



UNIVERSIDADE DA CORUÑA

Facultad de Informática
Departamento de Electrónica y Sistemas

PROYECTO DE FIN DE CARRERA
INGENIERÍA TÉCNICA EN INFORMÁTICA DE SISTEMAS

Visualización avanzada de nubes de puntos con OpenGL

Alumno: Antúnez González, David
Director: Álvarez Mures, Luis Omar
Tutor y Director: Padrón González, Emilio José
Fecha: 20 de septiembre de 2015

D. EMILIO JOSÉ PADRÓN GONZÁLEZ y D. LUIS OMAR ÁLVAREZ MURES
Departamento de Electrónica y Sistemas y Departamento de Tecnologías de la
Información y Comunicaciones
Universidad de A Coruña

CERTIFICAN: Que la memoria titulada *“Visualización avanzada de nubes de puntos con OpenGL”* ha sido realizada por DAVID ANTÚNEZ GONZÁLEZ bajo nuestra dirección y constituye su Proyecto de Fin de Carrera de Ingeniería Técnica en Informática de Sistemas.

En A Coruña, a 20 de septiembre de 2015

D. EMILIO JOSÉ PADRÓN GONZÁLEZ y D. LUIS OMAR ÁLVAREZ MURES
Directores del proyecto

Resumen:

El render de nubes de puntos ha adquirido un renovado interés en los últimos años con la popularización y proliferación de nuevos sistemas de adquisición de datos de coste medio/alto, habitualmente basados en la tecnología LiDAR de escaneo láser, o de bajo coste, empleando fotogrametría o cámaras infrarrojas.

Estos dispositivos obtienen una nube de puntos 3D (posición geométrica), con información adicional arbitraria asociada, normalmente el color como mínimo, aunque también frecuentemente otros datos como normales, temperatura, etc. Por las propias características que tiene el punto como primitiva gráfica, respecto por ejemplo de los tradicionales polígonos habitualmente empleados en informática gráfica (no tienen área, no tienen orientación, no están conectados, etc.), el render o visualización de nubes de puntos presenta una serie de retos si queremos obtener una visualización realista y de calidad.

Este proyecto se centra en la visualización avanzada de nubes de puntos, aplicando algunas de las más novedosas técnicas de render y haciendo uso de las características avanzadas de la API gráfica multiplataforma OpenGL, lo que nos permite explotar el hardware de las tarjetas gráficas modernas. El resultado del proyecto será la implementación de una herramienta multiplataforma para la visualización avanzada de nubes de puntos 3D.

Lista de palabras clave:

OpenGL, nubes de puntos, point-based rendering, LiDAR, deferred shading, gouraud shading, phong shading.

Buffalo buffalo Buffalo buffalo buffalo buffalo Buffalo buffalo.

Agradecimientos

A Ángela la persona a la que le dedico y dedicaré siempre.

A mi familia que me acompañó y pagó mis matrículas todos estos años y con las que se podrían haber comprado un piso en Marbella pero prefirieron que su hijo estudiara.

A Emilio y Omar, a los que considero amigos y que sin su ayuda quizás este instante de escribir dedicatorias no hubiese existido jamás. Gracias :)

Índice general

Introducción	xxi
1. Planificación y metodología	1
1.1. Desarrollo ágil de software	1
1.1.1. Agile Manifesto	2
1.1.2. Principios Ágiles	2
1.1.3. Aplicación	3
2. Fundamentos tecnológicos	9
2.1. Nubes de puntos	9
2.1.1. Adquisición	10
2.1.2. Visualización	11
2.2. OpenGL	13
2.2.1. Pipeline	14
2.2.2. Shaders	15
2.3. PCL	16
3. Fundamentos de la Programación Gráfica	19
3.1. Render	19
3.1.1. Modelos 3D	21
3.2. Espacios 3D y transformaciones	21
3.3. Camera model	23

3.3.1. Parámetros de la cámara	24
3.3.2. Funcionamiento interno de la cámara	25
4. Análisis y diseño de un visualizador	29
4.1. Análisis de requisitos	29
4.1.1. Casos de uso	30
4.2. Diagrama de clases	30
5. Visualización de nubes de puntos en GPU	33
5.1. Primitivas en nubes de puntos	33
5.1.1. Surfel	33
5.1.2. Splat	33
5.2. Técnicas de rasterización	34
5.2.1. Sized-Fixed Points	34
5.2.2. Image-aligned Squares	35
5.2.3. Affinely Projected Point Sprites	36
5.2.4. Perspective Correct Rasterization	37
5.3. Técnicas de Blending	39
5.3.1. Flat Shading	39
5.3.2. Gouraud Shading	40
5.3.3. Phong Shading	43
5.3.4. Deferred Shading	44
6. Resultados y rendimiento	47
6.1. Preparación de la prueba	47
6.2. Análisis de los métodos de rasterización	48
6.3. Análisis de los métodos de blending	51
6.3.1. Comparación de los sistemas de blending con puntos de luz	52
7. Conclusiones	55

7.1. Conclusiones	55
7.2. Posibles vías de desarrollo	56
A. Shaders	57
A.1. Sized-Fixed Points	57
A.2. Image-aligned Squares	58
A.3. Affinely Projected Point Sprites	59
A.4. Perspective Correct Rasterization	61
A.5. Gouraud	63
A.5.1. Visibility Pass	63
A.5.2. Blending Pass	65
A.5.3. Normalization Pass	68
A.6. Phong	69
A.6.1. Visibility Pass	69
A.6.2. Blending Pass	71
A.6.3. Normalization Pass	74
A.7. Deferred	75
A.7.1. Visibility Pass	75
A.7.2. Blending Pass	77
A.7.3. Normalization Pass	79

Índice de figuras

1.1.	Diagrama de Gant (I).	6
1.2.	Diagrama de Gant (II).	7
2.1.	Mapa de desviaciones de una pieza con respecto a un modelo <i>CAD</i> en un proceso industrial.	10
2.2.	Estación Leica LIDAR utilizado para el escaneo de edificios, formaciones rocosas, etc. con el objetivo de generar modelos 3D.	12
2.3.	Captura de pantalla de CloudCompare.	13
2.4.	Captura de pantalla de MeshLab.	14
2.5.	El pipeline de OpenGL, las fases es negrita son programables.	15
2.6.	Evolución del número de transistores en GPU y CPU (1999 - 2010).	16
3.1.	Fotorealistic (izquierda) vs. No fotorealísticos (derecha).	20
3.2.	Funcionamiento de una cámara estenopeica.	23
3.3.	Representación de las diferencias entre cámara en perspectiva (izquierda) y ortográfica (derecha).	25
3.4.	La matriz P se encarga de transformar la pirámide de visualización en un cubo unitario.	28
4.1.	Casos de uso de la aplicación.	31
4.2.	Diagrama de clases del visualizador de nubes de puntos.	32
5.1.	Representación de un <i>surfel</i> de una esfera.	34
5.2.	Construcción de un <i>splat</i>	34

5.3.	Todos los puntos de la nube son renderizados con un cuadrado del mismo tamaño para toda la nube.	35
5.4.	Dragon, modelo de 845,281 puntos renderizado mediante Affinely Projected Point Sprites. Este método causa huecos en ángulos extremos.	38
5.5.	Proyección y Ray Casting.	38
5.6.	Suzanne, modelo de 7,958 puntos renderizado mediante Perspective Correct Rasterization.	40
5.7.	Lucy, modelo de 397,664 puntos. Renderizado mediante <i>Flat Shading (izquierda)</i> , <i>Gouraud Shading (centro)</i> , y <i>Phong Shading (derecha)</i>	41
5.8.	Los tres pasos, <i>visibility</i> , <i>blending</i> , y <i>normalization</i>	42
5.9.	Los vectores n_i son representados como puntos homogéneos $(x,y,1)$ en un plano desplazado tangente.	43
5.10.	El renderizado mediante <i>Deferred Shading</i> acumula atributos como color y normales, seguido por una normalización y un shading pass.	44
6.1.	Gráfica comparativa de los 4 sistemas de rasterización.	50
6.2.	Comparativa de los resultados de render. <i>Sized-fixed (izquierda)</i> , <i>Square-aligned (centro izquierda)</i> , <i>Affinely (centro derecha)</i> y <i>Perspective Correct (derecha)</i>	50
6.3.	Gráfica comparativa de los 4 sistemas de blending.	51
6.4.	Comparación de los diferentes sistemas de blending, comparando luces con millones de puntos: flat y Gouraud.	53
6.5.	Comparación de los diferentes sistemas de blending, comparando luces con millones de puntos: Phong y Deferred.	54

Índice de cuadros

1.1.	Principales tareas llevadas a cabo en el proyecto.	5
1.2.	Estimación de coste.	5
6.1.	Resumen de las características del equipo de Test.	47
6.2.	Los modelos utilizados en el test.	48
6.3.	Formato del fichero de texto con los <i>logs</i> generados.	48
6.4.	Herramienta de apoyo en línea de comandos implementada para automatizar la creación de gráficas de resultados a partir del análisis de las ejecuciones de las pruebas.	49

Introducción

Cada vez más, la gestión y visualización de grandes *datasets* basados en nubes de puntos está adquiriendo una mayor relevancia con la popularización de distintos métodos de escaneo tridimensional, lo que además ha ayudado a que hoy sea mas económico y sencillo obtener estos escaneos. Curiosamente, a pesar de que el resultado de estos métodos de escaneado son nubes de puntos, es destacable la todavía carencia de aplicaciones potentes que trabajen de forma nativa con este tipo de datos [11].

El uso de este tipo de tecnologías está en la actualidad muy presente en diferentes aplicaciones, desde distintos campos de la Ingeniería y la Arquitectura, como la topografía, la ingeniería civil, los sistemas de información geográfica (GIS) o el control industrial; hasta su empleo en arqueología y la propia industria del entretenimiento, pasando también incluso por su aplicación en medicina, sobre todo en lo referente al diagnóstico médico.

Toda la gestión del gran volumen de datos que supone trabajar con el resultado de un escaneo 3D supone ya un problema en si mismo, con soluciones dentro del ámbito de la computación de altas prestaciones (HPC, *High Performance Computing*). No obstante, en este proyecto nos centraremos en la resolución del también importante problema de la visualización de este tipo de datos [8].

La «reconstrucción» de un objeto escaneado para poder ser *renderizado* presenta toda una serie de dificultades, motivadas por la falta de relación que los puntos tienen entre si. Así, nos encontramos con que, a día de hoy, en general, en muchas de las aplicaciones en las que es necesaria la visualización de estos datos se continúan usando mallas de triángulos, generadas a partir del análisis de las nubes de puntos, con los consiguientes errores debidos al ruido en los muestreos y la construcción de una topología poligonal, ante la ausencia de herramientas adecuadas para el trabajo con la primitiva punto.

En este proyecto, se presentan diferentes opciones para la visualización de nubes de puntos y la reconstrucción de su superficie sin necesidad de generar malla de polígonos, aprovechándose del *hardware* gráfico actual. Todas las técnicas empleadas forman parte del estado del arte más novedoso en el campo del *point-based rendering* y han sido imple-

mentadas usando C++ y la API gráfica multiplataforma OpenGL, lo que ha permitido obtener una herramienta portable y fácilmente extensible.

Contexto y Objetivos

Los objetivos concretos que cubre la elaboración de este Proyecto Fin de Carrera son:

1. Aprendizaje de la API gráfica OpenGL, que permite un *render* interactivo multiplataforma.
2. Visualización básica: Desarrollo de un visualizador multiplataforma de nubes de puntos 3D. Se hará una primera versión básica que irá evolucionando durante todo el proyecto, incorporando las distintas técnicas que se vayan estudiando e implementando.
3. Implementación de técnicas avanzadas de visualización de nubes de puntos:
 - a) Basadas en *rasterización*: discos orientados, corrección de perspectiva
 - b) Basadas en *blending*, para conseguir un acabado suave en el *render*
4. Propuesta de un algoritmo novedoso para la visualización avanzada de puntos con blending en un único pase, frente a los habituales 3 pasos de las técnicas de blending más usadas
5. Integración del visualizador con la librería PCL, especializada en la gestión de nubes de puntos. Esto nos permitirá acceder a un importante abanico de métodos y estructuras para el trabajo eficiente con grandes conjuntos de datos basados en la primitiva punto.

Estructura de la memoria

La memoria está constituida por un total de 7 capítulos, además de esta introducción, en los que se desarrollan los siguientes contenidos:

Capítulo 1. Se detalla la planificación y la metodología de desarrollo usadas en la realización del proyecto. Se introducen los métodos ágiles de desarrollo de *software* y se presenta su aplicación en este caso concreto. Además, se describe cómo fue el desarrollo del trabajo con un «histórico» en un diagrama de *Gantt*, incluyendo también una estimación de presupuesto para un proyecto como este.

Capítulo 2. En este capítulo se explican algunos de los principales fundamentos tecnológicos necesarios para comprender el proyecto, empezando por una descripción de la naturaleza de las nubes de puntos, la API OpenGL y la librería PCL.

Capítulo 3. Se introducen los fundamentos básicos de la programación gráfica: tipos de *render*, modelos 3D, espacios Euclídeos y transformaciones, cámaras y la naturaleza matricial de todas las matemáticas que hay detrás de una visualización 3D.

Capítulo 4. Se muestra el análisis y diseño realizado de la herramienta necesaria, previo a la construcción del visualizador 3D.

Capítulo 5. Este capítulo constituye el núcleo de este trabajo, presentándose los diferentes algoritmos y métodos que se han implementado para el dibujado de las nubes de puntos en pantalla, describiendo tanto algoritmos de rasterización como de *blending*.

Capítulo 6. En este capítulo se analizan los resultados obtenidos de la ejecución de los diferentes algoritmos presentados en el anterior capítulo, tanto en términos de calidad de los *renders* obtenidos como de rendimiento.

Capítulo 7. Por último, se muestran las principales conclusiones alcanzadas tras la finalización del proyecto, así como posibles líneas futuras a explorar como continuación del trabajo realizado.

INTRODUCCIÓN

Capítulo 1

Planificación y metodología

Este capítulo explica la metodología escogida para el desarrollo de este proyecto. Al tener este una importante componente investigadora, que complicaba una planificación y seguimiento tradicionales, basados en metodologías clásicas, se pensó que encajaría perfectamente un sistema ágil como el que a continuación se describe para el desarrollo del proyecto. Se incluye también un diagrama de *Gantt* con los plazos e hitos que se siguieron en este trabajo.

1.1. Desarrollo ágil de software

El desarrollo ágil de software refiere a métodos de ingeniería del software basados en el desarrollo iterativo e incremental, donde los requisitos y soluciones evolucionan mediante la colaboración de grupos auto organizados y multidisciplinarios [2]. Existen muchos métodos de desarrollo ágil; la mayoría minimiza riesgos desarrollando software en lapsos cortos. El software desarrollado en una unidad de tiempo es llamado una iteración, la cual debe durar de una a cuatro semanas. Cada iteración del ciclo de vida incluye: planificación, análisis de requisitos, diseño, codificación, revisión y documentación. Una iteración no debe agregar demasiada funcionalidad para justificar el lanzamiento del producto al mercado, sino que la meta es tener una «demo» (sin errores) al final de cada iteración. Al final de cada iteración el equipo vuelve a evaluar las prioridades del proyecto.

Los métodos ágiles enfatizan las comunicaciones cara a cara en vez de la documentación. La mayoría de los equipos ágiles están localizados en una simple oficina abierta, a veces llamadas «plataformas de lanzamiento» (*bulpen* en inglés). La oficina debe incluir revisores, escritores de documentación y ayuda, diseñadores de iteración y directores de proyecto. Los métodos ágiles también enfatizan que el software funcional es la primera

medida del progreso. Combinado con la preferencia por las comunicaciones cara a cara, generalmente los métodos ágiles son criticados y tratados como «indisciplinados» por la falta de documentación técnica.

Los métodos de desarrollo ágiles e iterativos pueden ser vistos como un retroceso a las prácticas observadas en los primeros años del desarrollo de software (aunque en ese tiempo no había metodologías formales). Inicialmente, los métodos ágiles fueron llamados métodos de «peso liviano».

1.1.1. Agile Manifesto

En el año 2001, miembros prominentes de la comunidad se reunieron en Snowbird, Utah, para discutir métodos de desarrollo ligero, publicando lo que denominaron *Manifesto for Agile Software Development* y sentando las bases de lo que ahora se conoce como Desarrollo ágil de software. Las principales ideas sobre las que se sostiene son:

- **Individuos e interacciones** auto-organización y la motivación son importantes. Otros valores promovidos son la co-ubicación y la programación en parejas.
- **Software funcionando** priorizando el tener *demos* funcionales frente a documentación.
- **Colaboración con el cliente** los requerimientos técnicos muchas veces no pueden ser definidos al principio del ciclo de desarrollo, por lo que la continua participación con el cliente es muy importante
- **Respuesta ante el cambio** centrándose en respuestas rápidas a los cambios y el desarrollo continuo.

1.1.2. Principios Ágiles

Este manifesto está basado en los siguientes 12 principios:

- Nuestra mayor prioridad es satisfacer al cliente mediante la entrega temprana y continua de software con valor.
- Aceptamos que los requisitos cambian, incluso en etapas tardías del desarrollo. Los procesos Ágiles aprovechan el cambio para proporcionar ventaja competitiva al cliente.

- Entregamos software funcional frecuentemente, entre dos semanas y dos meses, con preferencia al periodo de tiempo más corto posible.
- Los responsables de negocio y los desarrolladores trabajamos juntos de forma cotidiana durante todo el proyecto.
- Los proyectos se desarrollan en torno a individuos motivados. Hay que darles el entorno y el apoyo que necesitan, y confiarles la ejecución del trabajo.
- El método más eficiente y efectivo de comunicar información al equipo de desarrollo y entre sus miembros es la conversación cara a cara.
- El software funcionando es la medida principal de progreso.
- Los procesos Ágiles promueven el desarrollo sostenible. Los promotores, desarrolladores y usuarios debemos ser capaces de mantener un ritmo constante de forma indefinida.
- La atención continua a la excelencia técnica y al buen diseño mejora la Agilidad.
- La simplicidad, o el arte de maximizar la cantidad de trabajo no realizado, es esencial.
- Las mejores arquitecturas, requisitos y diseños emergen de equipos auto-organizados.
- A intervalos regulares el equipo reflexiona sobre cómo ser más efectivo para a continuación ajustar y perfeccionar su comportamiento en consecuencia.

1.1.3. Aplicación

En gran parte del proyecto fueron aplicadas premisas e ideas propuestas en este *Agile Manifesto*. Se hicieron reuniones semanales en las que se discutían las siguientes tareas a llevar a cabo, dividiendo objetivos principales en tareas simples que no supusiesen más de 2 semanas de desarrollo. El trabajo en cada iteración era constante, centrándose siempre en completar la tarea y no asumiendo cambios si no eran profundamente necesarios.

El diseño incremental también estuvo fuertemente presente, puesto que en cada momento el diseño era revisado para adaptarlo a los nuevos requisitos.

Las revisiones con las dos personas a cargo de la dirección de proyecto fueron, siempre que fue posible, cara a cara. En cada reunión se comentaba qué objetivos habían sido cumplidos desde la última vez, y qué objetivos se intentarían abordar en la siguiente iteración. Cuando estas reuniones no eran posibles, sistemas como el correo electrónico o plataformas de *chat* como *Hangouts* o *Telegram* fueron utilizados.

Para una adecuada gestión de las nuevas versiones producidas con la incorporación de las nuevas características introducidas en la aplicación durante cada iteración se empleó un sistema de control de versiones moderno, libre y distribuido: **Git**. Este sistema de administración de código fuente y distribución permite un trabajo eficiente con múltiples ramas de desarrollo, lo que resultó muy útil para el proyecto, facilitando la creación de nuevas ramas paralelas a la versión estable, con pruebas para nuevos algoritmos. Además, para mantener el código accesible en todo momento, desde el comienzo del desarrollo se utilizó un servicio de **Git** en la «nube», *github*¹, lo que no solo favoreció una interacción fluida con los directores, sino que sirvió además como un sistema de *backup* que permitía también llevar el seguimiento del desarrollo. Destacar también el aprovechamiento de la herramienta de integración continua **travis**², también en la «nube», que ayudó a mantener versiones compilables en las distintas plataformas del repositorio en *github*.

A pesar de que el evidente componente de investigación que tiene este proyecto, en el que se van a emplear técnicas que son ahora mismo estado del arte en computación gráfica, imposibilita en gran medida una adecuada estimación de tiempo por cada iteración, el diagrama de *Gantt* de las Figuras 1.1 y 1.2 viene a resumir los tiempos que llevó cada iteración. La tabla del Cuadro 1.1 muestra las principales tareas llevadas a cabo en esas interacciones.

