GRAPHICS SHADERS FOR SCIENTIFIC VISUALIZATION

DAVID FERNÁNDEZ ALCOBA

DOBLE GRADO EN INGENIERÍA INFORMÁTICA - MATEMÁTICAS
FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTESNE DE MADRID

Trabajo de Fin de Grado en Ingeniería Informática - Matemáticas

19 de septiembre de 2019

Directora:

Ana Gil Luezas

Resumen

En el mundo actual, las investigaciones y estudios científicos generan gran cantidad de datos que han de ser interpretados de una manera eficaz, con el fin de sacar las mejores conclusiones y obtener resultados fiables que no den lugar a la duda. En este contexto, una de las disciplinas más importantes es la de la visualización, ya que puede ayudar a entender, ilustrar y obtener información relevante acerca del fenómeno que se está estudiando.

De igual forma, en los últimos años se ha dado una expansión considerable de las capacidades de las GPUs, ofreciendo nuevas posibilidades dentro de la informática gráfica e incrementando el rendimiento tanto en computación paralela como en aplicaciones de visualización.

En este trabajo se exploran estas ideas, haciendo hincapié en las posibilidades que nos ofrecen los distintos tipos de shaders gráficos dentro de la especificación OpenGL, y la manera en la que nos pueden ser útiles a la hora de interpretar datos y obtener representaciones para problemas típicos de visualización científica, como puede ser la visualización de datos en tres dimensiones, visualización de volúmenes, renderizado de curvas y superficies, etc.

Palabras clave

Visualización científica, GPU, Shader, OpenGL, Bézier, Superficies.

Abstract

Nowadays, scientific studies and investigations generate a great amount of data that has to be well interpreted, so as to extract the best possible conclusions and obtain reliable results. Within this context, one of the most important disciplines is that of visualization, since it can help understand, illustrate and obtain relevant information about the phenomenon being studied.

Similarly, in the last few years, a considerable expansion of the capabilities of GPUs has been taking place, offering new possibilities within graphics computing and increasing performance both in parallel computing and in visualization applications.

In this text those ideas are explored, emphasizing the possibilities that the different kinds of graphic shaders have to offer within the OpenGL specification, and the way these can help interpret data and obtain representations for common scientific visualization topics, such as three dimensions data Visualization, volume visualization, surface rendering, etc.

Keywords

Scientific Visualization, Graphic Shaders, GPU, OpenGL.

Índice General

Índice de Figuras

Índice de Tablas

Capítulo 1

Introducción

Tal y como se cuenta en Defanti and Brown [1], los científicos computacionales basan su trabajo en fuentes de datos de gran volumen. Sin embargo, estos datos tienen tal magnitud que los científicos se ven, a menudo, superados. Entre las fuentes de datos de gran volumen se encuentran:

- Inteligencia militar, satélites, datos astronómicos y de tiempo atmosférico
- Sondas enviando datos desde otros planetas
- Radio telescopios terrestres
- Instrumentos capturando temperaturas oceánicas, movimientos tectónicos y actividad volcánica y sísmica
- Escáneres médicos empleando distintas técnicas de imagen como tomografía, resonancias magnéticas, etc

Simplemente con un formato numérico, el cerebro humano es incapaz de interpretar gigabytes de datos cada día, resultando en mucha información desperdiciada. De aquí surge la necesidad de una alternativa a los números. La posibilidad de los científicos para visualizar cómputos complejos y simulaciones es absolutamente esencial para asegurar la integridad de análisis y predicciones, así como presentar esta información al resto.

Esta capacidad de visualización se hace especialmente importante en el ser humano, puesto que, de todas nuestras funciones cerebrales, nuestro sistema de visión es el que mayor capacidad de procesamiento de información tiene. Según expertos en conocimiento, el procesamiento de información en el ser humano tiene dos formas: preconsciente y consciente.

El procesamiento de información preconsciente es involuntario, similar a la respiración. Este es el tipo de procesamiento que se da en información gráfica. [2]

Teniendo esto en cuenta y el hecho de que cada persona tiene una capacidad de visión espacial diferente, la informática gráfica puede ayudar a aquellos que tienen una mayor dificultad y que, de otro modo, serían incapaces de visualizar conceptos complejos.

Estos hechos muestran una necesidad que ha resultado en el surgimiento, en la última década, de una disciplina totalmente independiente, la visualización científica.

1.1. Motivación

La importancia de lo expuesto anteriormente sirve como suficiente motivación, aunque a esto se ha de añadir el reto personal de, con este trabajo, aprender y entender un área de la informática que no forma parte del itinerario en mi formación, como es la informática gráfica, y que engloba muchas de las materias vistas hasta ahora tanto en ingeniería informática como en matemáticas.

Además, esta rama dentro de la investigación científica es relativamente reciente, asociándose su nacimiento en 1987 al artículo de McCormick et al. [3], por lo que aún hay muchos retos y problemas por resolver, haciendo su estudio muy interesante.

1.2. Objetivos

El objetivo principal de este trabajo es el de aprender el funcionamiento básico de los gráficos y la aplicación de éstos a la investigación científica. Este objetivo se puede desglosar en otros subobjetivos más concretos y que marcan la línea de trabajo:

1. Comprender el pipeline de gráficos y la utilidad y funcionamiento de los shaders, así como aprender el lenguaje GLSL para su escritura.

2. Aprender las técnicas más conocidas de visualización científica y cómo desarrollar shaders que las implementen.

3. Desarrollar una aplicación que ponga de manifiesto lo aprendido, desarrollando shaders que ilustren algunas de las técnicas vistas.

1.3. Plan de trabajo

Con estos objetivos en mente, se desarrolló el siguiente plan de trabajo, acordado en reuniones iniciales entre tutora y autor del trabajo.

- **Toma de contacto con OpenGL** Durante esta fase se leyeron tutoriales sobre OpenGL y se experimentó con diversos shaders y librerías para familiarizarse con la tecnología, a la vez que se aprendía el lenguaje GLSL.

- **Documentación** Durante la duración completa del proyecto se llevó a cabo una documentación acerca de las distintas fuentes de información, con el objetivo de no olvidar incluir partes importantes en la memoria.

- **Comunicación con el tutor** Se concretaron diversas reuniones con la tutora durante las partes intensivas del proyecto con el fin de mostrar avances y acordar los siguientes pasos. Asimismo, se mantuvo una comunicación mediante correo electrónico para aquellas dudas menores que surgieron durante la realización del trabajo.

- **Preparación del entorno de desarrollo** Durante esta fase se preparó el equipo, instalando las librerías y programas necesarios para el correcto funcionamiento de la aplicación.

- **Desarrollo de la aplicación** Una vez preparado el entorno, se continuó durante toda la duración del trabajo con el desarrollo de la aplicación, incluyendo cada vez nuevas capacidades.

- **Redacción de la memoria** Se inició la redacción de la memoria una vez se tenían conocimientos suficientes, a mitad de la elaboración del trabajo. Una vez comenzada la redacción, se fue reeditando y mejorando en un proceso iterativo.

1.4. Estructura de la memoria

El siguiente capítulo, **OpenGL y DirectX** 2, presenta las dos grandes especificaciones dentro de la informática gráfica, centrándose en OpenGL y analizando sus características, capacidades y debilidades, así como las diferencias entre ambas.

Posteriormente, el capítulo **Shaders y Visualización Científica** 3 explora las técnicas más comunes dentro del campo de visualización científica y qué tipos de shaders son útiles para cada una de ellas, introduciendo algunos de los que más adelante se presentarán junto

a la aplicación.

En el capítulo **Aplicación** 4 se presenta la aplicación desarrollada, explicando el diseño, capacidades, experimentos realizados...

Por último el capítulo **Conclusiones y Trabajo Futuro** 5 incluye un análisis del trabajo realizado, el nivel de cumplimiento de los objetivos propuestos y posibles líneas de trabajo futuro.

El código de la aplicación desarrollada puede encontrarse en el siguiente enlace:
http://github.com/davidfdezalcoba/TFG

Capítulo 2

OpenGL y Direct3D

El objetivo de este capítulo es explicar en qué consiste OpenGL, así como su pipeline de gráficos, los tipos de shaders que incluye y las diferencias que presenta con Direct3D, su principal competidor.

2.1. ¿Qué es?

OpenGL [4] se define como una API *(application programming interface)*, que es simplemente una librería de software para acceder a capacidades del hardware de gráficos (ver Shreiner et al. [5]).

OpenGL está diseñado como una interfaz independiente del hardware que puede ser implementada en muchos sistemas hardware de gráficos diferentes, o completamente como software, en el caso de que el sistema no posea hardware de gráficos. OpenGL no proporciona ninguna funcionalidad para describir modelos en tres dimensiones ni operaciones para leer ficheros (como imágenes JPEG, por ejemplo). En su lugar, se deben construir los objetos tridimensionales a partir de un pequeño conjunto de primitivas geométricas—points, lines, triangles y patches.—(Ver Sección 2.3).

2.2. Breve historia de OpenGL

OpenGL nace a principios de los años 90, desarrollada por Silicon Graphics (SGI). En los años 80, Silicon Graphics poseía una API privada denominada IRIS GL, utilizada para

producir gráficos en sus estaciones de trabajo IRIS. Posteriormente, debido a la pérdida de cuota de mercado, decidió hacer su API pública. Sin embargo, a causa de problemas con patentes y el hecho de tener características poco relevantes para los gráficos 3D como la funcionalidad de ventanas, se decidió reescribir algunas de las partes y se lanzó lo que ahora se conoce como OpenGL.

Esta nueva especificación consiguió logros importantes para la informática gráfica, como estandarizar el acceso al hardware gráfico, trasladar a los fabricantes la responsabilidad del desarrollo de las interfaces con el hardware y delegar la funcionalidad de ventanas al sistema operativo. Todo esto supuso un gran impacto en la industria, al ofrecer a los desarrolladores una plataforma de alto nivel sobre la que trabajar.

En 1992, Silicon Graphics lideró la creación del OpenGL Architecture Review Board (OpenGL ARB) [6], un grupo de empresas del sector que sería la encargada de mantener y extender la especificación en los años siguientes. El OpenGL ARB estaba formado por 3Dlabs, Apple, ATI, Dell, IBM, Intel, Nvidia, SGI and Sun Microsystems.

En el año 2000 surge el grupo Khronos [7], una organización sin ánimo de lucro centrada en la creación de estándares abiertos para permitir la creación, reproducción multimedia y computación paralela en un amplio abanico de plataformas y dispositivos.

En otoño de 2006, el OpenGL ARB y los directores de Khronos Group votaron para transferir el control de OpenGL a Khronos Group. El secretario de la ARB Jon Leech observó: *"Hemos decidido mover OpenGL a Khronos para asegurar la salud futura de OpenGL en todas sus formas."* [6]

Khronos Group anunció en 2015 el lanzamiento de una nueva API llamada Vulkan [8], llamada a ser la sucesora de OpenGL. De hecho, en un principio esta API fue bautizada por Khronos como "La iniciativa OpenGL de nueva generación". Se trata de una API similar a OpenGL pero de más bajo nivel, basada en Mantle, cedida a Khronos por AMD [9].

La principal característica de Vulkan frente a OpenGL es la posibilidad de aprovechar los núcleos presentes en la CPU con el fin de incrementar drásticamente el rendimiento gráfico, además de permitir un mayor control sobre la GPU, disminuyendo la carga de la CPU.

Junto con Vulkan surgió SPIR-V *(Standard Portable Intermediate Representation)* [10], un lenguaje intermedio de computación paralela y gráficos, también perteneciente al grupo Khronos. Su principal finalidad era la de permitir a los desarrolladores la creación y distribución de binarios independientes del dispositivo.

Con su aparición, se pasó de utilizar un compilador de GLSL implementado en cada controlador de OpenGL a la utilización de shaders ya traducidos a un formato binario por SPIR-V en Vulkan, mejorando considerablemente la velocidad de inicialización de la

aplicación y el número de shaders a utilizar por escena.

En la actualidad conviven ambas APIs, aconsejándose en algunos casos más simples la utilización de OpenGL en lugar de Vulkan debido a su mayor facilidad de uso y menor carga de mantenimiento.

2.3. Diseño de OpenGL

OpenGL implementa el llamado pipeline de gráficos o pipeline de renderizado. Este pipeline consiste en una secuencia de etapas de procesamiento para convertir datos de una aplicación en una imagen final renderizada.

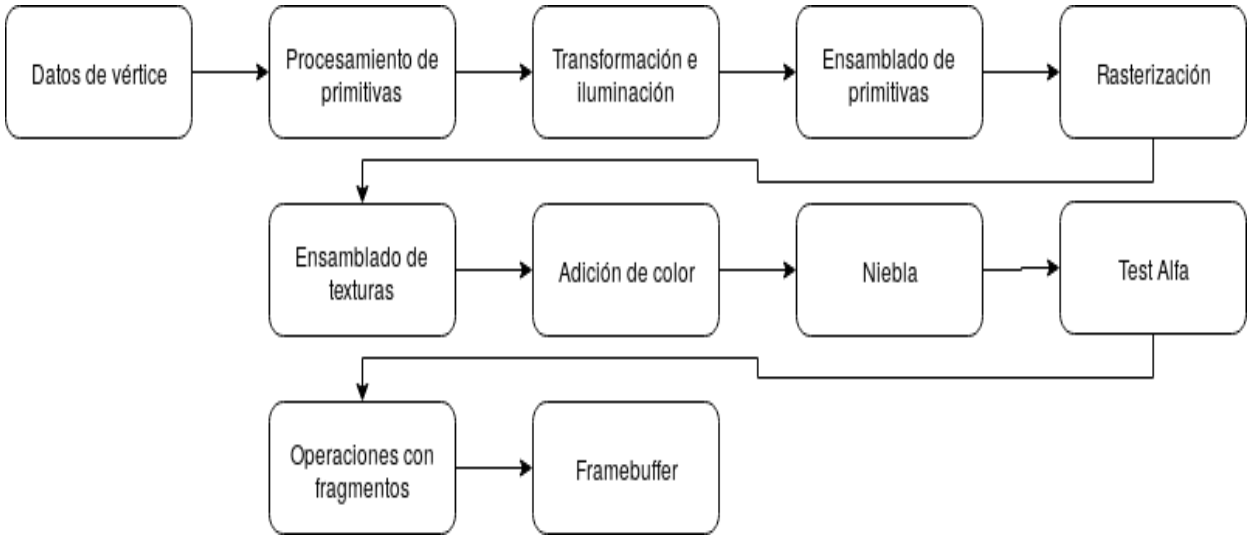


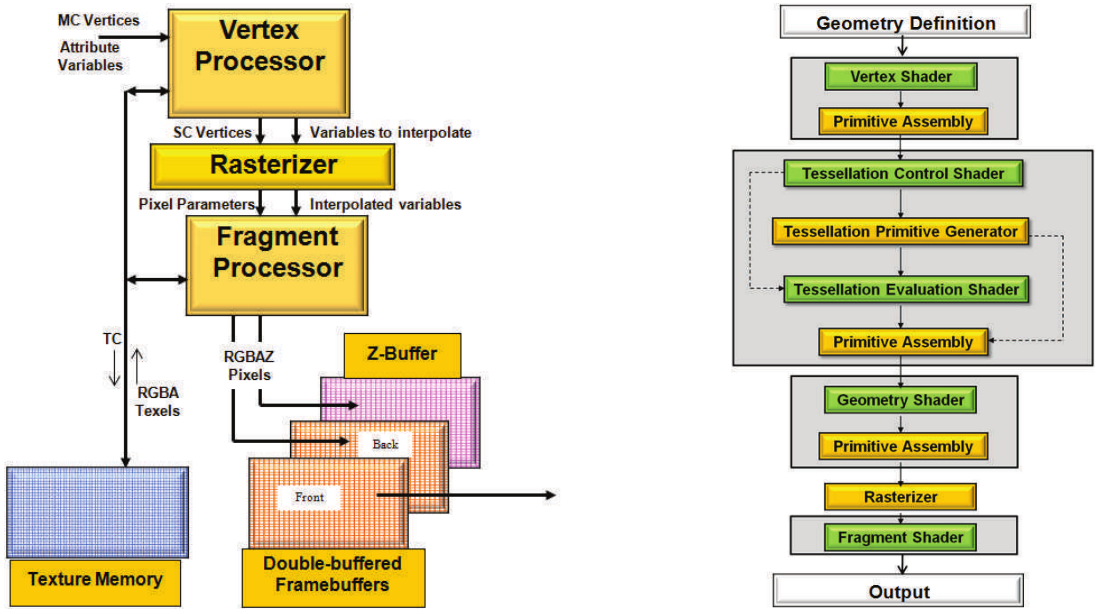
Figura 2.1: Pipeline no programable. OpenGL 1.5

En un principio, este pipeline de renderizado consistía en varias etapas fijas y no programables, en las que el programador simplemente elegía una serie de configuraciones para que OpenGL realizase las operaciones propias de cada etapa. Las etapas principales de este pipeline eran las siguientes (Ver Figura 2.1):

- Procesamiento de primitivas
- Transformación e iluminación
- Ensamblado de primitivas
- Rasterización

- Ensamblado de texturas
- Adición de color
- Niebla
- Test Alfa
- Operaciones con fragmentos

Sin embargo, con la versión 2.0 de OpenGL se introdujeron los *shaders*, explorados en detalle en el Capítulo 3, que sustituyeron algunas de estas etapas fijas por etapas programables, dando mayor flexibilidad al programador. La Figura 2.2 muestra el pipeline asociado a la versión 4.3 de OpenGL.



(a) Pipeline de OpenGL (b) Pipeline detallado

Figura 2.2: Pipeline de gráficos de OpenGL. Fuente: [11]

El renderizado de una imagen comienza con una llamada, desde la aplicación principal, a una función de dibujado de OpenGL. Con esta llamada, OpenGL toma datos sobre vértices contenidos en diferentes objetos y renderiza con estos datos una o más *primitivas* [12].

Las primitivas geométricas son las interpretaciones que OpenGL puede hacer sobre lo que representa un flujo de vértices. Por ejemplo, tres vértices pueden significar tres puntos independientes, dos líneas, un triángulo, etc. OpenGL contiene los siguientes tipos principales de primitivas:

2.3.1. Primitivas de OpenGL

Primitivas de Puntos

OpenGL contiene una única primitiva de puntos: `GL_POINTS`. Cuando se llama a una función de dibujado con esta primitiva, OpenGL interpreta el flujo de vértices como puntos independientes.

Primitivas de Línea

En cuanto a las primitivas de línea, hay tres tipos, basándose en distintas interpretaciones del flujo de vértices.

- `GL_LINES`: Los dos primeros vértices se consideran una línea, los dos siguientes otra línea, etc.
- `GL_LINE_STRIP`: Los vértices adyacentes se consideran una línea. Es decir el primero y el segundo forman una línea, el segundo y el tercero forman otra línea, etc.
- `GL_LINE_LOOP`: Como `GL_LINE_STRIP`, pero el primer y último vértices se consideran también una línea.

Primitivas de Triángulo

Un triángulo es una primitiva formada por tres vértices. También existen tres primitivas de triángulo:

- `GL_TRIANGLES`: los tres primeros vértices forman un triángulo, los tres siguientes otro, etc.
- `GL_TRIANGLE_STRIP`: cada grupo de tres vértices adyacentes forman un triángulo. El (0, 1, 2), el (1, 2, 3), etc.
- `GL_TRIANGLE_FAN`: El primer vértice queda fijo, y cada grupo de dos vértices adyacentes a partir de él forma un triángulo con el primero. Por ejemplo el (0, 1, 2), (0, 2, 3), etc.

En la Figura 2.3 se pueden ver las diferentes interpretaciones en forma de primitiva que

puede realizar OpenGL para renderizar un flujo de vértices.

Patches

La primitiva patches se utiliza principalmente para la etapa de teselación, explicada en la sección **??**. En este caso solo existe una primitiva, `GL_PATCHES`. Esta primitiva contiene el número de vértices definido por el usuario (n), siempre en el intervalo [1,`GL_MAX_PATCH_VERTICES`]. El valor de `GL_MAX_PATCH_VERTICES` depende de la implementación. Utilizando esta primitiva, cada grupo de n vértices del flujo de vértices conforma un patch diferente.

