueden ser mostradas en pantalla, aunque esta vez el tamaño está limitado a las medidas que se le dio a la textura.

Para la sección de shaders, se utilizarán dos grupos de shaders para hacer la labor completa. El primero se llamará GLuint depthProgramID y se encargará de revisar la profundidad de los elementos en la escena con respecto a la luz.

```
GLuint depthMatrixID = glGetUniformLocation(depthProgramID, "
    depthMVP");
```

Esta variable almacenará la trasformación de la posición de los objetos según la luz, la cual servirá para aplicarla dentro del shader.

```
glUseProgram(depthProgramID);
```

Esta función sirve para hacer uso del shader y además inicia el pipeline de los shaders.

```
glm::vec3 lightInvDir = glm::vec3(0.5f,2,2);
```

Esta variable almacena la dirección a la cual se dirige la luz.

```
glm::vec3 lightPos(5, 20, 20);
```

Se asigna una posición a la luz.

```
glm::mat4 depthProjectionMatrix = glm::perspective<float>(60.0f, 1.0
    f, 2.0f, 50.0f);
```

Es necesaria una matriz extra que se encargue de la perspectiva; sin esta matriz, por ejemplo, dos objetos de igual tamaño, a pesar de estar distantes, se verían iguales. El primer dato de la matriz es el ángulo del campo de vista: si ese valor fuera muy pequeño, se obtendría un espacio sin perspectiva. Por el contrario, si ese valor fuera muy grande, daría una sensación similar a la vista ojo de pescado. El segundo valor es el radio del tamaño de la ventana, que esta vez es 1, porque tanto la altura como el ancho son similares. Para terminar, siguen los valores de inicio y fin de espacio de perspectiva con respeto a la profundidad.

```
glm::mat4 depthViewMatrix = glm::lookAt(lightPos, lightPos-
    lightInvDir, glm::vec3(0,1,0));
```

Se crea una matriz que crea la visión de la luz desde su posición hasta la posición a la que apunta.

```
glm::mat4 depthModelMatrix = glm::mat4(1.0);
glm::mat4 depthMVP = depthProjectionMatrix * depthViewMatrix *
    depthModelMatrix;
```

Por último, se crea una matriz que servirá para aplicar la perspectiva al vector de la dirección de la luz.

```
glUniformMatrix4fv(depthMatrixID, 1, GL_FALSE, &depthMVP[0][0]);
```

Se guarda el puntero de la matriz perspectiva para ser usado en el shader.

```
glEnableVertexAttribArray(0);
glBindBuffer(GL_ARRAY_BUFFER, vertexbuffer);
glVertexAttribPointer(0,3,GL_FLOAT,GL_FALSE,0,(void*)0);
```

En este punto se le informa a los shaders los datos del vertexbuffer en donde van almacenados los vértices de las figuras en la escena para que sean utilizados en el vertex shader.

```
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, elementbuffer);
glDrawElements(GL_TRIANGLES,indices.size(),GL_UNSIGNED_SHORT,(void*)
    0);
```

El elementbuffer trae consigo bastante información, por lo cual se une a la llamada de dibujo. Esta vez se empieza a dibujar la escena en el espacio, trayendo consigo sus triángulos y colores, aunque se debe pasar por el pipeline antes de que esté completo.

Esta ejecución del pipeline es bastante simple. La imagen final que se debería obtener debe representar la escena cuyos colores son una escala de grises representando la profundidad.

```
#version 330 core
layout(location = 0) in vec3 vertexPosition_modelspace;
uniform mat4 depthMVP;
void main(){
        gl_Position =  depthMVP * vec4(vertexPosition_modelspace,1);
}
```

El código de este vertex shader es bastante simple. Como ya se explicó de dónde las variables, ahora se calcurán otras que serán usadas en el fragment shader. Esta vez solo se calculará gl_Position, que es la posición del vértice después de aplicarle una transformación al vertexPosition_modelspace que proviene del programa en OpenGL. Con esta transformación, la posición del vértice se puede adecuar a la posición del mundo con respecto al observador, realizando la transformación de perspectiva.

El fragment shader también es bastante simple, ya que en él solo se pintarán las profundidades.

```
#version 330 core
layout(location = 0) out float fragmentdepth;
void main(){
        fragmentdepth = gl_FragCoord.z;
}
```

out en el fragment shader es el color del pixel, ya que se conoce su posición, ahora solo es necesario pintarlo. Cabe recordar que se está observando la escena desde el punto de vista de la fuente de luz, así que la profundidad del objeto que se observa pertenece a dicha escena. gl_FragCoord almacena los valores de un punto en la pantalla, a través de 4 valores. Los primeros dos valores corresponden a las coordenadas x e y, las cuales son la posición del pixel actual sobre la pantalla. Por ejemplo, si la pantalla fuera de 1280 x 720 pixeles, la esquina inferior izquierda seria $x = 0{,}5$ e $y = 0{,}5$, mientras que la superior derecha seria $x = 1279{,}5$ e $y = 719{,}5$, ya que los

valores pasan justo por el medio de la posición del pixel, aunque estos valores pueden no estar al medio si es multi samples, ya que crearía más valores de colores. Para mejorar el aliasing, z, el tercer valor, sería la profundidad de $z_{near} - z_{far}$, cuyos valores irían de $[0, 1]$, mientras que el último valor será $1/w$, que es la perspectiva de la ecuación de proyección. Este valor será importante cuando se efectúen los cálculos de profundidad, ya que son necesarios para realizar la transformación con respecto a la perspectiva. Lo que sigue es pintar la profundidad para cada pixel, obteniendo como resultado el shadow map que se observa en la figura 3.2. Debido a que en un principio se unió al glBindFramebuffer(GL_FRAMEBUFFER, FramebufferName), la imagen obtenida se puede usar con depthTexture, la cual será importante para la siguiente parte del algoritmo.

En el programa en OpenGL se preparan las cosas para crear la imagen final que será mostrada en la pantalla.

```
glBindFramebuffer(GL_FRAMEBUFFER, 0);
glViewport(0,0,1024,768);
```

Esta vez se coloca 0 en el glBindFramebuffer, para que se muestre en pantalla lo que se dibuje en este paso, luego se da el tamaño de la imagen, que en este caso corresponde al total de la ventana creada en un principio, ya que los valores muestran qué tanto cubre, empezando desde la esquina inferior izquierda hasta la esquina superior derecha.

```
glEnable(GL_CULL_FACE);
glCullFace(GL_BACK);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

Se le dice al programa en OpenGL que se mostrará solo la parte más cerca de la cara de los objetos y se limpian las caras de los polígonos que no se verán.

```
glUseProgram(programID);
```

Se ejecuta otro shader, que será más complejo que el anterior, debido a que tendrá que realizar un algoritmo más complejo para encontrar y pintar las sombras.

```
glm::mat4 ProjectionMatrix = glm::perspective(45.0f, 4.0f / 3.0f,
    0.1f, 100.0f);
glm::mat4 ViewMatrix = glm::lookAt(position,position+direction,up);
glm::mat4 ModelMatrix = glm::mat4(1.0);
glm::mat4 MVP = ProjectionMatrix * ViewMatrix * ModelMatrix;

                glm::mat4 biasMatrix(
                        0.5, 0.0, 0.0, 0.0,
                        0.0, 0.5, 0.0, 0.0,
                        0.0, 0.0, 0.5, 0.0,
                        0.5, 0.5, 0.5, 1.0
                );

                glm::mat4 depthBiasMVP = biasMatrix*depthMVP;
```

Como ya se ha explicado previamente la perspectiva y la función lookAt, se detallará otro proceso de este paso, que es el armado de la cámara que se mostrará en la escena. Una de las formas más fáciles de trabajar en OpenGL con las cámaras es mover a todo el mundo menos la cámara, por lo cual es necesario empezar con un mundo en su posición normal. En este punto todos los objetos están en su posición absoluta, entonces se ubica a la cámara en un punto cualquiera y que mire hacia cualquier dirección; sin embargo, como resultaría difícil calcular bien todas las posiciones de los objetos en tiempo real para mostrarlos perfectamente en cámara, se lleva la cámara al origen, almacenando todas las transformaciones usadas para luego aplicar estas transformaciones a los objetos, así es como se mueven todos los objetos de tal forma que ahora la cámara pueda verlos de la misma manera que estaban en un inicio, pero ahora con la cámara en el origen.

```
GLuint TextureID  = glGetUniformLocation(programID, "
    myTextureSampler");
GLuint MatrixID = glGetUniformLocation(programID, "MVP");
GLuint ViewMatrixID = glGetUniformLocation(programID, "V");
GLuint ModelMatrixID = glGetUniformLocation(programID, "M");
GLuint DepthBiasID = glGetUniformLocation(programID, "DepthBiasMVP")
    ;
GLuint ShadowMapID = glGetUniformLocation(programID, "shadowMap");
GLuint lightInvDirID = glGetUniformLocation(programID, "
    LightInvDirection_worldspace");
```

Las constantes que se usarán en los shaders son las siguientes:

- TextureID: guarda las texturas del objeto.
- MatrixID: guarda la matriz de la transformación de los objetos.
- ViewMatrixID: guarda la matriz transformadora dependiendo hacia dónde esté mirando.
- ModelMatrixID: es una matriz identidad
- DepthBiasID: guarda la matriz de transformación, pero es necesario cambiar los límites de los valores de la matriz depthMVP, ya que se encuentra entre [0,1] y se requiere que esta variable esté entre [-1,1], por lo cual se le realiza una tranformación multiplicándola con otra matriz, quedando sus límites en [-0.5, 0.5]
- ShadowMapID: guardará el mapa de profundidad que se usará.
- lightInvDirID: guarda la dirección de la luz.