En el siguiente diagrama de *Gantt* (ver Figuras 1.1 y 1.2) se pueden ver las tareas que fueron definidas en cada una de las iteraciones, siendo las más importantes las que hacen referencia a la implementación del algoritmo *Perspective Correct* y *Gouraud Shading* en el que hubo que hacer modificaciones en el visualizador bastante críticas para poderlo adaptarlo a los requisitos de estos algoritmos.

Se tiene además un presupuesto (ver Tabla 1.2) para un proyecto como este. Incluyendo únicamente costes humanos y *hardware*, ya que el *software* fue siempre gratuito. El precio por hora del ordenador fue calculado pensando en una amortización de 5 años fiscales para el equipo.

¹<https://github.com/eipporko/Cube>

²<https://travis-ci.org/eipporko/Cube>

Tarea	Fecha de inicio	Fecha de fin
Creación de prototipo	1/05/15	5/05/15
Configuración del entorno de trabajo + CMake	1/05/15	1/05/15
Inicialización de contexto	4/05/15	4/05/15
Test de OpenGL + Shader básico	5/05/15	5/05/15
Inclusión de una cámara orbital	6/05/15	7/05/15
Instalación de GLM	6/05/15	6/05/15
Implementación de la cámara	7/05/15	7/05/15
Sized-Fixed Point	8/05/15	11/05/15
Generación de buffers en la GPU	8/05/15	8/05/15
Clase VAO + Shader Sized-Fixed Point	11/05/15	11/05/15
Port de GLUT a GLFW	12/05/15	12/05/15
Image-aligned Squares Shader	13/05/15	19/05/15
Sistema de cambio de Shader	20/05/15	20/05/15
Affinely Projected Point Sprites Shader	21/05/15	25/05/15
Soporte con PCL	26/05/15	27/05/15
Carga de formatos .PLY y .PCD	26/05/15	26/05/15
Refactorización hacia estructura cloud de PCL	26/05/15	27/05/15
Perspectively Correct Rasterization Shader	28/05/15	10/06/15
Radio del splat variable según vecindad	11/06/15	12/06/15
Gouraud Shading	15/06/15	2/07/15
Añadir al pipeline opción a sistema MultiPass	15/06/15	18/06/15
Gouraud Shading Shader	19/06/15	2/07/15
Implementación de FXAA	15/06/15	2/07/15
Gouraud Shading	3/07/15	6/07/15
Phong Shading Shader	7/07/15	13/07/15
Deferred Shading Shader	14/07/15	22/07/15
Luz orbital	23/07/15	23/07/15

Cuadro 1.1: Principales tareas llevadas a cabo en el proyecto.

Recurso	Cantidad	Horas	Coste por hora	Total
Software				0€
Ordenador	1	480	0,1372€	65,86€
Analista-Programador	1	480	15€	7200€
				7265,86€

Cuadro 1.2: Estimación de coste.

1.1. DESARROLLO ÁGIL DE SOFTWARE

PLANIFICACIÓN Y METODOLOGÍA

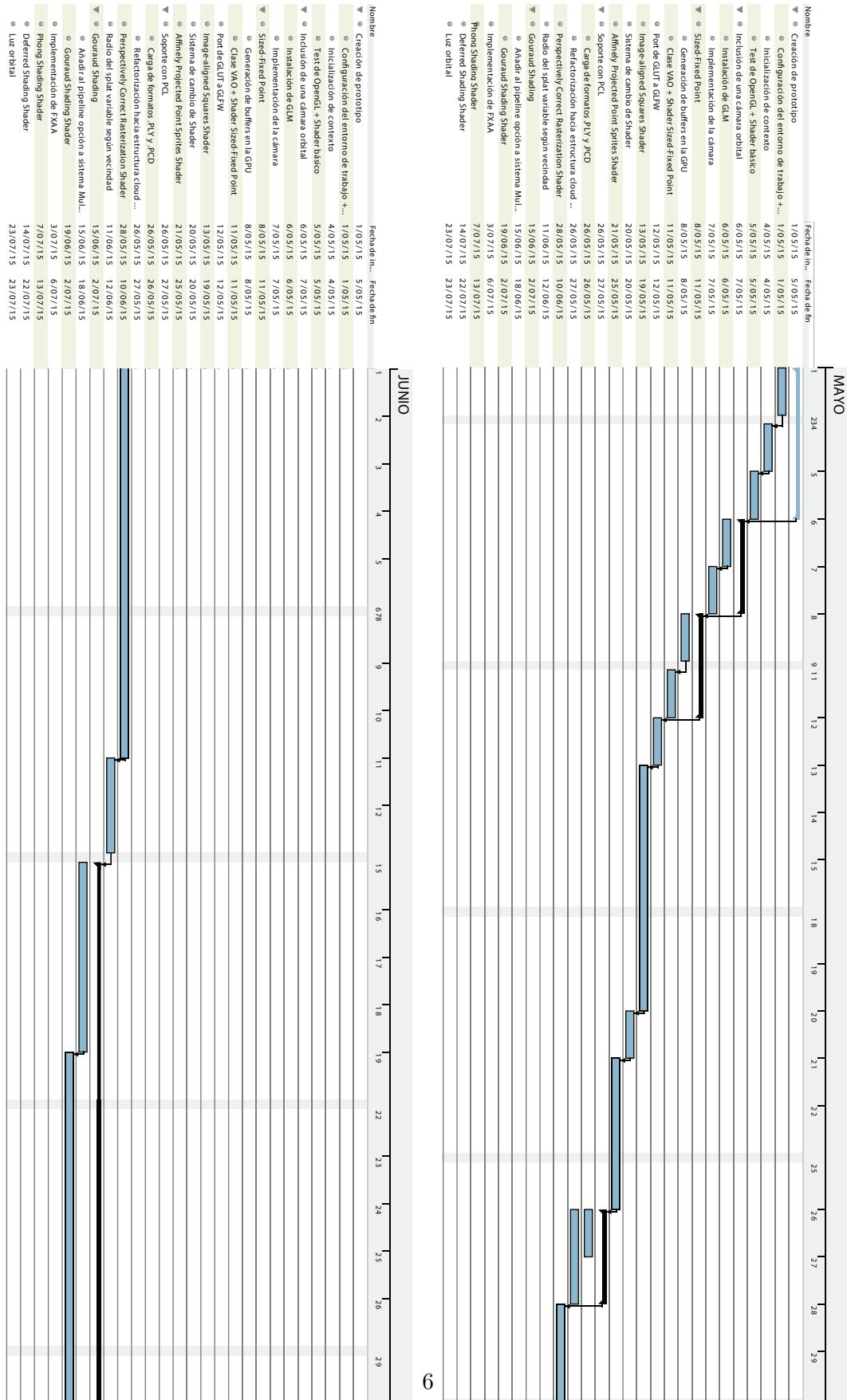


Figura 1.1: Diagrama de Gant (I).

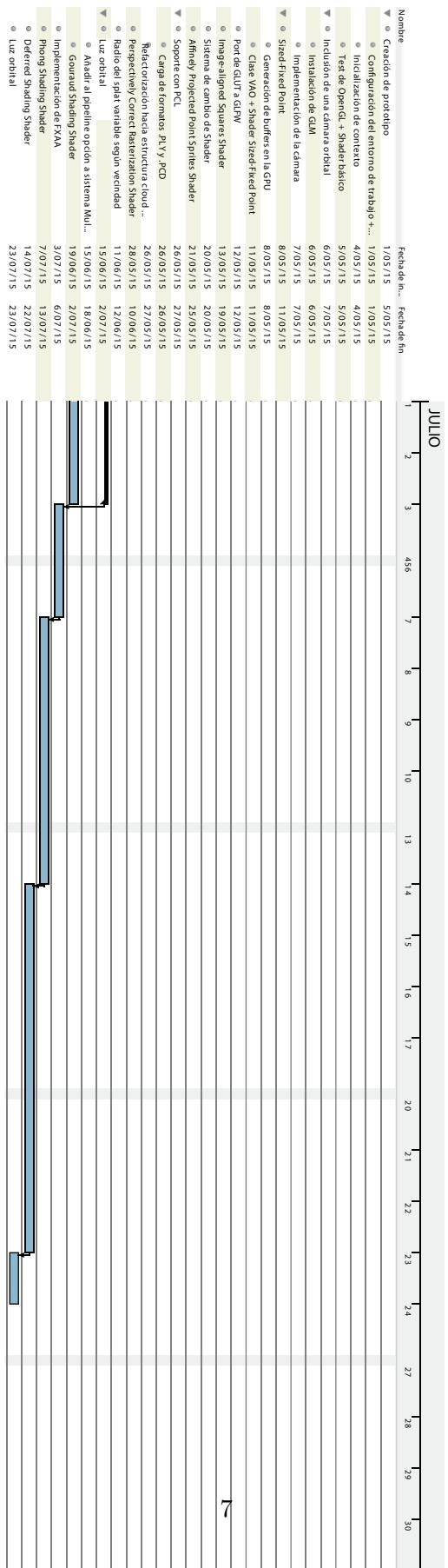


Figura 1.2: Diagrama de Gant (II).

Capítulo 2

Fundamentos tecnológicos

En este capítulo se detallan los principales fundamentos tecnológicos que están detrás de este trabajo. Así, el capítulo comienza describiendo el dominio en el que se enmarca el proyecto: el trabajo con nubes de puntos y las principales técnicas de adquisición para obtenerlas. A continuación, se lleva a cabo una introducción a la API multiplataforma para gráficos interactivos OpenGL, empleada para la visualización de todas las técnicas implementadas en el trabajo. Por último, se presenta la librería PCL, software libre que ha cogido bastante empuje en los últimos años en el campo de las nubes de puntos. La integración de nuestro visualizador con este software, permitiéndonos así aprovechar sus métodos para la realización de diferentes operaciones sobre la nube de puntos, no solo supone una importante ventaja en cuanto a ahorro y reutilización de código, sino que también otorga a este trabajo una importante proyección futura.

2.1. Nubes de puntos

Una nube de puntos es un conjunto de puntos en un sistema de coordenadas tridimensional. Estos puntos se identifican habitualmente como coordenadas X, Y, y Z y son representaciones de la superficie externa de un objeto.

Las nubes de puntos tienen múltiples aplicaciones, entre las que se incluyen la elaboración de modelos tridimensionales en CAD de piezas fabricadas, la inspección de calidad en metrología, y muchas otras en el ámbito de la visualización, animación, texturización y aplicaciones de personalización masiva.

Aunque las nubes de puntos se pueden revisar y texturizar directamente, habitualmente no se utilizan de esta forma en la mayoría de las aplicaciones tridimensionales, ya que se convierten en modelos de mallas poligonales o mallas triangulares irregulares, modelos de

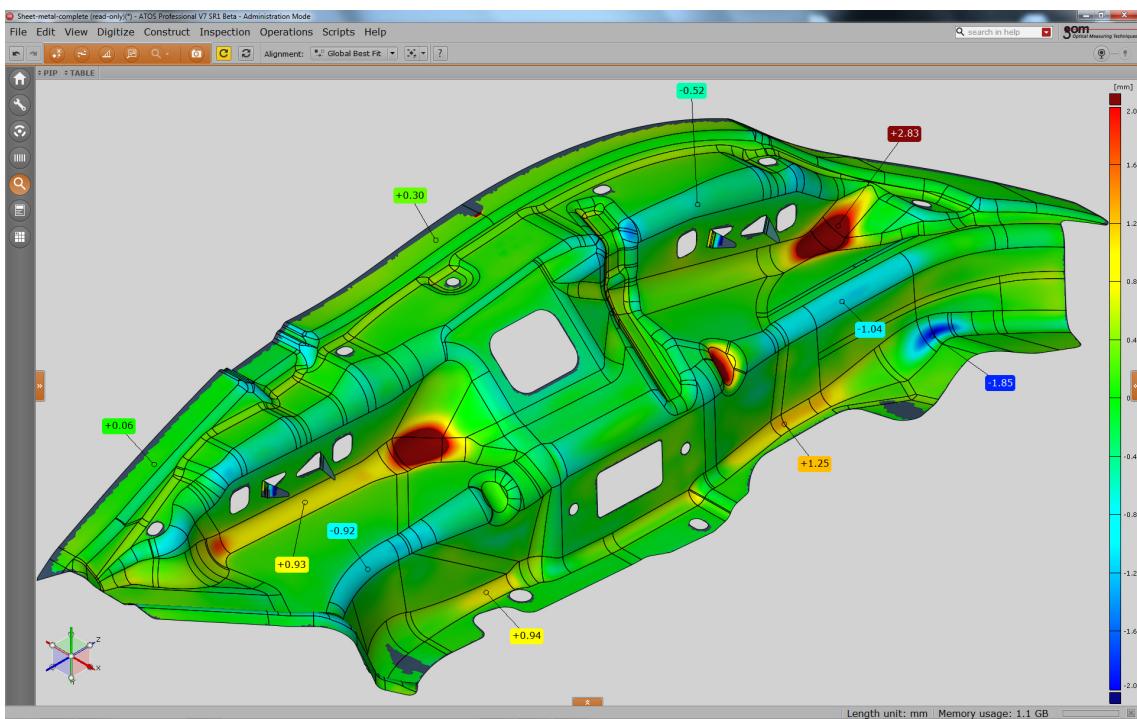


Figura 2.1: Mapa de desviaciones de una pieza con respecto a un modelo *CAD* en un proceso industrial.

superficie *NURBS*, o modelos de *CAD* mediante un proceso denominado reconstrucción de superficies.

Una aplicación en la que las nubes de puntos se utilizan directamente es la metroología y en la inspección industrial (ver Figura 2.1). La nube de una pieza fabricada se alinea a un modelo CAD (o a otra nube de puntos) y se comparan para verificar las diferencias. Éstas se visualizan como mapas de color que proporcionan un indicador visual de la desviación entre la pieza fabricada y el modelo CAD. Las dimensiones geométricas y tolerancias también se pueden extraer directamente de la nube de puntos.

Las nubes de puntos también se pueden utilizar para representar datos volumétricos como en la imagen médica. Se logra usando nubes de puntos de muestreo múltiple y compresión de datos.

En los Sistemas de Información Geográfica *GIS* las nubes de puntos son una de las fuentes de datos para construir modelos digitales del terreno.

2.1.1. Adquisición

Para la adquisición de estas nubes de puntos, existen actualmente diferentes métodos:

Escáner 3D. Un escáner 3D es un dispositivo que analiza un objeto o una escena para reunir datos de su forma y ocasionalmente su color. Diferenciando dos tipos de escáneres en función de si existe contacto con el objeto o no y dividiendo estos últimos en otras dos categorías (activos y pasivos).

- **Contacto** Los escáneres 3D examinan el objeto apoyando el elemento de medida (palpador) sobre la superficie del mismo, típicamente una punta de acero duro o zafiro. Una serie de sensores internos permiten determinar la posición espacial del palpador. Se usan en su mayoría en control dimensional en procesos de fabricación y pueden conseguir precisiones típicas de 0,01 mm.

▪ **Sin contacto**

- **Activos** Los escáneres activos (ver Figura 2.2) emiten alguna clase de señal y analizan su retorno para capturar la geometría de un objeto o una escena. Se utilizan radiaciones electromagnéticas (desde ondas de radio hasta rayos X) o ultrasonidos.
- **Pasivos** Los escáneres pasivos no emiten ninguna clase de radiación por sí mismos, pero en lugar se fía de detectar la radiación reflejada del ambiente. La mayoría de los escáneres de este tipo detectan la luz visible porque es una radiación ya disponible en el ambiente. Otros tipos de radiación, tal como el infrarrojo podrían ser utilizados también. Los métodos pasivos pueden ser muy baratos, porque en la mayoría de los casos estos no necesitan *hardware* particular.

Fotogrametría. La fotogrametría es una técnica para determinar las propiedades geométricas de los objetos y las situaciones espaciales a partir de imágenes fotográficas. Básicamente, es una técnica de medición de coordenadas 3D, que utiliza fotografías u otros sistemas de percepción remota junto con puntos de referencia, como medio fundamental para la medición.

2.1.2. Visualización

En el mercado existen varios programas para la visualización y procesamiento de estos datos, los más conocidos pueden ser *CloudCompare*¹ y *MeshLab*².

¹<http://www.danielgm.net/cc/>

²<http://meshlab.sourceforge.net/>



Figura 2.2: Estación Leica LIDAR utilizado para el escaneo de edificios, formaciones rocosas, etc. con el objetivo de generar modelos 3D.

2.1.2.1. CloudCompare

Originalmente creado durante una colaboración entre Telecom ParisTech y la division R&D de EDF, el proyecto comenzó como tesis de doctoramiento de Daniel Girardey-Montaut en la detección del cambio en información geométrica 3d. [7] al mismo tiempo, su principal objetivo era la detección del cambio rápidamente en nubes de puntos de alta densidad adquiridas mediante escáneres 3D en instalaciones industriales (como plantas de energía). Hoy en día es un proyecto independiente de *software open source* y libre

CloudCompare provee de un conjunto de herramientas para la edición manual y renderizado de nubes de puntos y mayas triangulares. Además también ofrece diversos algoritmos de procesado avanzado. Está disponible para plataformas *Windows*, *Linux*, y *Mac OS X* tanto arquitecturas de 32 como de 64 bits. Es desarrollado en C++ con *Qt*³.

Con todo esto, la visualización de las nubes de puntos es muy básica

³<http://www.qt.io/developers/>

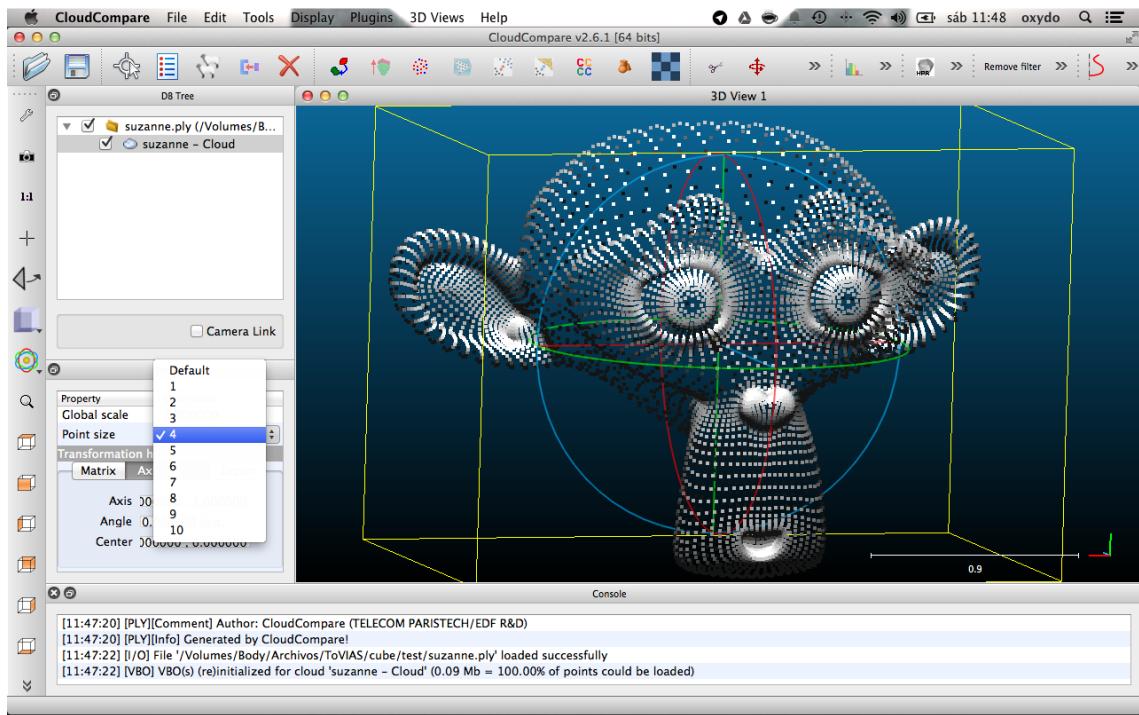


Figura 2.3: Captura de pantalla de CloudCompare.

2.1.2.2. MeshLab

Desarrollado por el centro de investigación INSTI - CNR, *MeshLab* es un *software* de procesamiento avanzado de malla 3D el cual es bien conocido en la mayor de los campos de desarrollo 3D y el manejo de datos.

Está enfocado al procesamiento y gestión de grandes nubes de puntos resultado de escaneo tridimensional y ofrece conjuntos de herramientas para la edición, limpieza, rendering y relleno de este tipo de datos.

Es multiplataforma, *open source* y libre, bajo la licencia de uso *GNU General Public License (GPL)*, versión 2 o posterior, y es usado tanto como aplicación como librería para dar soporte otro *software*.