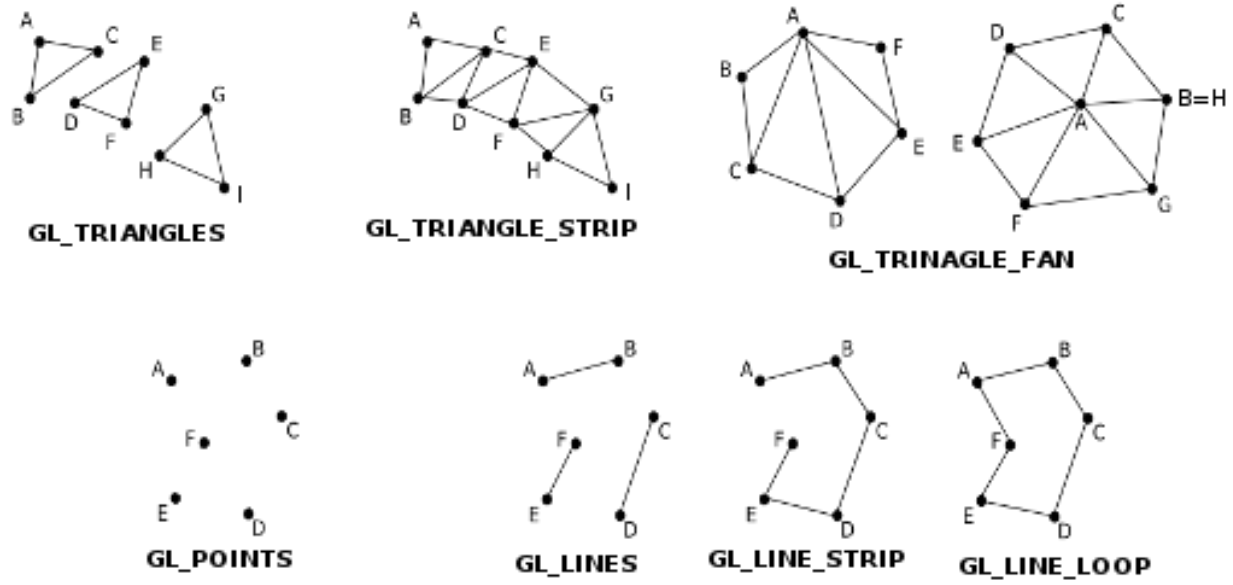


Figura 2.3: Primitivas de OpenGL. Fuente: [13]

Una vez se ha hecho la llamada a la función de dibujado, comienza el proceso de renderizado, recorriéndose las diferentes etapas que forman el pipeline de renderizado.

2.3.2. Procesamiento de Vértices

La primera etapa del pipeline programable —Vertex Processor, Figura 2.2a—, está formada a su vez por los shaders programables de vértice, teselación y geométrico *(vertex, tessellation y geometry shaders)*, que sustituyen a las etapas de transformación e iluminación del pipeline no programable, así como la etapa fija realizada por OpenGL conocida como

Ensamblado de Primitivas. Obtiene como entrada el flujo de vértices, normales, definiciones de primitivas geométricas, colores, parámetros de iluminación, materiales y coordenadas para las texturas. Esta etapa opera sobre estos datos y los organiza en las primitivas geométricas asociadas en preparación para la etapa de post-procesado de vértices, en la que se realizan, entre otras cosas, su recorte y rasterización. Así, la salida es un conjunto de vértices como píxeles, con su color, profundidad y coordenadas de textura.

2.3.3. Clipping

Posteriormente se realiza el recorte o *clipping*, que ocurre cuando, ocasionalmente, algunos vértices quedan fuera de lo que se denomina *viewport*—la región de la ventana donde se permite dibujar— modificando las primitivas geométricas para que nada quede fuera de ese espacio. (Ver Figura 2.4).

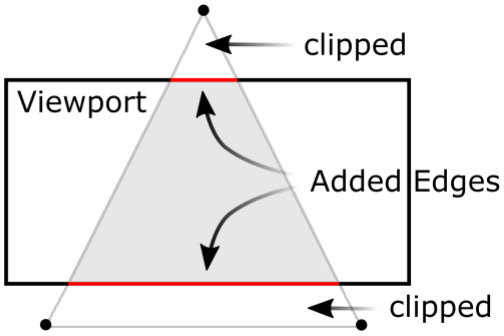


Figura 2.4: Clipping en OpenGL.
Fuente: [14]

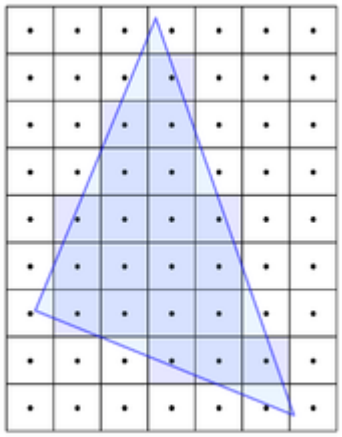


Figura 2.5: Rasterización en OpenGL.
Fuente: [15]

2.3.4. Rasterización

La siguiente etapa es la rasterización (ver Figura 2.5). Este paso implementa la transición de vértice a fragmento. Inmediatamente después del clipping, las primitivas geométricas actualizadas se mandan al rasterizador para la generación de fragmentos. Podemos considerar un fragmento como un "candidato a pixel", en el sentido de que un pixel reside en el *framebuffer*— un espacio de memoria manejado por el hardware gráfico que manda información al dispositivo de salida digital— mientras que un fragmento puede ser rechazado y

nunca actualizar su localización de pixel asociada. El framebuffer consta, a su vez, de tres buffers llamados *color buffer, depth buffer, stencil buffer*, que nos permiten realizar tests para descartar fragmentos.

2.3.5. Procesamiento de Fragmentos

En la última etapa, el procesado de fragmentos se realiza a su vez en dos pasos, uno de ellos programable con el *fragment shader* y otro que consta de distintas operaciones sobre fragmentos realizadas automáticamente por OpenGL. Durante este último procesamiento, la visibilidad de cada fragmento es determinada utilizando diferentes tests (depth, color, stencil...), para lo que se utilizan los buffers indicados en la sección anterior.

Cuando un fragmento pasa por todas estas etapas y todos los test activos, puede ser escrito directamente en el framebuffer, actualizando su color y valor de profundidad de su pixel o, si el *blending* está activado, mezclando su color con el del pixel actual para generar un nuevo color que se escribe en dicho framebuffer.

2.4. Diferencias con Direct3D

Direct3D es parte del conjunto de la API multimedia DirectX, propiedad de Microsoft [16]. Es la principal competidora de OpenGL, ofreciendo ambas un conjunto similar de funcionalidades. Aun así, se pueden observar varias diferencias en términos de disponibilidad, portabilidad, facilidad de uso, rendimiento, estructura y usuarios finales. En la tabla 2.1, se muestran algunas de ellas resumidas.

	OpenGL	Direct3D
Soporte de escritorio	Multiplataforma	Microsoft Windows Xbox
Soporte sistemas empotrados	Multiplataforma (OpenGL ES)	Windows Embedded Windows CE
Licencia	Libre (con características patentadas)	Privativa
Usuario final	Profesionales	Juegos
Lenguaje de shaders	GLSL	HLSL

Cuadro 2.1: Diferencias entre OpenGL y Direct3D

En cuanto a la portabilidad, DirectX solo está disponible para la familia de sistemas operativos Microsoft Windows, mientras que OpenGL tiene implementaciones en muchas plataformas, incluyendo Microsoft Windows y sistemas Unix.

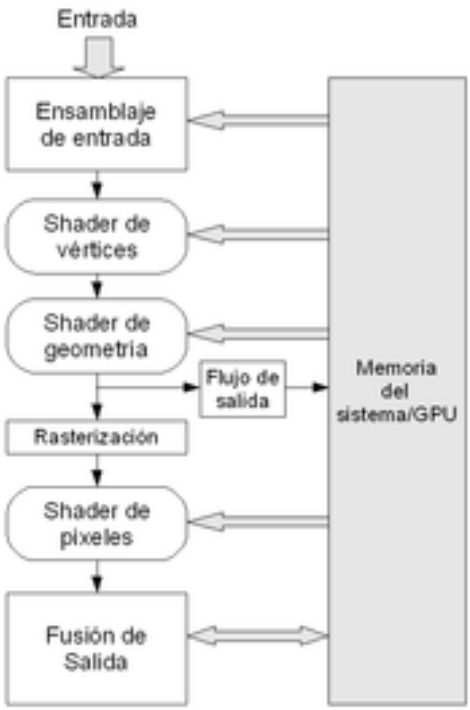


Figura 2.6: Pipeline de gráficos de Direct3D. Fuente: [17]

En términos de facilidad de uso hoy en día ambas APIs se encuentran bastante parejas, habiendo evolucionado desde las primeras versiones, en las que los desarrolladores instaban a Microsoft a unirse a la iniciativa OpenGL, debido a que suponía menos esfuerzo trabajar con esta última.

El diseño de ambas es también similar. Tras muchos años de evolución, el pipeline gráfico es bastante parecido, como puede apreciarse en la Figura 2.6.

Además, las etapas programables del pipeline de OpenGL (vertex, tessellation, geometry y fragment shader) se escriben en GLSL *(OpenGL Shading Language)* [18], un lenguaje creado específicamente para este propósito por el OpenGL ARB. En contraposición, las etapas programables del pipeline implementado por Direct3D (vertex, geometry y pixel shaders) se escriben en el lenguaje HLSL *(High Level Shader Language)* [19] desarrollado por Microsoft y análogo a GLSL.

En conclusión, hoy en día ambas se consideran bastante similares en cuanto a rendimiento y capacidades, siendo el soporte a otros sistemas y el carácter de las licencias las únicas grandes diferencias que hacen decantarse a los desarrolladores por una u otra.

Capítulo 3

Shaders Gráficos

Como se ha visto en la sección 2.3, el pipeline de gráficos de OpenGL tiene cuatro etapas programables:

- Shader de Vértices (Vertex Shader)
- Shaders de Teselación

  Shader de Control de Teselación (Tessellation Control Shader)

  Shader de Evaluación de Teselación (Tessellation Evaluation Shader)
- Shader Geométrico (Geometry Shader)
- Shader de Fragmento (Fragment Shader)

Los shaders gráficos son un tipo de programa utilizado inicialmente para producir niveles apropiados de luz, oscuridad y color en una imagen. Sin embargo, hoy en día se utilizan con diversas finalidades diferentes como efectos especiales, post procesado de vídeos, videojuegos, etc.

Los shaders se introdujeron en OpenGL en la versión 2.0, incluyendo el lenguaje de programación centrado en shaders OpenGL Shading Language, también conocido como GLSL [18]— un lenguaje tipo C creado específicamente para que los desarrolladores tuviesen más control sobre el pipeline de renderizado.—

Durante el proceso de desarrollo de shaders no es necesario incluir todas las etapas que se muestran en la Figura 2.2b, aunque normalmente, si se decide utilizar alguno de ellos, se

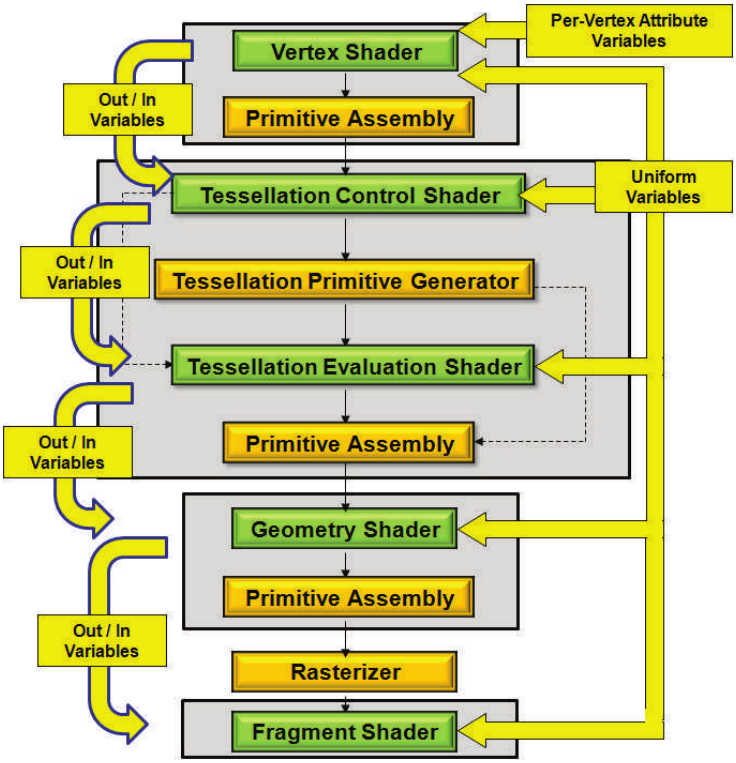


Figura 3.1: Comunicación entre shaders del pipeline. Fuente: [11]

requiere utilizar, al menos, un Vertex Shader y un Fragment Shader.

Los datos se proporcionan a los shaders de distintas maneras desde la aplicación principal. La primera es proporcionando los datos necesarios para el renderizado de vértices a la primera etapa, el Vertex Shader. A partir de ese momento los shaders del pipeline se comunican entre ellos mediante variables proporcionadas por GLSL o definidas por el programador en los shaders, siendo la salida de un shader la entrada del siguiente, como se muestra en la Figura 3.1. Un breve resumen acerca del lenguaje GLSL se incluye en el Apéndice A.

El segundo modo de especificar datos a los shaders es mediante variables `uniform`. Estas variables se definen en los shaders. Se trata de variables de solo lectura por parte de los shaders. Al contrario que las variables definidas como entrada y salida de un shader a otro, estas variables toman sus valores del programa principal cada vez que se invoca el shader.

Por último, es posible comunicarse con los shaders desde la aplicación principal mediante el uso de texturas. Con ellas, el shader puede acceder a la información contenida en la textura con el fin de renderizar imágenes o acceder a cualquier tipo de datos. Asimismo, desde la aplicación se pueden leer datos de texturas, permitiendo leer información generada por los shaders y codificada en forma de textura.

3.1. Vertex Shader

El Vertex Shader es la etapa del pipeline de renderizado que se encarga del procesamiento de los vértices individuales [20]. El Vertex Shader tiene como entrada unos atributos de vértice especificados desde un *Vertex Array Object (VAO)* por un comando de renderizado. Recibe un único vértice, formado por sus atributos, del flujo de vértices y genera un único vértice al flujo de salida. Por cada vértice de entrada ha de haber, necesariamente, uno de salida.

Este shader se invoca una vez por cada vértice en el flujo de entrada, exceptuando el caso en el que OpenGL detecte que una invocación a este shader con las exactamente las mismas entradas ya ha sido realizada, en cuyo caso se reutilizan los resultados de la invocación previa, resultando en un ahorro de tiempo valioso.

Normalmente, las operaciones que se realizan en el Vertex Shader son transformaciones para el espacio de post-proyección, iluminación por vértice o preparación para las siguientes etapas del pipeline.

3.2. Tessellation Shaders

La Teselación es la etapa opcional del pipeline de renderizado que consiste en subdividir patches de datos de vértices en primitivas más pequeñas y computar los valores de los nuevos vértices creados en el proceso. Está compuesta, a su vez, por otras tres etapas, dos de ellas programables en forma de shader, y una intermedia fija. Cada una de estas etapas se encarga de una parte del proceso de teselación.

En esta sección se explican los dos shaders involucrados, además de la etapa intermedia, llamada generador de primitivas de teselación, pues resulta importante para entender el proceso y las entradas y salidas a los shaders.

3.2.1. Tessellation Control Shader

El Tessellation Control Shader (TCS) [21] es la primera etapa del proceso de teselación, en el caso de ser utilizado. Se sitúa inmediatamente posterior al Vertex Shader e inmediatamente anterior al generador de primitivas de teselación. Controla cuánta teselación provocar en un patch determinado, así como el tamaño del patch de salida, permitiendo aumentar la cantidad de datos. Su función principal es la de comunicar al generador de primitivas de te-

selación el nivel de teselación deseado, así como proveerle los datos del patch al Tessellation Evaluation Shader mediante sus variables de salida. Es también el encargado de asegurar la continuidad entre patches. Así, en el caso de que dos patches adyacentes tengan diferentes niveles de teselación, las invocaciones al TCS para cada uno de estos patches han de utilizar el control de teselación para asegurarse de que los bordes compartidos utilizan el mismo nivel de teselación.

Como entrada, el TCS obtiene la salida del Vertex Shader organizada en un vector de tantos vértices como tenga el patch de entrada, así como determinadas variables de entrada integradas (Ver A). Las salidas de este shader pueden ser salidas de vértice, en un array del mismo tamaño que el número de vértices en el patch de salida; o salidas de patch, declaradas con la palabra reservada `patch`; además de las salidas integradas.

Por cada patch de entrada y cada vértice en el patch de salida se realiza una invocación al TCS, y cada invocación al TCS produce un único vértice como salida al patch de salida. Por ejemplo, si queremos dibujar 20 patches y cada patch de salida tiene 4 vértices, se realizarán 80 diferentes invocaciones al TCS.

En el caso de no utilizar un TCS, se pueden pasar valores por defecto a las siguientes etapas de teselación.

3.2.2. Tessellation Primitive Generator

El generador de primitivas de teselación (Tessellation Primitive Generator) [23] es la etapa que se encuentra entre los dos shaders de teselación, el TCS y el Tessellation Evaluation Shader. Esta etapa, fija en el pipeline, es la encargada de crear nuevas primitivas a partir del patch de entrada. La función principal de este sistema es la de determinar cuántos vértices crear, en qué orden hacerlo y qué clase de primitivas construir con ellos. Los datos reales de estos vértices, como color, posición, etc., han de ser generados por el TES. Debido a esto, el generador no tiene en cuenta el patch de salida producido por el TCS, sino que solo opera en términos de lo que se llama un abstract patch. En lugar de mirar la salida de patch del TCS, solo piensa en teselar un cuadrado o triángulo abstracto, o un bloque de isolíneas.

Esta etapa está supeditada al Tessellation Evaluation Shader, puesto que solo se ejecutará en el caso de que exista uno activo. La generación de primitivas en esta etapa se ve afectada por los siguientes factores:

- Niveles de teselación marcados por el TCS (O por defecto si no hay TCS)
- Espaciado de los vértices teselados, definido en el TES

- Tipo de primitiva, definido en el TES
- Orden de generación de primitivas, definido en el TES

La cantidad de teselación a realizar se define en niveles de teselación internos y externos (Ver Figura 3.2). Funcionan de la siguiente manera: un nivel de teselación 4 indica que un borde se convertirá en 4 bordes (2 vértices se convertirán en 5). El nivel externo define el grado de teselación para los bordes externos de la primitiva. Esto permite que dos patches distintos se conecten apropiadamente, a pesar de tener distintos niveles de teselación dentro del patch. El nivel interno hace referencia el número de teselaciones a realizar dentro del patch abstracto.

Cabe destacar que no todos los patches abstractos utilizan los mismos niveles de teselación. Por ejemplo, los triángulos utilizan un único nivel interno y tres niveles externos. El resto de posibles niveles son ignorados.

El espaciado entre vértices puede realizarse de las siguientes maneras: espaciado equidistante, espaciado fraccional par o espaciado fraccional impar.

El espaciado equidistante (ver Figura 3.3) divide el borde a teselar en segmentos de igual longitud. Solo acepta valores enteros, por lo que redondea el nivel de teselación hasta el siguiente entero. Este hecho causa que los segmentos aparezcan instantáneamente de un nivel a otro.

Para conseguir un comportamiento mas "suave" se tienen los otros dos modos de espaciado. Estos últimos son útiles especialmente cuando el nivel de teselación es dependiente del área vista desde la cámara. En el espaciado fraccional par el número de segmentos en los que dividir el borde (nivel de teselación efectivo) se redondea al siguiente entero par, mientras que en el espaciado fraccional impar se redondea al siguiente entero impar. Para estos modos de espaciado se necesita definir dos valores:

- n, el nivel de teselación efectivo, redondeado según lo anterior.
- f, el valor computado antes del redondeo. Un valor potencialmente fraccionario.