```
glUniformMatrix4fv(MatrixID, 1, GL_FALSE, &MVP[0][0]);
glUniformMatrix4fv(ModelMatrixID, 1, GL_FALSE, &ModelMatrix[0][0]);
glUniformMatrix4fv(ViewMatrixID, 1, GL_FALSE, &ViewMatrix[0][0]);
glUniformMatrix4fv(DepthBiasID, 1, GL_FALSE, &depthBiasMVP[0][0]);
glUniform3f(lightInvDirID, lightInvDir.x, lightInvDir.y, lightInvDir
    .z);
```

Se le asignan los punteros a las variables que se usarán en los shaders.

```
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, Texture);
glUniform1i(TextureID, 0);
```

GL_TEXTURE0 indica que es la primera variable que se pasará y guardará la textura; Texture es del tipo GLuint que almacena el puntero de la textura que se usará.

```
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, depthTexture);
glUniform1i(ShadowMapID, 1);
```

Similar a una textura normal, el shadow map se transfiere de esta manera al shader.

```
glEnableVertexAttribArray(0);
glBindBuffer(GL_ARRAY_BUFFER, vertexbuffer);
glVertexAttribPointer(0,3,GL_FLOAT,GL_FALSE,0,(void*)0);

glEnableVertexAttribArray(1);
glBindBuffer(GL_ARRAY_BUFFER, uvbuffer);
glVertexAttribPointer(1,2,GL_FLOAT,GL_FALSE,0,(void*));

glEnableVertexAttribArray(2);
glBindBuffer(GL_ARRAY_BUFFER, normalbuffer);
glVertexAttribPointer(2,3,GL_FLOAT,GL_FALSE,0,(void*)0);

glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, elementbuffer);

glDrawElements(GL_TRIANGLES,indices.size(),GL_UNSIGNED_SHORT,(void*)
    0);
```

En este paso se transfieren los datos al shader; el vertexbuffer contiene la posición de los vértices en el espacio, el uvbuffer contiene la posición de la textura que va en ese vértice y el normalbuffer almacena la normal del vértice. Este dato es importante para el sombreado debido que el valor de la reflexión de la luz depende la normal. Por último se llama a los elementos, informándole al programa que dibuje la escena con los elementos que se le ha entregado. La escena con todos los elementos será dibujada por el framebuffer al que se le asignó valor 0, por lo cual todo lo que se realice, se mostrará en pantalla.

Ahora se llamarán a otros shaders, los cuales están encargados de dibujar las sombras en la escena entregada, y que es más complejo que el shader que creaba los shadow map, por lo que se continuará con el paso de de los datos por el pipeline.

```
#version 330 core
layout(location = 0) in vec3 vertexPosition_modelspace;
layout(location = 1) in vec2 vertexUV;
layout(location = 2) in vec3 vertexNormal_modelspace;

out vec2 UV;
out vec3 Position_worldspace;
out vec3 Normal_cameraspace;
out vec3 EyeDirection_cameraspace;
out vec3 LightDirection_cameraspace;
out vec4 ShadowCoord;

uniform mat4 MVP;
uniform mat4 V;
uniform mat4 M;
uniform vec3 LightInvDirection_worldspace;
uniform mat4 DepthBiasMVP;
```

Los datos que se explicaron anteriormente pertenecen a la entrada del shader, por lo que ahora se hará mención a los de salida. Lo que sigue a continuación pertenece a la función main del shader.

```
gl_Position =  MVP * vec4(vertexPosition_modelspace,1);
```

Es necesario saber la posición del vértice, por lo cual se pregunta la posición absoluta del vértice a los datos ingresados, para luego transformarlos a la posición de la cámara.

```
ShadowCoord = DepthBiasMVP * vec4(vertexPosition_modelspace,1);
```

Este paso es similar al anterior, pero el resultado se usará para compararlo con el mapa de profundidad. El cambio en la matriz se realiza para que se pueda usar en la escala de [0,1]

```
Position_worldspace = (M * vec4(vertexPosition_modelspace,1)).xyz;
```

Esta transformación es similar a la primera, pero esta contiene un arreglo de 3 elementos.

```
EyeDirection_cameraspace = vec3(0,0,0)-(V*M*vec4(
    vertexPosition_modelspace,1)).xyz;
```

Este paso crea un vector que apunta desde el vértice a la cámara, cuando la cámara se encuentra en el origen.

```
Normal_cameraspace = ( V * M * vec4(vertexNormal_modelspace,0)).xyz;
```

En este paso se calcula la normal con respecto a la cámara.

```
UV = vertexUV;
```

Por último se agrega la variable de la posición de la textura que pertenece a este vértice.

En el siguiente paso del pipeline, después del cálculo de variables de los vértices por el vertex shader, sigue el fragment shader, el cual se encargará de analizar los datos de cada pixel y juzgar si es sombra o no.

```
#version 330 core

in vec2 UV;
in vec3 Position_worldspace;
in vec3 Normal_cameraspace;
in vec3 EyeDirection_cameraspace;
in vec3 LightDirection_cameraspace;
in vec4 ShadowCoord;

layout(location = 0) out vec3 color;

uniform sampler2D myTextureSampler;
uniform mat4 MV;
uniform vec3 LightPosition_worldspace;
uniform sampler2DShadow shadowMap;
```

Para el fragment shader se usan como entrada los datos obtenidos del vertex shader como parte pipeline, mientras que el dato de salida es el color del pixel y otros valores constantes entregados por el programa principal. Lo que sigue es parte de la función main del fragment shader.

```
vec3 LightColor = vec3(1,1,1);
float LightPower = 1.0f;
```

La luz blanca contiene todos los colores y se encuentra en su máxima potencia.

```
vec3 MaterialDiffuseColor = texture2D(myTextureSampler,UV).rgb;
vec3 MaterialAmbientColor = vec3(0.1,0.1,0.1) * MaterialDiffuseColor
    ;
vec3 MaterialSpecularColor = vec3(0.3,0.3,0.3);
```

Se almacena el color del pixel que se obtiene de la textura, luego se calcula el color del pixel con luz ambiental, siendo esta luz la única que ilumine al objeto si éste se encuentra en una sombra.

```
vec3 n = normalize( Normal_cameraspace );
vec3 l = normalize( LightDirection_cameraspace );
```

Se normalizan los valores de la normal de la cámara y la dirección de la luz.

```
float cosTheta = clamp( dot( n,l ), 0,1 );
```

Se calcula el producto punto de ambos vectores; la intensidad de la luz reflejada dependerá de qué tan cerca estén el uno del otro. El resultado que arroje esta operación se debe limitar entre 0 y 1, ya que si fuera menor a 0, el pixel no estaría reflejando la luz, debido a que estaría en un ángulo mayor a $180°$ y en este caso, solo se le aplica un valor 0 a la cantidad de luz reflejada.

```
vec3 E = normalize(EyeDirection_cameraspace);
```

Luego, se normaliza este vector.

```
vec3 R = reflect(-l,n);
float cosAlpha = clamp( dot( E,R ), 0,1 );
```

A continuación, se calcula el factor de la incidencia de la luz con respecto del vector de la cámara con el vector de la normal de la superficie, ya que el punto de mayor reflexión de luz para el ojo, es cuando el ángulo de incidencia de la luz es igual al ángulo que tiene el ojo con respecto a la normal.

```
float visibility=1.0;
float bias = 0.005;
```

En una siguiente etapa, se preparan dos variables. La primera es la potencia inicial de la luz, la cual será 1; en este punto el pixel está recibiendo el máximo de luz y aun no se decide si el pixel mostrado se encuentra en la sombra. Luego se tendrá el bias, el cual ayudará a que no se produzcan efectos indeseados en la sombra, ya que sin ese valor, se verían zonas de luz dentro de la sombra, pareciendo un área achurada, debido a un valor de comparación demasiado estricto para un mapa de profundidad de baja calidad.

```
if(textureProj(shadowMap, ShadowCoord.xyw).z<(ShadowCoord.z-bias)/
    ShadowCoord.w)
visibility=0.0;
```

Ahora se puede comparar el mapa de profundidad con el valor de la posición, usando textureProj, porque el valor está bajo una proyección. Luego se revisa si el valor se encuentra bajo el bias para no generar rayas extrañas en la sombra; si la comparación es correcta, se asigna 0 a visibility, ya que ese valor significa que se encuentra dentro de una sombra.