2.2. OpenGL

OpenGL (Open Graphics Library)⁴ [13] es una librería multiplataforma destinada al renderizado interactivo 2D y 3D. Esta API es típicamente usada para trabajar con la Unidad Gráfica de Procesado (GPU) para conseguir un renderizado acelerado mediante

⁴<https://www.khronos.org/opengl>

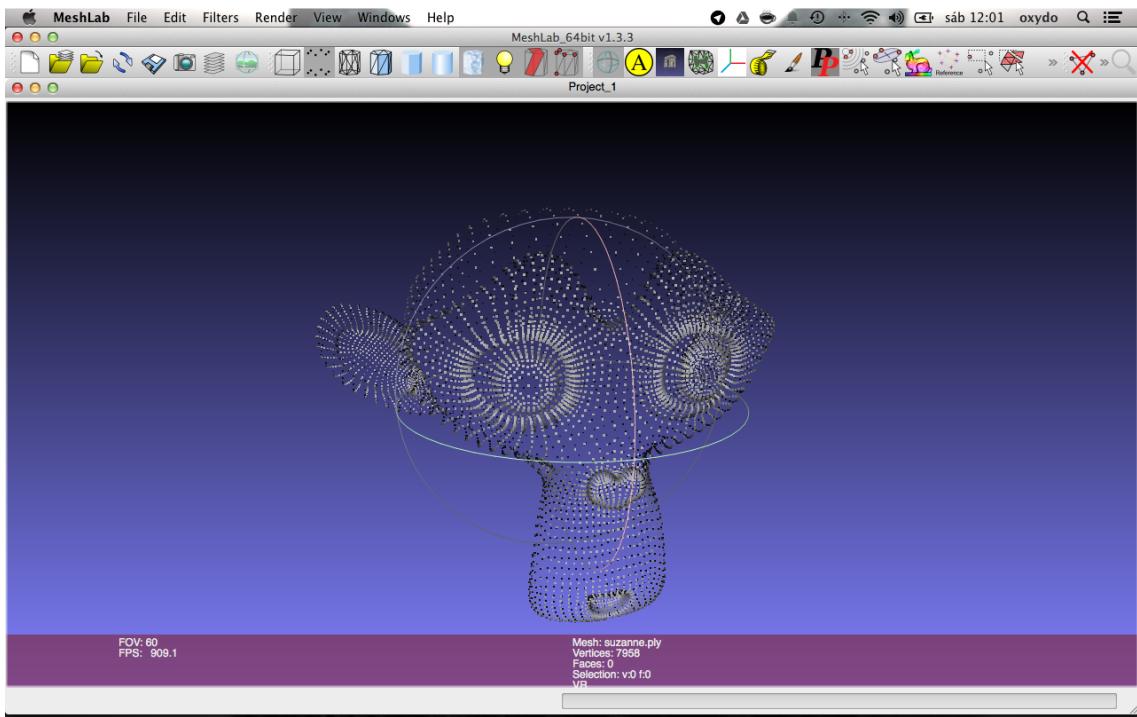


Figura 2.4: Captura de pantalla de MeshLab.

hardware, aunque es posible que esta sea implementada enteramente mediante *software*.

2.2.1. Pipeline

El *Pipeline* de render es una secuencia de pasos funcionales para la obtención de imágenes de escenas 3D. Idealmente, si se divide un proceso en n etapas se incrementará la velocidad del proceso en ese factor n . Así, la velocidad de la cadena viene determinada por el tiempo requerido por la etapa más lenta.

El *pipeline* que OpenGL sigue en la renderización de objetos (ver Figura 2.5) sería el siguiente:

1. Preparación del *Vertex Array Object* a renderizar.
2. Procesamiento de los vértices:
 - a) Cada vértice del flujo de datos (*stream*) es procesado mediante el *vertex shader* con motivo de realizar las transformaciones necesarias de estos a un espacio de coordenadas adaptado a su proyección.
 - b) Fase opcional de teselación.

- c) Fase opcional *geometry shader*. La salida es una secuencia de primitivas.
3. *Vertex Post-Processing*, la salida del anterior paso es ajustado (*primitive clipping* y transformación del *viewport* a coordenadas de la ventana).
 4. *Primitive Assembly*.
 5. Las primitivas son divididas en elementos discretos, los cuales generan un numero determinado de fragmentos.
 6. El *fragment shader* procesa cada fragmento generando un numero diferente de salidas.
 7. *Per-Sample Processing*, donde se descartan fragmentos, se combinan los colores de los fragmentos obtenidos del *fragment shader* con los que ya existen en el *buffer*, ademas de realizar diferentes operaciones lógicas.

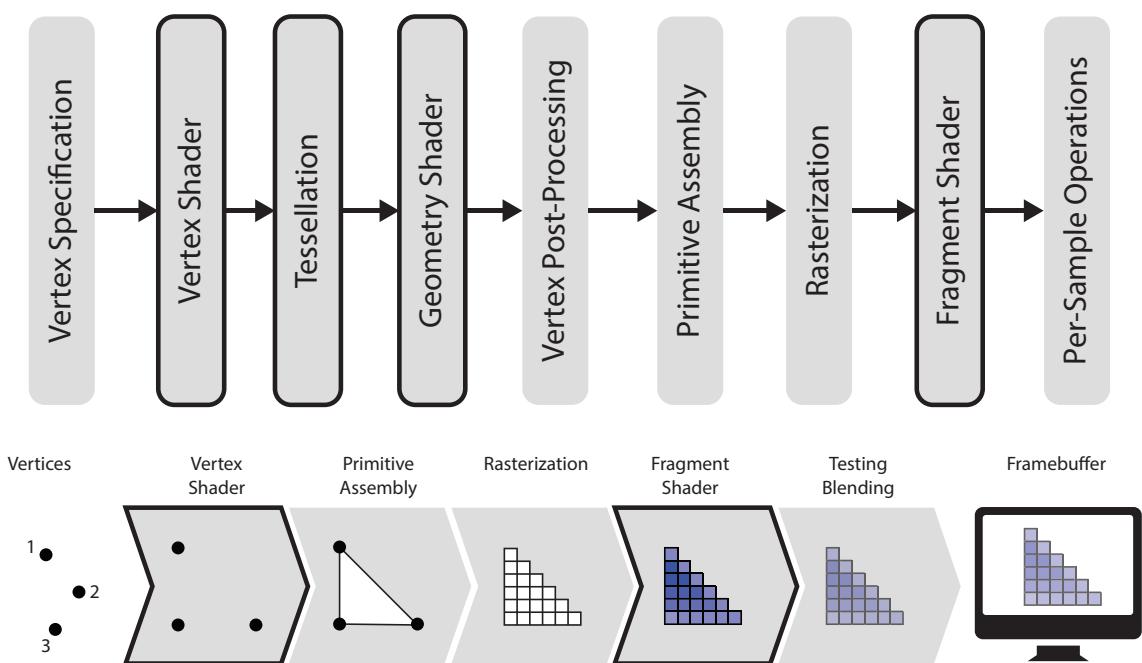


Figura 2.5: El pipeline de OpenGL, las fases es negrita son programables.

2.2.2. Shaders

El *hardware* de aceleración gráfica ha sufrido una importante transformación en la última década. Como se muestra en la Figura 2.6, en los últimos años el potencial de las *GPsUs* ha superado con creces al de la *CPU*.

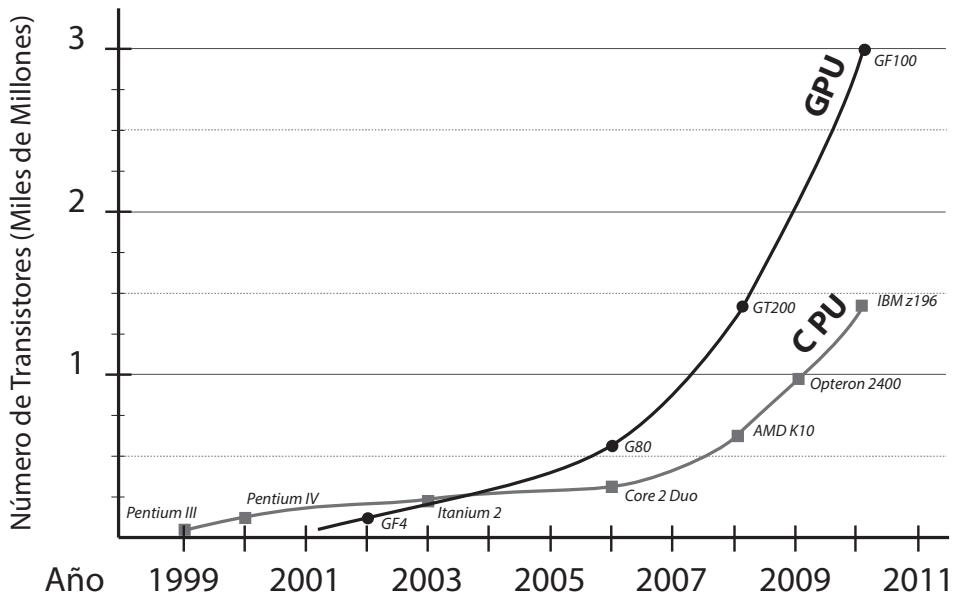


Figura 2.6: Evolución del número de transistores en GPU y CPU (1999 - 2010).

Esta tendencia en el *hardware* gráfico ha reemplazado la funcionalidad fija del antiguo *pipeline* con áreas programables mediante pequeños programas. De este modo algunas de las fases del *pipeline* pueden ser modificadas con pequeños programas escritos por el usuario denominados *shaders*. Estos programas se ejecutan directamente en la *GPU*, y permite realizar operaciones a diferentes niveles con alta eficiencia.

El lenguaje de OpenGL de *Shading (GLSL)* está basado en ANSI C y muchas de las características han sido mantenidas excepto las que entraban en conflicto con temas de rendimiento o implementación. C fué extendido con tipos nuevos para manejar estructuras como vectores o matrices, para hacer mas fácil operaciones típicas en gráficos 3D.

En la figura 2.5 se puede observar que en OpenGL las fases referentes al *vertex shader*, *tessellation*, *geometry shader* o *fragment shader* pueden ser programables.

2.3. PCL

*Point Cloud Library (PCL)*⁵ [12] es una librería *open-source* con algoritmos destinada a tareas de procesamiento de nubes de puntos y geometría 3D. La librería contiene algoritmos para estimación, reconstrucción de superficies, *model fitting* y segmentación. Está escrita en C++ y distribuida bajo una licencia *BSD*.

⁵<http://pointclouds.org>

La librería es usada en muchos de los campos en los que es habitual trabajar con nubes de puntos, por ejemplo, para percepción en robótica para filtrar ruido, unir nubes de puntos 3D, extraer *keypoints* y calcular descriptores para reconocimiento de objetos basados en su geometría y crear superficies a partir de nubes de puntos para visualizarlas.

El desarrollo de esta librería comenzó en Marzo de 2010 en la empresa *Willow Garage*. La primera versión oficial de *PCL* fue lanzada en Mayo de 2011.

Capítulo 3

Fundamentos de la Programación Gráfica

En este capítulo se introducen algunos conceptos básicos de informática gráfica, en especial los relativos a los fundamentos detrás de la obtención de un *render* 3D.

3.1. Render

Tradicionalmente la Programación Gráfica ha sido definida como la disciplina dedicada a sintetizar imágenes algorítmicamente mediante ordenadores [6]. Actualmente se pueden encontrar mas temas acerca del hiper realismo, técnicas de animación, realidad virtual, etc. Para generar imágenes a partir de escenas tridimensionales un proceso denominado *render* es usado. Este conjunto de acciones está al cargo del modelado y sus propiedades, iluminación y si es necesario la cámara que capturará todo.

Como bien se dijo antes, el *render* es el proceso computacional de generar una imagen a partir de un modelo. Desde esta definición, se podría sacar diferentes interpretaciones, desde crear una película de animación 3D a un diagrama de barras, y todas ellas serían igualmente válidas. Aunque el término que normalmente es usado es referido a un modelo tiene una naturaleza espacial y mas específicamente tridimensional

Diferentes clasificaciones de técnicas de *render* puede ser definidas, pero para esta introducción usaremos dos de ellas. Por un lado, si nos fijamos en el estilo de la imagen que se pretende conseguir:

- **Fotorealístico** que intenta ser lo más fiel como sea posible a la realidad. Este tipo de *render* puede ser subdividido en:



Figura 3.1: Fotorealistic (izquierda) vs. No fotorealísticos (derecha).

- **Render basado en Físicas** Intentando calcular una simulación lo más auténtica posible del tratamiento de la luz a través de la escena y su interacción con materiales y objetos. La precisión de la simulación dependerá de los modelos físicos y matemáticos escogidos.
- **Falseados** que utiliza trucos para reducir tiempos de renderizado.
- **No fotorealísticos** que pretende usar otros tipos de efectos para la renderización de la escena con motivos artísticos.

La Figura 3.1 muestra la diferencia entre ambas aproximaciones.

Por otro lado, si se comparan las opciones de interacción de la aplicación con el usuario final, se podría dividir las técnicas en 2 apartados:

- **Render Offline** Donde el proceso de generar la imagen es demasiado lento para responder instantáneamente a las interacciones del usuario, en general por la búsqueda de un resultado realista [10]. Una escala de tiempo de segundos a incluso días podría considerarse «lento». Esto ocurre por ejemplo cuando se generan fotogramas para una película.
- **Render Online, Real-time o Interactivo** Donde el tiempo para la obtención de las imágenes es suficientemente corto como para responder a la interacción del usuario en la aplicación [1], teniendo este una sensación de continuidad. Una escala de *milisegundos* tendría que ser alcanzada para obtener este efecto. Normalmente medida en *frames per second* (FPS). Un buen ejemplo de este grupo podrían ser los videojuegos.

La mayor parte del trabajo realizado en este documento se ha centrado en la obtención de un render interactivo, lo que en general implica sacrificar muchos aspectos presentes en una aproximación fotorealística.

3.1.1. Modelos 3D

El modelado describe el proceso de formar la figura de un objeto virtual en un ordenador. Existen realmente tres tipos diferentes de Modelos 3D:

- **Basado en Polígonos:** Donde se describe la superficie de un objeto mediante un conjunto de polígonos. La forma triangular es la mas comúnmente usada para esta tarea, puesto que son planos y trivialmente convexos. Prácticamente la totalidad del *hardware* gráfico está optimizado pensando en esta primitiva.

Aunque en el proceso del modelado de un objeto, este no tiene porque estar construido usando triángulos. En última instancia, es convertido a un conjunto de triángulos mediante un proceso conocido como teselación.

- **Basado en Voxels:** Divide el espacio tridimensional en una rejilla donde el *voxel* es la mínima unidad de medida. Cada una de estas celdas es rellenada o no, dependiendo de si se pretende que sea renderizada. La memoria necesaria aumenta en función de la precisión necesaria. Este tipo de procesos usualmente utilizado en Informática Biomédica.
- **Basada en Puntos:** Los objetos son representados mediante puntos que son producto de un muestreo de su superficie. Cada punto tiene una posición y alguna información relativa a la superficie a la que pertenece. Comparado con el modelo tradicional basado en polígonos esta primitiva necesita sus propias técnicas de *render*. Este es nuestro caso de estudio.

3.2. Espacios 3D euclídeos y transformaciones

En gráficos por computación normalmente trabajamos con espacios tridimensionales de geometría Euclídea. El término «Euclídeo» es usado para distinguir estos espacios y los espacios curvos de geometría no Euclídea. Las operaciones más típicas en geometría Euclídea puede ser representada con matrices de transformación si se usan sistemas de coordenadas homogéneos [6]. Debido a esto una transformación \mathbf{T} puede ser usada para diferentes motivos.

3.2. ESPACIOS 3D Y TRANSFORMACIONES FUNDAMENTOS DE LA PROGRAMACIÓN GRÁFICA

Usando conceptos básicos de álgebra lineal, una matriz 4×4 puede ser usada para expresar transformaciones lineales de un punto o un vector. Una transformación entonces será representada por los elementos de la matriz 4×4 . Ellas puede ser también usadas para realizar algunas transformaciones no lineales en espacios Euclídeos. Por esta razón, las matrices de transformación son altamente usadas en esta disciplina.

Generalmente una transformación es una función de puntos a puntos o vectores a vectores, por ejemplo:

$$p' = \mathbf{T}(p) \quad v' = \mathbf{T}(v) \quad (3.1)$$

Para transformar los puntos o vectores simplemente se tiene que realizar las apropiadas multiplicaciones de matrices. Esto también permite la composición de transformaciones, multiplicando las matrices de transformación.

Las transformaciones mas utilizadas comunmente son:

- **Rotación**, con el que se podrá rotar un punto o vector según un ángulo dado alrededor de un eje arbitrario o los ejes x , y o z .
- **Escalado**, para un punto o vector dado, esta transformación escalará los componentes x , y y z por un factor.
- **Translación**, únicamente afecta a puntos y podrá transladar las coordenadas x , y y z una cantidad.
- **Modelo**, esta matriz es útil para transformaciones del modelo localmente. De estar compuesto por un conjunto de matrices de rotación, escalado o translación. Cuando sea aplicada esta matriz a todos los puntos, los tendremos en coordenadas del mundo.
- **Vista**, inicialmente la cámara es localizada en el origen del espacio mundo. Para poder moverla alrededor del espacio mundo esta matriz es utilizada. Esta matriz también es denominada como “mirar a”. Luego de que esta transformación sea aplicada, tendremos los puntos en coordenadas de la cámara.
- **Proyección**, puesto que la escena tiene que ser proyectada tanto en una pantalla como en una imagen 2D, necesitamos otro proceso que convierta los puntos desde las coordenadas de la cámara a coordenadas homogéneas de modo que estas puedan ser proyectadas en pantalla.

Normalmente la composición de las matrices del modelo, vista y proyección es llamado *MVP* y es aplicada a cada punto que queramos dibujar.

Para ilustrar como las transformaciones son representadas por una matriz 4×4 , la ecuación 3.2 muestra la matriz genérica para la translación:

$$\mathbf{T}(\Delta x, \Delta y, \Delta z) = \begin{pmatrix} 1 & 0 & 0 & \Delta x \\ 0 & 1 & 0 & \Delta y \\ 0 & 0 & 1 & \Delta z \\ 0 & 0 & 0 & 0 \end{pmatrix} \quad (3.2)$$

Esta transformación es aplicada al punto $P = (x, y, z)$ de la siguiente forma:

$$\begin{pmatrix} 1 & 0 & 0 & \Delta x \\ 0 & 1 & 0 & \Delta y \\ 0 & 0 & 1 & \Delta z \\ 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x + \Delta x \\ y + \Delta y \\ z + \Delta z \\ 1 \end{pmatrix} \quad (3.3)$$

3.3. Camera model

Casi todo el mundo actualmente ha usado una cámara y conoce su propósito: se quiere obtener una imagen del mundo (normalmente presionando un botón) y la imagen es entonces almacenada en un *film*. Una de las cámaras mas simples es la cámara estenopeica. Este tipo de cámaras están compuestas por: una caja estanca, un pequeño orificio por donde entra la luz (estenopo) y un material fotosensible. De modo que cuando el pequeño orificio es descubierto la luz entra a través de el y alcanza la pieza de material fotosensible que está dispuesto al otro lado de la caja (ver Figura 3.2). Hoy en día, las cámaras son mucho mas complejas de funcionamiento que esta simple cámara, pero es un buen comienzo para explicar como el funcionamiento de nuestra simulación.

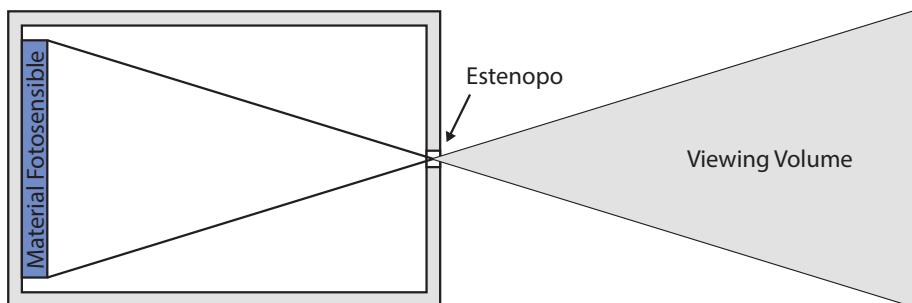


Figura 3.2: Funcionamiento de una cámara estenopeica.

Lo más importante es definir que parte de la escena será reflejada en el papel fotosensible.

Conectando el estenopo con las aristas del papel se genera una doble pirámide que extiende dentro de la escena. Los objetos que no estean dentro de esta pirámide no quedarán reflejados en la imagen del film. Como las cámaras de hoy en dia son mas complejas, se referirá a la región que puede ser reflejada en la imagen final como *viewing volume*.