Según este esquema, los bordes a teselar se subdividen en dos conjuntos de segmentos. El primero con $n-2$ segmentos de igual longitud; el otro con 2 segmentos de igual longitud entre ellos, pero no necesariamente de igual longitud que los del otro conjunto. Estos dos segmentos tendrán menor longitud que los otros en general. La longitud de estos es exactamente $n - f$. Por tanto, cuando se cumple que $n - f = 0$ se tiene que todos los segmentos tienen igual longitud. Estos comportamientos se pueden observar en las Figuras 3.4, 3.5.

3.2.3. Tessellation Evaluation Shader

El Tessellation Evaluation Shader (TES) [24] es la etapa opcional que se encuentra entre el generador de primitivas de teselación y el geometry shader. Su función es la de coger los resultados obtenidos en la etapa anterior y, a partir de estos, computar las posiciones interpoladas de cada vértice y otros datos.

Como entrada, el TES obtiene del generador de primitivas de teselación un patch abstracto, así como datos de los vértices y del patch, provenientes del TCS. Los datos de vértices se obtienen del TCS como un array indexado por el índice del vértice y tiene tantos elementos como vértices hay en el patch de entrada. Los datos del patch se pueden obtener del TCS, de nuevo, mediante la palabra reservada `patch`. Además posee variables de entrada integradas en GLSL. (Ver A).

Como salida se produce, por cada invocación a este shader, un vértice particular. Junto con este vértice se tienen tanto variables de salida definidas por el usuario como variables integradas en GLSL.

Esta es la etapa donde el programador implementa el algoritmo que se usa para computar las nuevas posiciones, normales, coordenadas de texturas, etc. Como se ha expuesto antes, este shader es el que determina si ocurrirá o no la etapa de generación de primitivas de teselación, puesto que solo se ejecutará si existe un TES activo.

El TES, en caso de ser utilizado, debe especificar el tipo de primitiva que servirá como entrada al geometry shader. Este tipo puede ser puntos, isolíneas, triángulos o cuadriláteros.

3.3. Geometry Shader

El Geometry Shader [25] es un programa escrito en GLSL que corresponde a la etapa del pipeline programable que se encuentra entre el TES o el Vertex Shader (dependiendo de si existe o no teselación) y la etapa fija de post-procesado de vértices. Este shader es opcional y no es necesaria su utilización.

Las invocaciones a este shader toman como entrada una única primitiva geométrica y puede dar como salida cero o más primitivas, aunque existe un límite de primitivas que se pueden generar en cada invocación, dependiendo de la implementación. Los shaders geométricos están diseñados para aceptar como entrada una primitiva específica y dar como salida otra.

Sus usos varían bastante, pudiéndose utilizar como una manera de amplificar la geometría, sirviendo como una especie de teselación, así como para realizar un renderizado por capas o incluso para la realización de tareas de cómputo en la GPU.

Entre las primitivas de entrada aceptadas por el geometry shader se encuentran las siguientes:

Entrada	Primitiva	Parámetro TES	Vértices
`points`	`GL_POINTS`	`point_mode`	1
`lines`	`GL_LINES, GL_LINE_STRIP, GL_LINE_LIST`	`isolines`	2
`lines_adjacency`	`GL_LINES_ADJACENCY, GL_LINE_STRIP_ADJACENCY`	`N/A`	4
`triangles`	`GL_TRIANGLES, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN`	`triangles, quads`	3
`triangles_adjacency`	`GL_TRIANGLES_ADJACENCY, GL_TRIANGLE_STRIP_ADJACENCY`	`N/A`	6

Cuadro 3.1: Primitivas de entrada al Geometry Shader

Las primitivas de salida pueden ser únicamente alguna de las siguientes:

- `points`
- `line_strip`
- `triangle_strip`

Los shaders geométricos pueden generar tantos vértices como permita el límite de implementación. Para ello, el programador genera los valores que necesite para el nuevo vértice y, una vez estos valores sean correctos, una llamada a la función `EmitVertex()` produce el vértice deseado. Una vez llamada esta función, los valores escritos para el vértice son reseteados, teniendo que volver a escribirlos para generar otro vértice.

De igual modo, para generar una primitiva, debemos especificar del modo anterior todos los vértices que forman esa primitiva y posteriormente llamar a la función `EndPrimitive()`. De esta forma, si se desea generar más de una primitiva, se deben especificar los vértices que forman la primera, llamar a `EndPrimitive()`, generar los vértices que forman la segunda y llamar de nuevo a `EndPrimitive()` para generar la segunda primitiva.

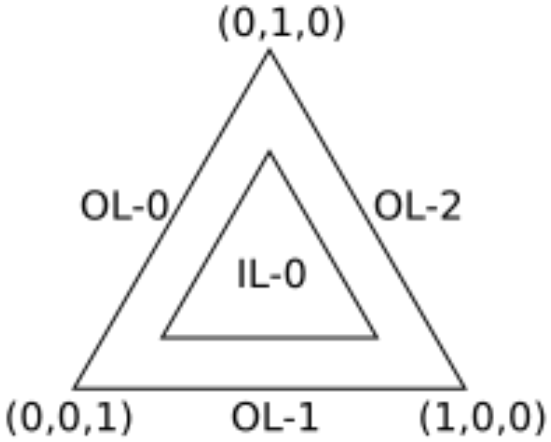
3.4. Fragment Shader

El Fragment Shader [26] es la etapa posterior a la rasterización. Por cada uno de los píxeles cubiertos por una primitiva, se genera un fragmento. Cada uno de estos fragmentos tiene una posición en la espacio de ventana, así como otros valores procedentes de la etapa de procesamiento de vértices.

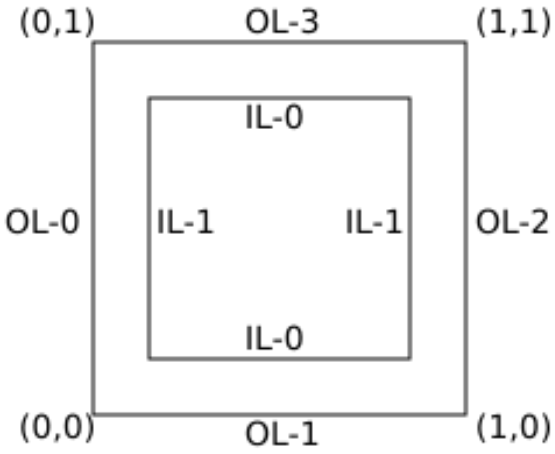
La salida del fragment shader consta de un depth value, un posible stencil value (que no es modificado por el shader) y cero o más color values para ser potencialmente escritos en los buffers del framebuffer actual. Estos shaders toman como entrada un único fragmento, producido por el rasterizador, y dan como salida otro único fragmento.

Técnicamente, la utilización de estos shaders es también opcional, puesto que de no utilizarlo, los valores de color del fragmento de salida quedarán indefinidos, pero los valores de depth y stencil en la salida serán los mismos que los de entrada. Esto puede ser interesante en el caso de solo estar interesados en los valores de profundidad computados por el sistema en lugar de otro valor calculado por el programador.

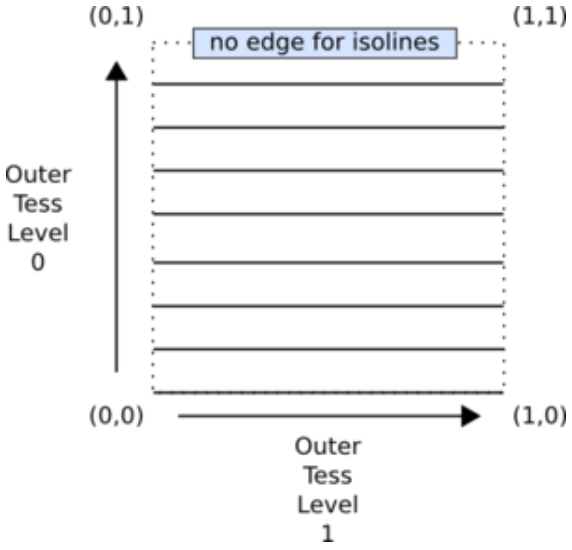
Este shader también tiene operaciones especiales no presentes en los otros tipos de shader, como puede ser la instrucción `discard`, cuyo objetivo es descartar los valores de salida generados durante la ejecución del shader para un fragmento en concreto, haciendo que este fragmento no pase a las siguientes etapas del pipeline. Esto puede ser útil para descartar fragmentos cuyos valores generados en la ejecución se queden fuera de unos límites impuestos por el programador.



(a) Niveles de triángulos



(b) Niveles de cuadriláteros



(c) Niveles de isolíneas

Figura 3.2: Teselación - Funcionamiento de los niveles. Fuente: [22]

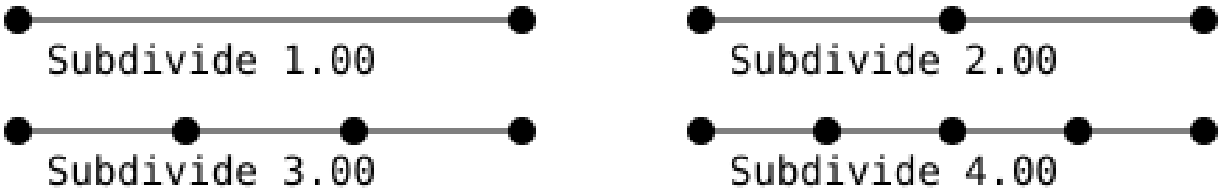


Figura 3.3: Teselación - Espaciado equidistante. Fuente: [22]

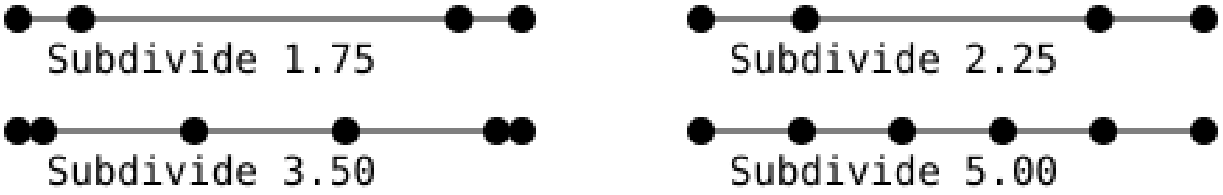


Figura 3.4: Teselación - Espaciado fraccional par. Fuente: [22]

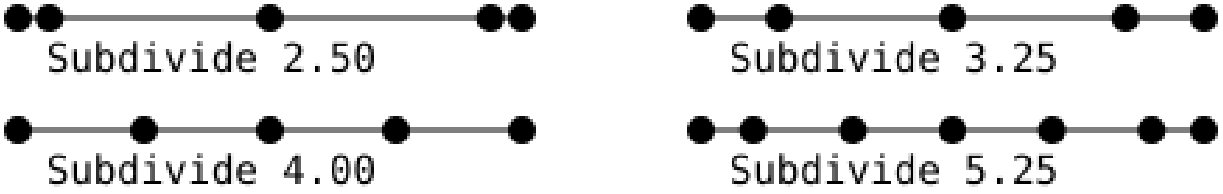


Figura 3.5: Teselación - Espaciado fraccional impar. Fuente: [22]

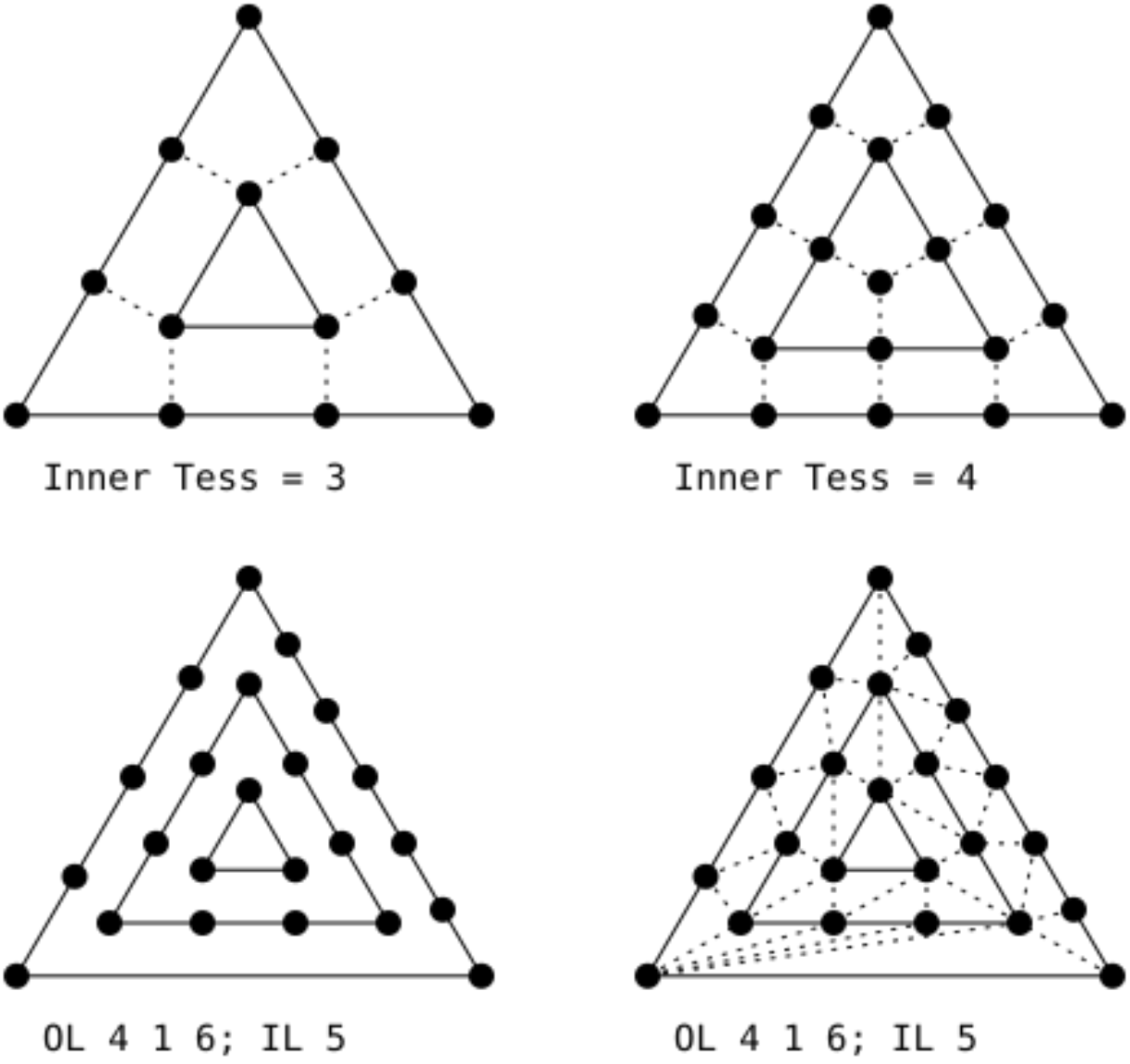


Figura 3.6: Diferentes niveles de teselación en triángulos. Fuente: [22]

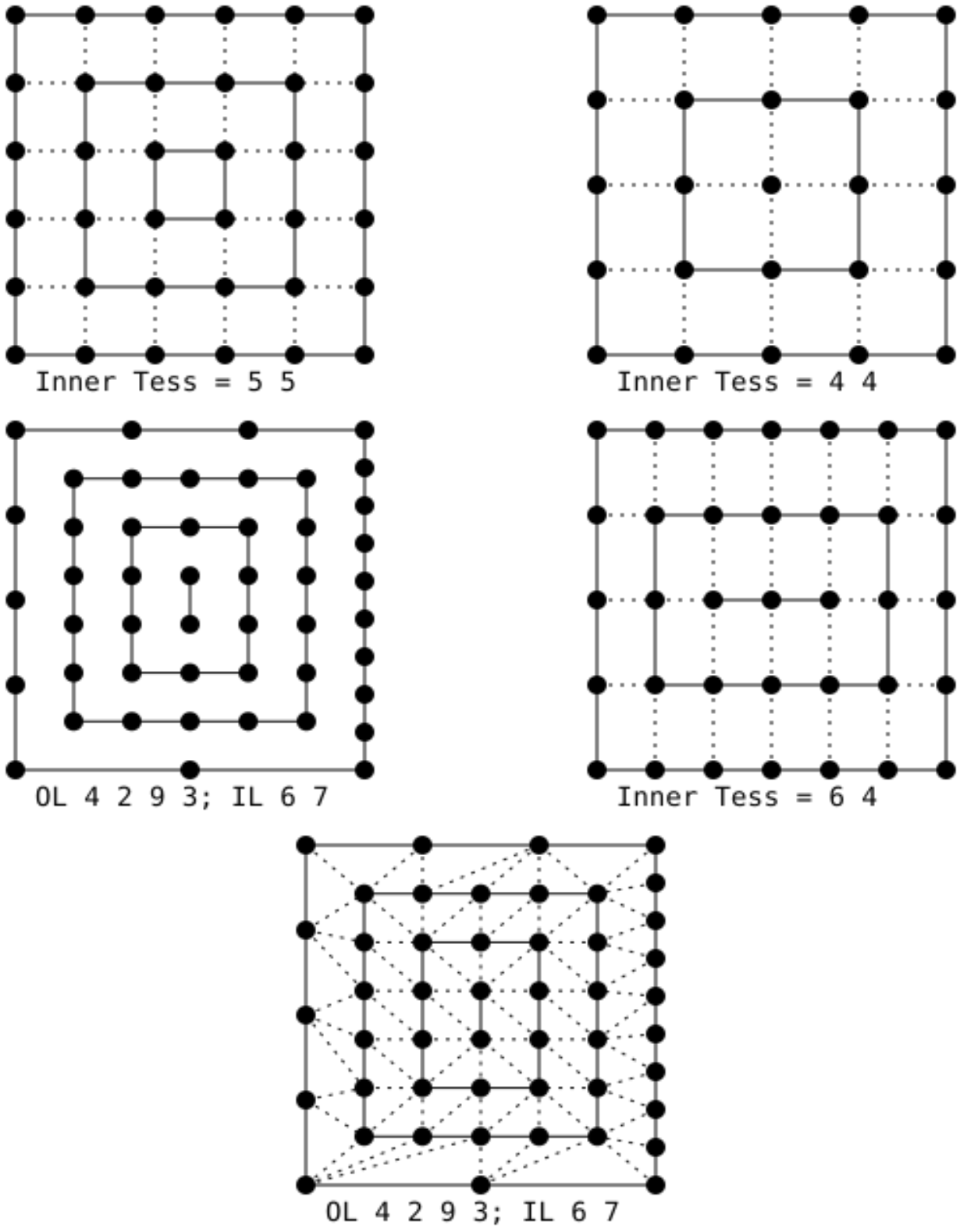


Figura 3.7: Diferentes niveles de teselación en cuadriláteros. Fuente: [22]

Capítulo 4

Visualización Científica

En este capítulo se muestran técnicas relevantes de visualización científica. Algunas de ellas serán implementadas mediante shaders en nuestra aplicación, explicando los shaders utilizados en ella.

4.1. Procesamiento de imágenes

El procesamiento de imágenes es la disciplina que incluye las diferentes técnicas de filtrado, combinación o modificación de imágenes con el fin de resaltar características deseables de la imagen. En visualización científica, por ejemplo, se puede utilizar el negativo de la imagen de una estructura ósea con el fin de detectar defectos que de otro modo resulta más complicado ver; utilizar una mezcla de imágenes astronómicas en diferentes espectros para conseguir una imagen mas realista; o aumentar el brillo y nitidez de una imagen para conseguir una visualización más eficaz.

Entre estas técnicas podemos encontrar, por ejemplo:

- Negativo
- Detección de bordes
- Cambios de brillo, contraste y nitidez
- Mezcla de imágenes

4.2. Curvas y superficies de Bézier

Se denominan curvas de Bézier a un sistema de trazado de dibujos técnicos ideado en los años 60 por Pierre Bézier. El modelo se basa en una descripción matemática que se utiliza extensivamente en programas tipo CAD, y son muy útiles para modelar curvas suaves y fáciles de manipular mediante sus puntos de control. Éstas pueden ser, generalmente, lineales, cuadráticas o cúbicas, aunque se puede generalizar a cualquier grado.