```
color = MaterialAmbientColor +
visibility * MaterialDiffuseColor * LightColor * LightPower *
    cosTheta+
visibility * MaterialSpecularColor * LightColor * LightPower * pow(
    cosAlpha,5);
```

En esta última etapa se le da el color al pixel. Primero se encuentra la variable MaterialAmbientColor, que representa a la luz remanente; las sombras no son totalmente negras, ya que la luz se está reflejando por distintas superficies, por lo que es posible que una parte de esa luz llegue a esa zona, entonces a esa luz se le denomina luz ambiental. La segunda variable, MaterialDiffuseColor, corresponde a la reflexión difusa; hay materiales que tienen mayores valores para esta reflexión, debido a que sus superficies reflejan en todas direcciones. Finalmente, la tercera variable, MaterialSpecularColor, representa la reflexión directa, la que ocurre cuando la superficie es relativamente plana y tiende a reflejar en una sola dirección. El material que tiene mayor factor en este tipo de reflexión es el metal.

Shadow Volume

El volumen está creado por varios polígonos que se extienden desde el borde de la figura que crea la sombra y se proyecta hasta el infinito en la dirección contraria en que se encuentra la luz, formando una especie de manta sobre la figura. En OpenGL

se puede crear este volumen, pero será necesario conocer la forma en que están organizados los vértices y luego usar el geometry shader para crear los polígonos.

Antes que todo, para crear el volumen es necesario encontrar los bordes que dan inicio al polígono. Para realizar esta tarea es necesario ver a los objetos que están en la escena como un grupo de triángulos, ya que ese es el polígono más básico que existe en la computación gráfica, los cuales serán capaces de encontrar las aristas que pertenecen al borde, ya que si, al analizar dos triángulos vecinos que comparten una arista en común mientras son iluminados, se da el caso de que uno de ellos refleja la luz y el otro no, la arista en común pertenecerá al borde que genera la sombra. Saber si un triángulo refleja la luz es fácil, ya que se puede calcular el producto punto entre la normal y la dirección de la luz; si este valor es menor que 0, entonces esa cara no recibe luz. Debido a esto, el primer trabajo del programa es encontrar los triángulos y los vecinos de estos según las aristas que comparten.

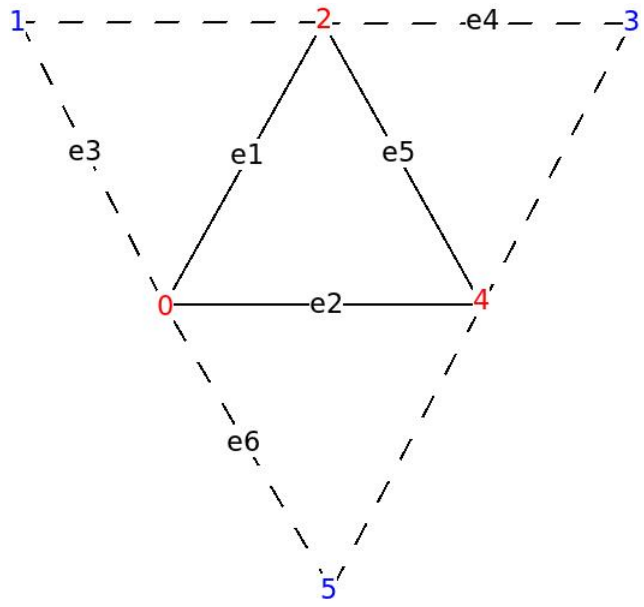


Fig. 4.4: Fuente : OGL dev, Chapter 39: Silhouette Detection - `http://ogldev.atspace.co.uk/www/tutorial39/tutorial39.html`

En cada instancia del geometry shader se pueden usar hasta 6 vértices, como se ve

en la figura 4.4, los cuales corresponden al triángulo analizado y a sus vecinos. Esto facilita la búsqueda de aristas que pertenecen al borde, así que se deben crear grupos de triángulos a través de sus vértices, para lo cual se usará la librería Assimp[29], pues contiene funciones interesantes para el desarrollo de este algoritmo.

```
for (uint i=0; i<paiMesh->mNumFaces;i++){
        const aiFace& face = paiMesh->mFaces[i];
        Face Unique;

        for (uint j=0;j<3;j++){
                uint Index = face.mIndices[j];
                aiVector3D& v = paiMesh->mVertices[Index];

                if (m_posMap.find(v) == m_posMap.end()) {
                        m_posMap[v] = Index;
                }
                else {
                        Index = m_posMap[v];
                }
                Unique.Indices[j] = Index;
        }

        m_uniqueFaces.push_back(Unique);

        Edge e1(Unique.Indices[0], Unique.Indices[1]);
        Edge e2(Unique.Indices[1], Unique.Indices[2]);
        Edge e3(Unique.Indices[2], Unique.Indices[0]);

        m_indexMap[e1].AddNeigbor(i);
        m_indexMap[e2].AddNeigbor(i);
        m_indexMap[e3].AddNeigbor(i);
}
```

paiMesh es una variable del tipo aiMesh, que contiene una gran cantidad de información del polígono en la escena. Esta vez se quiere obtener la información de las

caras del polígono, como cada cara es un triángulo y esa variable guarda la información de los 3 vértices que la componen, entonces se almacena en una variable el índice del vértice y se procura que no esté duplicado, para no malgastar recursos. Una vez listo este proceso, se crean las aristas que están compuestas de dos vértices.

```
std::map<Edge, Neighbors, CompareEdges> m_indexMap;
```

Estos datos se ingresan en un mapa que almacena las aristas y a qué triángulo pertenecen, mientras que las variables Edge y Face son estructuras que almacenan los índices de dos o tres vértices respectivamente, luego hace falta una segunda parte para especificar qué triángulos son vecinos.

```
for (uint i=0;i<paiMesh->mNumFaces;i++){
        const Face& face = m_uniqueFaces[i];

        for (uint j=0;j<3;j++){
        Edge e(face.Indices[j], face.Indices[(j + 1) % 3]);
                assert(m_indexMap.find(e) != m_indexMap.end());
                Neighbors n = m_indexMap[e];
                uint OtherTri = n.GetOther(i);

                assert(OtherTri != -1);

                const Face& OtherFace = m_uniqueFaces[OtherTri];
                uint OppositeIndex = OtherFace.GetOppositeIndex(e);

                Indices.push_back(face.Indices[j]);
                Indices.push_back(OppositeIndex);
        }
}
```

Neighbors es un struct que guarda el índice de dos caras, ya que cada arista es parte de dos triángulos, que contiene la función GetOther(), la cual sirve para seleccionar al triángulo vecino del original. A continuación, se selecciona el vértice que se encuentra en este otro triángulo pero que no está en el original, ya que el struct Face tiene una

función relacionada, llamada GetOppositeIndex, que selecciona al vértice que no se encuentra en los vértices de la arista que se ingresa, este vértice se almacena en los índices junto con el primer vértice analizado. luego se hace el mismo ejercicio con los demás vértices, hasta que se obtienen los 6 vértices necesarios para tener al triángulo y a los 3 triángulos limítrofes.

```
GLuint vertexbuffer;
glGenVertexArrays(1, &m_VAO);
glBindVertexArray(m_VAO);
GLuint m_Buffers[NUM_VBs];
glGenBuffers(ARRAY_SIZE_IN_ELEMENTS(m_Buffers), m_Buffers);

vector<uint> Indices;
Indices.reserve(NumIndices);

glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, m_Buffers[INDEX_BUFFER]);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(Indices[0]) * Indices.
    size(), &Indices[0], GL_STATIC_DRAW);

glBindVertexArray(0);
```

En la variable Indices se han almacenado los vértices de la figura, por lo que ahora es necesario guardarlo en un vertexarray buffer, específicamente en m_Buffers. Si bien se puede almacenar la información de otros buffers, por ahora sólo es necesario mostrar los vértices.

Con los vertex buffers listos, se puede hacer uso de los shaders. Esta vez el fragment shader no será usado, ya que no se dibujará la escena, por lo cual el paso será desde el vertex shader al geometry shader.

```
#version 330

layout (location = 0) in vec3 Position;
layout (location = 1) in vec2 TexCoord;
layout (location = 2) in vec3 Normal;

out vec3 PosL;

void main()
{
    PosL = Position;
}
```

El vertex shader es simple y sin complicaciones, ya que solo se deben pasar los vértices al siguiente shader que hará todo el trabajo con ellos.

Se explicará paso a paso lo que el geometry shader va a hacer para crear los volúmenes de sombra.

```
#version 330

layout (triangles_adjacency) in;
layout (triangle_strip, max_vertices = 18) out;

in vec3 PosL[];

uniform vec3 gLightPos;
uniform mat4 gWVP;

float EPSILON = 0.0001;
```

El shader tiene como entrada el triángulo y sus adyacentes, los cuales suman 6 vértices, por otra parte; la cantidad máxima de vértices en los datos de salida será 18, número que considera los vértices que se puedan crear en el proceso.

En la variable PosL[] se almacenarán los 6 vértices mencionados anteriormente; la variable gLightPos tiene la posición de la luz, la cual será importante a la hora de lanzar las aristas del volumen; y la variable gWVP almacenará la matriz de transformación del mundo en la escena.

Por último se tiene una variable EPSILON, la cual servirá para alejar el nuevo vértice del oclusor original, para que no se produzcan errores en un proceso posterior.

```
void EmitQuad(vec3 StartVertex, vec3 EndVertex)
{
    vec3 LightDir = normalize(StartVertex - gLightPos);
    gl_Position = gWVP * vec4((StartVertex + LightDir * EPSILON),
        1.0);
    EmitVertex();

    gl_Position = gWVP * vec4(LightDir, 0.0);
    EmitVertex();

    LightDir = normalize(EndVertex - gLightPos);
    gl_Position = gWVP * vec4((EndVertex + LightDir * EPSILON), 1.0)
        ;
    EmitVertex();

    gl_Position = gWVP * vec4(LightDir , 0.0);
    EmitVertex();

    EndPrimitive();
}
```

La función EmitQuad está encargada de realizar la proyección de una arista al infinito, la cual crea una cara del volumen de sombra. Para lograrlo, son necesarios dos vértices, los que pertenecen a una de las tantas aristas que tendrá el oclusor. LightDir es el vector unitario que apunta desde el vértice inicial de la luz, hasta el vértice con el

que se está trabajando, y que posteriormente servirá para proyectar la sombra. El primer vértice que se crea se encuentra cerca del vértice dado inicialmente, separado una cantidad EPSILON del vector LightDir, distancia que evita que se produzcan errores, luego se agrega el punto proyectado con el vector LightDir, realizando en seguida la misma acción, pero con el otro vértice, para finalmente unir todos los vértices, creando un polígono.

```
void main()
{
    vec3 e1 = PosL[2] - PosL[0];
    vec3 e2 = PosL[4] - PosL[0];
    vec3 e3 = PosL[1] - PosL[0];
    vec3 e4 = PosL[3] - PosL[2];
    vec3 e5 = PosL[4] - PosL[2];
    vec3 e6 = PosL[5] - PosL[0];

    vec3 Normal = normalize(cross(e1,e2));
```

Es necesario conocer las normales de los triángulos, para lo cual se calculan las aristas, luego se aplica el producto cruz entre dos aristas para obtener la normal. Al primer triángulo para el cual es necesario calcular la normal es el de la cara principal, ya que después se debe comparar con los demás triángulos.