Puesto que la imagen obtenida es la situada enfrente de el estenopo, este comunmente es también referido como *ojo*. Una cantidad de luz viajará del mundo al plano donde está situado el material fotosensible. Varios tests serán realizados por OpenGL dependiendo del tipo de cámara para establecer que puntos pueden ser vistos y tendrán que ser representados en el *frame-buffer*.

El modelo de la cámara usado en este proyecto soporta diferentes configuraciones. Primero se explicará como estos parámetros afectan al modelo de la cámara, luego se explicará como el modelo interacciona con el proceso de *ray tracing*.

3.3.1. Parámetros de la cámara

El primer parámetro que el modelo de la cámara soporta es la **posición**. Este parámetro establece donde la cámara tendrá que estar situada en el espacio de coordenadas del mundo (x,y,z) .

El siguiente parámetro soportado es el de la rotación. Este parámetro es un punto en el mundo hacia donde la cámara estará mirando. También necesitamos saber cómo orientar la cámara a lo largo de la dirección de visión implícita por los dos primeros parámetros .

También es necesario conocer como está orientada la cámara con respecto a la dirección en la que se está mirando implicado por los 2 primeros parámetros. El parámetro *up vector* nos da esa orientación.

Podremos ver como estos parámetros están organizados en el espacio de la cámara en .

Otros parámetros importantes en el modelo de la cámara son los planos de *clipping*, que nos darán el rango a lo largo del eje z que dejará que los objetos dentro de el sean renderizados. **Near** establece la posición del plano cercano y **far** indica la posición del plano de *clipping* lejano.

El último parámetro que la cámara soporta es el **campo de visión**. El ángulo de visión describe la extensión angular de la escena capturada por la cámara horizontal y verticalmente .

Los planos de *clipping* y los ángulos de visión define el *viewing volume* en el modelo, también conocido como *viewing frustum*.

El usuario también tiene la opción de usar entre una cámara con proyección **perspectiva** o **ortográfica** dependiendo de las necesidades. En arquitectura por ejemplo se suele preferir una cámara ortográfica, mientras que un usuario normal suele preferir una en perspectiva puesto que el campo de visión da un resultado más natural. (ver Figura 3.3)

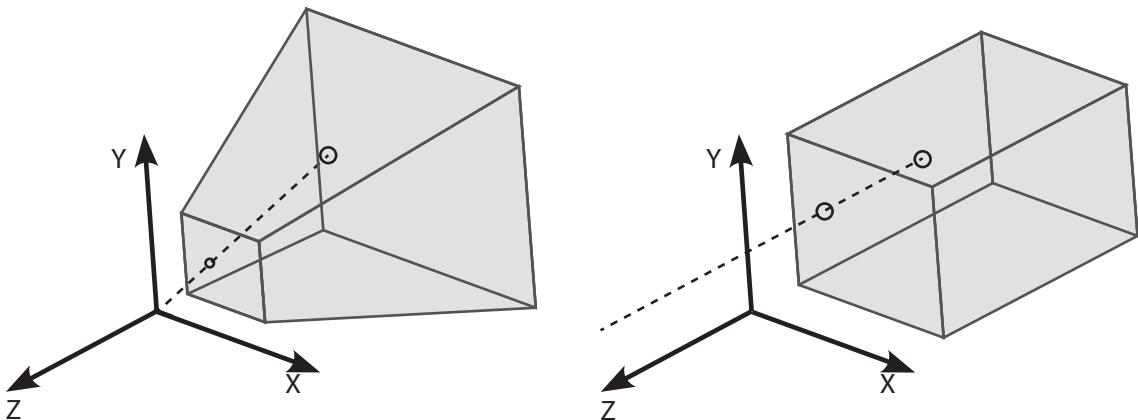


Figura 3.3: Representación de las diferencias entre cámara en perspectiva (izquierda) y ortográfica (derecha).

La proyección ortográfica es mas comúnmente utilizada en ingeniería puesto que las dimensiones son mostradas de forma inequívoca, si hay alguna linea de unidad en cualquier lado de la escena, esta se mostrara con la misma longitud en cualquier lado. También cualquier par de líneas que sean paralelas lo serán también en la realidad. Con la proyección en perspectiva, líneas de idéntico tamaño en la vida real podrán aparecer de longitud diferente debido al *escorzo*.¹ Esto hace que sea difícil juzgar las dimensiones relativas de un objeto cuando está lejano.

3.3.2. Funcionamiento interno de la cámara

Una vez que se tienen los parámetros que se necesitan para localizar la cámara en escena, usaremos una transformación *mirar a* para este propósito. Usando esta transformación como matriz de vista nos dará una transformación entre coordenadas del mundo y la cámara como se mencionó anteriormente. La matriz es construida usando la posición (p), orientación (o) y up vector (u).

Primero, un vector dirección f es construido:

$$f = \frac{p - o}{\|p - o\|} \quad (3.4)$$

¹Los objetos se vuelven más pequeños en la imagen en cuanto estén situados en un plano más lejano.

Después, el vector *right r*:

$$r = f \times u \quad (3.5)$$

Luego, un nuevo vector *up u'* es calculado en función de la referencia de la cámara:

$$u' = r \times f \quad (3.6)$$

Con éstos, una matriz de rotación se puede construir lo que representa una reorientación en la base ortonormal de nueva creación:

$$R = \begin{bmatrix} r_x & u'_x & f_x & 0 \\ r_y & u'_y & f_y & 0 \\ r_z & u'_z & f_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (3.7)$$

Por último, para transformar objetos en el marco de la cámara; no sólo todo tiene que ser orientada correctamente, sino que también tiene que ser traducido desde el origen hasta la posición de la cámara. Esto se logra mediante la concatenación de la matriz de rotación *R* con la matriz correspondiente:

$$V = \begin{bmatrix} r_x & u'_x & f_x & -p_x \\ r_y & u'_y & f_y & -p_y \\ r_z & u'_z & f_z & -p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (3.8)$$

La ecuación 3.8 es la resultante matriz *mirar a V* que será usada para transformar los puntos del espacio mundo a coordenadas de la cámara.

La siguiente fase dependerá del tipo de cámara que el usuario elija, ortogonal o perspectiva. En función de esta elección una matriz de proyección será escogida.

Si es escogida una cámara perspectiva, tendremos que tener en cuenta el efecto del escorzo mencionado anteriormente. Esta proyección no conserva distancias, ángulos, y paralelismo entre líneas. *x* e *y* serán modificadas de acuerdo al correspondiente valor de *z*, que establece como de cerca a la cámara está un punto. Para obtener este efecto, una matriz perspectiva será utilizada. Una matriz comúnmente utilizada es la matriz *frustum*. En ella el espacio homogéneo obtiene la forma de una pirámide truncada. Los parámetros usados para la construcción de esta matriz serían el **izquidara** (*l*), **derecha** (*r*), **arriba** (*t*), **abajo** (*b*), plano **near** (*n*) y **far** (*f*) del *frustum*.

t es obtenida de la siguiente forma usando los parámetros de la cámara del **campo de**

visión (*fov*) y el plano *near* (*n*):

$$t = n \cdot \tan(fov/2) \quad (3.9)$$

Luego, *r* es obtenida usando el ancho (*w*) y alto (*h*) de la ventana:

$$r = t \cdot w/h \quad (3.10)$$

Puesto que el *frustum* es simétrico, *l* y *b* serán lo mismo que *r* y *t* pero de signo opuesto. Una vez se tienen todos los parámetros se podrá construir las matrices de perspectiva u ortográfica:

$$P = \begin{bmatrix} \frac{2 \cdot n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2 \cdot n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{n+f}{n-f} & \frac{2 \cdot n \cdot f}{n-f} \\ 0 & 0 & -1 & 0 \end{bmatrix} \quad (3.11)$$

En contraste, si la cámara requerida es la ortográfica, una matriz algo más simple se puede usar. Los puntos serán proyectados al plano *near*. Este tipo de transformaciones no da el efecto de escorzo, manteniendo paralelas y manteniendo la distancia relativa entre objetos. Esta transformación mantiene *x* e *y* invariables, pero cambia el valor de *z* al plano *near*. Los parámetros para la construcción de esta matriz son también **izquierda** (*l*), **derecha** (*r*), **arriba** (*t*), **abajo** (*b*) y plano **near** (*n*) del *frustum*:

$$P = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & \frac{l+r}{l-r} \\ 0 & \frac{2}{t-b} & 0 & \frac{b+t}{b-t} \\ 0 & 0 & \frac{2}{n-f} & \frac{n+f}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (3.12)$$

Una de estas dos matrices (3.11 or 3.12) representarán la matriz de proyección *P*.

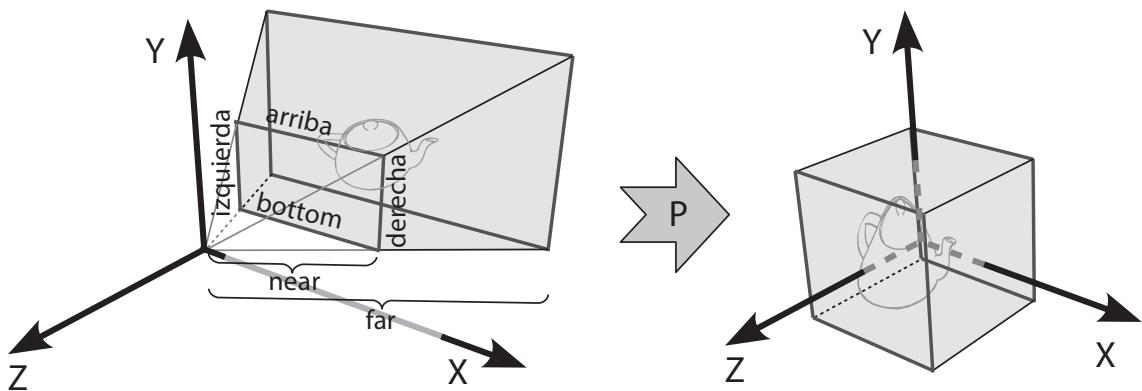


Figura 3.4: La matriz P se encarga de transformar la pirámide de visualización en un cubo unitario.

La composición de estas matrices producirá la matriz MVP. Esta es la expresión matemática usada para transformar cada punto:

$$\begin{aligned} v' &= \mathbf{P} \cdot \mathbf{V} \cdot \mathbf{M} \cdot v \\ &= \mathbf{MVP} \cdot v \end{aligned} \tag{3.13}$$

Estos son los paso necesarios para la creación de un *frame*, pero puesto que le visualizador es interactivo, tendremos que repetir este proceso varias veces por segundo.

Capítulo 4

Análisis y diseño de un visualizador

En este capítulo se presentará el resumen de diseño que se siguió en la creación del visualizador de nubes de puntos.

4.1. Análisis de requisitos

Ante la necesidad de un visualizador en el que poder probar los algoritmos que se iban a implementar, se pensó que sería buena idea el desarrollo de uno propio de manera que fuera simple, sencillo y fácilmente modificable.

El visualizador tendría que cumplir los siguiente requisitos:

- Tener una gestión de *shaders* de manera que fuera sencillo compilar y enviar a *GPU* estos programas, además de facilitar la incorporación de nuevos en el proyecto.
- Carga de modelos desde ficheros en disco.
- Un sistema para almacenar las nubes de puntos y poderlas enviar a *GPU* para su dibujado.
- Una cámara para moverse alrededor de los objetos cargados.
- Fuentes de luz para la iluminación de la escena.
- Un *framebuffer* donde almacenar los resultados.

4.1.1. Casos de uso

Se establecerán los siguientes casos de uso (ver Figura 4.1), siendo los más significativos:

- La carga de modelos mediante ficheros externos (formatos .PLY o .PCD) destinados al almacenamiento de este tipo de datos.
- La opción para desactivar el color RGB de los puntos, de manera que sea mas claro evaluar si la reconstrucción de la superficie es correcta.
- La capacidad de moverse por el espacio tridimensional.
- Modificar el radio de los puntos manualmente o que el programa estipule un radio de manera automatica.
- Poder escoger entre los diferentes métodos de rasterización o *blending* implementados para variar la visualización de la nube.
- Cambiar el sistema de iluminación.

4.2. Diagrama de clases

Una vez definidos los requisitos, se pasó a diseñar un esqueleto el cual se podrá ver en el siguiente diagrama de clases (ver Figura 4.2):

La idea de tener un *framebuffer* propio donde renderizar de manera separada (clase **FBO**), permitió independizar el visualizador de la librería que fue usada para la creación de ventanas en el Sistema Operativo¹, de forma que en caso de necesitar cambiarla, esta escasa dependencia facilitaría la tarea.

La clase **Shader**, vendría a dar la solución al manejo de los *vertex* y *fragment shaders* en el visualizador, ofreciendo métodos para compilar y enviar estos programas a *GPU*.

La clase **Models** será la encargada de almacenar los modelos que se carguen. Estos almacenarán los datos de la nube (posición, color, normal y radios).

El objetivo de la clase **VAO** será la de enviar la información de la nube de puntos al *pipeline* y encargarse de reservar la memoria en la tarjeta gráfica.

La clase Camera viene a ser la implementación de todo lo explicado en la sección 3.3, ofreciendo las matrices de transformación pertinentes y que serán necesarias para el render.

¹GLFW: <http://www.glfw.org/>



Figura 4.1: Casos de uso de la aplicación.

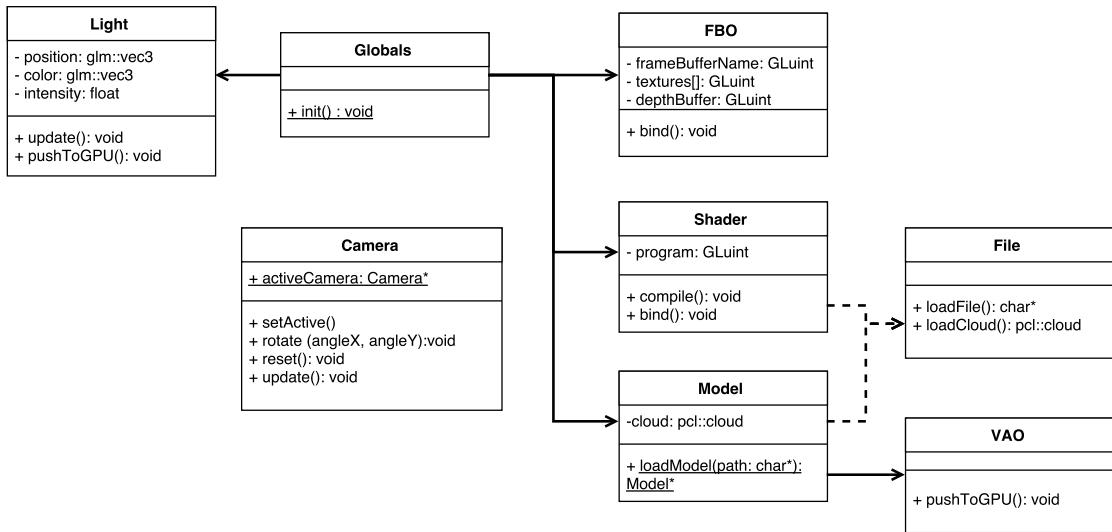


Figura 4.2: Diagrama de clases del visualizador de nubes de puntos.

En este caso el funcionamiento de la cámara es orbital, rotando alrededor de origen de coordenadas, pudiendo acercar o alejar la vista.

Se implementaron tres tipos diferentes de luz según su comportamiento: una luz orbital que gira alrededor de un eje, una luz que se denominó *camera light* de forma que se podía fijar a una cámara para que iluminara desde su posición y otra estática con motivo de tener luces de relleno inmóviles en la escena. Estas tres luces heredan de la superclase **Light**.

La clase **Globals** nos permite almacenar variables donde almacenar las diferentes configuraciones que el usuario puede usar para ajustar la visualización a su gusto, además de registrar otra información relevante así como la posición del ratón, dimensiones de la ventana, título ...

Capítulo 5

Visualización de nubes de puntos en GPU

Este capítulo se presenta como un resumen a los diferentes métodos de render basados en GPU. La sección 5.2 empieza mostrando los diferentes algoritmos que existen para la rasterización de splats. La sección 5.3 se centra en el blending de los diferentes splats, rasterizados en el paso previo, teniendo en cuenta la implicación que toman diferentes aspectos como la iluminación.

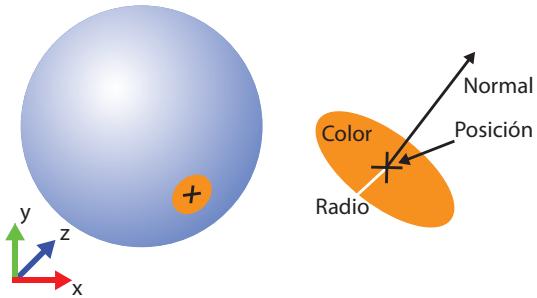
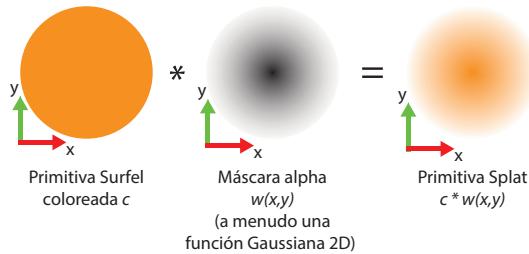
5.1. Primitivas en nubes de puntos

5.1.1. Surfel

Los puntos de una nube son el resultado del muestreo de la superficie que representan, cada punto corresponde a un elemento de superficie describiendo la superficie de una pequeña vecindad. El *surfel* (ver Figura 5.1) es una primitiva diseñada para el renderizado de puntos y podría ser descrita únicamente mediante su posición y color o añadiendo otros parámetros como pueden ser la normal, radio, etc.

5.1.2. Splat

Se podría ver la necesidad de trabajar con *splats* que tuvieran asociado una máscara que modifique el canal *alpha* (ver Figura 5.2), normalmente debido a que la muestra suele tener más relevancia en el centro del *splat* debido a que el punto es la representación de un elemento diferencial de superficie. Este tipo de primitiva es conocida como *splat*.

Figura 5.1: Representación de un *surfel* de una esfera.Figura 5.2: Construcción de un *splat*.

5.2. Técnicas de rasterización

El primer paso en el render de nubes de puntos, es determinar según la proyección de los *surfels* que píxeles en el framebuffer serán cubiertos por ellos. Puesto que no existe soporte en las librerías actuales para representar este tipo de primitivas, los *surfels* tendrán que ser representados por otras como puntos o triángulos. De entre esas dos opciones, se considera como más eficiente los puntos por delante de los triángulos, ya que por cada *surfel* únicamente hay que almacenar la posición de un vértice en lugar de tres. Finalmente para renderizar la nube, habrá que dibujar el *VBO*, pasándole el tipo *GL_POINTS*

```
glDrawArrays(GL_POINTS, &surfels[0], surfels.size());
```

5.2.1. Sized-Fixed Points

De una forma un poco primitiva y puesto que un punto a priori puede no tener un tamaño definido, a la hora de mostrar una representación de la nube, se podría dibujar cada coordenada del conjunto de puntos, con un cuadrado de un tamaño definido. Para ello basta con activar en el programa.

```
 glEnable(GL_PROGRAM_POINT_SIZE);
```

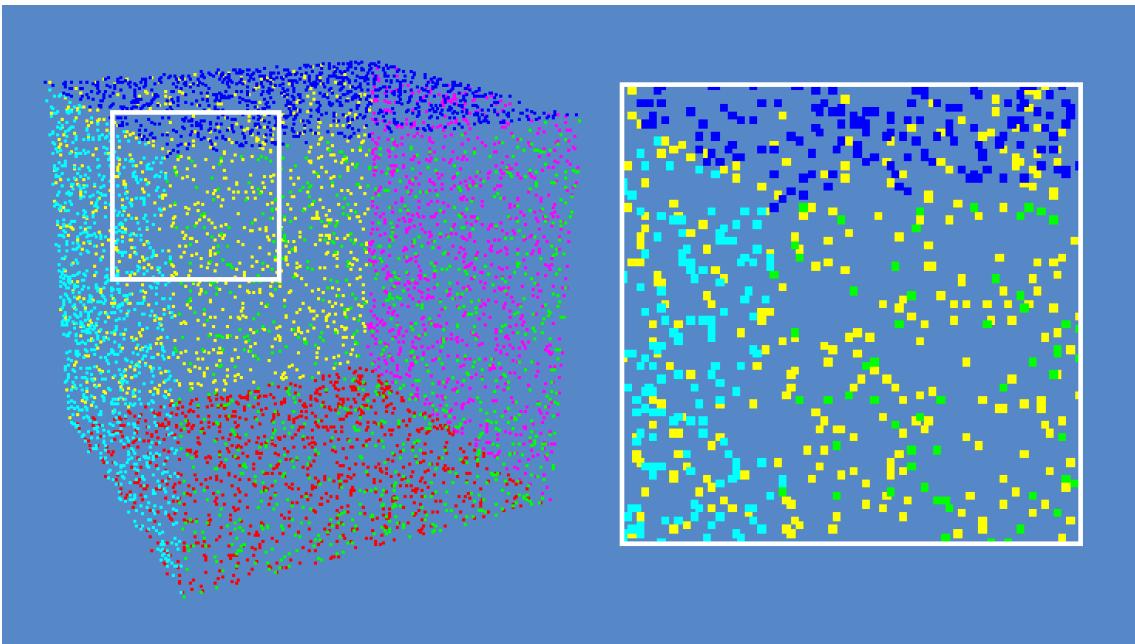


Figura 5.3: Todos los puntos de la nube son renderizados con un cuadrado del mismo tamaño para toda la nube.