Las curvas de Bézier vienen definidas mediante el llamado Polinomio de Bézier $B(t)$ y el grado de este polinomio es el que determina el grado de la curva. Las curvas de Bézier lineales tienen dos puntos de control $\mathbf{P_0}$ y $\mathbf{P_1}$, por lo que la curva resultante es en realidad una recta entre esos dos puntos. Siguen la fórmula general de una recta, es decir:

$$B(t) = \mathbf{P_0} + (\mathbf{P_1} - \mathbf{P_0})t = (1-t)\mathbf{P_0} + t\mathbf{P_1},\ t \in [0,1] \tag{4.1}$$

Las cuadráticas tienen tres puntos de control $\mathbf{P_0}$, $\mathbf{P_1}$ y $\mathbf{P_2}$ y sigue la trayectoria marcada por la función $B(t)$:

$$B(t) = (1-t)^2\mathbf{P_0} + 2t(1-t)\mathbf{P_1} + t^2\mathbf{P_2},\ t \in [0,1] \tag{4.2}$$

En las curvas cúbicas se utilizan cuatro puntos de control $\mathbf{P_0}$, $\mathbf{P_1}$, $\mathbf{P_2}$ y $\mathbf{P_3}$, siguiendo la trayectoria:

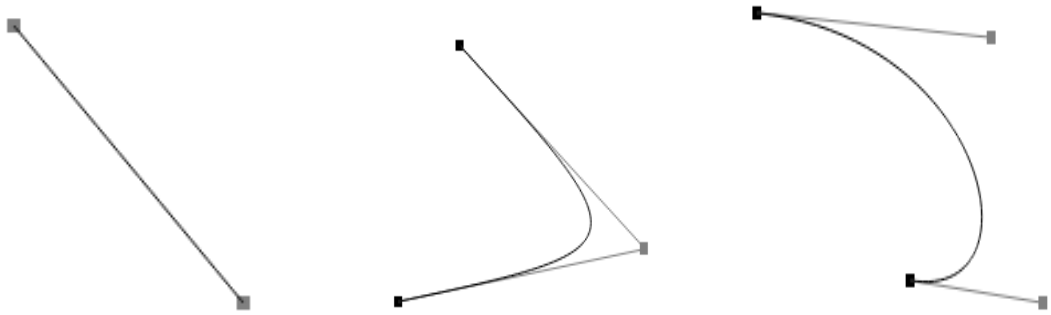
$$B(t) = (1-t)^3\mathbf{P_0} + 3t(1-t)^2\mathbf{P_1} + 3t^2(1-t)\mathbf{P_2} + t^3\mathbf{P_3},\ t \in [0,1] \tag{4.3}$$

Si generalizamos este polinomio se pueden generar curvas de grado n siguiendo la fórmula:

$$B(t) = \sum_{k=0}^{n} \mathbf{P_k} BEZ_{k,n}(t),\ t \in [0,1] \tag{4.4}$$

donde $BEZ_{k,n}(t)$ son conocidos como los polinomios de Bernstein:

$$BEZ(t) = \binom{n}{k} t^k (1-t)^{n-k},\ k = 0, \ldots, n \tag{4.5}$$



(a) Lineal (b) Cuadrática (c) Cúbica

Figura 4.1: Curvas de Bézier.

En cuanto a las superficies de Bézier, se generan mediante dos conjuntos de curvas de Bézier ortogonales, especificadas mediante una malla de puntos de control. La ecuación para estas superficies es:

$$B(t,u) = \sum_{j=0}^{m} \sum_{k=0}^{n} \mathbf{P_{j,k}} BEZ_{j,m}(t) BEZ_{k,n}(u) \tag{4.6}$$

Las curvas y superficies de Bézier tienen propiedades muy interesantes, como la posibilidad de calcular las primeras y segundas derivadas de la curva en los puntos finales, posibilitando su empalme con otras curvas suavemente. Para leer más acerca de las curvas y superficies de Bézier se pueden consultar Bailey and Cunningham [11], Hearn and Baker [27].

4.3. Visualización de datos en 3D

Otra de las grandes áreas de la visualización científica es la de visualizar conjuntos grandes de datos en tres dimensiones. Estos datos pueden ser tanto escalares como vectoriales. Por ejemplo, se pueden monitorizar los datos de temperatura en la atmósfera durante un determinado período de tiempo, queriendo después visualizar las áreas que han estado más calientes de media.

Otro tipo de visualización de datos en tres dimensiones es la visualización de datos vectoriales en tres dimensiones, con el fin de capturar, por ejemplo, flujos vectoriales de fluidos al rededor de un objeto, etc.

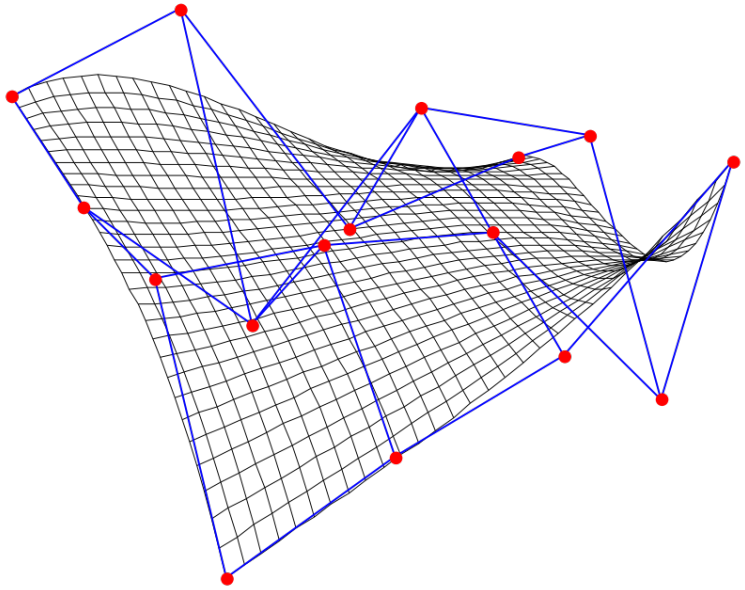


Figura 4.2: Superficie de Bézier con 16 puntos de control. Fuente: [28]

Para este tipo de tareas existen diferentes métodos de visualización, entre los que se encuentran los siguientes:

- Nubes de puntos (Figura 4.3a)
- Visualización de volumen (Figura 4.3b)
- Isosuperficies (Figura 4.3c)
- Planos cortados (Figura 4.3d)
- Trazado de partículas (Figura 4.3e)
- Visualización de vectores en 3D (Figura 4.3f)

4.4. Sólidos de revolución

Los sólidos de revolución han sido siempre objeto de estudio en matemáticas, física e ingeniería. Debido a la facilidad para calcular su área, son utilizadas ampliamente en diseño industrial y modelado científico.

Este tipo de superficies surgen al rotar una curva planar —la generatriz— en torno a un eje. Como se ha dicho antes, se conocen fórmulas exactas para su área y volumen. Por ejemplo, cuando la revolución se produce en torno al eje y, se tiene las ecuación (4.7) para el área, y la ecuación (4.8) para el volumen del sólido encerrado por esta superficie. Sin embargo, si la revolución se realiza en torno al eje x, las ecuaciones de área y volumen son (4.9) y (4.10).

$$A_y = 2\pi \int_a^b x(t)\sqrt{\left(\frac{dx}{dt}\right)^2 + \left(\frac{dy}{dt}\right)^2}dt \tag{4.7}$$

$$V = 2\pi \int_a^b xf(x)dx \tag{4.8}$$

$$A_y = 2\pi \int_a^b y(t)\sqrt{\left(\frac{dx}{dy}\right)^2 + \left(\frac{dy}{dt}\right)^2}dt \tag{4.9}$$

$$V = \pi \int_a^b f(x)^2 dx \tag{4.10}$$

4.5. Coloreado de terrenos

Otro de los ejemplos típicos en visualización científica consiste en modelar un terreno, ya sea en la Tierra o en otros planetas captando datos con sondas, y colorearlo por alturas para hacerse una idea visual de lo que nos están diciendo realmente esos datos. Un ejemplo de este coloreado de mapas puede verse en la figura 4.4.

4.6. Visualización de campos vectoriales

La última de las técnicas de visualización que vamos a ver tiene que ver con la visualización de campos vectoriales en dos dimensiones. Para ello veremos un método importante denominado Line Integral Convolution.

El método Line Integral Convolution fue originalmente propuesto en el artículo de Cabral and Leedom [31]. Se utiliza para visualizar campos vectoriales densos. En este método, se

utiliza una imagen de ruido blanco junto con datos de un flujo vectorial, modificando esta imagen para mostrar el flujo de los datos. (ver Figura 4.5).

En Petkov [32] se muestra como funciona el método de Line Integral Convolution, señalando los principales pasos que sigue:

Para cada pixel en la imagen de entrada, hacer:

1. Computar la línea de flujo *(streamline)* para una longitud determinada por el usuario en direcciones positivas y negativas. (Figura 4.6)

2. Para cada punto en la streamline, computar su peso de convolución h_i. (Figura 4.7)

3. Computar el valor de salida del pixel con los valores de entrada y los pesos computados en 2. (Figura 4.8)

Como imagen de entrada se puede utilizar, realmente, cualquier imagen, haciendo ver las líneas de flujo con los colores de la imagen original. Sin embargo, con el fin de que perturbaciones en la imagen original no condicionen el análisis del flujo a visualizar, la opción recomendada y casi siempre utilizada en este método es la del ruido blanco, debido a la distribución uniforme de los colores de sus píxeles.

Computar la línea de flujo

Para computar la línea de flujo se ha de utilizar algún método numérico. Entre las posibilidades se encuentran el método de Euler, utilizado por Petkov [32], el método de Euler de paso variable, utilizado por los autores originales en Cabral and Leedom [31], o el método de Runge-Gutta de orden 4, propuesto como alternativa en ambos textos. En la sección 5.4.3, se explican estos métodos como preparación para su implementación. De esta manera, se consiguen una serie de puntos a lo largo de la línea de flujo, dependiendo del paso escogido en el método numérico y la longitud de la línea escogida por el usuario.

Computar los pesos de convolución

Este paso consiste en encontrar la integral exacta del núcleo de convolución k en cada punto de la línea de flujo computado en el paso anterior. Es decir, se ha de resolver la integral (4.11).

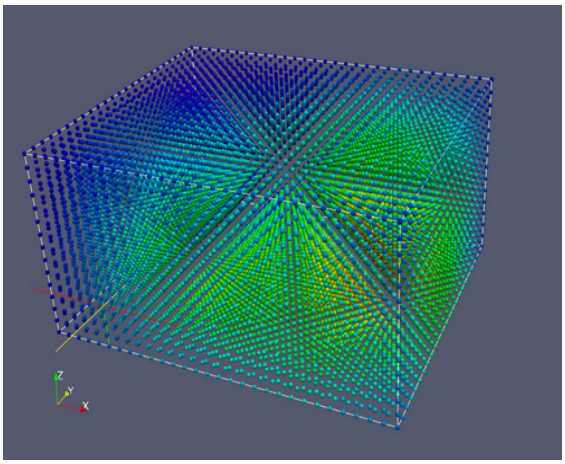
$$h_i = \int_a^b k(\omega) d\omega \tag{4.11}$$

donde i representa el índice del punto actual en la línea de flujo, a es la distancia a lo largo de la línea de flujo entre el punto actual y el pixel para el que queremos calcular el valor de salida y b es igual a a más el tamaño del paso utilizado en el paso actual Δs_i. Más información acerca del núcleo de convolución puede encontrarse en Cabral and Leedom [31].

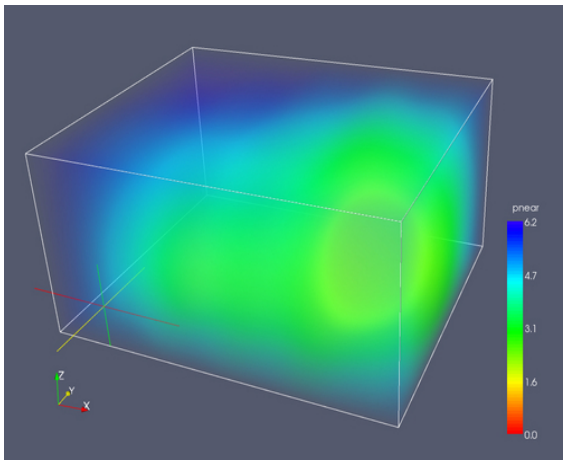
Computar los valores de salida del pixel

Para calcular los valores de salida del pixel una vez calculados los pesos de los puntos de la línea de flujo, se sigue la siguiente ecuación:

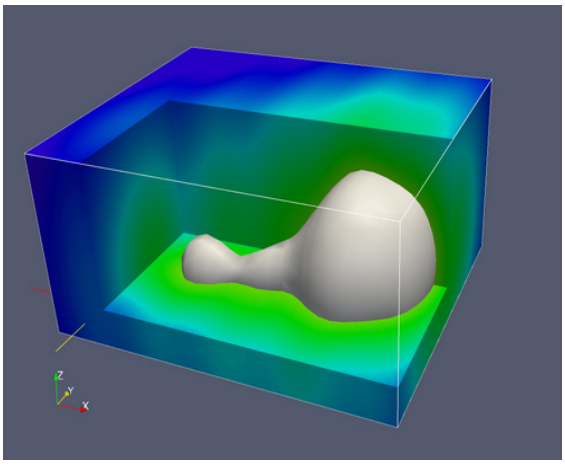
$$F_{out}(x, y) = \frac{\sum_{i=0}^{l} F_{in}(P_i) h_i + \sum_{i=0}^{l'} F_{in}(P'_i) h'_i}{\sum_{i=0}^{l} h_i + \sum_{i=0}^{l'} h'_i} \tag{4.12}$$



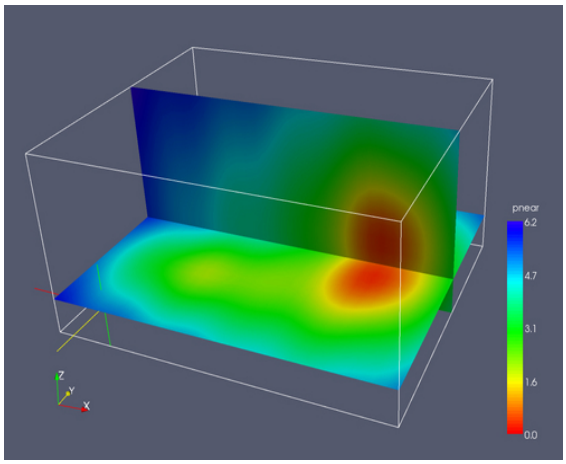
(a) Nube de puntos



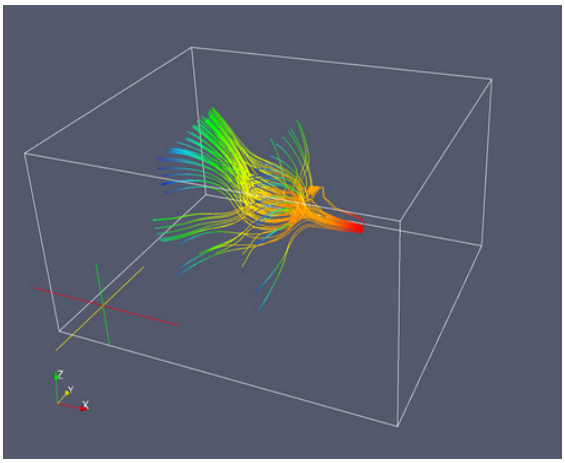
(b) Visualización de volumen



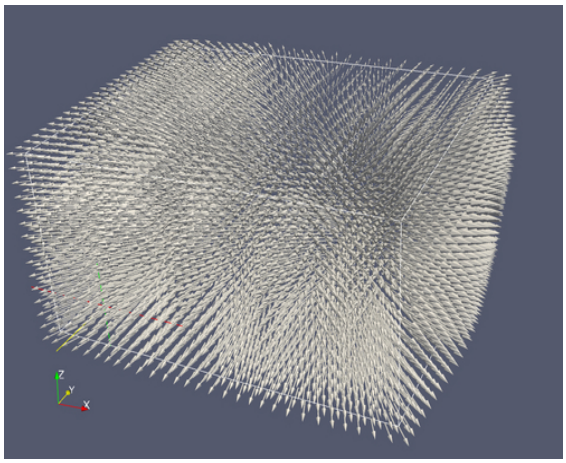
(c) Isosuperficie



(d) Planos cortados



(e) Trazado de partículas



(f) Visualización de vectores

Figura 4.3: Técnicas de visualización en 3D. Fuente: [29]

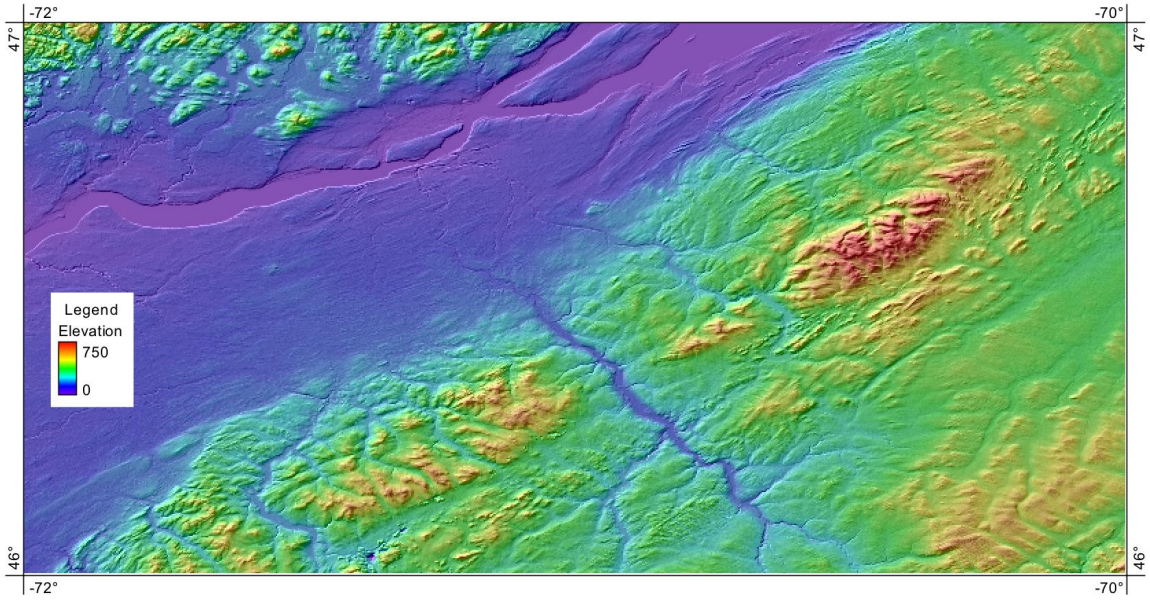
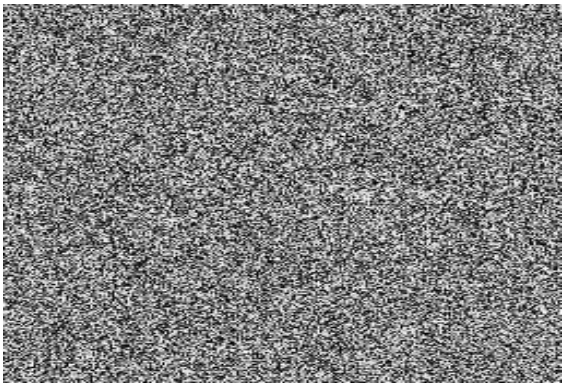
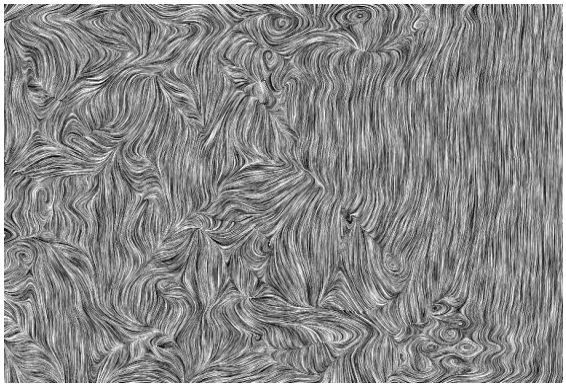


Figura 4.4: Coloreado de terreno por alturas. Fuente: [30]



(a) Imagen utilizada para el método del LIC.