```
vec3 LightDir = normalize(gLightPos - PosL[0]);
if (dot(Normal, LightDir) > 0) {
```

Luego, se obtiene la dirección de la luz incidente a través de la recta que se forma desde la fuente de la luz hasta el triángulo. Este dato ayuda a saber si el triángulo está iluminado, ya que si se calcula el producto punto entre la luz incidente y la normal, el objeto iluminado debería dar un valor mayor a 0, ya que el ángulo formado por ambas rectas debería ser menor que $90°$.

```
Normal = cross(e3,e1);
if (dot(Normal, LightDir) <= 0) {
        vec3 StartVertex = PosL[0];
        vec3 EndVertex = PosL[2];
        EmitQuad(StartVertex, EndVertex);
}

Normal = cross(e4,e5);
LightDir = gLightPos - PosL[2];

if (dot(Normal, LightDir) <= 0) {
        vec3 StartVertex = PosL[2];
        vec3 EndVertex = PosL[4];
        EmitQuad(StartVertex, EndVertex);
}

Normal = cross(e2,e6);
LightDir = gLightPos - PosL[4];

if (dot(Normal, LightDir) <= 0) {
        vec3 StartVertex = PosL[4];
        vec3 EndVertex = PosL[0];
        EmitQuad(StartVertex, EndVertex);
}
```

El calculo del producto punto se realiza luego para todos los triángulos, aunque esta vez se busca que los triángulos no estén iluminados, ya que la arista de dos triángulos en que uno esté iluminado y el otro no, forma parte del borde que forma la sombra; si esta condición se cumple, se manda a dibujar la cara del polígono del volumen de sombra.

```
LightDir = (normalize(PosL[0] - gLightPos));
gl_Position = gWVP * vec4((PosL[0] + LightDir * EPSILON),
    1.0);
EmitVertex();

LightDir = (normalize(PosL[2] - gLightPos));
gl_Position = gWVP * vec4((PosL[2] + LightDir * EPSILON),
    1.0);
EmitVertex();

LightDir = (normalize(PosL[4] - gLightPos));
gl_Position = gWVP * vec4((PosL[4] + LightDir * EPSILON),
    1.0);
EmitVertex();
EndPrimitive();

LightDir = PosL[0] - gLightPos;
gl_Position = gWVP * vec4(LightDir, 0.0);
EmitVertex();

LightDir = PosL[4] - gLightPos;
gl_Position = gWVP * vec4(LightDir, 0.0);
EmitVertex();

LightDir = PosL[2] - gLightPos;
gl_Position = gWVP * vec4(LightDir, 0.0);
EmitVertex();

EndPrimitive();
```

El volumen de sombra se completa con todos los triángulos del objeto que reciben luz de forma directa de la fuente de luz. El volumen debe estar cerrado por todos lados; la parte superior corresponde a la parte iluminada de la figura, los costados son

la proyección de las aristas de los límites de la zona iluminada con la zona oscurecida, faltando la parte inferior, la cual se cierra proyectando la zona iluminada a la misma distancia que se proyectan los vértices de las aristas de los bordes, por lo cual el volumen está conformado de dos secciones que son similares a la zona iluminada, uno en la posición original de la zona iluminada y el otro proyectado en dirección de la luz, estando ambas secciones unidas por los bordes por medio de aristas.

```
glStencilFunc(GL_ALWAYS, 0, 0xff);
glStencilOpSeparate(GL_BACK, GL_KEEP, GL_INCR_WRAP, GL_KEEP);
glStencilOpSeparate(GL_FRONT, GL_KEEP, GL_DECR_WRAP, GL_KEEP);
```

Finalmente, el volumen de sombra está listo para usarse. El siguiente paso es volver al programa en OpenGL para trabajar con el stencil buffer, el cual tiene la característica de aceptar o no un punto dependiendo de si pasa o falla el test, siendo necesario, en este caso, saber si el punto que se está apuntando se encuentra dentro de una sombra o no. Al hacer el test, se lanza una línea imaginaria y por cada vez que esta línea entre en el volumen, se le sumará 1 al buffer, restando 1 por cada vez que salga. La función glStencilFunc se le dice que dibuje cuando el buffer sea igual a 0, es decir, cuando no se encuentre dentro de un volumen, ya que si este valor fuera mayor que 0, implicaría que se encuentra dentro de un volumen de sombra.

```
        glDrawBuffer(GL_BACK);
        glStencilFunc(GL_EQUAL, 0x0, 0xFF);
        glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);
```

Una vez configurado el stencil buffer, se dibuja toda la escena con luz ambiental, sin tener ningún tipo de shading. Se dibuja por segunda vez la escena, por encima de la primera, solo en los sectores en que el stencil buffer sea 0, con toda la intensidad de la luz que provenga de la fuente de luz. De esta forma, en las secciones iluminadas, estará la escena dibujada por el stencil buffer, mientras que en las zonas con sombras, quedará la sección dibujada con luz ambiental.

4.2.4. Técnicas actuales de sombra difusa

PCF (Percentage closer filtering)

El método PCF es la base para los algoritmos actuales de sombra difusa, debido a que es un método bastante maleable, liviano y fácil de implementar, porque es una modificación del método de shadow mapping, compartiendo la mayor parte del algoritmo. La única diferencia comienza en el fragment shader, ya que el tratamiento que se le da al shadow mapping y al z-buffer es distinto, debido a que el método PCF analiza varias secciones del shadow map a la vez para cada pixel, lo que facilita el cálculo de la sombra. Por otra parte, también se ahorra la creación de más shadow maps, ya que solo necesita uno para funcionar.

```
uniform float xPixelOffset ;
uniform float yPixelOffset ;
```

La primera variable necesaria para este algoritmo es la fracción del shadow map, esta variable sirve para moverse de pixel en pixel, ya que las posiciones del shadow map van medidas de [0,1], entonces cada pixel está a una distancia de 1 dividido por el ancho en pixeles del shadow map.

```
float lookup( vec2 offSet)
{
        return texture(shadowMap, vec3((ShadowCoord.x+(offSet.x*
            xPixelOffset))/ShadowCoord.w,(ShadowCoord.y+(offSet.y*
            xPixelOffset))/ShadowCoord.w, (ShadowCoord.z-bias)/
            ShadowCoord.w));
}
```

Esta función permite analizar otra sección del shadow map utilizando el PixelOffset y saber si existe una sombra bajo ese cambio; si el offset fuese vec2(0,0), el pixel analizado sería el pixel que se da por defecto. los valores resultantes son la posición relativa dada por el pixel que se está revisando actualmente.

```
        float shadow ;
        if (ShadowCoord.w > 1.0)
        {
                float x,y;
                for (y = -3.5 ; y <=3.5 ; y+=1.0)
                        for (x = -3.5 ; x <=3.5 ; x+=1.0)
                                shadow += lookup(vec2(x,y));
                shadow /= 64.0;
    }
```

Para obtener el porcentaje de iluminación que tiene un pixel en particular, se deben revisar los diferentes valores kernel, que son las muestras que se recogen en una zona del shadow map. Se puede usar un tamaño arbitrario del kernel, aunque entre mayor sea su número, mejor es el resultado; sin embargo, el tiempo de procesamiento crecerá en una potencia de 2. Para este caso, se usaron 64 kernels, en una estructura 8x8, con separación de 1.0. Es posible utilizar otros valores para calcular estos kernels, ya sea la cantidad o la distancia, pero se debe considerar que estos valores no cambian durante la ejecución del programa.

```
gl_FragColor = (shadow+0.2) * gl_Color;
```

Finalmente, a la variable del color del pixel se le asigna el resultado de la luz ambiental más el porcentaje de luz obtenido del algoritmo de PCF, resultando una sombra con una penumbra homogénea en todo su perímetro.

PCSS (Percentage closer Soft Shadow)

El método PCSS es una modificación del método PCF que permite generar sombras difusas sin mayores cambios al algoritmo general, con la diferencia de que el tamaño del kernel es variable, dependiendo de la distancia del oclusor a la superficie iluminada.

Si la zona analizada es sombra, es fácil saber los valores para conocer el tamaño del kernel; en cambio, si no lo es, habrá que empezar una búsqueda para saber si existe una sombra cerca en el shadow map. Los datos necesarios para generar el tamaño del kernel son la distancia del objeto a la luz, la que se consigue fácilmente con el deep buffer; la distancia del oclusor, dato que se obtiene con el shadow map, (si el oclusor está en una sombra, basta con calcular su valor, pero si está en una zona iluminada, se debe buscar la zona de sombra más cercana por medio del shadow map); por último, se necesita conocer el tamaño de la luz, el cual es un valor predeterminado dado al comienzo del programa.

$$W_{Penumbra} = (D_{superficie} - D_{oclusor}) * W_{luz}/D_{Oclusor}$$

Los datos son ingresados dentro de la fórmula para conocer el tamaño del kernel que se usará. Cada pixel que se calcule en el fragment shader tendrá su propio tamaño de kernel, lo que permite que la penumbra de la sombra tenga diferentes tamaños dependiendo del punto del borde en que se encuentra, siendo este tamaño lo más cercano posible a la realidad.