Mientras que en el *vertex shader*, se define con *glPointSize* el tamaño en pixels que se pretende que ocupe el punto al ser rasterizado. Este cuadrado tendrá como centro las coordenadas del punto \mathbf{p} .

Como resultado, se obtiene una representación en la que cada punto de la nube se renderizará con el tamaño descrito.

Ver Apéndice Shaders A.1

5.2.2. Image-aligned Squares

En el anterior paso, el tamaño de la representación que cada punto tomaba en el *frame-buffer* era de alguna forma independiente a la cámara y al modelo.

De esta manera, si en lugar de entender el tamaño del punto como los pixeles que va a ocupar en la imagen final, se interpreta como una medida dentro del modelo, que podría ser el radio r del splat, al representarse, estos puntos tendrían que ser coherentes con su proyección.

Así que el tamaño en el *screen-space* del punto OpenGL proyectado, tiene que ser ajustado en el *vertex shader*. El tamaño [15] se aproxima haciendo el escorzo (*perspective foreshort*) del radio r del *surfel* usando el *depth value* del centro \mathbf{p} que ha de estar en

coordenadas de la cámara.

$$gl_PointSize = 2r \cdot \frac{n}{p_z} \cdot \frac{g}{t - b'} \quad (5.1)$$

Donde n , t y b son los parametros *near/top/bottom* del *view frustum* y h denota la altura (en pixels) del *viewport*. En esta fórmula el término n/z corresponde a la proyección en el plano *near*, mientras que $f/(t - b)$ escala el resultado desde el plano *near* a coordenadas de la imagen.

Todos los pixels, generados de un punto, tienen el mismo *depth value* lo que imposibilita hacer uso de las técnicas blending en zonas que visualmente se estén superponiendo. (ver sección 5.3)

Ver Apéndice Shaders A.2

5.2.3. Affinely Projected Point Sprites

Una mejor aproximación para representar superficies afines a nubes de puntos, es con discos orientados según un vector normal \mathbf{n} , este método fue presentado en *Bosch and Kobbelt* [4] .

Para ello se tendría que ajustar el tamaño del punto en el *vertex shader* de la misma forma que en el paso anterior 5.2.2. Pero añadiendo a mayores en el *fragment shader* alguna manera de determinar si este se corresponde con un fragmento de la proyección de un disco orientado con centro en la posición \mathbf{p} , de radio r y con la normal \mathbf{n} .

Para cada uno de los pixeles $(x, y) \in [-r, r]$, se puede calcular un valor de desplazamiento en profundidad δz como una función lineal dependiente del vector en coordenadas de la cámara $\mathbf{n} = (n_x, n_y, n_z)^T$:

$$\delta z = -\frac{n_x}{n_z} \cdot x - \frac{n_y}{n_z} \cdot y \quad (5.2)$$

Este desplazamiento es usado para calcular la distancia 3D desde el centro del punto \mathbf{p} , de manera que el fragmento (x, y) corresponde al punto si $\|(x, y, \delta z)\| \leq r$. Los que no cumplan esta condición pueden ser fácilmente descartados usando el comando *discard*.

La variable vector *gl_PointCoord* del *fragment shader* contiene las coordenadas bidimensionales que indican donde dentro de un punto OpenGL el fragmento está situado, con valores desde el 0,0 al 1,0 estos han de desplazarse a un rango $[-0,5, 0,5]$ para usarse en

el cálculo de δz .

Una posible implementación de este algoritmo en *GLSL* sería:

```
vec3 test;

test.x = gl_PointCoord.x - 0.5;
test.y = gl_PointCoord.y - 0.5;
test.z = -(normal.x/normal.z) * test.x - (normal.y/normal.z) * test.y;

if (length(test) > 0.5)
    discard;
```

Uno de los inconvenientes usando Image-aligned Squares 5.2.2 era que el valor de profundidad era constante por *surfel*. Pero δz puede ser usado también para corregir esto. Partiendo de un valor de profundidad en espacio de la cámara $z' = p_z + \delta z$ y el frustum el valor de profundidad del $zbuffer(x, y)$ puede ser modificado

$$zbuffer(x, y) = \frac{1}{z'} \cdot \frac{fn}{f-n} + \frac{f}{f-n} \quad (5.3)$$

En comparación con la propuesta de *Image-aligned Squares*, este método ofrece una mejor aproximación, especialmente en los contornos del objeto. Sin embargo el valor de profundidad δz es solo una aproximación, ya que se asume una proyección paralela (5.2), causando errores que provocan que los discos se vuelvan demasiado finos cuando estos son mirados bajo ángulos planos, lo que puede resultar en huecos en la imagen renderizada (ver Figura 5.4).

Ver Apéndice Shaders A.3

5.2.4. Perspective Correct Rasterization

El método de *Affinely Projected Point Sprites* (5.2.3) causaba errores debido a la proyección paralela que en esta se asumía. Puesto que la proyección puede o no ser paralela, se tendrá que usar una aproximación mas precisa. Esta técnica fue la propuesta por *Bosch et al.* [5] .

La idea principal de este procedimiento parte en determinar el punto 3D correspondiente al pixel 2D mediante un *ray casting* local.

Fijándose en el *pipeline* de transformaciones de OpenGL el primer paso para calcular \mathbf{q} es invertir la transformación *window-to-viewport*, de esta manera se mapeará el pixel (x, y)

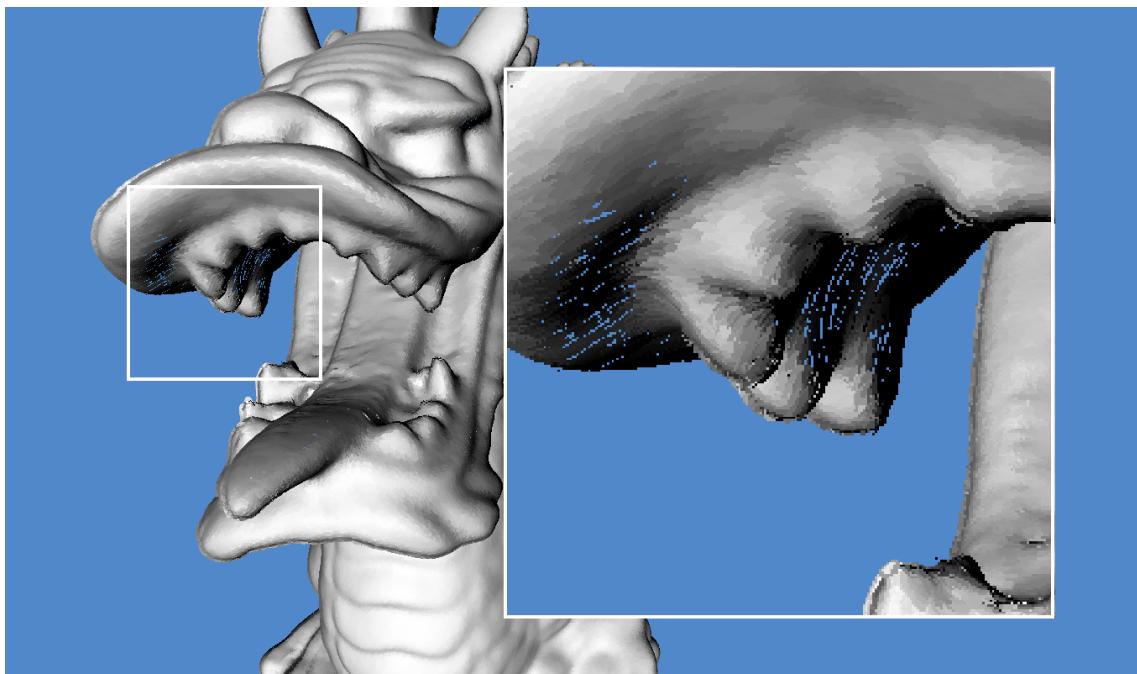


Figura 5.4: Dragon, modelo de 845,281 puntos renderizado mediante Affinely Projected Point Sprites. Este método causa huecos en ángulos extremos.

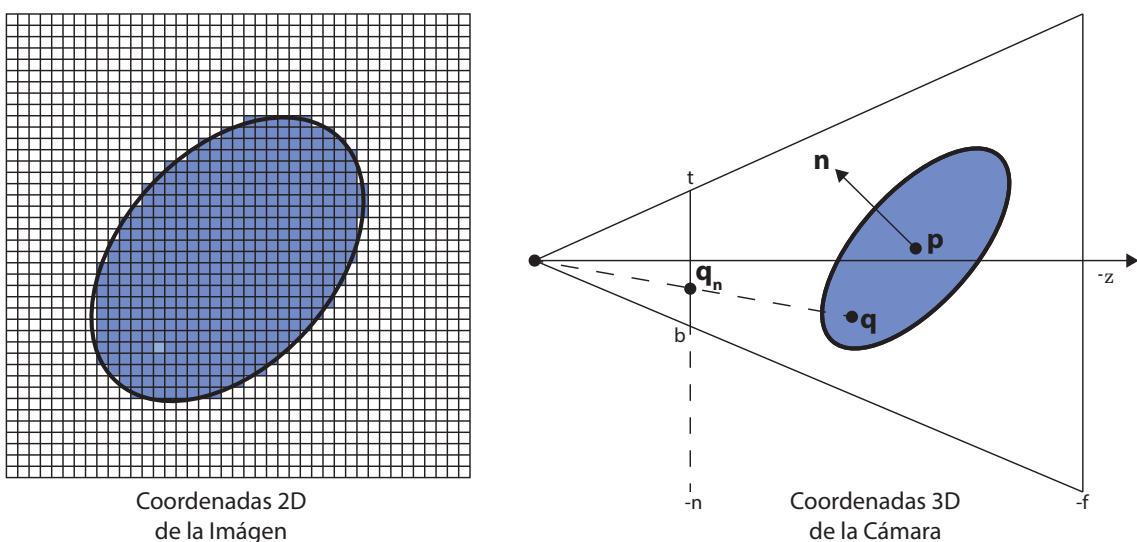


Figura 5.5: Proyección y Ray Casting.

a un punto 3D \mathbf{q}_n en el plano *near*.

$$\mathbf{q}_n = \begin{pmatrix} x \cdot \frac{r-l}{w} - \frac{r-l}{2} \\ y \cdot \frac{t-b}{h} - \frac{t-b}{2} \\ -n \end{pmatrix} \quad (5.4)$$

Donde x/y son las coordenadas relativas a la ventana *gl_FragCoord.xy*, $b/t/l/r$ son los parámetros *bottom/top/left/right* del *viewing frustum*, y w/h denotan la *altura/anchura* del *viewport* (ver Figura 5.5).

Se proyectará un rayo desde el origen (el ojo de la cámara) a través de \mathbf{q}_n y la intersección con el plano del *surfel* \mathbf{p} dará como resultado el punto \mathbf{q} . En caso de que $\mathbf{q}_n \cdot \mathbf{n} = 0$ se determinará que el el ángulo el por lo que se podrá descartar el *surfel*.

$$\mathbf{q} = \mathbf{q}_n * \frac{\mathbf{p} \cdot \mathbf{n}}{\mathbf{q}_n \cdot \mathbf{n}} \quad (5.5)$$

Finalmente el pixel será descartado si $\|\mathbf{q} - \mathbf{p}\| > r$. Por otro lado el valor de profundidad del *z-buffer* puede ser ajustado insertando q_z como valor de z' en la ecuación (5.3).

Ver Apéndice Shaders A.4

5.3. Técnicas de Blending

Luego de determinar los pixels que son cubiertos por la proyección de los *surfels* en el anterior paso de rasterización, el siguiente implica iluminación y en general el acabado estético de la superficie *shading*.

5.3.1. Flat Shading

Puesto que cada punto tiene asociado un vector normal \mathbf{n} , para unas ciertas propiedades de material y reflectancia una iluminación local puede ser evaluada por *surfel*. Como resultado se obtendrá una constante de color \mathbf{c}_i para cada *surfel*, similar al *flat shading* en modelos poligonales. Ya que los puntos tienden a intersecarse unos con otros, las representaciones mediante *flat shading* llevan a discontinuidades en el color.

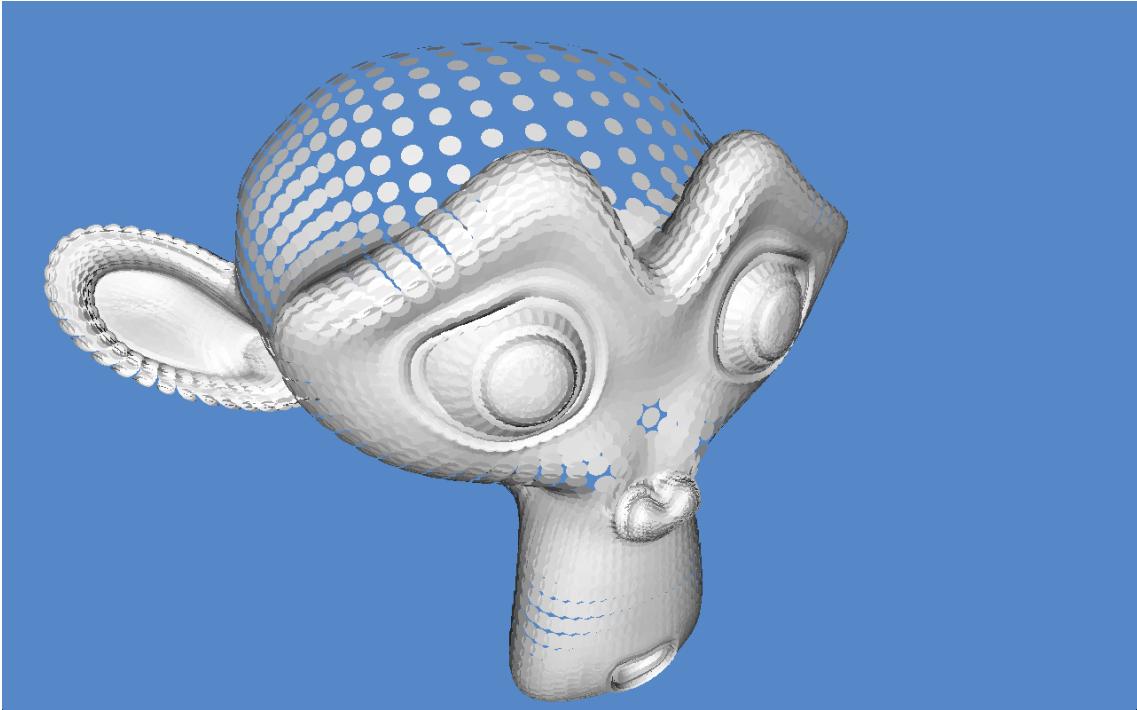


Figura 5.6: Suzanne, modelo de 7,958 puntos renderizado mediante Perspective Correct Rasterization.

5.3.2. Gouraud Shading

Para conseguir un render mas fino, las discontinuidades en el color provocadas por el *shading* pueden ser emborronadas mediante *blending* de los valores de color \mathbf{c}_i de los puntos que se están superponiendo. Puesto que los cálculos de iluminación todavía son computados por *surfel*, este tipo de *blending* del color, conceptualmente corresponde al *Gouraud shading* de modelos poligonales.

Para implementar este tipo de *blending* cada punto de la nube irá asociado con una función, de modo que para cada *surfel* sus fragmentos irán teniendo menor peso \mathbf{r} cuanto mas alejados de su centro estén dando como lugar a un *splat*, teniendo así menor impacto en la mezcla final de color $\mathbf{c}(\mathbf{x}, \mathbf{y})$. De manera que en la reconstrucción final del *framebuffer*, el color de la posición (x, y) vendrá dado por el peso promedio de todos los fragmentos \mathbf{i} que intenten cubrirlo.

$$\mathbf{c}(x, y) = \frac{\sum_i r_i(x, y) c_i}{\sum_i r_i(x, y)} \quad (5.6)$$

Esta media puede ser implementada en OpenGL usando 2 pasos de render. Primero, el *alpha-blending* es configurado con funciones separadas de *blend* para las componentes



Figura 5.7: Lucy, modelo de 397,664 puntos. Renderizado mediante *Flat Shading* (izquierda), *Gouraud Shading* (centro), y *Phong Shading* (derecha).

RGB y alpha *EXT_blend_func_separate* con

```
glEnable(GL_BLEND);
glBlendFuncSeparateEXT(GL_SRC_ALPHA, GL_ONE, GL_ONE, GL_ONE);
```

Así los *splats* irán acumulando los valores de color y peso en las componentes RGB y alpha del *framebuffer* como $(\sum_i r_i c_i, \sum_i r_i)$. Finalmente tendrá que normalizarse la componente *RGB* de cada pixel, dividiéndose por su componente *alpha*. Esto puede ser conseguido enviando el resultado del anterior pase a un siguiente como una textura, renderizando un rectángulo de tamaño de la ventana y mapeando esta textura en el. Este truco conocido como *render-to-texture*, envía de nuevo cada pixel a través del *pipeline* de OpenGL, de modo que un simple *fragment shader* puede realizar esta normalización, como la propuesta en *Botsch and Kobbelt* [4] y en *Guennebaud and Paulin* [9]. Esta acumulación debe de ser realizada usando *buffers* con precisión de punto flotante de 16 bits, con motivo de evitar saturaciones o artefactos.

Para restringir el *blend* únicamente a la superposición de los *splats* vecinos de la misma superficie. Es necesario un tercer pase, denominado de visibilidad. Para cada *frame*, el pase de visibilidad primero renderiza la nube únicamente en el *depth buffer*. Luego en el

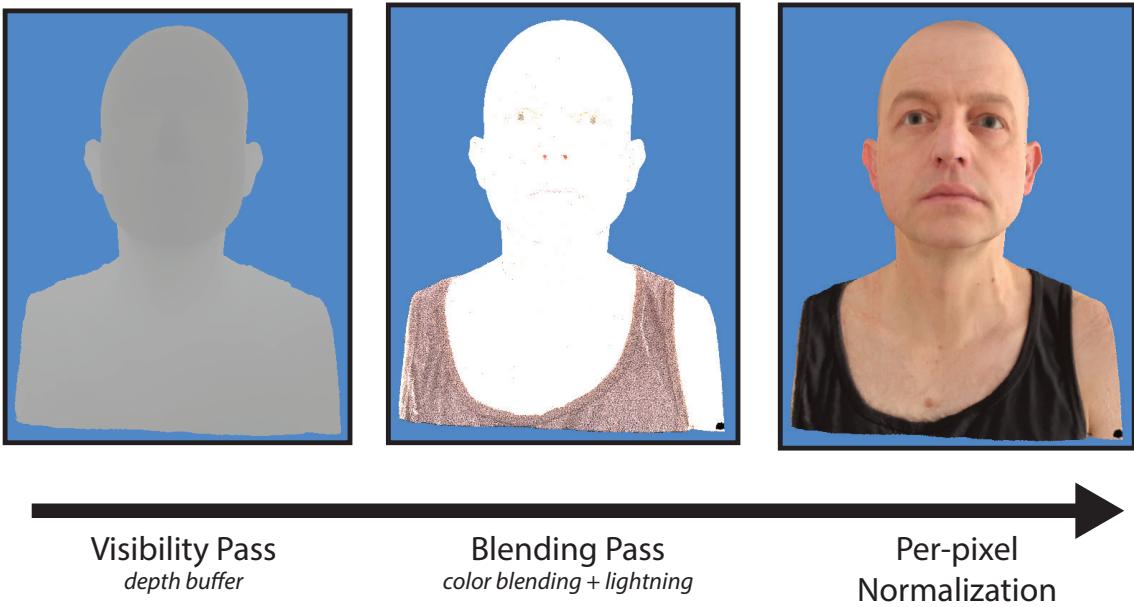


Figura 5.8: Los tres pasos, *visibility*, *blending*, y *normalization*.

pase de *blending* se renderiza la nube de nuevo, pero esta vez calculando iluminación y acumulando el valor resultante de color usando *additive alpha-blending* como lo descrito anteriormente. En este pase no se debe de actualizar el *z-buffer* calculado en el pase de visibilidad, pero sí se añade un desplazamiento ϵ a todos los valores de profundidad [14], lo que provocará que todos los fragmentos dentro de una distancia ϵ sean mezclados. El pase final de normalización realiza la necesaria división por el componente *alpha* como se describió anteriormente. El resultado de los tres pasos de render es representado en la Figura 5.8.