(b) Resultado de aplicar el método del LIC a la imagen de la izquierda con un flujo vectorial. Fuente: [31]

Figura 4.5: Método Line Integral Convolution.

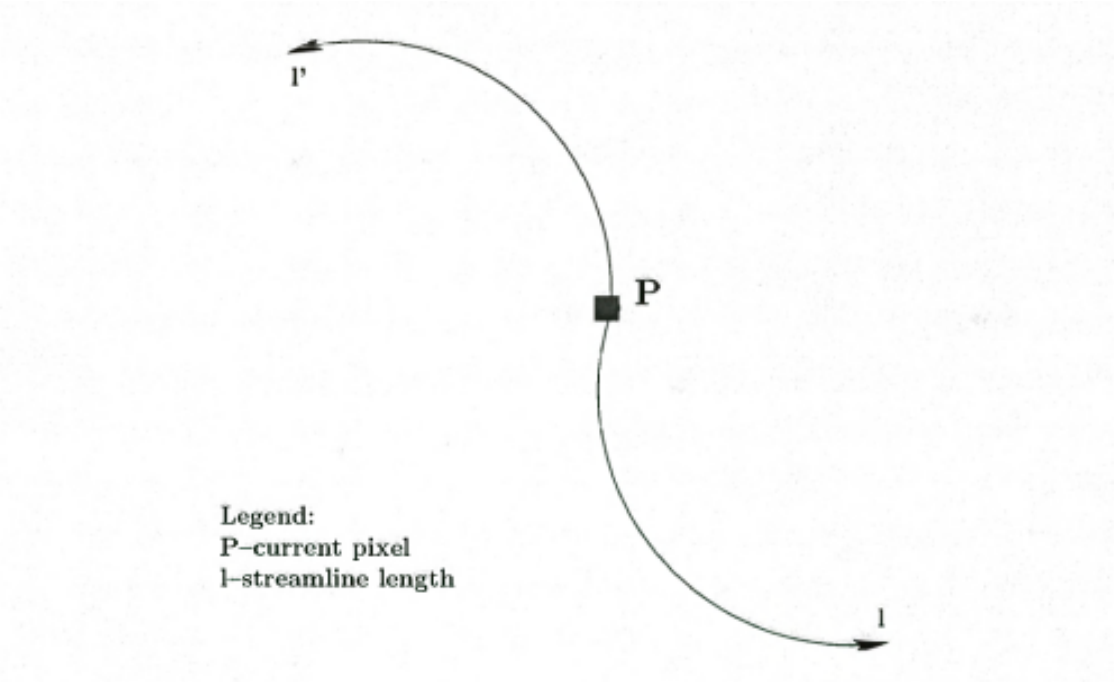


Figura 4.6: Computar la línea de flujo. Fuente: [32]

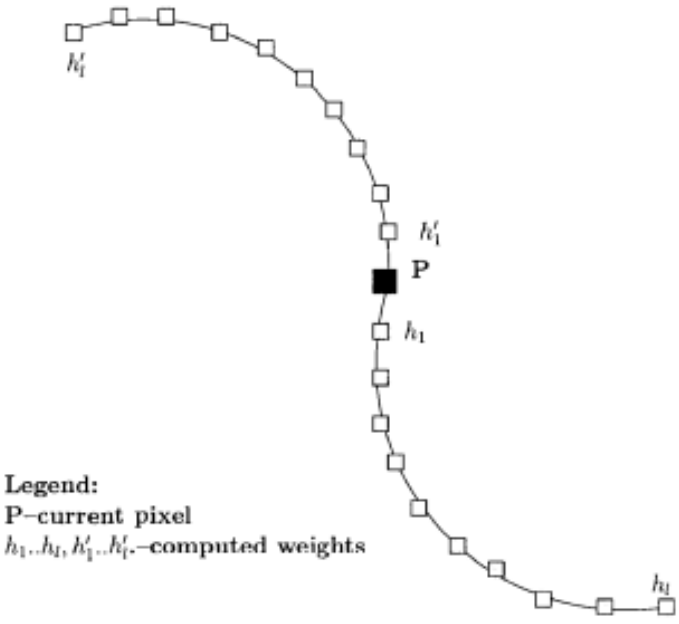


Figura 4.7: Computar los pesos. Fuente: [32]

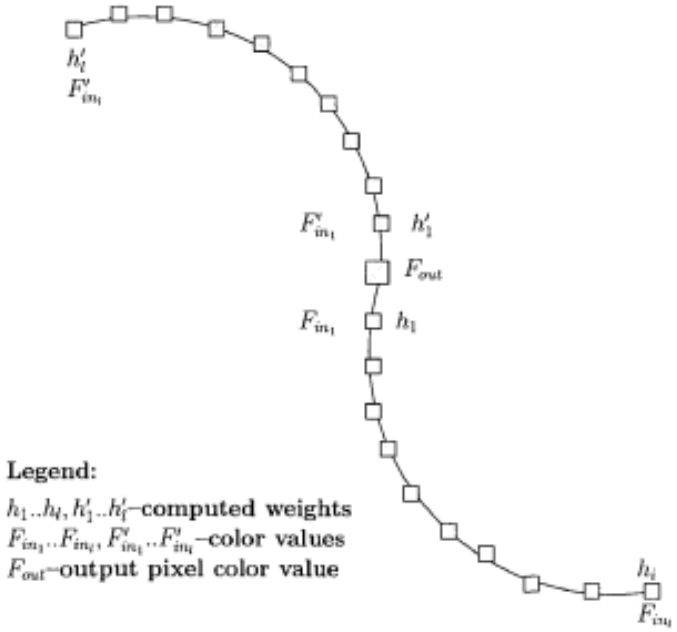


Figura 4.8: Computar valores de salida del pixel. Fuente: [32]

Capítulo 5

Aplicación desarrollada

En este capítulo se introducirá la aplicación que se ha desarrollado con el fin de demostrar los conceptos expuestos en los capítulos anteriores. En concreto se desarrollarán shaders para los siguientes problemas:

- Coloreado de terrenos
- Curvas de Bézier
- Superficies de Bézier
- Sólidos de revolución
- Nubes de puntos
- Negativo de una imagen
- Line Integral Convolution

Para el desarrollo de la aplicación y los shaders será necesario introducir algunos conceptos matemáticos importantes, que se explican en la sección 5.4.

5.1. Plan de desarrollo

Durante las primera semanas de desarrollo de la aplicación lo más importante fue realizar un exhaustivo estudio del funcionamiento de OpenGL, así como la lectura y realización de tutoriales sobre la materia. Una vez adquirido el conocimiento necesario, se comenzó a desarrollar el esqueleto principal de la aplicación, sobre el cual se incorporarían después los distintos tipos de visualización a realizar.

Una vez desarrollado este esqueleto se procedió a la preparación de los shaders que se utilizarían en cada uno de los problemas, para lo que se utilizó en gran medida la información expuesta en el texto Bailey and Cunningham [11].

El siguiente paso fue conseguir datos y prepararlos adecuadamente para poder mostrar las capacidades de visualización de la aplicación, así como comprobar su correcto funcionamiento.

Lo anterior fue reiterado con cada uno de los problemas, añadiéndose cada vez más a lo largo del desarrollo de todo el proyecto.

5.2. Herramientas de desarrollo

Como entorno de desarrollo principal se ha utilizado el sistema operativo Ubuntu Linux 18.04 LTS [33]. En este sistema, además, se han utilizado las siguientes herramientas de desarrollo:

- Vim como editor de textos. [34]
- Git para el control de versiones. [35]
- Github como respositorio. [36]
- GCC como compilador. [37]
- GDB para la depuración. [38]
- Make para la gestión de dependencias. [39]

.

Asimismo, siguiendo las recomendaciones del tutorial de Vries [40], se han utilizado las siguientes librerías:

- GLFW [41]. GLFW es una librería orientada específicamente a OpenGL que proporciona las necesidades básicas para el renderizado en pantalla. Permite crear un contexto de OpenGL, definir parámetros de ventana y manejar la entrada del usuario. Estas son las funciones que utilizaremos en la aplicación.

- GLAD [42]. La localización de funciones de OpenGL depende tanto del controlador gráfico utilizado como del sistema operativo utilizado. Esta localización es desconocida en tiempo de compilación y ha de ser conseguida en tiempo de ejecución. Es, pues, tarea del programador conseguir la localización de estas funciones. GLAD es una librería que realiza esta tarea automáticamente.

- Assimp [43]. Assimp — *The Open-Asset-Importer-Lib* — es una librería que permite importar diferentes formatos de modelos 3D de una manera uniforme. Será utilizado para cargar los modelos para el colorado de terrenos.

- GLM [44]. GLM — OpenGL Mathematics — es una librería para matemáticas en software gráfico en C++ basada en las especificaciones del lenguaje GLSL. Proporciona funciones diseñadas e implementadas con el mismo convenio de nombres y funcionalidades que GLSL. Proporciona capacidades como transformaciones de matrices, cuaterniones, empaquetado de datos, aleatoriedad, ruido...

- Otras librerías especificas del sistema operativo, como Pthreads, xrandr, x11, xi, xcursor, etc.

5.3. Diseño de la aplicación

Para esta aplicación se ha optado por un diseño modular orientado a objetos, en el que poder incrementalmente añadir distintos tipos de visualización sin tener demasiados problemas. Como se ha expuesto en la sección 5.1 la aplicación consta de un esqueleto principal utilizado por todos los tipos de visualización. Este esqueleto consta de una ventana principal, creada en el programa principal, en la que se renderizará el objeto particular que representa el tipo de visualización. Así, con el fin de añadir un nuevo tipo de visualización solo se tendrían que realizar las siguientes acciones:

1. Crear un nuevo objeto que implemente los métodos necesarios de la clase `Object`.

2. Añadir un nuevo modo al `enum Modes`.

3. Añadir las opciones necesarias para dicho objeto en el programa principal e incluir la nueva clase `#include "class.h"`.

4. Actualizar las dependencias en el Makefile.

En esta ventana principal se tiene por defecto un sistema de cámara en primera persona, con la capacidad de moverse para visualizar mejor detalles del objeto en cuestión. Este sistema puede ser sobreescrito en el objeto específico en caso de necesitar otro comportamiento.

En la clase `Object` existen tres métodos virtuales puros, que han de ser implementados por las clases específicas de cada tipo de visualización:

- `draw()`, que ha de encargarse de dibujar el objeto en cuestión.

- `processInput(GLFWwindow * window)`, que ha de especificar que hacer con la entrada del usuario para este tipo de visualización.

- `setUniforms()`, que ha de especificar las variables `uniform` que utilicen los shaders de este tipo de visualización.

Estos métodos se llaman una vez por vuelta del bucle principal. En la sección siguiente se introducen las matemáticas necesarias e importantes para el desarrollo de la aplicación y los shaders concretos.

5.4. Matemáticas necesarias

Con el fin de desarrollar el sistema de cámaras que utiliza la aplicación es necesario conocer varios conceptos importantes sobre álgebra, geometría lineal y y proyectiva y transformaciones matriciales, así como los ángulos de Euler y su relación con los cuaterniones. También se explorarán distintos métodos numéricos, relevantes en el método de Line Integral Convolution, así como fórmulas matemáticas específicas de cada tipo de visualización.

5.4.1. Transformaciones matriciales

Como vamos a trabajar con objetos tridimensionales y una cámara móvil, necesitamos realizar transformaciones sobre los vértices que componen nuestros objetos para que estos aparezcan en su lugar y con sus dimensiones adecuadas. Es deseable, además, realizar estas operaciones con vectores matricialmente, puesto que éstas permiten presentar transforma-

ciones arbitrarias en un formato consistente y apto para la computación. Así, se pueden concatenar diferentes transformaciones de manera sencilla multiplicando sus matrices.

Entre estas transformaciones podemos encontrar lineales y no lineales. Sin embargo, las transformaciones no lineales y proyectivas no se pueden representar con una matriz de dimensión la dimensión del espacio Euclídeo. Así, para representar matricialmente transformaciones no lineales en un espacio Euclídeo n-dimensional $\mathbb{R}^n$ se ha de utilizar una transformación lineal en el espacio $(n+1)$-dimensional $\mathbb{R}^{n+1}$. Esta es la razón por la que utilizaremos geometría proyectiva y *coordenadas homogéneas* en la aplicación.

En esta sección se presentan las coordenadas homogéneas y transformaciones lineales y afines más habituales y necesarias para nuestra aplicación, así como sus formas matriciales.

Coordenadas Homogéneas

Las coordenadas homogéneas son un sistema de coordenadas utilizado en geometría proyectiva, del mismo modo que las coordenadas cartesianas se utilizan en la geometría Euclídea. Tienen la ventaja de que las coordenadas de todos los puntos del espacio, incluidos los puntos en el infinito, se pueden representar mediante coordenadas finitas.

Debido a que este sistema permite la representación de puntos en el infinito, el número de coordenadas necesarias para permitirlo ha de ser uno más que la dimensión del espacio proyectivo considerado. Este sistema, además, permite la representación de transformaciones afines y proyectivas de forma matricial, proporcionando un método rápido y eficiente de calcular proyecciones consecutivas. Como en nuestra aplicación estamos interesados principalmente en el espacio tridimensional, las matrices que utilizaremos serán 4×4. Y los puntos se representarán con 4 coordenadas. Cabe destacar algunas de las propiedades de las coordenadas homogéneas antes de continuar con las transformaciones que vamos a utilizar.

- Un punto en el espacio tridimensional está representado por 4 coordenadas homogéneas (x, y, z, w), donde x, y, z y w no son todas 0.

- Dos tuplas de coordenadas homogéneas p y q que cumplen la relación $p = a\dot{q}, a \in \mathbb{R}$, es decir, que sólo se diferencian en la multiplicación por un escalar, representan el mismo punto en el espacio.

- Si $w \neq 0$, entonces el punto representado es el punto $(\frac{x}{w}, \frac{y}{w}, \frac{z}{w})$ del espacio Euclídeo tridimensional.

- Si $w = 0$, entonces el punto representado en un punto del infinito.

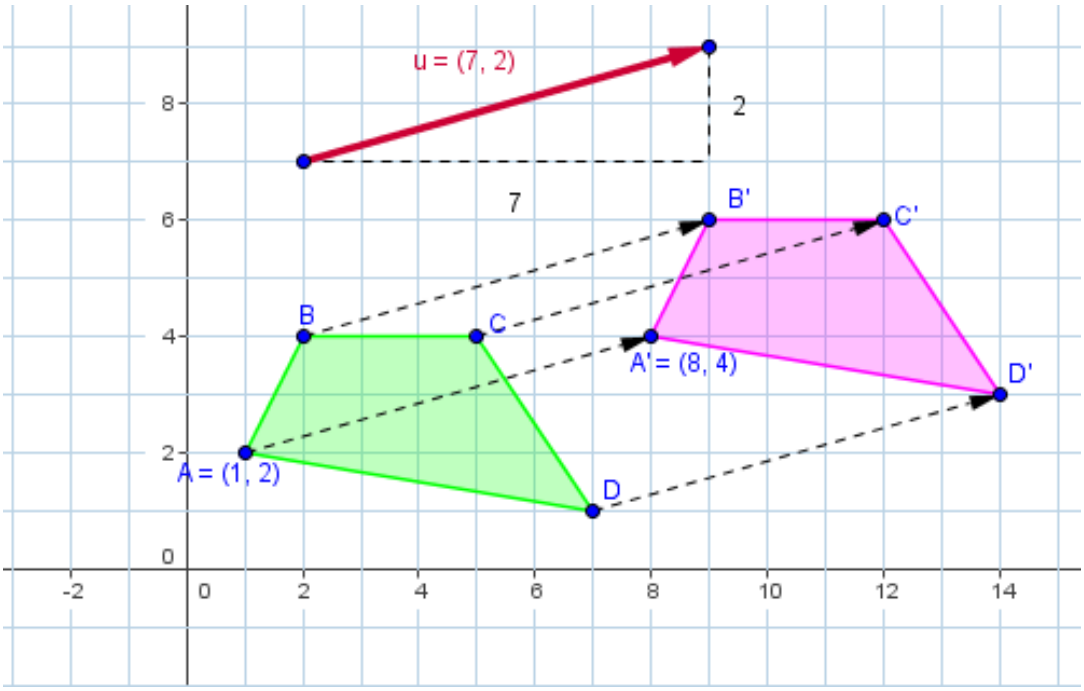


Figura 5.1: Traslación de un vector. Fuente: [45]

Traslación

Se denomina traslación a la operación consistente en *mover* un punto en una posición a otra nueva posición. Supongamos, pues, que queremos trasladar un punto $p = (x, y, z)$ en la dirección marcada por el vector $\overrightarrow{t} = (t_1, t_2, t_3)$ como se muestra en la figura 5.1. Para ello realizaríamos la siguiente operación:

$$p' = p + \overrightarrow{t} = \begin{pmatrix} x + t_1 \\ y + t_2 \\ z + t_3 \end{pmatrix} \tag{5.1}$$

La traslación se trata de una transformación afín sin puntos fijos. Como se ha expuesto previamente, para poder representar esta transformación de forma matricial se ha de recurrir a un espacio de una dimensión más. Por tanto, se recurre a las coordenadas homogéneas para representar la traslación de un espacio vectorial con multiplicación de matrices. Escribiendo el punto $p = (x, y, z)$ utilizando una cuarta coordenada homogénea $p = (x, y, z, 1)$. Esta operación se muestra en (5.2).

$$\overrightarrow{v}' = \begin{pmatrix} 1 & 0 & 0 & t_1 \\ 0 & 1 & 0 & t_2 \\ 0 & 0 & 1 & t_3 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x + t_1 \\ y + t_2 \\ z + t_3 \\ 1 \end{pmatrix} \tag{5.2}$$

Escalado

El escalado de un vector es la operación consistente en modificar la longitud del vector. Para ello, debemos multiplicar cada una de sus coordenadas por el factor de escalado deseado en cada eje. Es decir, para escalar un vector $\overrightarrow{v}$ por un factor de 0,5 en el eje x y 3 en el eje y, la operación a realizar sería la siguiente:

$$\overrightarrow{v} = \begin{pmatrix} 2 \\ 3 \end{pmatrix} \qquad \overrightarrow{v}' = \begin{pmatrix} 2 \cdot 0{,}5 \\ 3 \cdot 3 \end{pmatrix} = \begin{pmatrix} 1 \\ 9 \end{pmatrix} \tag{5.3}$$

Esta operación de escalado se puede escribir matricialmente en coordenadas homogéneas como sigue. Supongamos que tenemos un vector $\overrightarrow{v} = (x, y, z)$ y lo queremos escalar por un factor $fac = (F_1, F_2, F_3)$. Entonces podemos escribir la operación anterior con la matriz FAC como sigue:

$$\overrightarrow{v}' = \begin{pmatrix} F_1 & 0 & 0 & 0 \\ 0 & F_2 & 0 & 0 \\ 0 & 0 & F_3 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} F_1 \cdot x \\ F_2 \cdot y \\ F_3 \cdot z \\ 1 \end{pmatrix} \tag{5.4}$$

Rotación

Al contrario de los casos de la rotación y traslación de vectores expuestas anteriormente, el caso de la rotación requiere un estudio más profundo en el caso tridimensional para la matemática aplicada. Por esto, se dedica una sección exclusiva para este tema. (Ver sección 5.4.2).