```
float tam_luz;
tamkernel = (tam_luz/texture(shadowMap, vec3(ShadowCoord.xy/
    ShadowCoord.w,0.0))) * (ShadowCoord.z - texture(shadowMap, vec3(
    ShadowCoord.xy/ShadowCoord.w,0.0)));
```

Esta sería la fórmula del tamaño del kernel que estaría dentro del fragment shader, siendo lo único complejo la búsqueda de la posición del shadow map, ya que esta fórmula pertenecería a un pixel que se encuentra dentro de una sombra; si un pixel se encontrase en una zona iluminada, tal como se mencionó anteriormente, se tendría que buscar la ubicación más cercana a la actual en donde el shadow map arroje una sombra y se utiliza este punto. Este proceso puede ser un poco pesado dependiendo de cómo se realice.

CHS (Contact Hardening Soft Shadow)

El método CHS es parecido a PCSS debido a que hereda todo lo relacionado al uso de un kernel, diferenciándose ambos métodos en la búsqueda del tamaño del kernel. El primer paso de CHS es crear el nuevo buffer, el cual es similar a la textura del shadow map y que necesita de varios pasos para ser creado. Para empezar, se separan arreglos 3x3 de la textura, los cuales son el pixel analizado y todos los demás que le rodean; luego, se multiplica por una matriz que sirve para detectar los bordes, realizándose una operación similar para cada pixel en la textura.

```
glm::mat3 matrizborde(
        0.0, 1.0, 0.0,
        1.0, -4.0, 1.0,
        0.0, 1.0, 0.0
);
```

La matriz está construida de esta forma para poder trabajar con el nuevo buffer, siendo la forma más eficiente para trabajarlo hacerlo a través de los shaders, ya que permite analizar la textura a través de la tarjeta gráfica. Adicionalmente, los datos de entrada son diferentes a los datos de salida, por lo que es fácil de paralelizar.