El *pseudocode* sería el siguiente

```

// visibility pass
glDepthMask(GL_TRUE);
glDisable(GL_BLEND);
bind_visibility_shaders();
glDrawArrays(GL_POINTS, &cloud[0], cloud.size());

// blending pass
glDepthMask(GL_FALSE);
 glEnable(GL_BLEND);
bind_blending_shaders();
glDrawArrays(GL_POINTS, &cloud[0], cloud.size());

// normalization pass
glCopyTexSubImage2D(GL_TEXTURE_RECTANGLE_ARB, 0, 0, 0, 0, w, h);
bind_normalization_shaders();
  
```

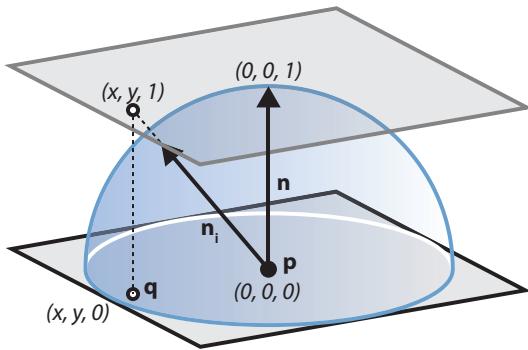


Figura 5.9: Los vectores n_i son representados como puntos homogéneos $(x,y,1)$ en un plano desplazado tangente.

```
draw_rectangle();
```

Ver Apéndice Shaders A.5

5.3.3. Phong Shading

Cuando se comparan los resultados de las diferentes técnicas en la figura 5.7, *Gouraud* realmente elimina las indeseadas discontinuidades de color provocadas en el *Flat Shading*, pero también añade cierto emborrone a la imagen notablemente. Para renderizado de polígonos es bien conocido que *Phong* es superior que *Gouraud Shading*. En lugar de calcular la iluminación por cada vértice y linealmente interpolar el color resultante dentro de los triángulos, *Phong* interpola las normales en los vértices, seguido de una iluminación por píxel basada en la resultante función definida a trozos.

Pero ya que en el caso de las nubes de puntos falta la relación de conectividad, no es posible la interpolación de las normales de los puntos vecinales, de modo que el campo de normales deberá ser construido de otra forma.

El método de *Phong* de *Botsch et al.* [5] asigna explícitamente un campo de normales lineal $\mathbf{n}_i(x, y)$ para cada *splat* en lugar de mantener su normal asociada constante. Durante la rasterización, la normal es evaluada para cada pixel basándose en sus parámetros locales (x, y) , y el vector resultante $\mathbf{n}_i(x, y)$ es usado para computar la iluminación. Ya que los valores de color resultantes siguen teniendo problemas de discontinuidad, estos son acumulados y mezclados usando el mismo sistema de 3 pasos de *Gouraud*. La única diferencia es saber cómo computar el campo de normales lineal $\mathbf{n}_i(x, y)$. Cada normal será representada por un punto (x, y) en el plano tangente con distancia 1, similar a las coordenadas homogéneas (Figura 5.9).

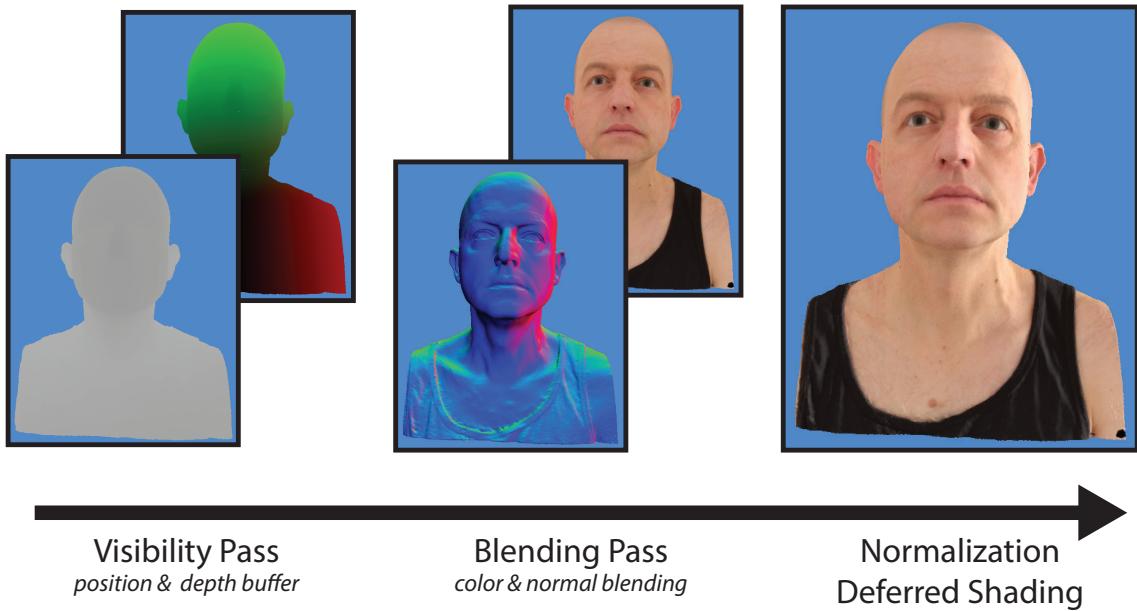


Figura 5.10: El renderizado mediante *Deferred Shading* acumula atributos como color y normales, seguido por una normalización y un shading pass.

$$n_i = \frac{(q + n) - p}{\|(q + n) - p\|} \quad (5.7)$$

Donde \mathbf{q} es el punto 3D del *splat* calculado al igual que en el método de *Perspective Correct Rasterization* (Ecuación 5.5), \mathbf{n} es la normal del *splat* y \mathbf{p} es el centro de este en coordenadas de la cámara.

Ver Apéndice Shaders A.6

5.3.4. Deferred Shading

Básicamente existen dos formas para generar interpolaciones suavizadas de vectores de normales. La primera es *Phong* que como se comentó en el anterior apartado, utiliza funciones para calcular el campo de normales de un *splat*. Una segunda opción pasaría por la denominada como *deferred* en la que en lugar de mezclar en un misma fase color y normales, estas se calculan de forma separada.

Recientes generaciones de *GPU* contienen todo lo necesario para implementar esta aproximación, que gracias al denominado *multiple render targets* (*ARB_DRAW_BUFFERS*) permite que en un simple pase de *render* se puedan sacar diferentes resultados a diferentes *buffers*, estas características habilitan la opción para poder implementar una iluminación por pixel (*deferred*) en este contexto, como se demostró en *Botsch et al. [3]*

Primeramente en el pase de visibilidad (ver Figura 5.10, izquierda), se hará una modificación para poder sacar en un target a parte la posición 3D del fragmento, \mathbf{q} (ver. Ecuación 5.5) ya que será necesario para el pase final. Luego, en el pase de acumulación y blend (ver Figura 5.10, centro) se usarán 2 targets adicionales para sacar la salida referente al color y de las normales de manera separada, usando el mismo proceso de acumulación que en *Gouraud*. Finalmente estos 3 *buffers* serán enviados de nuevo al *pipeline* de OpenGL, usando el método de *render to texture* comentado previamente, con motivo de su normalización.

Este último pase (ver Figura 5.10, derecha), corresponde a la fase de normalización de *Gouraud* pero en el que a mayores se van a realizar los cálculos pertinentes para la iluminación. Por cada pixel, una media de la normal y el color pueden ser obtenidos, que junto a la posición obtenida en la primera fase de visualización puede ser fácilmente calcular el color final iluminado.

Es importante darse de cuenta de que en este método los cálculos derivados de la iluminación son computados una única vez por pixel, en contraste con las aproximaciones que no son *deferred* que incorporan este proceso en el momento de la rasterización, consiguiendo el color final del pixel a partir de la mezcla de muchos pixels de diferentes *splats* iluminados previamente. Dependiendo del número de *splats* superpuestos, esto puede llegar a multiplicar el numero de cálculos derivados de la iluminación.

Además a esto, *deferred shading* añade también una clara separación entre la fase de rasterización y de iluminación.

Ver Apéndice Shaders A.7

Capítulo 6

Resultados y rendimiento

En este capítulo se pasarán a analizar los resultados obtenidos de la ejecución de los diferentes algoritmos de visualización, tanto en términos de rendimientos como de calidad del *render* obtenido. Se empezará por los 4 métodos de rasterización: *Sized-Fixed* (ver. 5.2.1), *Image-aligned Squares* (ver. 5.2.2), *Affinely Projected* (ver. 5.2.3) y *Perspective Correct* (ver. 5.2.4). Seguidamente se comentarán los resultados obtenidos de la comparación de los 4 sistemas de *blending*: *Flat* (ver. 5.3.1), *Gouraud* (ver. 5.3.2), *Phong* (ver. 5.3.3), *Deferred* (ver. 5.3.4) y la estabilidad de estos algoritmos en temas de iluminación.

6.1. Preparación de la pruebas

Se dispuso del mismo equipo (las características se muestran en la tabla del Cuadro 6.1) para la obtención de los datos necesarios con motivo de analizar el rendimiento que presentan los diferentes algoritmos mencionados anteriormente. Así mismo se dispuso de un conjunto de 6 modelos que fueron muestreados a diferentes densidades (ver la tabla del Cuadro 6.2).

Equipo de Test	
CPU:	Intel Core i7-4930K
Tarjeta Gráfica:	NVIDIA GTX 780Ti
RAM:	16 GB DDR3 a 2133 MHz
HDD:	WD Caviar Black HDD
SO:	Windows 7

Cuadro 6.1: Resumen de las características del equipo de Test.

Modelo	Número de puntos
Suzanne	473672
Head (Ten24)	1223693
Dorna	2585907
Dragon	8628877
Happy Budha	26519484
Lucy	43982232

Cuadro 6.2: Los modelos utilizados en el test.

```
CUBE | Sized-Fixed Points | Flat Shading | RGB | 6000 Points | 0 Lights | 640x480
0.0152009
0.016681
0.016721
0.016681

CUBE | Sized-Fixed Points | Flat Shading | RGB | 6000 Points | 0 Lights | 1920x1058
0.0144408
0.016641
0.016681
0.016641
0.016641
0.016681
```

Cuadro 6.3: Formato del fichero de texto con los *logs* generados.

El parámetro de rendimiento que se estudiará será el tiempo (en segundos) que se tarda en renderizar un *frame*. El visualizador consta con un modo *DEBUG* en el que automáticamente escribe en un *log* los datos resultantes de la sesión. El formato de este *log* consta de una línea con la cabecera en la que se resume el escenario actual de ejecución, al que siguen los tiempos medios resultantes de los últimos 25 *frames*. El Cuadro 6.3 muestra un ejemplo de este formato.

Con el objetivo de facilitar y agilizar todas las tareas relacionadas con el análisis de los datos obtenidos, se creó una herramienta para automatizar el análisis de este fichero *log* y generar automáticamente las gráficas necesarias para su comprensión. El Cuadro 6.4 muestra la interfaz en línea de comandos de esta herramienta.

6.2. Análisis de los métodos de rasterización

Se renderizaron los 6 modelos mediante los 4 métodos de rasterización, sin filtro de antialiasing, iluminación o *blending*. El resultado de este análisis lo podemos ver en la gráfica de la Figura 6.1.

Se encontró que en cuanto aumentó notablemente el número de puntos a dibujar, el

CUBE GRAPH GENERATOR

```
USAGE: cubeGraphGen.py [ options ] logFile
```

OPTIONS:

-h	Display available options
--help	Display available options
-v	Run in verbose mode
May the Force be with you :)	

Cuadro 6.4: Herramienta de apoyo en línea de comandos implementada para automatizar la creación de gráficas de resultados a partir del análisis de las ejecuciones de las pruebas.

rendimiento del supuestamente más ligero de todos los algoritmos *Sized-Fixed* cayó estrepitosamente, debido a que al tener el tamaño de punto definido de forma constante este estaría generando más fragmentos en comparación con los otros 3 que corrigen su tamaño en función de la profundidad y de la distancia con los n-vecinales.

Además, en comparación con los algoritmos que utilizan las normales para la representación, en el algoritmo *Affinely Projected* al no poder hacer *backface culling*¹ aumentan el número de puntos a mostrar aumentando el tiempo de *render*.

Con respecto al aspecto visual (podemos ver un ejemplo comparando los cuatro algoritmos en las capturas de la Figura 6.2), encontramos que:

- El algoritmo *Sized-Fixed* crea agujeros de forma incontrolable, ya que dependiendo de la distancia a la que en la que se encuentren los puntos, puede dar sensación de superficie cerrada o de nube dispersa.
- El algoritmo de *Square Aligned* consigue cerrar la superficie pero deforma los contornos notablemente.
- El algoritmo *Affinely Projected* da una sensación buena, pero da lugar a fallos con discos con ángulos perpendiculares al plano *near* de la cámara. (como se muestra en la Figura 5.4)
- El método de *Perspective Correct* consigue mantener el contorno, además de evitar los agujeros de la anterior aproximación.

¹Técnica que consiste en eliminar del *pipeline de render* las primitivas cuya normal apunta en dirección opuesta a la cámara, es decir, aquellas superficies que son vistas por detrás en lugar de frontalmente y, por lo tanto, no deberían tener influencia visual en el render (aunque sí en tiempo si no son eliminadas).

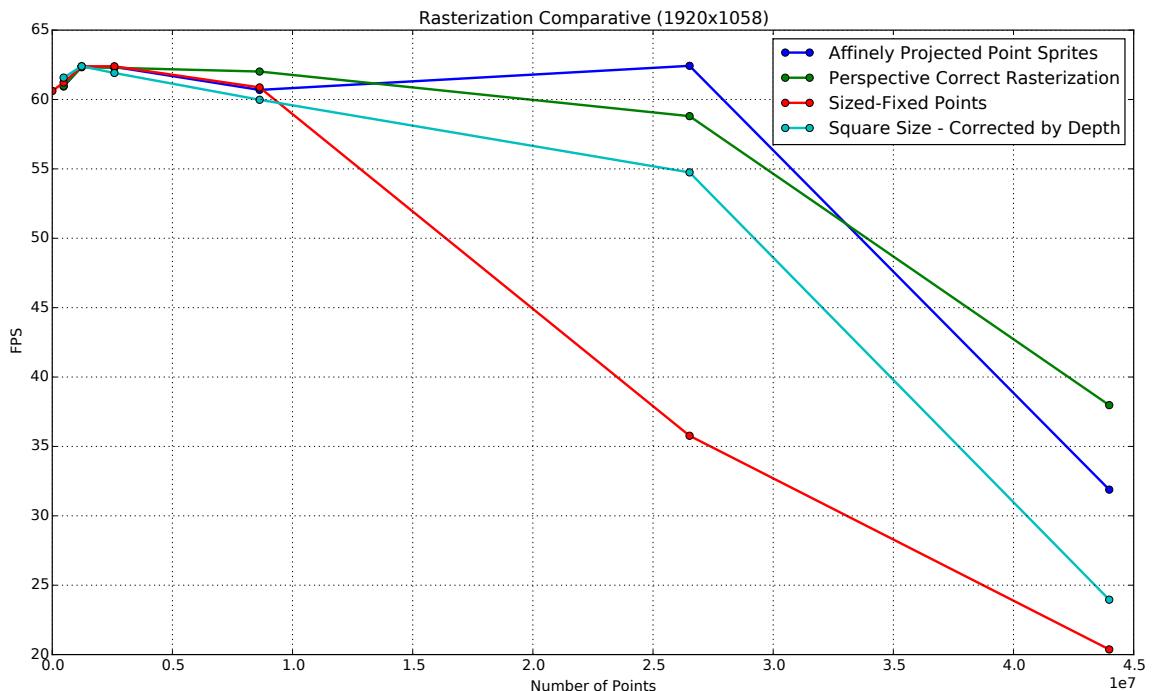


Figura 6.1: Gráfica comparativa de los 4 sistemas de rasterización.

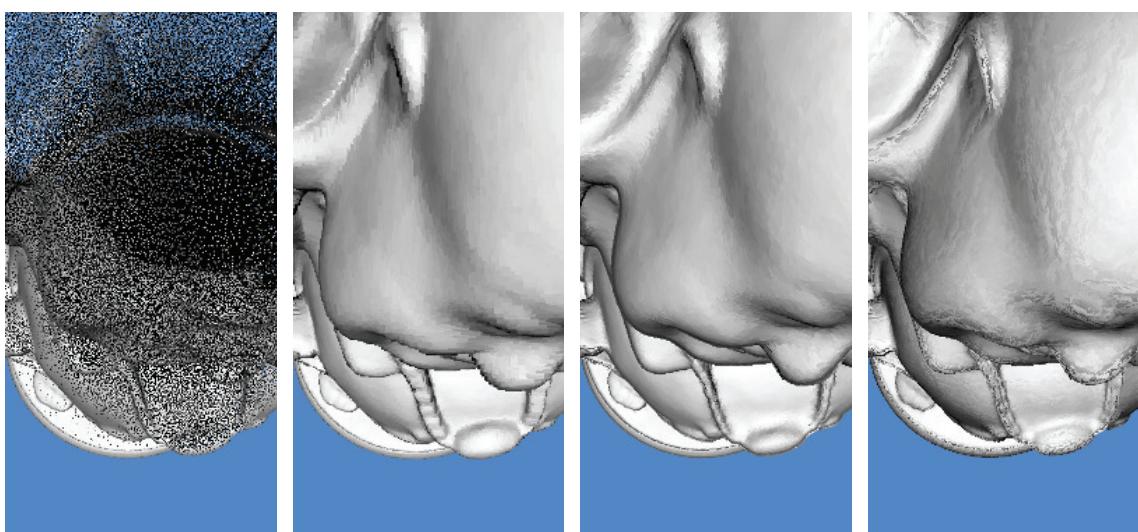


Figura 6.2: Comparativa de los resultados de render. *Sized-fixed* (**izquierda**), *Square-aligned* (**centro izquierda**), *Affinely* (**centro derecho**) y *Perspective Correct* (**derecha**).

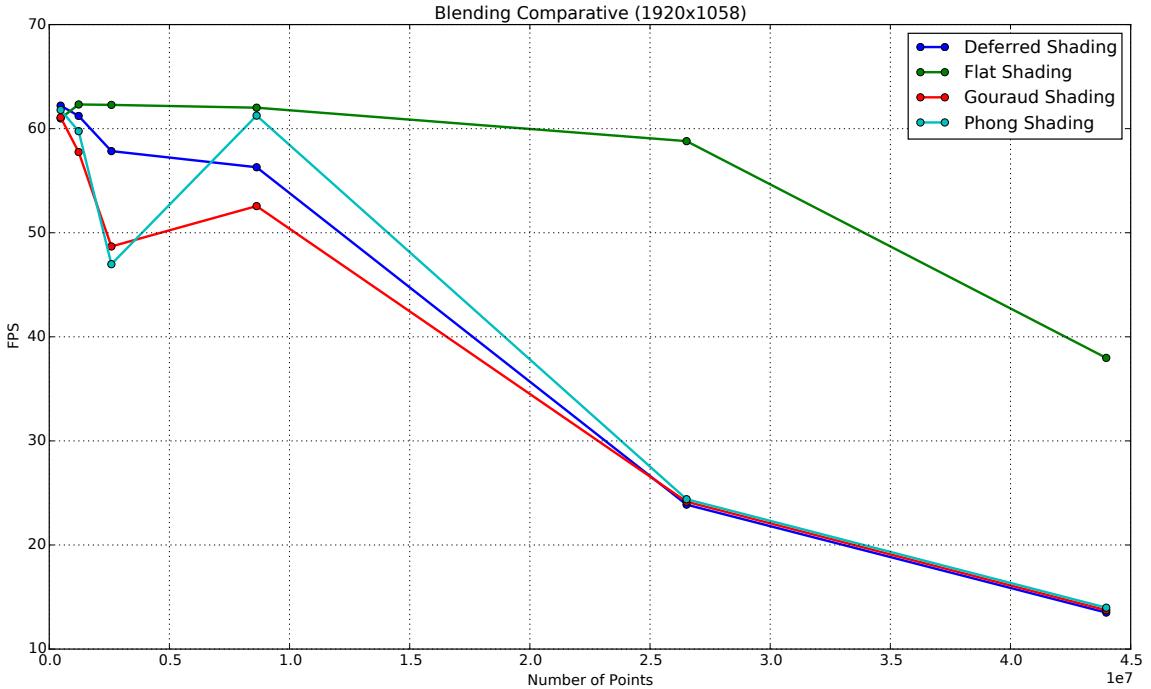


Figura 6.3: Gráfica comparativa de los 4 sistemas de blending.