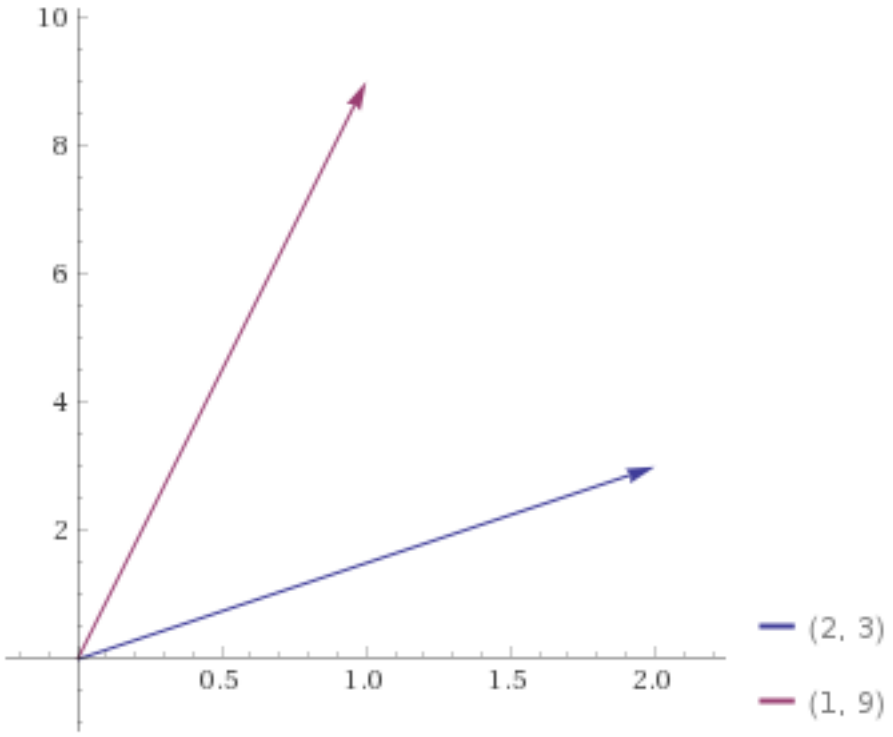


Figura 5.2: Escalar un vector. Fuente: [46]

Perspectiva

Además de las transformaciones anteriores, se hace necesario realizar una última, muy importante en el caso de la visualización. Se trata de la perspectiva. Con esta transformación se conseguirá que los objetos más lejanos aparezcan más pequeños mientras que los más cercanos aparecerán más grandes.

En la creación de esta matriz aparecen diferentes parámetros a tener en cuenta, que son los siguientes:

- *aspect* - El ratio entre la anchura y altura del rectángulo en el que se realizará esta proyección
- fov - El campo de visión vertical. Es el ángulo vertical de la cámara mediante la que estamos viendo la escena
- *near* - La localización del plano Z cercano. Se utiliza para realizar clipping sobre los objetos que quedan demasiado cerca de la cámara

- far - La localización del plano Z lejano. Se utiliza para realizar clipping sobre los objetos que quedan demasiado lejos de la cámara

La matriz correspondiente a esta transformación es la siguiente:

$$\begin{pmatrix} \frac{1}{aspect*\tan\frac{fov}{2}} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan\frac{fov}{2}} & 0 & 0 \\ 0 & 0 & -\frac{far+near}{far-near} & -\frac{2far\cdot near}{far-near} \\ 0 & 0 & -1 & 0 \end{pmatrix} \tag{5.5}$$

La derivación de esta matriz puede verse en Meiri [47].

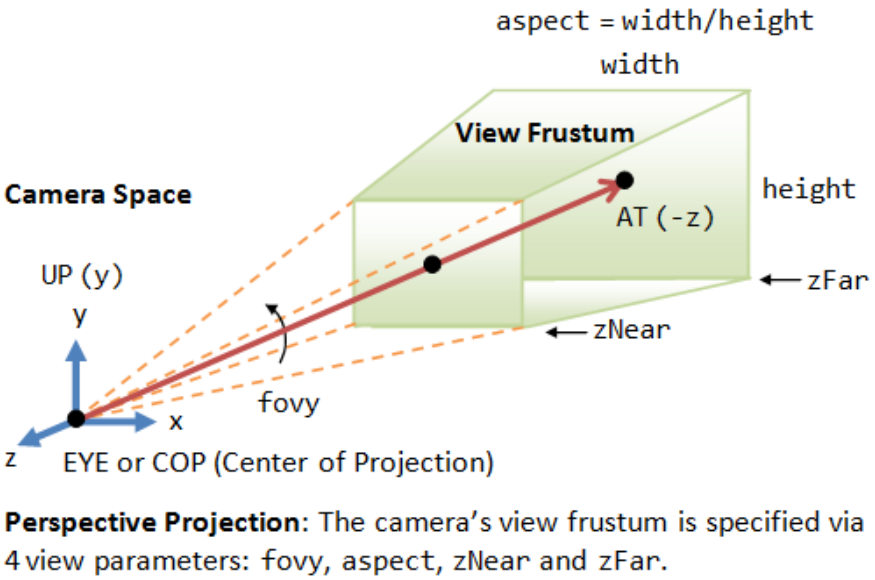


Figura 5.3: Proyección perspectiva - Parámetros y significado. Fuente: [48]

5.4.2. Rotación: Ángulos de Euler y Cuaterniones

La rotación tridimensional es un caso particularmente complejo. Esto se debe a determinados problemas que surgen a la hora de formalizar matemáticamente este movimiento.

La manera más intuitiva de pensar en la rotación consiste en especificar esta rotación mediante un eje de giro y un ángulo. Con esto, el movimiento consistiría en rotar el vector en torno al eje de giro. (Ver Figura 5.4).

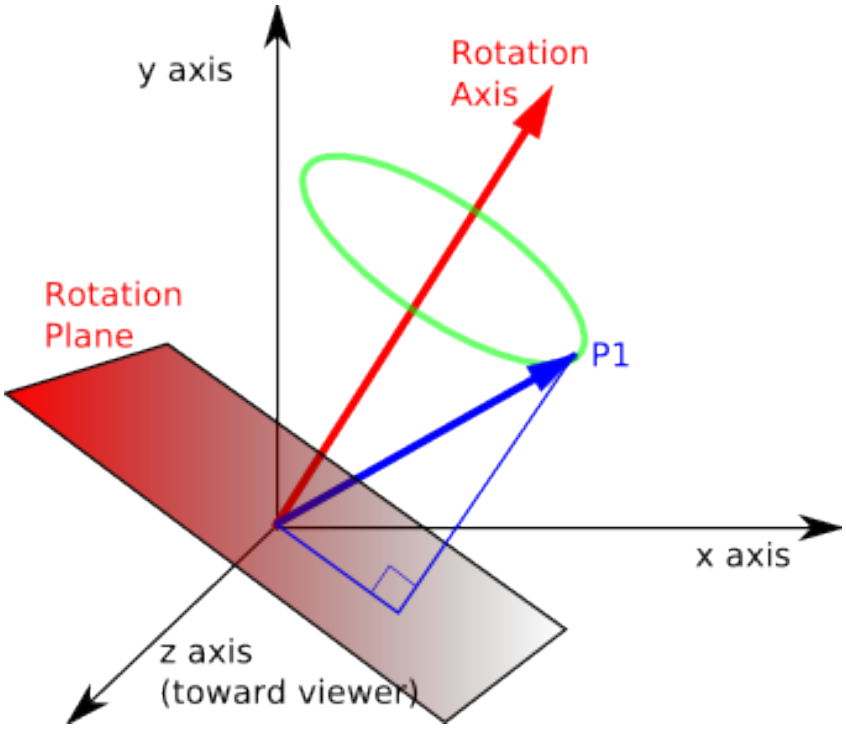


Figura 5.4: Rotación de un vector 3D. Fuente: [49]

Existen diversas maneras de formular matemáticamente este movimiento. Para ello existe un grupo, llamado $SO(3)$, que es el grupo de todas las rotaciones en torno al origen del espacio vectorial Euclídeo $\mathbb{R}^3$ bajo la operación de la composición. Una rotación en $\mathbb{R}^3$ es una aplicación lineal. Como toda aplicación lineal en un espacio vectorial, ésta puede ser representada mediante una matriz, dando lugar a la representación matricial de la rotación.

Ángulos de Euler y Matriz de rotación

En el espacio vectorial $\mathbb{R}^3$, los vectores se representan a partir de una base. Esta base estará formada por tres vectores unitarios. Por tanto, si queremos realizar una rotación de cualquier otro vector en $\mathbb{R}^3$, bastará con rotar los vectores de la base acorde al eje de rotación y después calcular el vector a rotar en términos de la base rotada.

Los vectores rotados que forman la nueva base definen completamente la rotación y, escritos como una matriz nos da la matriz de rotación. Esta matriz, al ser los vectores una base ortonormal de $\mathbb{R}^3$, forman una matriz ortogonal. Por tanto, el grupo $SO(3)$ se identifica con el grupo formado por las matrices ortogonales 3×3 bajo la operación de la multiplicación. Estas matrices se conocen como *Matrices Ortogonales Especiales (Special Orthogonal Matrices)*, de ahí la notación de $SO(3)$.

Como sabemos, dos rotaciones en $SO(3)$ se pueden concatenar mediante la composición, dando lugar a otra nueva rotación. Lo mismo pasa con la representación matricial, multiplicando dos matrices de rotación. Para nuestra aplicación, es conveniente realizar las rotaciones como composición de rotaciones en torno a los ejes de referencia. Para ello definimos las matrices de rotación, de nuevo utilizando coordenadas homogéneas acorde a lo explicado anteriormente.

$$R_x(\theta) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{5.6}$$

$$R_y(\theta) = \begin{pmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{5.7}$$

$$R_z(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{5.8}$$

Esto da lugar a los conocidos como ángulos de Euler, llamados en inglés *yaw, pitch, roll.* (Ver Figura 5.5). Así, una rotación cualquiera en el espacio tridimensional viene dada por:

$$R = Roll(\phi)Pitch(\theta)Yaw(\varphi) = R_x(\phi)R_y(\theta)R_z(\varphi) \tag{5.9}$$

Ahora bien, existen diversos problemas derivados de la utilización de este sistema. El más importante es el llamado Bloqueo del Cardán [50]. Aunque los formalismos matemáticos son más complejos, intuitivamente este bloqueo ocurre debido a que la descripción de cualquier rotación tridimensional mediante ángulos de Euler no es única y existen puntos sobre los que no todo cambio en el espacio de rotaciones puede ser expresado mediante cambios en el espacio de los ángulos de Euler.

Con el fin de disminuir el riesgo del bloqueo, se puede prescindir de la multiplicación de matrices utilizando una sola matriz utilizando un eje de giro arbitrario unitario $u = (u_x, u_y, u_z)$ y un ángulo de giro θ. La procedencia de dicha matriz se puede consultar en Rodrigues [51].

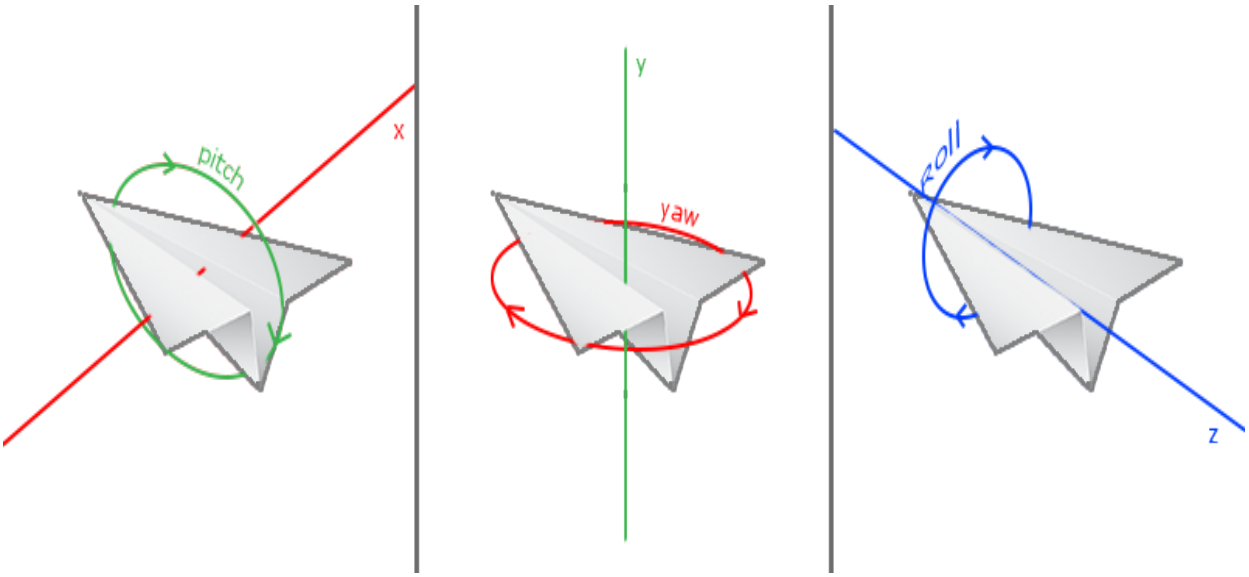


Figura 5.5: Ángulos de Euler. Fuente: de Vries [40]

$$R = \begin{pmatrix} \cos\theta + u_x^2(1-\cos\theta) & u_x u_y(1-\cos\theta) - u_z\sin\theta & u_x u_z(1-\cos\theta) + u_y\sin\theta & 0 \\ u_y u_x(1-\cos\theta) + u_z\sin\theta & \cos\theta + u_y^2(1-\cos\theta) & u_y u_z(1-\cos\theta) - u_x\sin\theta & 0 \\ u_z u_x(1-\cos\theta) - u_y\sin\theta & u_z u_y(1-\cos\theta) + u_x\sin\theta & \cos\theta + u_z^2(1-\cos\theta) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{5.10}$$

Cuaterniones

Debido a la importancia de la precisión en muchos sistemas informáticos que necesitan cálculos de rotaciones y el problema que supone el bloqueo del Cardán, se hace importante la mención de los cuaterniones como sistema de representación de rotaciones en tres dimensiones, aunque no los vayamos a utilizar en nuestra aplicación.

Los cuaterniones, también conocidos como cuaternios, suponen una notación para representar orientaciones y rotaciones en tres dimensiones. En contraposición con los ángulos de Euler, son más sencillos de componer y previenen el bloqueo del Cardán. Comparados con las matrices de rotación, suponen un método más compacto, más estable numéricamente y más eficiente.

Los cuaterniones son una extensión de los números complejos, descritos por primera vez por William Rowan Hamilton en 1843, definiéndolos como el cociente de dos lineas dirigidas en un espacio tridimensional. Cuando se utilizan para representar rotaciones se les denomina

también cuaterniones de rotación, puesto que representan el grupo $SO(3)$.

Generalmente, se representan de la siguiente forma:

$$a + b\boldsymbol{i} + c\boldsymbol{j} + d\boldsymbol{k}$$

donde a, b, c y d son números reales y $\boldsymbol{i}, \boldsymbol{j}$ y $\boldsymbol{k}$ son las unidades fundamentales del cuaternión. Además, los cuaterniones siguen las reglas algebraicas usuales, excepto la de la propiedad conmutativa de la multiplicación, y cumplen la siguiente propiedad:

$$\boldsymbol{i}^2 = \boldsymbol{j}^2 = \boldsymbol{k}^2 = \boldsymbol{ijk} = -1$$

Es conveniente verlos representados como un escalar más un vector, es decir:

$$a + b\boldsymbol{i} + c\boldsymbol{j} + d\boldsymbol{k} = a + \overrightarrow{v}$$

considerando la parte imaginaria $b\boldsymbol{i} + c\boldsymbol{j} + d\boldsymbol{k}$ como un vector $\overrightarrow{v} = (b, c, d)$.

Recordemos que cualquier rotación en el espacio tridimensional puede verse como una giro de ángulo θ en torno a un eje definido por un vector unitario u. Esto puede ser representado mediante un cuaternión de la siguiente forma:

$$\boldsymbol{q} = \exp\frac{\theta}{2}(u_x\boldsymbol{i} + u_y\boldsymbol{j} + u_z\boldsymbol{k}) = \cos\frac{\theta}{2} + (u_x\boldsymbol{i} + u_y\boldsymbol{j} + u_z\boldsymbol{k})\sin\frac{\theta}{2}$$

Se puede demostrar que la rotación deseada se puede aplicar a un vector ordinario $\boldsymbol{p} = (p_x, p_y, p_z) = p_x\boldsymbol{i} + p_y\boldsymbol{j} + p_z\boldsymbol{k}$ en el espacio tridimensional, considerado como un cuaternión con parte real igual a cero, evaluando la conjunción de $\boldsymbol{p}$ por $\boldsymbol{q}$:

$$\boldsymbol{p'} = \boldsymbol{qpq^{-1}}$$

donde $\boldsymbol{p'}$ es la nueva posición del vector $\boldsymbol{p}$ tras la rotación. La parte vectorial del este cuaternión es el vector deseado. Se puede leer más acerca de cuaterniones utilizados para las rotaciones en Vicci [52].

5.4.3. Métodos Numéricos para la resolución de Ecuaciones Diferenciales

Como ya se vio en la sección 4.6, en el método del Line Integral Convolution se ha de utilizar un método numérico para computar la línea de flujo. En esta sección se analizan algunos de los métodos utilizados y propuestos tanto por Cabral and Leedom [31] como por Petkov [32].

Para poder analizar los métodos hemos de introducir algunas definiciones relacionadas con ellos. Recordemos que un método numérico sirve para calcular de manera aproximada la solución de una ecuación diferencial en un intervalo $[t_0, T]$. Para ello, dividiremos el intervalo en una serie de puntos, dando lugar a los siguientes conceptos [53]:

- **Puntos de red.** Cada uno de los puntos en los que se divide el intervalo $[t_0, T]$. Los subintervalos resultantes pueden ser de longitud constante (Redes uniformes de paso $h = \frac{T-t_0}{N}$) o de longitud variable, dando lugar a los métodos númericos de paso variable.

- **Esquema Numérico.** Se denomina esquema numérico al proceso iterativo mediante el cual, conociendo los r primeros valores $x_0, x_1, \ldots, x_{r-1}$, que son aproximaciones de los valores exactos $x(t_0), x(t_1), \ldots, x(t_{n-1})$ de la solución de la ecuación diferencial en los puntos $t_0, t_1, \ldots, t_{r-1}$ podemos calcular todos los demás valores $x_n, n = r, \ldots, N$, que son aproximaciones de los valores exactos en los puntos de red.

  El esquema numérico se escribe de la siguiente forma:

$$\begin{cases} x_0, x_1, \ldots, x_{r-1} \text{ dados} \\ x_{n-r} = \Phi(t_n, x_n, x_{n+1}, \ldots, x_{n+r-1}, h), \ n = 0, \ldots, N-r \end{cases} \tag{5.11}$$

- **Error Local de Truncamiento.** Dado un esquema como el anterior, suponiendo que las soluciones exactas verifican

$$x(t_{n+r}) = \Phi(t_n, t_{n+1}, \ldots, t_{n+r-1}, x(t_n), x(t_{n+1}), \ldots, x(t_{n+r-1})) + h\tau_{n+r}(h) \tag{5.12}$$

  el error local de truncamiento viene dado por

$$\tau(h) := \max_{n=0,\ldots,N-r} |\tau_{n+r}(h)| \tag{5.13}$$

que consiste en el error que se comete al calcular el valor exacto utilizando el esquema numérico.

- **Consistencia.** Se dice que un método es consistente si

$$\lim_{h\to 0} \tau(h) = 0 \tag{5.14}$$

Se dice que es consistente de orden p si

$$\tau(h) = O(h^p) \tag{5.15}$$

- **Error Global de Discretización.** Siguiendo la notación anterior, el Error global de discretización viene dado por

$$\epsilon(h) := \max_{n=0,\dots,N} |x_n - x(t_n)| \tag{5.16}$$

- **Convergencia.** Se dice que un método es convergente cuando se verifica lo siguiente:

$$si \quad \max_{k=0,\dots,r-1} |x_k - x(t_k)| \xrightarrow{h\to 0} 0 \quad entonces \quad \lim_{h\to 0} \epsilon(h) = 0 \tag{5.17}$$

Se dice que es convergente de orden p si

$$\epsilon(h) = O(h^p) \tag{5.18}$$

Con estas nociones ya podemos empezar a analizar los distintos métodos utilizados, para así conocer cuáles proporcionarán mejores resultados.

Método de Euler

El método de Euler es el más sencillo de los métodos numéricos a considerar. Se trata de un método por aproximación de la derivada. Para ver cómo aparece este método, hemos de recurrir a la definición de derivada. Para cada $t \in (t_0, T)$

$$x'(t) = \lim_{h \to 0} \frac{x(t+h) - x(t)}{h} \tag{5.19}$$

Si $h > 0$ es suficientemente pequeño, podemos suponer que

$$x'(t_n) \approx \frac{x(t_n + h) - x(t_n)}{h} = \frac{x(t_{n+1}) - x(t_n)}{h} \tag{5.20}$$

lo que nos conduce a que

$$x(t_{n+1}) \approx x(t_n) + hf(t_n, x(t_n)) \tag{5.21}$$

De aquí aparece el esquema numérico del método de Euler

$$\begin{cases} x_{n+1} = x_n + hf(t_n, x_n) \; n = 0, \ldots, N-1 \\ x_0 \approx a \end{cases} \tag{5.22}$$

En Arrieta et al. [53] se puede ver una demostración de que el método de Euler supone un método **consistente de orden 1** y **convergente**.