```
vec3 dim1 = vec3(shadow2D(ShadowMap, vec3(coord.x-1,coord.y
    -1,0)).x, shadow2D(ShadowMap, vec3(coord.x, coord.y-1,0)).x
    , shadow2D(ShadowMap, vec3(coord.x+1, coord.y-1,0)).x);
vec3 dim2 = vec3(shadow2D(ShadowMap, vec3(coord.x-1,coord.y
    ,0)).x, shadow2D(ShadowMap, coord,0).x, shadow2D(ShadowMap,
    vec3(coord.x+1,coord.y,0)).x);
vec3 dim3 = vec3(shadow2D(ShadowMap, vec3(coord.x-1,coord.y
    +1,0)).x, shadow2D(ShadowMap, vec3(coord.x, coord.y+1,0)).x
    , shadow2D(ShadowMap, vec3(coord.x+1, coord.y+1,0)).x);
mat3 matrizshadow = mat3(dim1, dim2, dim3);
color = (matrizshadow * matrizborde)[1][1];
```

Se aplica la matriz y se guarda el valor del resultado de la multiplicación dentro del nuevo buffer. En adelante, cada vez que se necesite un kernel para pintar la penumbra

y la zona que se desee pintar se encuentre iluminada, la búsqueda se realizará a través del nuevo buffer creado.

5. ANÁLISIS DE LAS TÉCNICAS ACTUALES DE SOMBRA DIFUSA

5.1. Metodología

Las sombras difusas en tiempo real son una característica que en el último tiempo se ha intentado explotar en los videojuegos más demandantes gráficamente. Actualmente ya se cuenta con la potencia suficiente en el hardware para poder correr este tipo de sombras con todas las complejidades que tienen estos algoritmos, existiendo dos grandes implementaciones funcionales de este tipo de sombra.

NVIDIA ha trabajado en una solución basada en la técnica PCSS, la cual se encuentra en el conjunto de soluciones gráficas NVIDIA ShadowWorks. Este conjunto de tecnologías trae consigo las últimas técnicas de iluminación y sombras, tales como la oclusión ambiental, varios tipos de sombras en tiempo real y entre estas, las sombras difusas.

AMD, a través de su iniciativa Gaming Evolved, ha hecho llegar a los videojuegos su alternativa para las sombras difusas, utilizando la técnica de CHS para generarlas.

Ya que existen dos implementaciones reales de sombras difusas, se procederá a realizar pruebas sobre ellas, utilizando dos criterios: el rendimiento que entrega dicha técnica de sombra con respecto a otras y la calidad de la sombra creada.

En el apartado de rendimiento se medirá el rendimiento de ambas sombras más el

rendimiento de una sombra dura para saber contra qué se analiza, todo esto se hará través del promedio de fotogramas por segundos que se obtenga a partir de las pruebas que se realice, analizándose además la carga de CPU y GPU. Las pruebas se realizarán sobre un mismo equipo y la diferencia en el rendimiento permitirá conocer la cantidad de potencia extra que necesita una sobre otra, el rendimiento se obtendrá a través de la experimentación y de la información que se encuentre en Internet, ya que es necesario saber cómo corren dichas técnicas en tarjetas gráficas de ambos fabricantes, ya que son ellos mismos los que desarrollaron estas técnicas.

La calidad de la sombra es un factor importante, aunque es más difícil de medir, ya que no tiene una característica cuantitativa con la cual se pueda comparar, por lo que se decidió realizar una encuesta dirigida a varias personas preguntándoles sobre cuál sombra creán que tenía un mejor aspecto. Los resultados de dicha encuesta serán usados para comprender la razón detrás de los motivos mencionados por las personas.

5.2. Rendimiento

El rendimiento de las sombras se debe probar bajo las diferentes condiciones que existen, ya que este puede variar dependiendo del fabricante de la tarjeta de video y de su potencia, debido a que ambas tecnologías son desarrolladas por las dos compañías más grandes en tarjeta gráficas de alto rendimiento, así que éstas podrían utilizar mejor una tecnología en una tarjeta en detrimento de la otra.

Se tomaron los datos de las tarjeta de video de las dos empresas, pertenecientes al juego GTA V, el cual contiene los dos métodos actuales de sombra difusa. Estas tarjetas además de correr las sombras difusas, también corrieron otros tipos de tecnologías, así que lo más importante a considerar es la brecha del rendimiento entre una y otra tecnología, así como también el rendimiento de ellas en ambos tipos de tarjeta.

Tab. 5.1: Tabla de rendimiento de diferentes sombras difusas en tarjetas gráficas de diferentes fabricantes [32]

Tarjeta	CHS(fps)	PCSS(fps)	Diferencia(fps)	Porcentaje
Nvidia GTX 980	52,1	49,8	2,3	4,6 %
AMD R9 290X	41,1	39,9	1,2	3,0 %

En la tabla 5.1 se muestra el rendimiento de las tarjetas gráficas medido en cuadros por segundo (fps), dividiéndolo en la tarjeta de AMD y la tarjeta de Nvidia corriendo los métodos PCSS y CHS. lo importante de este benchmark no son los rendimientos por sí solos, sino la diferencia de rendimiento de una técnica sobre otra y ver en general cómo corren en estas tarjetas. En este caso, la diferencia en el rendimiento es mínima, lo que quiere decir que ese 3 % o 4 % de diferencia se reflejaría en unos pocos cuadros por segundos, por esta razón se pueden hacer pruebas sobre cualquier tarjeta gráfica sin que ésta refleje una gran diferencia por ser de un fabricante u otro. Por lo tanto, se realizarán pruebas propias bajo condiciones controladas para tener una mejor visión del rendimiento.

Para realizar las pruebas se utilizó un PC de escritorio con las siguientes características:

- Procesador AMD 8320E@4.0Ghz
- Memoria Ram 4 Gigabyte@1866mhz
- Tarjeta de video AMD R9 380 2 Gigabyte

Todo esto se realizó a una resolución de 1920x1080. En la figura 5.1 se muestran las opciones gráficas que se configuraron; la memoria RAM de la tarjeta de video se queda corta, por lo cual se debieron recortar varios efectos para que el juego pudiera correr; adicionalmente, se le realizó un overclock al procesador, ya que limitaba los cuadros por segundo, por esto en ciertas condiciones los fps promedio pueden ser parecidos. Para las pruebas se corrió el benchmark que tiene incluido el juego y se monitoreó el rendimiento de la máquina, por esta razón se logró visualizar que la única

prueba que no era limitada por el procesador era el test 3, el cual también contaba con muchas luces en escena donde se puede medir el rendimiento de los algoritmos.

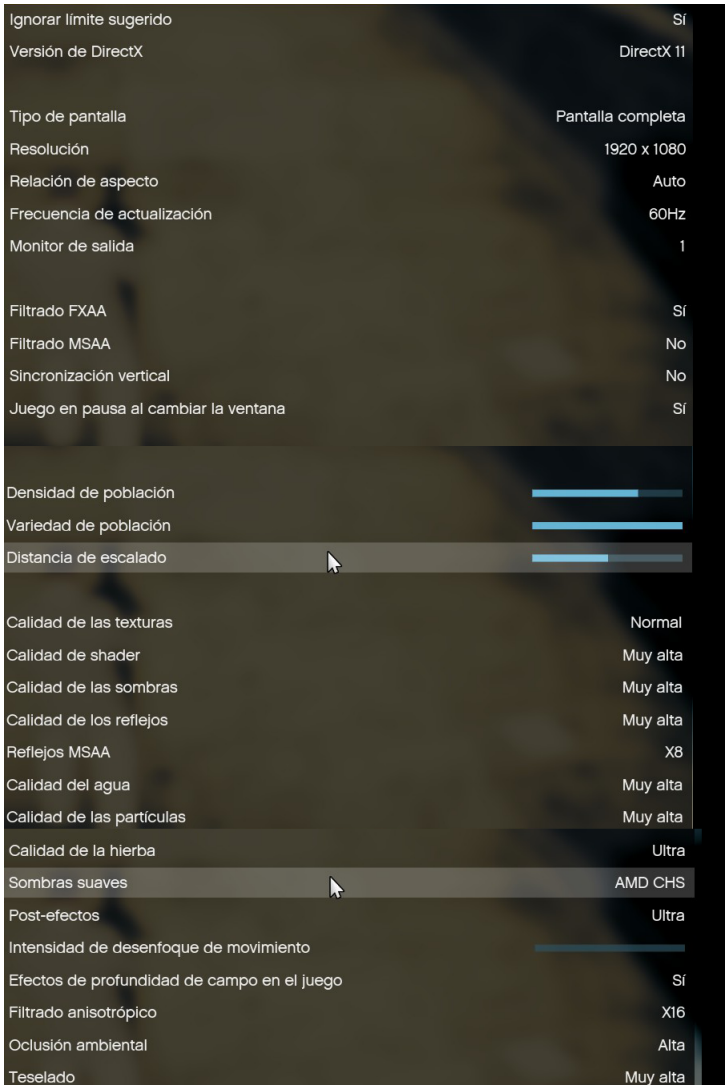


Fig. 5.1: Configuración gráfica del benchmark

Fuente: Videojuego GTA V

En la tabla 5.2 se muestra que en promedio existe una clara pérdida de fps, lo que es previsible por la cantidad de procesamiento extra que se le pedía a la tarjeta gráfica. Los cuadros por segundo máximos son similares, ya que se presentan en casos que la luz no tiene tanta importancia y no logra verse una diferencia clara entre usar las dife-

rentes técnicas. Entre CHS y PCSS existe una caída de 9 % en el rendimiento, que no significa mucho si el juego logra mantener más de 30 cuadros por segundo. Por otra parte, ambas técnicas logran mantener el tiempo entre cuadros bajo los 33 milisegundos en un 99 % de ellos, lo que significa que rara vez existe una caída de cuadros por segundo, además siempre se pueden modificar los ajustes para lograr un buen desempeño. Finalmente, se debe tener en consideración que para estas pruebas se utilizó una tarjeta gráfica de gama media, es decir, con una tarjeta de gama alta se obtendrían mejores resultados, por esta razón ambas técnicas logran un buen desempeño en tiempo real.

Tab. 5.2: Tabla de rendimiento de diferentes tipos de sombras difusas

Fuente: Elaboración propia

	FPS promedio	FPS máximos	Porcentaje de FPS promedio
Sombra dura	71	133	100 %
Sombra difusa CHS	67	134	94,3 %
Sombra difusa PCSS	61	135	85,9 %

5.3. Calidad

Para poder analizar las diferencias entre las sombras y conocer cuál tiene mejor calidad, se utilizarán sombras difusas reales para compararlas a las generadas por estos algoritmos. La fuente de luz que se utilizará para las pruebas será el sol, debido a que estas sombras deben ser similares bajo cualquier condición, ya como el sol siempre tiene el mismo tamaño y se encuentra siempre a la misma distancia, entonces se fotografiarán sombras difusas producidas por el sol y se compararán con las del juego.

Una de las ventajas de analizar las sombras difusas provocadas por el sol es que se puede realizar un estudio exacto de su tamaño, incluso se puede saber el tamaño de la sombra difusa a través de la relación entre el tamaño del sol y de la distancia de este

con la tierra. Para esto, se establece una proporcionalidad, porque si se mantiene la proporción del tamaño y la distancia, se puede seguir realizando el mismo ejercicio, por eso se puede calcular el tamaño de la penumbra que provoca.

El diámetro del sol es de 1.391.600 km., mientras que la distancia del sol a la tierra es de 149.600.000 km., la proporción de las dos medidas es de 107,5, valor que puede ser usado para conocer el tamaño de la penumbra. Esto es posible si se divide ese número por la distancia del oclusor a la superficie; por ejemplo, si la distancia fuera de 1 metro, el tamaño de la penumbra de la sombra difusa sería de 9,3 mm; si fueran 2 metros, el tamaño de la penumbra sería el doble. Estos tamaños son visibles desde una distancia cercana, y es por eso que este tipo de sombras difusas, creadas por el sol, son vistas.



Fig. 5.2: Fotografía a sombras difusas

Fuente : Elaboración propia



Fig. 5.3: Escena con sombras duras