6.3. Análisis de los métodos de blending

Se renderizaron los seis modelos de prueba mediante los cuatro métodos de *blending*, sin filtro de *antialiasing* ni iluminación. El resultado de este análisis lo podemos ver en la gráfica de la Figura 6.3.

Se concluyó que como se esperaba, el método de renderizado en un pase (sin *blending*) alcanza mejores tiempos de *render* en comparación con los otros tres algoritmos que constan de tres pasos para su dibujado. Teniendo estos tres un resultado bastante similar y peor con respecto al *Flat Shading*.

Comparando al acabado visual de los algoritmos (mostrado en la Figura 5.7) encontramos que:

- *Flat Shading* no elimina las discontinuidades consecuencia de la intersección y solape de los discos.
- *Gouraud Shading* elimina las discontinuidades añadiendo cierto desenfoque en su acabado.
- *Phong* como *Deferred Shading* eliminan las discontinuidades, añadiendo mas definición que el anterior, puesto que se hace una interpolación de las normales. Siendo el

resultado de estos dos métodos es prácticamente similar.

6.3.1. Comparación de los sistemas de blending con puntos de luz

Para esta prueba, los seis modelos fueron renderizados con los cuatro algoritmos de *blending* sin antialiasing y bajo la influencia de diferentes conjuntos de luces radiales. Así, para la iluminación se contó con distintas disposiciones con un número de luces variable 0, 1, 3, 5, 7 o 9 luces radiales. Los resultados se muestran en las gráficas de las Figuras 6.4 y 6.5.

Analizando los resultados obtenidos, se puede concluir que la iluminación con *Deferred Shading* es más estable ante un aumento en el número de puntos de luz de la escena. De esta forma se evidencia la superioridad de la iluminación por *píxel* en contrapuesta con los algoritmos que iluminan por *fragmento*.

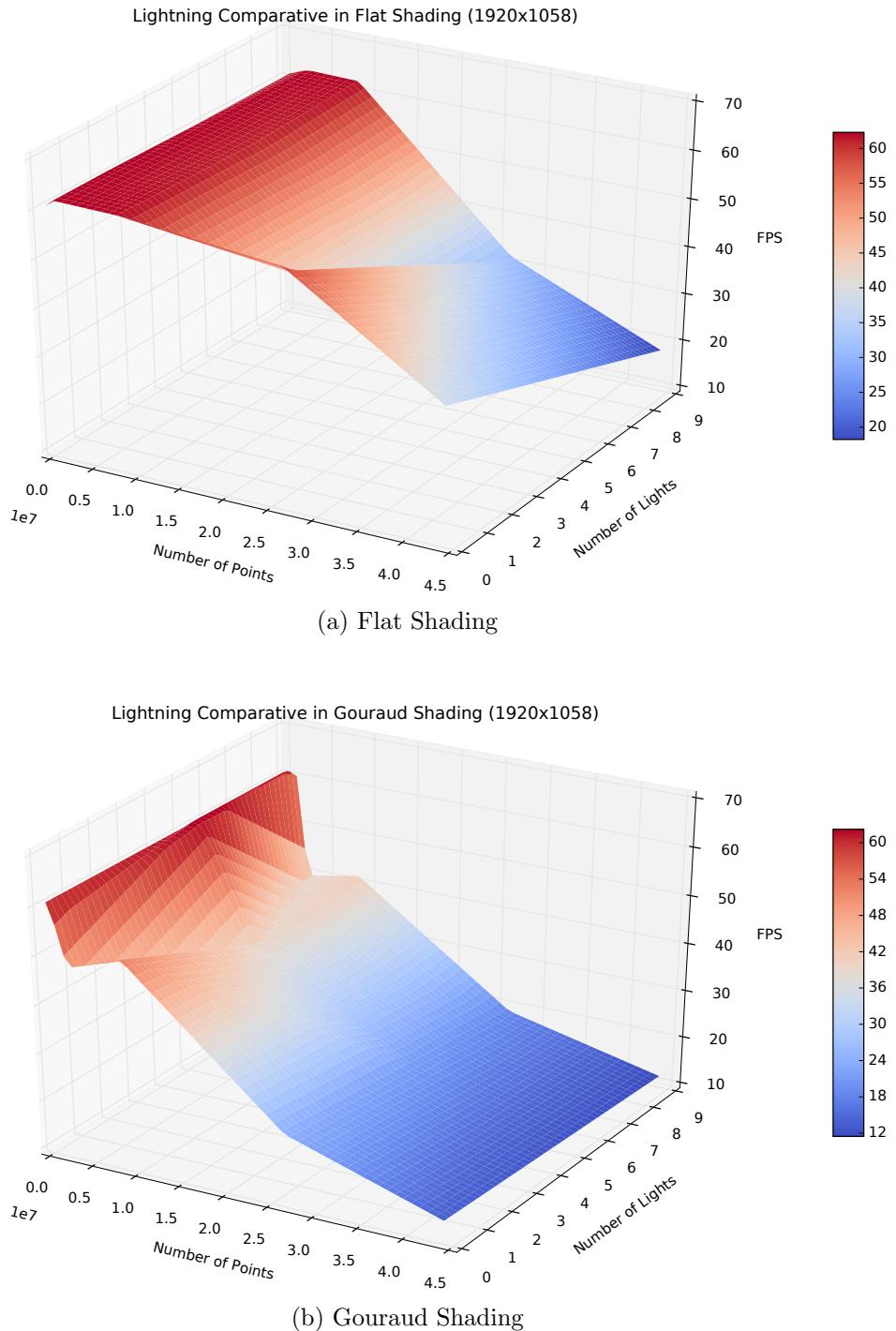


Figura 6.4: Comparación de los diferentes sistemas de blending, comparando luces con millones de puntos: flat y Gouraud.

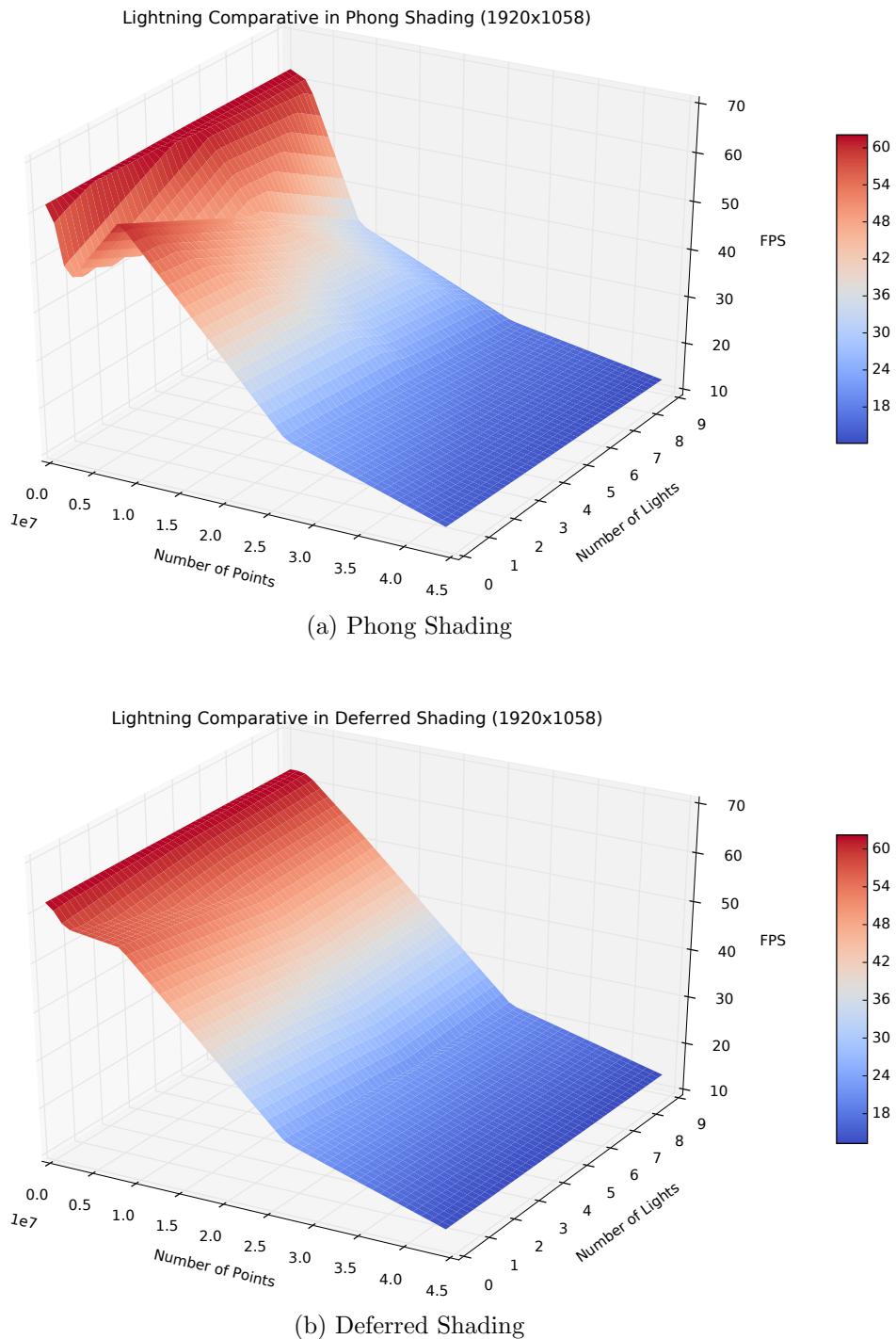


Figura 6.5: Comparación de los diferentes sistemas de blending, comparando luces con millones de puntos: Phong y Deferred.

Capítulo 7

Conclusiones

En este capítulo se comentan brevemente las principales conclusiones alcanzadas luego de la finalización de este Proyecto Fin de Carrera, continuando con las posibles líneas de trabajo que se podrían derivar del mismo.

7.1. Conclusiones

En relación a las metas fijadas antes del comienzo del proyecto, se podría decir que los siguientes objetivos fueron alcanzados:

- Aprendizaje en el uso de una librería gráfica como OpenGL.
- Programación de programas para GPU con GLSL.
- Implementación de un visualizador multiplataforma de modelos de puntos en tiempo real, que sirviera para el propósito de estudiar los diferentes algoritmos en *GPU* que existen para su *render*.
- Diseño de herramientas para el análisis de resultados mediante lenguajes ágiles.

Luego de la finalización de este proyecto, estas son las principales conclusiones alcanzadas en relación a la visualización de las nubes de puntos:

- **El mismo tamaño para todos los puntos es un error:** El tiempo necesario para calcular como mínimo la corrección de profundidad garantiza un mejora en la percepción visual además de mejoras en el tiempo de renderizado para nubes de alta densidad.

- **El aumento en la cantidad de puntos no siempre es mejor:** Al representar nubes con una muy alta densidad de puntos, implica un aumento desproporcionado del esfuerzo computacional necesario para acabar en muchos de los casos con resultados semejantes al renderizado con algoritmos simples. Incluso provocando que los algoritmos de *blending* no tengan efecto al probocar superficies de solape ínfimas impidiendo que funcionen los algoritmos correctamente.
- **Investigar en sistemas de render que no usen normales:** A pesar de que los resultados obtenidos del render mediante algoritmos como *deferred* son muy buenos. Las nubes de puntos en mucho de los *datasets* carecen de este dato. Por lo que se exige de una precomputación que una vez superando cierto tamaño empieza a no ser abordable, además de que las normales obtenidas en muchos de los casos son erroneas probocando que las imágenes resultantes no se vean correctamente.

7.2. Posibles vías de desarrollo

Con la finalización del desarrollo de este proyecto, surgen diferentes ideas para continuar con el trabajo iniciado en el mismo:

- **Rendimiento:** A pesar de que se intentó que el rendimiento obtenido del programa fuera lo más eficiente posible, habría que repasar y analizar bien las especificaciones del *hardware* gráfico para que las transacciones de datos fuera mas eficientes, intentando evitar sobrecomputación evitable con buenas praxis.
- **Interfaz de usuario:** Hacer una migración de *GLFW* a *Qt* con la idea de añadir una interfaz de usuario y poder ofrecer más información y opciones al usuario.
- **Shaders:** Permitir al usuario el escribir sus propios shaders y probarlos online con los modelos cargados.
- **Cámara:** Mejorar el funcionamiento de la cámara orbital, además de añadir mejoras en esta para poder moverse con más libertad por la escena.
- **Frustum culling:** Ahora mismo todos los puntos de la nube son enviados a GPU, sería interesante en desarrollar algún sistema para filtrar estos datos mediante algún método de *hashing* espacial en función de la cámara antes de ser enviados para dibujar.
- **Order-independent transparency:** Implementar un sistema para intentar simplificar el *blending* en tres pasos a un sistema en un único pase que además pueda ser aplicado a todos los algoritmos de rasterización.

Apéndice A

Shaders.

A.1. Sized-Fixed Points

Vertex Shader

```
#version 400
uniform mat4 viewMatrix, projMatrix;
uniform mat3 normalMatrix;

in vec3 in_Position;
in vec3 in_Color;

out vec3 ex_Color;

void main(void) {
    gl_Position = projMatrix * viewMatrix * vec4(in_Position, 1.0);
    gl_PointSize = 2;

    vec3 color = vec3 (0.0, 0.0f, 0.0f);

    ex_Color = in_Color;
}
```

Fragment Shader

```
#version 400
in vec3 ex_Color;
out vec4 out_Color;

void main(void) {
    out_Color = vec4(ex_Color, 1.0);
}
```

A.2. Image-aligned Squares

Vertex Shader

```
#version 400
uniform mat4 viewMatrix, projMatrix;
uniform mat3 normalMatrix;
uniform int h; //Height of the viewport
uniform float n; //Near parameter of the viewing frustum
uniform float t; //Top parameter of the viewing frustum
uniform float b; //Bottom parameter of the viewing frustum

in float in_Radius;
in vec3 in_Position;
in vec3 in_Color;

out float ex_Radius;
out vec3 ex_Color;

vec4 ccPosition; //position in Camera Coordinates

void main(void) {
    ex_Radius = in_Radius;

    ccPosition = viewMatrix * vec4(in_Position, 1.0);
    gl_Position = projMatrix * ccPosition;
    gl_PointSize = 2 * ex_Radius * (n / ccPosition.z) * (h / (t-b));

    vec3 color = vec3(0.0, 0.0f, 0.0f);

    ex_Color = in_Color;
}
```

Fragment Shader

```
#version 400
uniform float n; //Near parameter of the viewing frustum
uniform float f; //Far parameter of the viewing frustum

in vec3 ex_Color;
out vec4 out_Color;

void main(void) {
    out_Color = vec4(ex_Color, 1.0);
}
```

A.3. Affinely Projected Point Sprites

Vertex Shader

```
#version 400
uniform mat4 viewMatrix, projMatrix;
uniform mat3 normalMatrix;
uniform int h; //Height of the viewport
uniform float n; //Near parameter of the viewing frustum
uniform float t; //Top parameter of the viewing frustum
uniform float b; //Bottom parameter of the viewing frustum

in float in_Radius;
in vec3 in_Position;
in vec3 in_Color;
in vec3 in_Normals;

out float ex_Radius;
out vec3 ex_Color;
out vec3 ex_Normals;
out float ex_Pz; //z in Camera Coordinates

vec4 ccPosition; //position in Camera Coordinates

void main(void) {
    ex_Normals = normalize(normalMatrix * in_Normals);

    if (abs(ex_Normals.z) <= 0.1)
        ex_Normals.z = 0.1;

    ex_Radius = in_Radius;

    ccPosition = viewMatrix * vec4(in_Position, 1.0);
    gl_Position = projMatrix * ccPosition;
    gl_PointSize = 2*ex_Radius * (n / ccPosition.z) * (h / (t-b));

    //BackFace Culling
    if (dot(ccPosition.xyz, ex_Normals) > 0)
        gl_Position.w = 0;

    vec3 color = vec3(0.0, 0.0f, 0.0f);

    ex_Color = in_Color;

    ex_Pz = ccPosition.z;
}
```

Fragment Shader

```
#version 400
uniform float n; //Near parameter of the viewing frustum
uniform float f; //Far parameter of the viewing frustum

in vec3 ex_Color;
in vec3 ex_Normals;
in float ex_Pz;

out vec4 out_Color;

vec3 test;
float zBuffer;

void main(void) {
    test.x = gl_PointCoord.x - 0.5;
    test.y = gl_PointCoord.y - 0.5;
    test.z = -(ex_Normals.x/ex_Normals.z) * test.x - (ex_Normals.y/
        ex_Normals.z) * test.y;
    if (length(test) > 0.5)
        discard;

    zBuffer = (ex_Pz + test.z * 0.1);
    gl_FragDepth = ((1.0 / zBuffer) * ( (f * n) / (f - n) ) + ( f / (f - n)
        ));

    out_Color = vec4(ex_Color, 1.0);
}
```

A.4. Perspective Correct Rasterization

Vertex Shader

```
#version 400
uniform mat4 viewMatrix, projMatrix;
uniform mat3 normalMatrix;
uniform int h; //Height of the viewport
uniform float n; //Near parameter of the viewing frustum
uniform float t; //Top parameter of the viewing frustum
uniform float b; //Bottom parameter of the viewing frustum
uniform float userRadiusFactor; //Splat's radii
uniform bool automaticRadiusEnabled;

in float in_Radius;
in vec3 in_Position;
in vec3 in_Color;
in vec3 in_Normals;

out vec3 ex_Color;
out float ex_Radius;

out vec4 ccPosition; //position in Camera Coordinates
out vec3 normals;

void main(void) {
    normals = normalize(normalMatrix * in_Normals);

    ex_Radius = in_Radius;

    ccPosition = viewMatrix * vec4(in_Position, 1.0);
    gl_Position = projMatrix * ccPosition;
    gl_PointSize = 2 * ex_Radius * (n / ccPosition.z) * (h / (t-b));

    //BackFace Culling
    if (dot(ccPosition.xyz, normals) > 0)
        gl_Position.w = 0;

    ex_Color = in_Color;
}
```

Fragment Shader

```

#version 400
uniform mat4 viewMatrix;
uniform float n; //Near parameter of the viewing frustum
uniform float f; //Far parameter of the viewing frustum
uniform float t; //Top parameter of the viewing frustum
uniform float b; //Bottom parameter of the viewing frustum
uniform float r; //Right parameter of the viewing frustum
uniform float l; //Left parameter of the viewing frustum
uniform int h; //Height of the viewport
uniform int w; //Width of the viewport
uniform int lightCount;
uniform vec3 lightPosition[16];
uniform vec3 lightColor[16];
uniform float lightIntensity[16];

in float ex_Radius;
in vec3 ex_Color;
in vec3 normals;
in vec4 ccPosition;

out vec4 out_Color;

void main(void) {
    vec3 qn;
    qn.x = (gl_FragCoord.x) * ((r - l)/w) - ((r - l)/2.0);
    qn.y = (gl_FragCoord.y) * ((b - t)/h) - ((b - t)/2.0);
    qn.z = -n;

    float denom = dot (qn, normals);
    if (denom == 0.0)
        discard;

    float timef = dot (ccPosition.xyz, normals) / denom;
    vec3 q = qn * timef;
    vec3 testq = q;
    vec3 dist = (q - ccPosition.xyz);

    if ((dist.x * dist.x) + (dist.y * dist.y) + (dist.z * dist.z) > pow(ex_Radius, 2))
        discard;

    gl_FragDepth = ((1.0 / q.z) * ((f * n) / (f - n)) + (f / (f - n)));
    out_Color = vec4(ex_Color, 1.0f);
}

```

A.5. Gouraud

A.5.1. Visibility Pass

Vertex Shader

```
#version 410
uniform mat4 viewMatrix, projMatrix;
uniform mat3 normalMatrix;
uniform int h; //Height of the viewport
uniform float n; //Near parameter of the viewing frustum
uniform float t; //Top parameter of the viewing frustum
uniform float b; //Bottom parameter of the viewing frustum

in vec3 in_Position;
in vec3 in_Normals;
in float in_Radius;

out float ex_Radius;

out vec4 ccPosition; //position in Camera Coordinates
out vec3 normals;

void main(void) {
    ex_Radius = in_Radius;

    normals = normalize(normalMatrix * in_Normals);

    ccPosition = viewMatrix * vec4(in_Position, 1.0);
    gl_Position = projMatrix * ccPosition;
    gl_PointSize = 2 * ex_Radius * (n / ccPosition.z) * (h / (t-b));
}
```