Método de Euler de paso variable

El método de Euler de paso variable sigue la misma idea general que el método de Euler de paso uniforme. En este caso, sin embargo, el siguiente punto en el que calcular la solución aproximada se calcula en cada paso. Para ello, se ha de fijar previamente una tolerancia permitida para el error cometido. Por tanto, necesitamos ser capaces de **estimar** el error que vamos a cometer al escoger el siguiente paso. Entonces, si el error estimado es mayor que la tolerancia fijada, el cálculo se descarta y se vuelve a realizar, y si éste es menor que la tolerancia, entonces se acepta y se pasa al siguiente cálculo.

Con el fin de realizar esta estimación, como hemos dicho, necesitamos introducir los siguientes conceptos:

- **Error local relativo al paso h.** Se define el error local relativo al paso h en el nodo $t_{n+1} = t_n + h$ como

$$ERR_n(h) := \frac{|y(t_n + h) - y_1(t_n, h)|}{h} \tag{5.23}$$

donde la función $y = y(t)$ es la solución del problema y la función $y_1(t_n, h)$ representa la aproximación numérica desde el instante t_n en un paso h,dada por $y_1(t_n, h) := x_n + h\Phi(t_n, x_n, h)$ Con esta notación, $x_{n+1} = y_1(t_n, hn + 1)$.

- **Error local relativo.** Se define el error local relativo como

$$ERR(h_1, \ldots, h_N) := \max_{n=1,\ldots,N-1} |ERR_n(h_n + 1)| \tag{5.24}$$

De nuevo en Arrieta et al. [53] podemos encontrar demostrado que si un método de paso adaptativo es consistente, estable y de orden p, entonces

$$\tau(h_1, \ldots, h_N) \leq Ch_{max}^p$$

para cierta constante C y, por tanto, si $x_0 \to a$ y $h_{max} \to 0$, obtenemos que $\epsilon(h_1, \ldots, h_N) \to 0$.

El esquema para este tipo de métodos es:

$$\begin{cases} x_{n+1} = x_n + h_{n+1}\Phi(t_n, x_n, h_{n+1}) & n = 0, \ldots, N-1 \\ x_0 \sim a \end{cases} \tag{5.25}$$

El método de Euler adaptativo es, por tanto:

$$\begin{cases} x_{n+1} = x_n + h_{n+1}f(t_n, x_n) & n = 0, \ldots, N-1 \\ x_0 \sim a \end{cases} \tag{5.26}$$

Este método es propuesto en Cabral and Leedom [31].

Métodos de Runge-Kutta

Los últimos métodos que vamos a considerar, y en particular el que vamos a implementar en nuestra aplicación para el método del Line Integral Convolution, son los métodos de Runge-Kutta. Esta familia de métodos se basa en añadir puntos intermedios entre los puntos del mallado t_n y t_{n+1} en la media ponderada.

Como ejemplo, si consideramos una media ponderada entre las pendientes en los puntos t_n y $t_n + ch$ con $c \in (0, 1]$ podemos aproximar el valor de $x(t_{n+1})$ por

$$x(t_{n+1}) \approx x(t_n) + h\left[b_1 f(t_n, x(t_n)) + b_2 f(t_n + ch, x(t_n + ch))\right]$$

donde $b_1 + b_2 = 1$, para que sea realmente una media. El problema ahora radica en cómo calcular el valor $x(t_n + ch)$. Una manera de hacerlo es utilizar el método de Euler, es decir,

$$x(t_n + ch) \approx x(t_n) + chf(tn, x(t_n))$$

De esta forma, la aproximación de $x(t_{n+1})$ queda:

$$x(t_{n+1}) \approx x(t_n) + h\Big[b_1 f(t_n, x(t_n)) + b_2 f\Big(t_n + ch, x(t_n) + chf(t_n, x(t_n))\Big)\Big]$$

obteniéndose la familia de métodos Runge-Kutta

$$x_{n+1} = x_n + h\Big[b_1 f(t_n, x_n + b_2 f\Big(t_n + ch, x_n + chf(t_n, x_n)\Big)\Big]$$

que suele escribirse como

$$\begin{cases} K_1 = f(t_n, x_n) \\ K_2 = f(t_n + ch, x_n + chK_1) \\ x_{n+1} = x_n + h(b_1 K_1 + b_2 K_2) \end{cases}$$

En este caso, el método tiene 2 etapas. Esta familia de métodos se puede generalizar, dando lugar a los métodos de Runge-Kutta de s etapas. La forma de escribir estos métodos es la siguiente:

$$\begin{cases} K_i = f\left(t_n + c_i h, x_n + h\sum_{j=1}^{s} a_{ij} K_j\right), \quad i = 1, 2, 3, \ldots, s, \\ x_{n+1} = x_n + h\sum_{i=1}^{s} b_i K_i \end{cases} \tag{5.27}$$

El que vamos a utilizar en nuestra aplicación es el método de Runge-Kutta de orden 4, que tiene el siguiente esquema numérico:

$$\begin{cases} K_1 = f(t, x) \\ K_2 = f\left(t + \frac{h}{2}, x + \frac{h}{2} K_1\right) \\ K_3 = f\left(t + \frac{h}{2}, x + \frac{h}{2} K_2\right) \\ K_4 = f\left(t + h, x + h K_3\right) \\ x_{n+1} = x_n + \frac{h}{6}[K_1 + 2K_2 + 2K_3 + K_4] \end{cases} \tag{5.28}$$

La demostración de que este método es consistente de orden 4 se puede encontrar en Arrieta et al. [53].

5.4.4. Otras Fórmulas

Para el desarrollo de la aplicación también se han tenido que utilizar fórmulas matemáticas que describen el comportamiento de las curvas y superficies de Bézier, ya explicadas en la sección 4.2, así como las fórmulas para la obtención de los valores de los píxeles de salida en el método del Line Integral Convolution (Sección 4.6).

5.5. Shaders en la aplicación

En esta sección se presentan ya los shaders desarrollados para cada uno de los tipos de visualización considerados en el proyecto, analizando los distintos cálculos realizados y explorando las entradas y salidas de cada uno de ellos.

5.5.1. Coloreado de terrenos

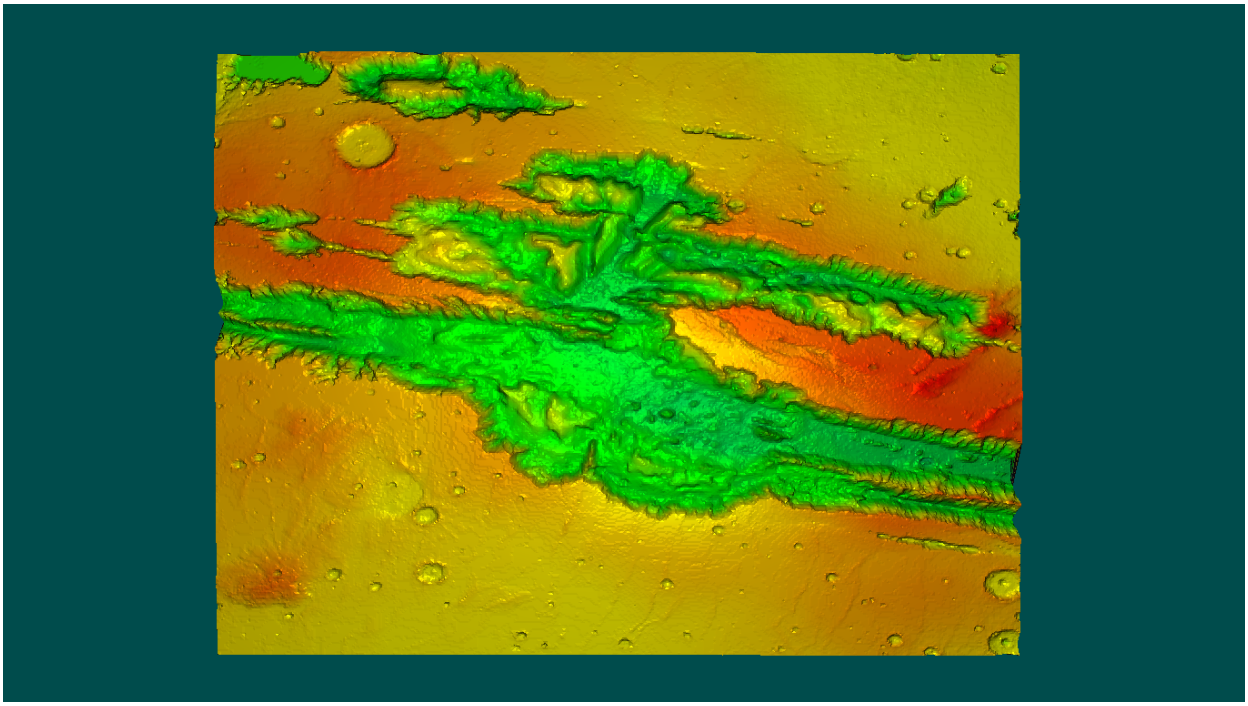


Figura 5.6: Coloreado de Valles Marineris. Modelo: [54]

Para este problema, descrito en la sección 4.5, se ha elaborado un vertex shader y un fragment shader. Partiendo del modelo de un terreno existen dos posibilidades: que las alturas vengan codificadas en forma de textura o que venga implícitamente en la posición de los vértices. En este último caso, en la aplicación se lee el valor más alto y el valor más bajo en el eje y para realizar la coloración.

Así, el vertex shader toma como entrada la posición y los normales de los vértices, sacados de un modelo tridimensional.

```
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;
```

Como solo se están utilizando el vertex shader y fragment shader, la salida del vertex shader se da como entrada al fragment shader. Esta salida corresponde al normal, que es el mismo que entra; la posición del fragmento y el color del fragmento. Estas dos últimas variables se calculan en el vertex shader utilizando la altura máxima y mínima del modelo, que se pasan mediante variables `uniform`. Además, con el fin de mejorar el rendimiento, las transformaciones de los vértices, que vienen dadas por las matrices de modelo, vista y proyección, se calculan también en este shader. El siguiente fragmento de código muestra las salidas del vertex shader junto con las variables `uniform` utilizadas:

```
out vec3 vNormal;
out vec3 vFragPos;
out vec3 vFragColor;

uniform float uMaxHeight;
uniform float uMinHeight;
uniform mat4 uModel;
uniform mat4 uView;
uniform mat4 uProjection;
```

El color del fragmento se calcula realizando una interpolación entre el valor máximo de altura(rojo), el valor tres cuartos de altura(amarillo), el valor medio de la altura(verde) y el valor mínimo de altura(azul). El código siguiente muestra el cálculo del color de un fragmento entre la altura máxima y los tres cuartos:

```
if (aPos.z > threecuarters) {
    alpha = smoothstep(uMaxHeight, threecuarters, aPos.z);
    vFragColor = mix(RED, YELLOW, alpha);
}
```

Asimismo, para calcular la posición del fragmento se utiliza el siguiente código:

```
vFragPos = vec3(uModel * vec4(aPos, 1.0));
gl_Position =  uProjection * uView * vec4(vFragPos, 1.0);
```

Esto es todo lo que realiza el vertex shader, y sería suficiente para renderizar el terreno coloreado. Sin embargo, con el fin de darle más realismo, en esta ocasión se ha utilizado el fragment shader para dotar de iluminación a la escena. Así, se han utilizado los normales, la posición y el color del fragmento para elaborar un modelo de iluminación ADS *(Ambient, Diffusion, Specular)* [11]. Esto es todo lo que realiza el fragment shader, quedándonos lo siguiente:

```
//Ambient
vec3 ambient = uLight.ambient * vFragColor;

//Difuse
vec3 norm = normalize(uNormalMatrix * vNormal);
vec3 lightDir = normalize(uLight.position - vFragPos);
float diff = max(dot(norm, lightDir), 0.0);
vec3 diffuse = uLight.diffuse * (diff * vFragColor);

//Specular
vec3 viewDir = normalize(uViewPos - vFragPos);
```

```
vec3 reflectDir = reflect(-lightDir, norm);
float spec = pow(max(dot(viewDir, reflectDir), 0.0), uShininess);
vec3 specular = uLight.specular * (spec * vFragColor);

vec3 light = ambient + diffuse + specular;
fFragColor = vec4(light, 1.0);
```

El resultado final de la utilización de estos shaders se puede ver en la figura 5.6, para la que se ha utilizado un modelo de la NASA de Valles Marineris, un sistema de cañones que recorre el ecuador de marte.

5.5.2. Curvas de Bézier

Para mostrar las curvas de Bézier se han utilizado un vertex shader, un geometry shader y un fragment shader. Además, se ha utilizado otro par de shaders para renderizar tanto los ejes como los puntos de control de la curva.

En este caso, todo el trabajo se realiza en el geometry shader. El vertex shader simplemente se encarga de realizar la transformación mediante

```
gl_Position = uProjection * uView * uModel * vec4(aPos, 1.0);
```

mientras que el fragment shaders solo se encarga de darle color a la curva mediante la instrucción

```
fFragColor = vec4(ORANGE, 1.0);
```

Así pues, veamos qué acciones realizar en el geometry shader para conseguir, mediante los 4 vértices de control, renderizar una curva de Bézier en nuestra aplicación. Este shader es la versión de Bailey and Cunningham [11].

Recordemos de la sección 3.3 que para el shader geométrico ha de especificarse la primitiva de entrada y la de salida. Esto se realiza mediante las líneas

```
layout( lines_adjacency ) in;
layout( line_strip, max_vertices=256 ) out;
```

Utilizamos `lines_adjacency` puesto que como entrada tomaremos los 4 puntos de control (y esta es la única entrada que toma 4 puntos. Ver tabla 3.1) y como salida `line_strip` pues queremos renderizar nuestra curva como una serie de segmentos. El nivel de detalle de nuestra curva vendrá dado por este número de segmentos, que se especifica mediante una

variable `uniform`.

```
float dt = 1. / float(uNum);
float t = 0.;
for( int i  = 0; i <=uNum; i++, t+= dt){
    float omt = 1. -t;
    float omt2 = omt * omt;
    float omt3 = omt * omt2;
    float t2 = t * t;
    float t3 = t * t2;
    vec4 xyzw = omt3 * gl_PositionIn[0] +
    3. * t * omt2 * gl_PositionIn[1] +
    3. * t2 * omt * gl_PositionIn[2] +
    t3 * gl_PositionIn[3];
    gl_Position = xyzw;
    EmitVertex( );
}
```

El código anterior supone la totalidad del geometry shader. En él, mediante el uniform `uNum` indicamos la cantidad de puntos que queremos incluir de la curva, obteniendo mayor detalle cuanto más alto es este número. El vector `xyzw` se calcula mediante la ecuación (4.3). El resultado de este shader puede verse en la Figura 5.7.

5.5.3. Superficies de Bézier

En este caso, en lugar de utilizar el shader geométrico como en el caso anterior, utilizaremos los shaders de teselación. Por tanto, tendremos, al igual que en el caso anterior, un vertex shader, un tessellation control shader, un tessellation evaluation shader y un fragment shader, además de los mismos shaders que en el caso anterior para renderizar los ejes y los puntos de control.

El vertex y fragment shader son exactamente iguales que en el caso de la curva de Bézier, por lo que nos centraremos exclusivamente en los shaders de teselación. El TCS ha de marcar el nivel de teselación, acorde a lo explicado en la sección 3.2, y, debido a que queremos utilizar cuadriláteros y guiándonos por la Figura 3.2b, debemos especificar los siguientes niveles de teselación:

```
gl_TessLevelOuter[0] = gl_TessLevelOuter[2] = uOuter02;
gl_TessLevelOuter[1] = gl_TessLevelOuter[3] = uOuter13;
gl_TessLevelInner[0] = uInner0;
gl_TessLevelInner[1] = uInner1;
```

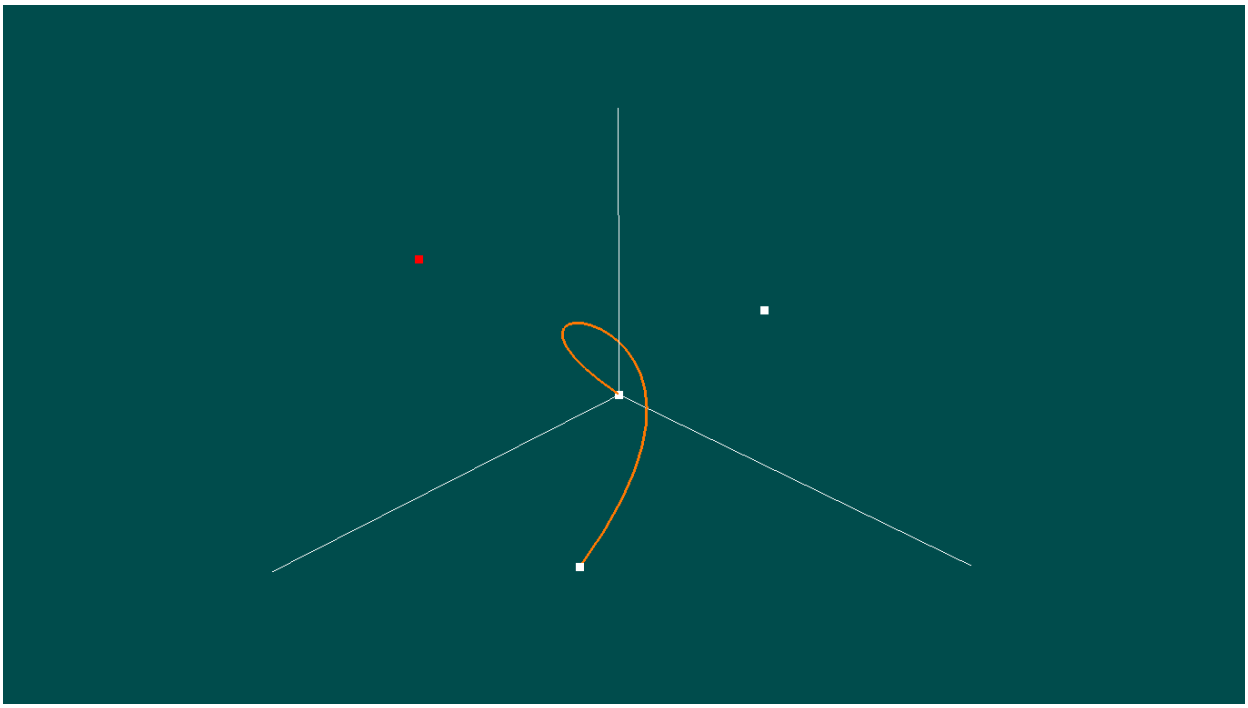


Figura 5.7: Bézier Curve

Además se pasa al TES la posición del vértice dentro del patch mediante la instrucción:

```
gl_out[ gl_InvocationID ].gl_Position =
    gl_in[ gl_InvocationID ].gl_Position;
```

En el TES es donde realmente se realizan los cálculos necesarios para obtener la superficie de Bézier. El siguiente código, que realiza el cálculo de la ecuación (4.6) es una versión simplificada del que se puede encontrar en Bailey and Cunningham [11]. Este último utiliza también las propiedades de las derivadas de las superficies de Bézier para calcular el normal con el fin de utilizar un modelo de luz como el utilizado en 5.5.1.

```
vec4 p00 = gl_in[0].gl_Position;
vec4 p10 = gl_in[1].gl_Position;
vec4 p20 = gl_in[2].gl_Position;
vec4 p30 = gl_in[3].gl_Position;
vec4 p01 = gl_in[4].gl_Position;
vec4 p11 = gl_in[5].gl_Position;
vec4 p21 = gl_in[6].gl_Position;
vec4 p31 = gl_in[7].gl_Position;
vec4 p02 = gl_in[8].gl_Position;
vec4 p12 = gl_in[9].gl_Position;
```