Fuente: Videojuego GTA V



Fig. 5.4: Escena con sombras difusas PCSS

Fuente: Videojuego GTA V



Fig. 5.5: Escena con sombras difusas CHS

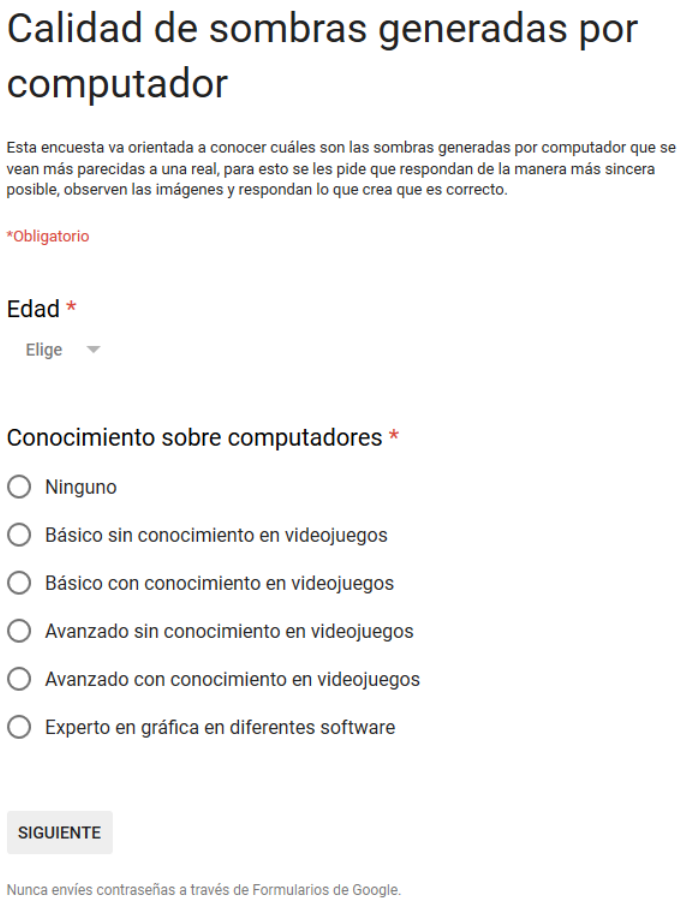
Fuente: Videojuego GTA V

La figura 5.2 pertenece a una fotografía sacada a unas sombras generadas por el sol. La sombra difusa es fácilmente visible, tanto la generada por la persona, como la generada por el árbol. En la sombra del árbol, no se pueden distinguir individualmente las hojas, ya que la penumbra va envolviendo circularmente a la umbra y como son varias, estas penumbras se van sumando y se forman secciones de luz con bordes curvos. En la figura 5.3 aparece una escena del videojuego realizada con sombras duras, el límite de sombra y no sombra es bastante claro debido a que solo tiene una sección de umbra, aunque en el borde se puede apreciar un cierto aliasing, producido por el tamaño del shadow map, si el shadow map creado fuera más grande, los bordes de la sombra pueden ser aún más definidos. Finalmente, las figuras 5.4 y 5.5 muestran sombras difusas utilizando las técnicas PCSS y CHS respectivamente.

En la figura 5.4, la sombra está más difuminada y la penumbra tiene un tamaño más natural, en especial si se observa la sombra del objeto más lejano. La sombra de la izquierda pertenece a una torre de un generador eólico, por lo cual la distancia es de más o menos 30 metros, por consiguiente, el tamaño de la penumbra sería de, al

menos, 28 centímetros, lo que es bastante grande y apreciable.

En la figura 5.5, aún se pueden ver algunos bordes nítidos en donde la sombra es cercana al oclusor, mientras que la penumbra se puede ver mejor cuando la sombra está bastante lejos del oclusor. De este modo, se puede concluir que ambas técnicas logran un buen resultado.



Calidad de sombras generadas por computador

Esta encuesta va orientada a conocer cuáles son las sombras generadas por computador que se vean más parecidas a una real, para esto se les pide que respondan de la manera más sincera posible, observen las imágenes y respondan lo que crea que es correcto.

*Obligatorio

Edad *

Elige

Conocimiento sobre computadores *

- Ninguno
- Básico sin conocimiento en videojuegos
- Básico con conocimiento en videojuegos
- Avanzado sin conocimiento en videojuegos
- Avanzado con conocimiento en videojuegos
- Experto en gráfica en diferentes software

SIGUIENTE

Nunca envíes contraseñas a través de Formularios de Google.

Fig. 5.6: Primera parte de la encuesta sobre sombras difusas

Fuente: Elaboración propia

Para conocer mejor cuáles sombras tienen mejor calidad, se procedió a realizar una encuesta sobre las sombras difusas a varias personas. Esta encuesta es totalmen-

te anónima, pero se hicieron algunas preguntas para conocer el perfil de la persona encuestada. En la figura 5.6, aparecen las preguntas que se formularon en la primera sección, las que tienen relación con algunas características de las personas encuestadas.

La primera pregunta es sobre la edad, ya que la percepción de las sombras puede variar con ella. Para poder comparar la respuesta de personas de diferentes edades, la segunda pregunta es sobre el acercamiento que ha tenido la persona con un computador. Esta pregunta tiene 2 objetivos: el primero, definir el conocimiento de la persona sobre computadores, ya que este dato podría arrojar algunas pistas sobre el acercamiento de la persona con gráficos generados por un computador. El segundo objetivo está relacionado a la insteracción con los videojuegos, ya que es posible que las personas que han interactuado con videojuegos noten ciertas relaciones o vean ciertos patrones en los gráficos que las demás personas no pueden.

En la figura 5.7 está la segunda sección de preguntas. Antes de contestarlas, el encuestado debe ver 4 imágenes: la primera es una fotografía de sombras; la segunda, que es la sombra A, es la imagen del juego con sombras duras; la tercera y cuarta imagen, correspondientes a las sombras B y C respectivamente, son la sombra difusa PCSS y la sombra difusa CHS, aunque en la encuesta no se menciona el método, para que el encuestado no se desvíe de su respuesta. La primera pregunta busca saber si el encuestado logra ver alguna diferencia entre las sombras; la segunda pregunta trata sobre la opinión del encuestado sobre si la sombra le parece natural o similar a una sombra real, mientras que la tercera busca conocer todo lo contrario, es decir, cuál sombra le parece más falsa. al final se deja un espacio de respuesta opcional en donde el encuestado puede expresar un comentario sobre la encuesta o sobre alguna sombra en particular.

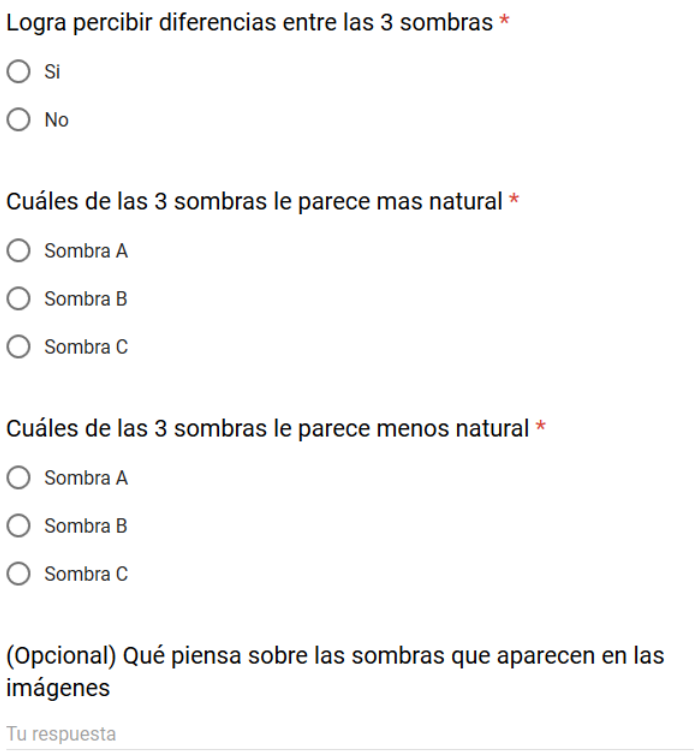


Fig. 5.7: Segunda parte de la encuesta sobre sombras difusas

Fuente: Elaboración propia

La encuesta fue respondida por 11 personas personas entre las edades de 10 a 59 años. Estas personas tienen diferentes experiencias con un computador y con videojuegos; sin embargo, al analizar la segunda sección de la encuesta, las respuestas no varían dependiendo de estas dos características, es decir, dentro de un grupo las respuestas pueden ser bastante heterogéneas y dos personas diferentes pueden tener la misma respuesta. Por estas razones, lo mejor para analizar los datos obtenidos es mirarlos como un todo.

En la primera pregunta sobre las sombras existe una respuesta predominante pero no absoluta, aunque una de las tres va quedando descartada como se ve en la figura 5.8. La sombra PCSS se ubica en primer lugar; la penumbra de esta sombra es bastante

grande, por lo cual, ésta puede ser la razón de que les parezca mas natural, la diferencia no es tan grande con la sombra CHS, en donde la penumbra es más pequeña y por lo tanto no parece haber una opinión tajante sobre cuál sombra difusa es más natural. Lo que sí es claro es que ambas sombras tienen más preferencias que la sombra dura, que apenas aparecen en este gráfico. Esto se refuerza en la figura 5.9, en donde la sombra dura tiene la mayor parte de las preferencias sobre cuál sombra parece menos natural, así que la sombra dura queda relegada frente a las otras dos opciones. No es totalmente importante que las sombras sean exactamente igual a las reales, lo que sí importa es que se parezcan a las reales en sus características, esto se debe a que aunque estén ahí, las sombras se dan por hechas, no se observan en profundidad, sino que siempre que muestren ciertas características, estas parecerán reales.

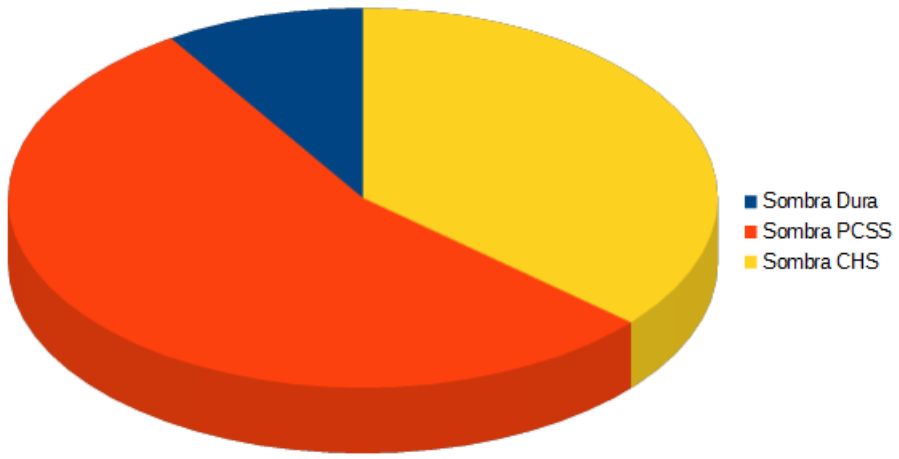


Fig. 5.8: Gráfico sobre las respuestas dadas a la pregunta sobre cuál sombra parece más natural

Fuente : Elaboración propia

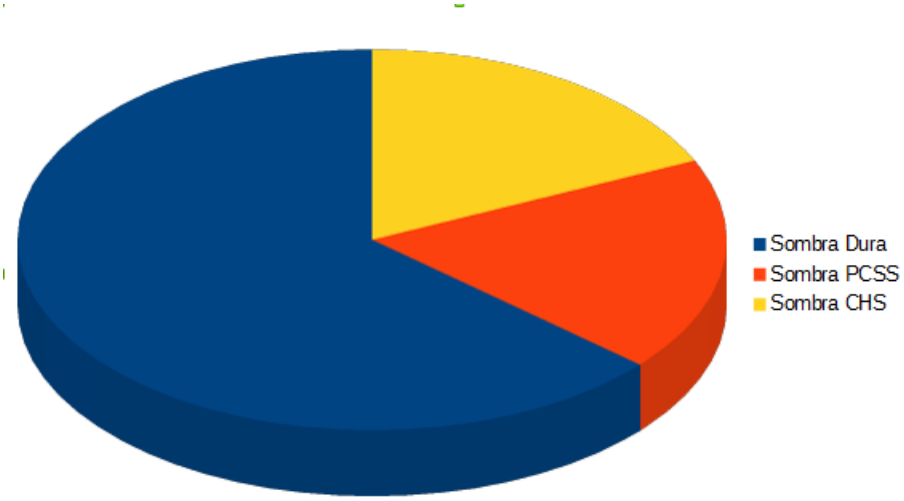


Fig. 5.9: Gráfico sobre las respuestas dadas a la pregunta sobre cuál sombra parece menos natural

Fuente : Elaboración propia

Al parecer, el algoritmo CHS de las sombras difusas selecciona las secciones de sombra a través de un recuadro, pudiendo observarse esta sombra en la imagen lejana. En otro ejemplo, en donde existen dos sombras que se superponen, la sombra de la persona y la sombra de un cerro que está en la lejanía, éstas tienen penumbras de diferente tamaño debido a que se encuentran a diferentes distancias de la superficie, lo que se observa en la figura 5.10. Allí se ve que existe una sección de la sombra del cerro que tiene una penumbra de tamaño similar a la de la persona, mientras que se distingue un límite en donde vuelve a recuperar el tamaño de la sombra del cerro. Esto muestra claramente que el algoritmo divide las secciones de la sombra con tamaño de penumbra fijo, incluso si en esa sección tiene varias sombras que se superponen. Por otro lado, en la figura 5.11, que muestra una sombra PCSS , se ve el gran tamaño de la penumbra debido a que el cerro está muy lejos y que la sombra de la persona solo se ve cuando acaba la penumbra de la sombra del cerro. Esto se puede deber a que el algoritmo toma al objeto más cercano al sol y le aplica un kernel del tamaño que se obtiene por dicha lejanía y solo a esa sombra, ya que si se sumaran las sombras,

la sombra de la persona debería ser visible a través de la penumbra de la sombra del cerro.



Fig. 5.10: Sombra de dos objetos interpuestos utilizando el método CHS

Fuente : Videojuego GTA V



Fig. 5.11: Sombra de dos objetos interpuestos utilizando el método PCSS

Fuente : Videojuego GTA V

5.4. Comparación

En ambos tipos de sombras, existe una pérdida en el rendimiento, aunque no es grave, ya que si el programa tiene una buena cantidad de cuadros por segundos, la

pérdida de un 10 % no debería ser muy significativa y aún sería lo suficientemente bueno para ser observado.

La sombra CHS luce muy bien, existe una difuminación que hace ver bastante definida a la sombra; sin embargo, el tamaño de la penumbra es demasiado pequeño, ya que no logra acercarse al modelo real. Por otra parte, el método no calcula bien la penumbra para todo el borde de la sombra, aunque esto se dejó así, para que el programa tuviera un mejor rendimiento.

La sombra PCSS calcula mejor la penumbra, pues el ancho mostrado se acerca bastante a la realidad, y cada borde se calcula independientemente, Si bien es cierto que la sombra no luce tan bien de cerca, puede verse natural desde una distancia mayor, a cambio de una penalización en el rendimiento.