Fragment Shader

```
#version 410
uniform float n; //Near parameter of the viewing frustum
uniform float f; //Far parameter of the viewing frustum
uniform float t; //Top parameter of the viewing frustum
uniform float b; //Bottom parameter of the viewing frustum
uniform float r; //Right parameter of the viewing frustum
uniform float l; //Left parameter of the viewing frustum
uniform int h; //Height of the viewport
uniform int w; //Width of the viewport

in float ex_Radius;
in vec3 normals;
in vec4 ccPosition;

void main(void) {
    vec3 qn;
    qn.x = (gl_FragCoord.x) * ((r - l)/w) - ((r - l)/2.0);
    qn.y = (gl_FragCoord.y) * ((b - t)/h) - ((b - t)/2.0);
    qn.z = -n;

    float denom = dot(qn, normals);

    if (denom == 0.0)
        discard;

    float timef = dot(ccPosition.xyz, normals) / denom;

    vec3 q = qn * timef;

    vec3 dist = (q - ccPosition.xyz);

    if ((dist.x * dist.x) + (dist.y * dist.y) + (dist.z * dist.z) > pow(ex_Radius, 2))
        discard;

    gl_FragDepth = ((1.0 / q.z) * ((f * n) / (f - n)) + (f / (f - n)));
}
```

A.5.2. Blending Pass

Vertex Shader

```
#version 410
uniform mat4 viewMatrix, projMatrix;
uniform mat3 normalMatrix;
uniform int h; //Height of the viewport
uniform float n; //Near parameter of the viewing frustum
uniform float t; //Top parameter of the viewing frustum
uniform float b; //Bottom parameter of the viewing frustum

in float in_Radius;
in vec3 in_Position;
in vec3 in_Color;
in vec3 in_Normals;

out vec3 ex_Color;
out float ex_Radius;

out vec4 ccPosition; //position in Camera Coordinates
out vec3 normals;

void main(void) {
    normals = normalize(normalMatrix * in_Normals);

    ex_Radius = in_Radius;

    ccPosition = viewMatrix * vec4(in_Position, 1.0);
    gl_Position = projMatrix * ccPosition;
    gl_PointSize = 2*ex_Radius * (n / ccPosition.z) * (h / (t-b));

    //Backface Culling
    if (dot(ccPosition.xyz, normals) > 0)
        gl_Position.w = 0;

    ex_Color = in_Color;
}
```

Fragment Shader

```

#version 410
uniform mat4 viewMatrix;
uniform float n; //Near parameter of the viewing frustum
uniform float f; //Far parameter of the viewing frustum
uniform float t; //Top parameter of the viewing frustum
uniform float b; //Bottom parameter of the viewing frustum
uniform float r; //Right parameter of the viewing frustum
uniform float l; //Left parameter of the viewing frustum
uniform int h; //Height of the viewport
uniform int w; //Width of the viewport

uniform int lightCount;
uniform vec3 lightPosition[16];
uniform vec3 lightColor[16];
uniform float lightIntensity[16];

in float ex_Radius;
in vec3 ex_Color;
in vec3 ex_UxV;
in vec3 normals;
in vec4 ccPosition;

layout (location = 0) out vec4 out_Color;

void main(void) {
    vec3 qn;
    qn.x = (gl_FragCoord.x) * ((r - l)/w) - ((r - l)/2.0);
    qn.y = (gl_FragCoord.y) * ((b - t)/h) - ((b - t)/2.0);
    qn.z = -n;

    float denom = dot (qn, normals);

    if (denom == 0.0)
        discard;

    float timef = dot (ccPosition.xyz, normals) / denom;

    vec3 q = qn * timef;
    vec3 testq = q;

    vec3 dist = (q - ccPosition.xyz);

    if ((dist.x * dist.x) + (dist.y * dist.y) + (dist.z * dist.z) > pow(
        ex_Radius, 2))
        discard;
}

```

```
vec3 epsilon = normalize(qn)/40.0f;
q = q - epsilon;

gl_FragDepth = ((1.0 / q.z) * ( (f * n) / (f - n) ) + ( f / (f - n) ));
float weight = (1.0f - length(dist)/ex_Radius);

vec3 color = ex_Color;

//Diffuse
vec3 dotValue = vec3(0,0,0);
for (int i = 0; i < lightCount; i++) {
    vec3 ccLightPosition = (viewMatrix * vec4(lightPosition[i], 1.0f)).xyz
    ;
    vec3 lithToQ = normalize(ccLightPosition - testq);
    dotValue += vec3(max(dot(normals, lithToQ), 0.0)) * lightIntensity[i]
        * lightColor[i];
}

out_Color = vec4(dotValue + color, 1.0f);
}
```

A.5.3. Normalization Pass

Vertex Shader

```
#version 410
in vec3 in_Position;
in vec3 in_Color;

out vec4 out_Color;

void main(void) {
    gl_Position = vec4(in_Position, 1.0);
}
```

Fragment Shader

```
#version 410
uniform sampler2DRect blendTexture;

out vec4 out_Color;

void main(void) {
    vec4 textureColor = texture(blendTexture, gl_FragCoord.xy);

    if (textureColor.a <= 0.0f)
        discard;

    out_Color = vec4(textureColor.rgb / textureColor.a, 1.0f);
}
```

A.6. Phong

A.6.1. Visibility Pass

Vertex Shader

```
#version 410
uniform mat4 viewMatrix, projMatrix;
uniform mat3 normalMatrix;
uniform int h; //Height of the viewport
uniform float n; //Near parameter of the viewing frustum
uniform float t; //Top parameter of the viewing frustum
uniform float b; //Bottom parameter of the viewing frustum

in vec3 in_Position;
in vec3 in_Normals;
in float in_Radius;

out float ex_Radius;

out vec4 ccPosition; //position in Camera Coordinates
out vec3 normals;

void main(void) {
    ex_Radius = in_Radius;

    normals = normalize(normalMatrix * in_Normals);

    ccPosition = viewMatrix * vec4(in_Position, 1.0);
    gl_Position = projMatrix * ccPosition;
    gl_PointSize = 2 * ex_Radius * (n / ccPosition.z) * (h / (t-b));
}
```

Fragment Shader

```
#version 410
uniform float n; //Near parameter of the viewing frustum
uniform float f; //Far parameter of the viewing frustum
uniform float t; //Top parameter of the viewing frustum
uniform float b; //Bottom parameter of the viewing frustum
uniform float r; //Right parameter of the viewing frustum
uniform float l; //Left parameter of the viewing frustum
uniform int h; //Height of the viewport
uniform int w; //Width of the viewport

in float ex_Radius;
in vec3 normals;
in vec4 ccPosition;

void main(void) {
    vec3 qn;
    qn.x = (gl_FragCoord.x) * ((r - l)/w) - ((r - l)/2.0);
    qn.y = (gl_FragCoord.y) * ((b - t)/h) - ((b - t)/2.0);
    qn.z = -n;

    float denom = dot(qn, normals);

    if (denom == 0.0)
        discard;

    float timef = dot(ccPosition.xyz, normals) / denom;

    vec3 q = qn * timef;

    vec3 dist = (q - ccPosition.xyz);

    if ((dist.x * dist.x) + (dist.y * dist.y) + (dist.z * dist.z) > pow(ex_Radius, 2))
        discard;

    gl_FragDepth = ((1.0 / q.z) * ((f * n) / (f - n)) + (f / (f - n)));
}
```

A.6.2. Blending Pass

Vertex Shader

```
#version 400
uniform mat4 viewMatrix, projMatrix;
uniform mat3 normalMatrix;
uniform int h; //Height of the viewport
uniform float n; //Near parameter of the viewing frustum
uniform float t; //Top parameter of the viewing frustum
uniform float b; //Bottom parameter of the viewing frustum

in float in_Radius;
in vec3 in_Position;
in vec3 in_Color;
in vec3 in_Normals;

out vec3 ex_Color;
out float ex_Radius;

out vec4 ccPosition; //position in Camera Coordinates
out vec3 normals;

void main(void) {
    normals = normalize(normalMatrix * in_Normals);

    ex_Radius = in_Radius;

    //p. 277
    ccPosition = viewMatrix * vec4(in_Position, 1.0);
    gl_Position = projMatrix * ccPosition;
    gl_PointSize = 2 * ex_Radius * (n / ccPosition.z) * (h / (t-b));

    //BackFace Culling
    if (dot(ccPosition.xyz, normals) > 0)
        gl_Position.w = 0;

    ex_Color = in_Color;
}
```

Fragment Shader

```

#version 400
uniform mat4 viewMatrix;
uniform float n; //Near parameter of the viewing frustum
uniform float f; //Far parameter of the viewing frustum
uniform float t; //Top parameter of the viewing frustum
uniform float b; //Bottom parameter of the viewing frustum
uniform float r; //Right parameter of the viewing frustum
uniform float l; //Left parameter of the viewing frustum
uniform int h; //Height of the viewport
uniform int w; //Width of the viewport
uniform bool colorEnabled;
uniform int lightCount;
uniform vec3 lightPosition[16];
uniform vec3 lightColor[16];
uniform float lightIntensity[16];

in float ex_Radius;
in vec3 ex_Color;
in vec3 ex_UxV;
in vec3 normals;
in vec4 ccPosition;

out vec4 out_Color;

void main(void) {
    vec3 qn;
    qn.x = (gl_FragCoord.x) * ((r - l)/w) - ((r - l)/2.0);
    qn.y = (gl_FragCoord.y) * ((b - t)/h) - ((b - t)/2.0);
    qn.z = -n;

    float denom = dot (qn, normals);

    if (denom == 0.0)
        discard;

    float timef = dot (ccPosition.xyz, normals) / denom;

    vec3 q = qn * timef;
    vec3 testq = q;

    vec3 dist = (q - ccPosition.xyz);

    if ((dist.x * dist.x) + (dist.y * dist.y) + (dist.z * dist.z) > pow(
        ex_Radius, 2))
        discard;
}

```

```
vec3 epsilon = normalize(qn)/40.0f;
q = q - epsilon;

gl_FragDepth = ((1.0 / q.z) * ( (f * n) / (f - n) ) + ( f / (f - n) ));

//Phong
float weight = (1.0f - length(dist)/ex_Radius);
vec3 phongNormal = normalize((q + normals) - ccPosition.xyz);

vec3 color = ex_Color;
//Diffuse
vec3 dotValue = vec3(0,0,0);
for (int i = 0; i < lightCount; i++) {
    vec3 ccLightPosition = (viewMatrix * vec4(lightPosition[i], 1.0f)).xyz
    ;
    vec3 lithToQ = normalize(ccLightPosition - testq);
    dotValue += vec3(max(dot(normals, lithToQ), 0.0)) * lightIntensity[i]
        * lightColor[i];
}

out_Color = vec4(dotValue + color, 1.0f * weight);
}
```

A.6.3. Normalization Pass

Vertex Shader

```
#version 410
in vec3 in_Position;
in vec3 in_Color;

out vec4 out_Color;

void main(void) {
    gl_Position = vec4(in_Position, 1.0);
}
```

Fragment Shader

```
#version 410
uniform sampler2DRect blendTexture;

out vec4 out_Color;

void main(void) {
    vec4 textureColor = texture(blendTexture, gl_FragCoord.xy);

    if (textureColor.a <= 0.0f)
        discard;

    out_Color = vec4(textureColor.rgb / textureColor.a, 1.0f);
}
```

A.7. Deferred

A.7.1. Visibility Pass

Vertex Shader

```
#version 410
uniform mat4 viewMatrix, projMatrix;
uniform mat3 normalMatrix;
uniform int h; //Height of the viewport
uniform float n; //Near parameter of the viewing frustum
uniform float t; //Top parameter of the viewing frustum
uniform float b; //Bottom parameter of the viewing frustum

in vec3 in_Position;
in vec3 in_Normals;
in float in_Radius;

out float ex_Radius;

out vec4 ccPosition; //position in Camera Coordinates
out vec3 normals;

void main(void) {
    ex_Radius = in_Radius;

    normals = normalize(normalMatrix * in_Normals);

    ccPosition = viewMatrix * vec4(in_Position, 1.0);
    gl_Position = projMatrix * ccPosition;
    gl_PointSize = 2 * ex_Radius * (n / ccPosition.z) * (h / (t-b));
}
```

Fragment Shader

```
#version 410

uniform float n; //Near parameter of the viewing frustum
uniform float f; //Far parameter of the viewing frustum
uniform float t; //Top parameter of the viewing frustum
uniform float b; //Bottom parameter of the viewing frustum
uniform float r; //Right parameter of the viewing frustum
uniform float l; //Left parameter of the viewing frustum
uniform int h; //Height of the viewport
uniform int w; //Width of the viewport

in float ex_Radius;
in vec3 normals;
in vec4 ccPosition;

layout (location = 1) out vec3 out_Position;

void main(void) {
    vec3 qn;
    qn.x = (gl_FragCoord.x) * ((r - l)/w) - ((r - l)/2.0);
    qn.y = (gl_FragCoord.y) * ((b - t)/h) - ((b - t)/2.0);
    qn.z = -n;

    float denom = dot (qn, normals);

    if (denom == 0.0)
        discard;

    float timef = dot (ccPosition.xyz, normals) / denom;

    vec3 q = qn * timef;

    vec3 dist = (q - ccPosition.xyz);

    if ((dist.x * dist.x) + (dist.y * dist.y) + (dist.z * dist.z) > pow(ex_Radius, 2))
        discard;

    out_Position = q;
    gl_FragDepth = ((1.0 / q.z) * ((f * n) / (f - n)) + (f / (f - n)));
}
```

A.7.2. Blending Pass

Vertex Shader

```
#version 410
uniform mat4 viewMatrix, projMatrix;
uniform mat3 normalMatrix;
uniform int h; //Height of the viewport
uniform float n; //Near parameter of the viewing frustum
uniform float t; //Top parameter of the viewing frustum
uniform float b; //Bottom parameter of the viewing frustum

in float in_Radius;
in vec3 in_Position;
in vec3 in_Color;
in vec3 in_Normals;

out vec3 ex_Color;
out float ex_Radius;

out vec4 ccPosition; //position in Camera Coordinates
out vec3 normals;

void main(void) {
    normals = normalize(normalMatrix * in_Normals);

    ex_Radius = in_Radius;

    ccPosition = viewMatrix * vec4(in_Position, 1.0);
    gl_Position = projMatrix * ccPosition;
    gl_PointSize = 2*ex_Radius * (n / ccPosition.z) * (h / (t-b));

    //Backface Culling
    if (dot(ccPosition.xyz, normals) > 0)
        gl_Position.w = 0;

    ex_Color = in_Color;
}
```

Fragment Shader

```

#version 410
uniform float n; //Near parameter of the viewing frustum
uniform float f; //Far parameter of the viewing frustum
uniform float t; //Top parameter of the viewing frustum
uniform float b; //Bottom parameter of the viewing frustum
uniform float r; //Right parameter of the viewing frustum
uniform float l; //Left parameter of the viewing frustum
uniform int h; //Height of the viewport
uniform int w; //Width of the viewport

in float ex_Radius;
in vec3 ex_Color;
in vec3 normals;
in vec4 ccPosition;

layout (location = 0) out vec4 out_Color;
layout (location = 1) out vec4 out_Normals;

void main(void) {
    vec3 qn;
    qn.x = (gl_FragCoord.x) * ((r - 1)/w) - ((r - 1)/2.0);
    qn.y = (gl_FragCoord.y) * ((b - t)/h) - ((b - t)/2.0);
    qn.z = -n;

    float denom = dot (qn, normals);
    if (denom == 0.0)
        discard;

    float timef = dot (ccPosition.xyz, normals) / denom;
    vec3 q = qn * timef;
    vec3 dist = (q - ccPosition.xyz);

    if ((dist.x * dist.x) + (dist.y * dist.y) + (dist.z * dist.z) > pow(ex_Radius, 2))
        discard;

    vec3 epsilon = normalize(qn)/40.0f;
    q = q - epsilon;

    gl_FragDepth = ((1.0 / q.z) * ((f * n) / (f - n)) + (f / (f - n)));
    float weight = (1.0f - length(dist)/ex_Radius);

    out_Color = vec4(ex_Color.rgb, 1.0f * weight);
    out_Normals = vec4(normals, 1.0f * weight);
}

```

A.7.3. Normalization Pass

Vertex Shader

```
#version 400
in vec3 in_Position;
in vec3 in_Color;

out vec4 out_Color;

void main(void) {
    gl_Position = vec4(in_Position, 1.0);
}
```

Fragment Shader

```
#version 410
uniform mat4 viewMatrix;
uniform float n; //Near parameter of the viewing frustum
uniform float f; //Far parameter of the viewing frustum
uniform float t; //Top parameter of the viewing frustum
uniform float b; //Bottom parameter of the viewing frustum
uniform float r; //Right parameter of the viewing frustum
uniform float l; //Left parameter of the viewing frustum
uniform int h; //Height of the viewport
uniform int w; //Width of the viewport

uniform int lightCount;
uniform vec3 lightPosition[16];
uniform vec3 lightColor[16];
uniform float lightIntensity[16];

uniform sampler2DRect blendTexture;
uniform sampler2DRect normalTexture;
uniform sampler2DRect positionTexture;

out vec4 out_Color;

void main(void) {
    vec4 textureColor = texture(blendTexture, gl_FragCoord.xy);

    if (textureColor.a <= 0.0f)
        discard;

    vec4 textureNormal = texture(normalTexture, gl_FragCoord.xy);

    vec4 normalizedColor = vec4(textureColor.rgb / textureColor.a, 1.0f);
```

```
vec4 normalizedNormal = vec4(textureNormal.xyz / textureNormal.w, 1.0f);
vec3 q = texture(positionTexture, gl_FragCoord.xy).xyz;

vec3 color = normalizedColor.rgb;

//Lightning with the resultant normalized textures
vec3 dotValue = vec3(0,0,0);
for (int i = 0; i < lightCount; i++) {
    vec3 ccLightPosition = (viewMatrix * vec4(lightPosition[i], 1.0f)).xyz;
    vec3 ligthToQ = normalize(ccLightPosition - q);
    dotValue += vec3(max(dot(normalize(normalizedNormal.xyz), ligthToQ),
        0.0)) * lightIntensity[i] * lightColor[i];
}

out_Color = vec4(dotValue + color, 1.0f);
}
```

Bibliografía

- [1] Tomas Akenine-Moller, Eric Haines, and Naty Hoffman. *Real-Time Rendering*. AK Peters/CRC Press; Third edition, 2008. 20
- [2] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change*. Addison Wesley; Second edition, 2004. 1
- [3] Mario Botsch, Alexander Hornung, Matthias Zwicker, and Leif Kobbelt. High-quality surface splatting on today's gpus. In *Proc. of Symposium on Point-based Graphics 2005*, pages 17–24, 2005. 44
- [4] Mario Botsch and Leif Kobbelt. High-quality point-based rendering on modern gpus. In *Proc. of Pacific Graphics 03*, pages 335–343, 2003. 36, 41
- [5] Mario Botsch, Michael Spernati, and Leif Kobbelt. Phong splatting. In *Proc. of Symposium on Point-based Graphics 04*, pages 25–32, 2004. 37, 43
- [6] James D. Foley, Andries van Dam, and Steven K. Feiner. *Computer Graphics: Principles and Practice*. Addison Wesley; Third edition, 2013. 19, 21
- [7] D. Girardeau-Montaut. *Détection de changement sur des données géométriques tridimensionnelles*. PhD thesis, Telecom ParisTech, 2006. 12
- [8] Markus Gross and Hanspeter Pfister. *Point-Based Graphics*. Morgan Kaufmann Publishers Inc., 2007. xxi
- [9] Gaël Guennebaud and Mathias Paulin. Efficient screen space approach for hardware accelerated surfel rendering. In *Vision, Modeling and Visualization (VMV'03)*, pages 485–495, 2003. 41
- [10] Matt Pharr and Greg Humphreys. *Physically Based Rendering: From Theory To Implementation*. Morgan Kaufmann; Second edition, 2010. 20

- [11] Szymon Rusinkiewicz and Marc Levoy. Qsplat: A multiresolution point-rendering system for large meshes. *Computer Graphics, SIGGRAPH 2000 Proceedings*, pages 343–352, 2000. xxI
- [12] Radu Bogdan Rusu and Steve Cousins. 3D is here: Point Cloud Library (PCL). In *Proc. of IEEE International Conference on Robotics and Automation (ICRA)*, Shanghai, China, May 9-13 2011. 16
- [13] Graham Sellers, Richard S Wright Jr., and Nicholas Haemel. *OpenGL SuperBible: Comprehensive Tutorial and Reference*. Addison-Wesley Professional; 6th edition, 2013. 13
- [14] Matthias Zwicker, Hanspeter Pfister, Jeroen van Baar, and Markus Gross. Surface splatting. *Computer Graphics, SIGGRAPH 2001 Proceedings*, pages 371–378, 2001. 42
- [15] Matthias Zwicker, Jussi Räsänen, Mario Botsch, Carsten Dachsbacher, and Mark Pauly. Perspective accurate splatting. In *Proc. of Graphics Interface (GI'04)*, pages 247–254, 2004. 35