Estos dos métodos son excelentes, tanto en calidad como en rendimiento. el método PCSS tiene una calidad un poco mayor a costa del rendimiento, en cambio el método CHS tiene una calidad menor, pero se logra un mejor rendimiento. Ambas técnicas logran resultados increíbles a un pequeño costo, gracias a su base común, el método PCF, que logra estos resultados realizando pequeñas modificaciones al algoritmo base de Mapping Shadow.

6. CONCLUSIONES

6.1. Sombras Difusas

El primer acercamiento al tema de esta memoria fue investigar sobre las sombras difusas en Volume Shadow y Mapping Shadow; sin embargo, este acercamiento tuvo un error, ya que no se tomaron en cuenta los métodos basados en PCF. Buena parte de ese tiempo se investigó e intentó ver la implementación de métodos mucho menos eficiente que los métodos actuales, lo que finalmente permitió apreciar la facilidad y el rendimiento que daban los métodos basados en PCF. Fue bastante sorprendente encontrar los algoritmos basados en PCF, ya que el trabajo extra que requerían era mínimo comparado a los resultados que lograban, incluso si se mira el algoritmo PCF por sí solo, logra unos resultados bastante decentes frente a la sombra dura producida por el algoritmo mapping shadow, sin la necesidad de tener un shadow map de alta resolución. Una sombra dura corriente requiere de un shadow map de alta resolución para eliminar el aliasing sobre la superficie receptora, en cambio la técnica de PCF puede tener un shadow map de menor resolución y lograr mejores resultados. Esto conlleva algunas ventajas, primero sobre la memoria gráfica, ya que un shadow map de alta resolución necesita mucha memoria gráfica para ser almacenado y en una escena compleja pueden haber más de un shadow map. La otra ventaja podría estar sobre la creación del shadow map, ya que se necesita realizar un renderizado de escena para crearlo, lo que pone a trabajar al hardware gráfico; por esto, entre mayor sea la resolución del shadow map más poder gráfico será necesario para crearlo.

La técnica de Shadow Mapping se impuso sobre Volume Shadow, principalmente

por la técnica de PCF que decantó en dos implementaciones, siendo el hecho de que hayan aparecido dichas implementaciones es una prueba de la idoneidad del algoritmo PCF. La implementación de las sombras difusas vienen de la mano con dos sucesos: el primero, la aparición de los algoritmos que permiten crear sombras difusas a partir del hardware gráfico, ya que el cálculo puede ser hecho para cada pixel en especial; y el segundo, el aumento de la potencia gráfica que permite realizar este trabajo extra en tiempo real. En estos últimos años, la potencia del hardware gráfico ha dejado de medirse en gigaflops y en su lugar se utiliza el teraflops, ya que la cantidad de núcleos gráficos que tiene cada chip y la frecuencia que estos pueden alcanzar, todo unido a la memoria gráfica que es cada vez más grande y más rápida, permiten ir aún más lejos en lo que se refiere a efectos basados en la luz.

Una de las primicias con que se empezó a investigar las sombras difusas era que el ambiente generado fuera lo más fotorrealista posible, ya que esa es una de las metas en computación gráfica. Esto se vuelve un reto cuando se quieren mover entornos gráficos en tiempo real. Las sombras difusas son un completo desafío, ya que son muchos los factores que participan en su creación. Las técnicas antiguas resolvían el problema creando muchos shadow maps, pero estos shadow maps son pesados y para conseguir un buen resultado se necesitan muchos, lo cual terminaría llenando la memoria. Todo esto desaparece con la llegada de los métodos PCF, ya que se ahorra los recursos de memoria, debido a que utiliza un solo shadow map y es una técnica simple de usar. Las complejidades que puede tener se van agregando con las investigaciones que permitan explotarla mejor, pero más allá del buen resultado, las personas resultaban conformes con que se les diera la sensación de una sombra difusa. Aunque existe preferencia por el método PCSS, no es tan marcada para concluir de que la prefieren sobre el método CHS; por esto, el desarrollo de las sombras difusas tiene que ir más allá de su apariencia, ya que las sombras son un elemento extra que se dan por hechas, no se les da mayor importancia, así que mientras tengan una buena apariencia, se ignorará si están lo suficientemente bien hechas.

6.2. Trabajo a futuro

Los efectos de la luz en los entornos de computación gráfica están evolucionando, no solo las sombras importan, otra técnica que está tomando relevancia es la técnica de oclusión ambiental. Esta es una de las técnicas que más recursos consumen y que más se acerca al fotorrealismo. Existen programas que la usan, pero no son en tiempo real y el resultado es increíble, pero han ido apareciendo técnicas bastante buenas que las usan en computación en tiempo real.

También están apareciendo nuevas APIs gráficas tales como Vulkan o Directx 12, que son más eficientes que las actuales, por lo que podría investigarse la implementación de las sombras sobre dichas APIs y de este modo conocerlas mejor, también existen motores gráficos de fácil acceso que hacen este trabajo mas simple tales como Unreal Engine y Unity, ya que estas técnicas explotan mejor las cualidades de estos motores gráficos.

Bibliografía

[1] Crow, Franklin. Shadow algorithms for computer graphics. ACM SIGGRAPH Computer Graphics. 11(2):242-248, 1977

[2] L. Williams. Casting Curved Shadows on Curved Surfaces. ACM SIGGRAPH Computer Graphics. 12(3):270-274, 1978

[3] M. Segal, C. Korobkin, R. V. Widenfelt, et al. Fast Shadows and Lighting Effects Using Texture Mapping. ACM SIGGRAPH Computer Graphics. 26(2):249-252, 1992

[4] The Cg Tutorial, Chapter 9: Advanced Topics [en línea] <http://http.developer.nvidia.com/CgTutorial/cg_tutorial_chapter09.html>

[5] Heidmann T. Real shadows, real time. Iris Universe, Silicon Graphics Inc. 18:23-31, 1991

[6] Fuchs H, Kedem Z M, Naylor B F. On visible surface generation by a priori tree structures. ACM SIGGRAPH Computer Graphics. 14(3): 124-133, 1980

[7] Chin N, Feiner S. Near real-time shadow generation using BSP trees. ACM Computer Graphics. 23(3): 99-106, 1989

[8] Batagelo H C, Jnior I C. Real-time shadow generation using BSP trees and stencil buffers. In SIBGRAPI. 12: 93-102, 1999

[9] Liu L M y Wu E H. A. Real-Time algorithm of 3-D shadow generation for point light sources. Journal of Software. 11(6):785-790, 2000

[10] McCool M D. Shadow volume reconstruction from depth maps. ACM Transactions on Graphics. 19(1):1-26, 2000

[11] Brabec S, Seidel H P. Shadow volumes on programmable graphics hardware. Computer Graphics Forum. 25(3), 2003

[12] Bilodeau B. Of Creative Labs Inc. Slide printout from a presentation at the Computer Game Developers Conference. 1999

[13] Carmack J. Carmack On Shadow Volumes (Un correo personal entre Carmack y Kilgard). 2002

[14] Brett Keating y Nelson Max. Shadow Penumbras for Complex Objects by Depth-Dependent Filtering of Multi-Layer Depth Images En su : Rendering Techniques 99, España, Springer-Verlag, 1999, pp.205-220

[15] J.M. Hasenfratz, M. Lapierre, N. Holzschuch y F. Sillion. A Survey of Real-time Soft Shadows Algorithms. Computer Graphics Forum. 22(4):753-774, 2003

[16] Steven Parker, Peter Shirley, Brian Smits. Single sample soft shadows. Technical Report UUCS-98-019, Despartamento de las ciencias de la computación, Unversidad de Utah, Octubre 1998.

[17] Eric Haines. Soft planar shadows using plateaus. Journal of Graphics Tools, 6(1): 19-27, 2001

[18] Eric Chan y Fredo Durand. Rendering fake soft shadows with smoothies En su Rendering Techniques 2003, Belgica, Springer-Verlag, 2003.

[19] Tomas Akenine-Möller and Ulf Assarsson. Approximate soft shadows on arbitrary surfaces using penumbra wedges En su Rendering Techniques 2002 (13th Eurographics Workshop on Rendering), Italia, Springer-Verlag, pp. 297-306

[20] Reeves W.T., Salesin D. H. y Cook R. I. Rendering Antialiased Shadows with Depth Maps. En Computer Graphics (SIGGRAPH 1987 Proceedings), Julio 1987, pp. 283-291

[21] GPU Gems, capitulo 11. Shadow Map Antialiasing [en línea] <http://http.developer.nvidia.com/GPUGems/gpugems_ch11.html>

[22] Fernando R. Percentage-Closer Soft Shadows. ACM SIGGRAPH 2005 Sketches, 2005, articulo numero 35

[23] Wyman C y Hansen C D. Penumbra maps:Approximate soft shadows in real-time En su Rendering Techniques, 2003, 202-207

[24] 20th International Conference on Computer Graphics, Visualization and Computer Vision 2012. (20ª, 2012, Pilsen, Republica Checa). Contact Hardening Soft Shadows using Erosion. Eurographics Association. 2012

[25] OpenGL [en línea] <https://www.opengl.org/>

[26] GLEW [en línea] <http://glew.sourceforge.net/>

[27] GLFW [en línea] <http://www.glfw.org/>

[28] GLM [en línea] <http://glm.g-truc.net/0.9.7/index.html>

[29] ASSIMP [en línea] <http://assimp.sourceforge.net/>

[30] GLUT [en línea] <https://www.opengl.org/resources/libraries/glut/>

[31] FreeGLUT [en línea] <http://freeglut.sourceforge.net/>

[32] Grand Theft Auto V Video Card Performance Preview [en línea] <https://m.hardocp.com/article/2015/04/20/grand_theft_auto_v_video_card_performance_preview/6>