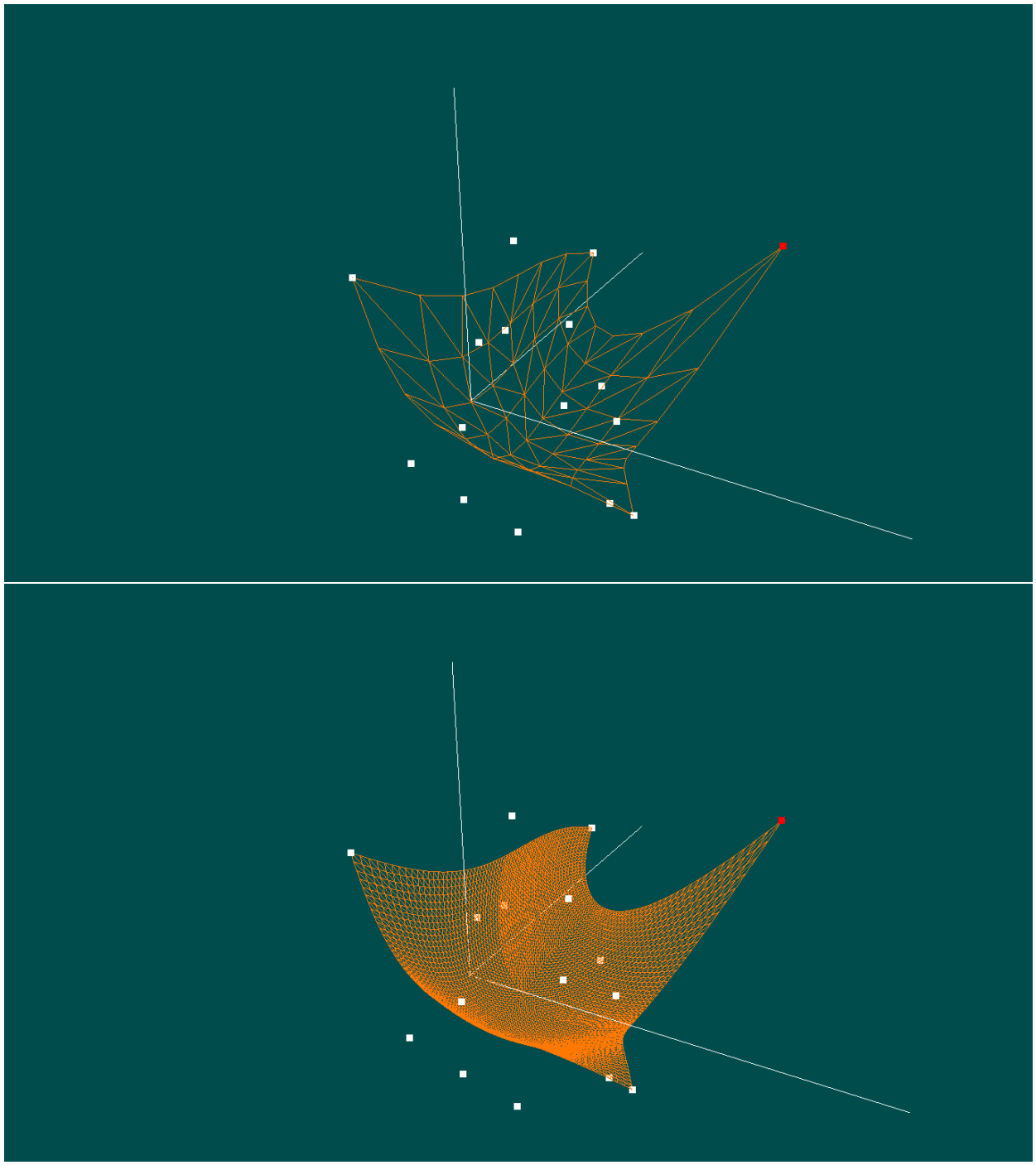


Figura 5.8: Superficie de Bézier

```
vec4 p22 = gl_in[10].gl_Position;
vec4 p32 = gl_in[11].gl_Position;
vec4 p03 = gl_in[12].gl_Position;
vec4 p13 = gl_in[13].gl_Position;
vec4 p23 = gl_in[14].gl_Position;
vec4 p33 = gl_in[15].gl_Position;

float u = gl_TessCoord.x;
float v = gl_TessCoord.y;

float bu0 = (1.-u) * (1.-u) * (1.-u);
float bu1 = 3. * u * (1.-u) * (1.-u);
float bu2 = 3. * u * u * (1.-u);
float bu3 = u * u  * u;

float bv0 = (1.-v) * (1.-v) * (1.-v);
float bv1 = 3. * v * (1.-v) * (1.-v);
float bv2 = 3. * v * v * (1.-v);
float bv3 = v * v * v;

gl_Position =
      bu0 * ( bv0*p00 + bv1*p01 + bv2*p02 + bv3*p03 )
    + bu1 * ( bv0*p10 + bv1*p11 + bv2*p12 + bv3*p13 )
    + bu2 * ( bv0*p20 + bv1*p21 + bv2*p22 + bv3*p23 )
    + bu3 * ( bv0*p30 + bv1*p31 + bv2*p32 + bv3*p33 );
```

El resultado de estos shaders se puede ver en la Figura 5.8.

5.5.4. Sólidos de revolución

En este caso, de nuevo, tanto el vertex shader como el fragment shader son los más simples posible, como en los casos anteriores. Todo el trabajo recae en el geometry shader. Así, dados los vértices que forman una curva, se van generando vértices en torno al eje y, tantos como indica la variable `uNum`. El resultado de revolucionar una curva en torno al eje y puede verse en la Figura 5.9. El código del geometry shader utilizado es el siguiente:

```
for( int i  = 0; i <= uNum; i++){
    float grados = 2.0*PI / uNum * i;

    gl_Position = gl_in[0].gl_Position;
```

```
    gl_Position.x = cos(grados)*gl_PositionIn[0].x
            + sin(grados)*gl_PositionIn[0].z;
    gl_Position.z = cos(grados)*gl_PositionIn[0].z
            - sin(grados)*gl_PositionIn[0].x;

    gl_Position = uProjection * uView * uModel * gl_Position;
    EmitVertex( );

    gl_Position = gl_PositionIn[1];
    gl_Position.x = cos(grados)*gl_PositionIn[1].x
            + sin(grados)*gl_PositionIn[1].z;
    gl_Position.z = cos(grados)*gl_PositionIn[1].z
            - sin(grados)*gl_PositionIn[1].x;

    gl_Position = uProjection * uView * uModel * gl_Position;
    EmitVertex( );
}
```

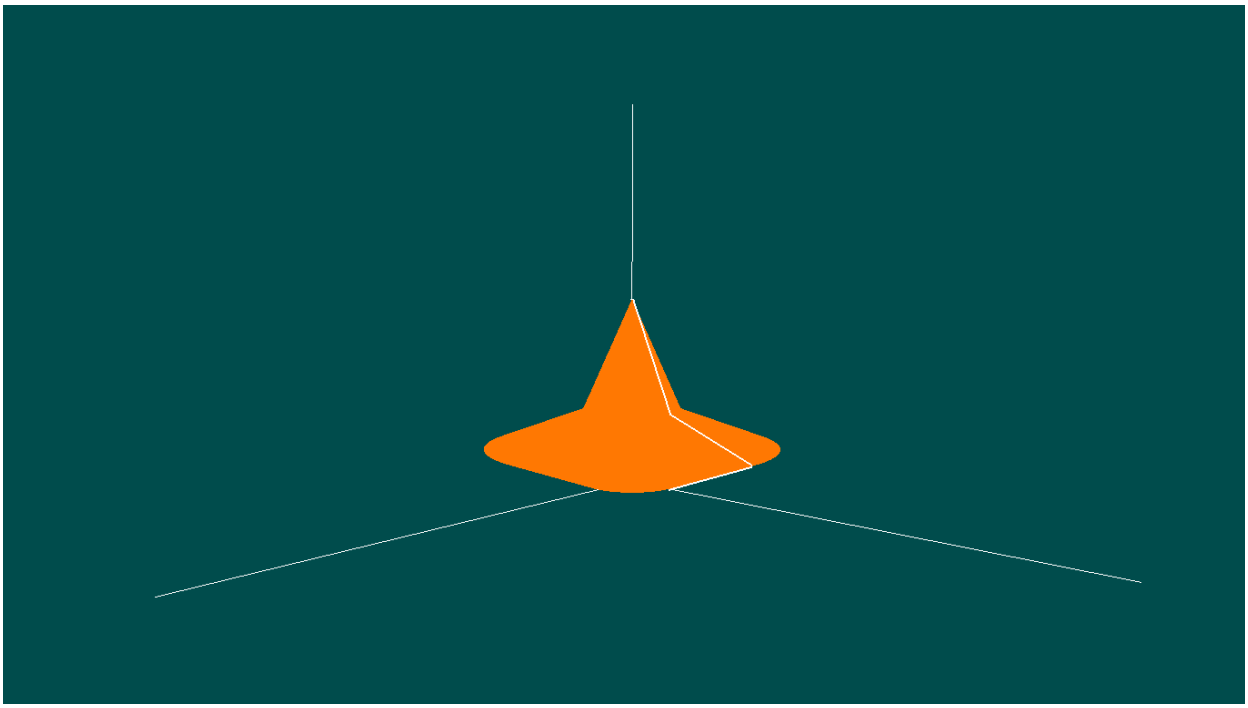


Figura 5.9: Sólido de revolución

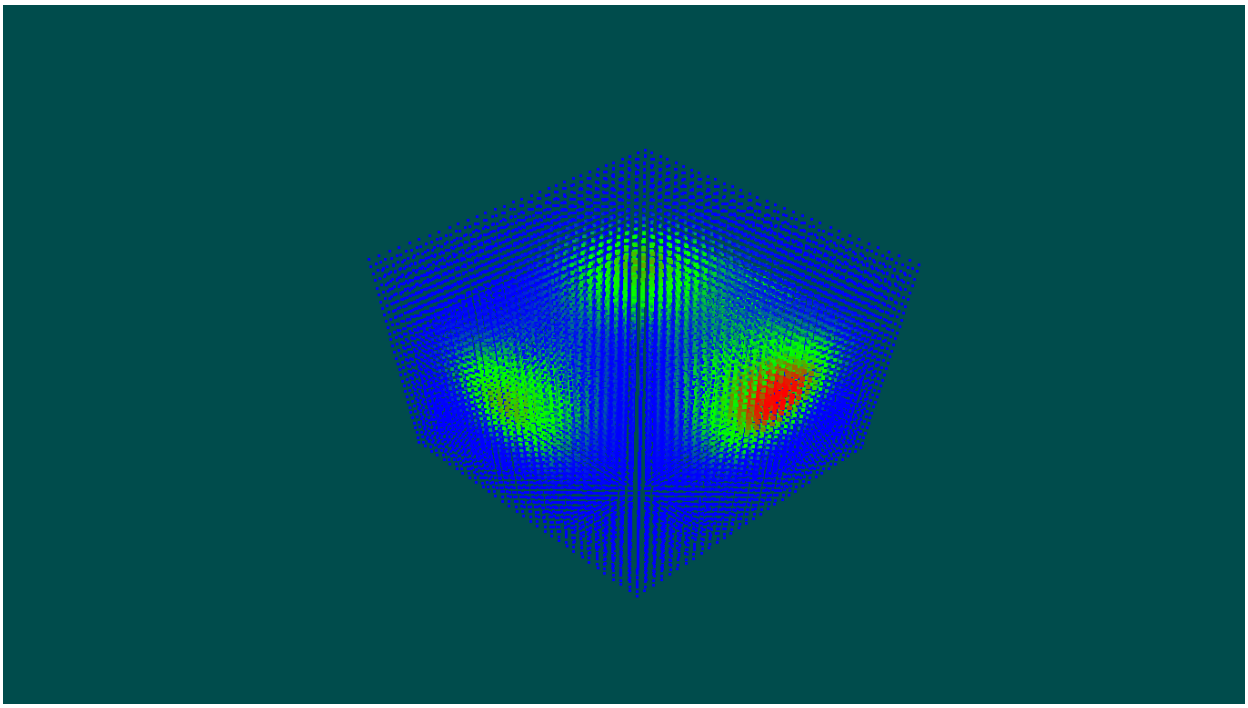


Figura 5.10: Nube de puntos

5.5.5. Nube de puntos

Para realizar esta visualización se han utilizado únicamente un vertex shader y un fragment shader. Como entradas, el vertex shader utiliza tanto la posición del vértice en la nube de puntos como el escalar que representa la información que queremos visualizar.

En el vertex shader, además de definir la posición mediante las transformaciones necesarias, utilizamos la siguiente instrucción

```
gl_PointSize = 2 + pow((smoothstep(0.0, uMaxData, aScalar) + 1), 2);
```

para definir el tamaño del punto acorde al escalar. Así, los puntos que se correspondan con escalares más altos tendrán un tamaño mayor. El valor escalar `aScalar` es pasado también al fragment shader para realizar una coloración del punto acorde a este valor.

Además, en el fragment shader se ha declarado la variable `uniform uMax` que se utiliza para descartar los puntos que tengan valores escalares menores que él. Así, podemos visualizar de una manera más efectiva datos por encima de cierto límite. Este comportamiento puede verse en la Figura 5.11. El código para este shader se incluye a continuación.

```
if(vScalar < uMax) discard;
```

```
float alpha;
float middle = uMaxData / 2;
if(vScalar >= middle){
    alpha = smoothstep(middle, uMaxData, vScalar);
    fFragColor = vec4(mix(GREEN, RED, alpha), 1.0f);
}
else{
    alpha = smoothstep(0.0, middle, vScalar);
    fFragColor = vec4(mix(BLUE, GREEN, alpha), 1.0f);
}
```

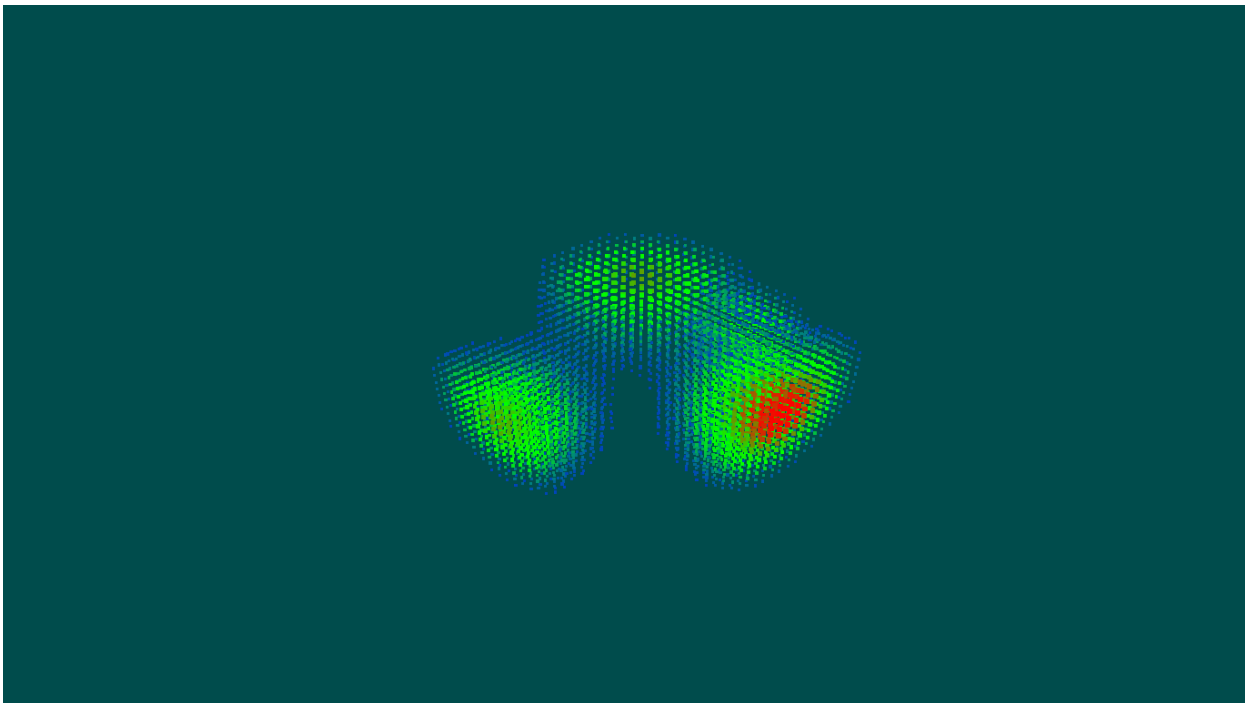
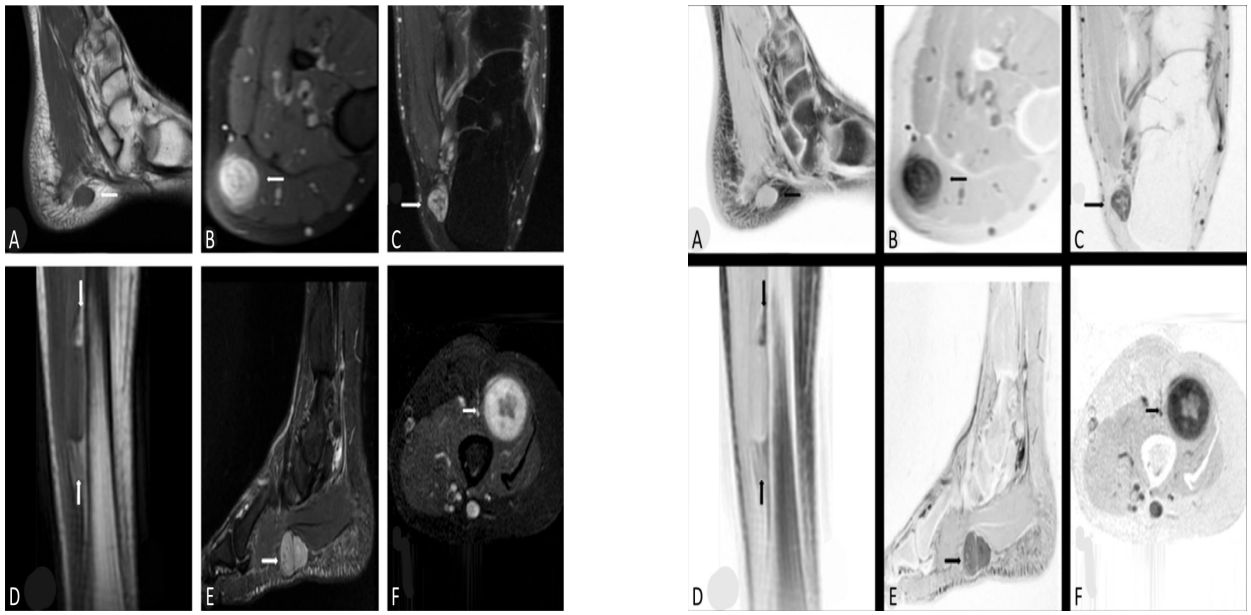


Figura 5.11: Nube de puntos con valores descartados

5.5.6. Negativo de una imagen

En este caso, al tratarse de una imagen, los únicos elementos que necesitamos son cuatro vértices cuya posición se corresponda con cada una de las esquinas de la imagen y la textura para la imagen. Así pues, el vertex shader toma como entrada la posición de estos cuatro vértices junto a las coordenadas que le corresponden en la textura. Estas coordenadas de textura se definen en el intervalo $(0, 1) \times (0, 1)$, siendo el punto $(0, 0)$ la esquina inferior izquierda de la textura y el punto $(1, 1)$ la esquina superior derecha. Con estos datos, el vertex shader escribe la posición de los cuatro vértices y pasa las coordenadas de textura al fragment shader, que es el que las utilizará.



(a) Imagen original (b) Negativo

Figura 5.12: Negativo de una imagen. Fuente: [55]

El fragment shader, por su parte, obtiene el color original de los fragmentos con la instrucción

```
vec3 irgb = texture( texture1, vTexCoords ).rgb;
```

para posteriormente computar el color negativo y escribirlo como salida para el pixel. Esta tarea se realiza con las siguientes instrucciones:

```
vec3 neg = vec3(1.,1.,1.) - irgb;
fFragColor = vec4( mix( irgb, neg, uT ), 1. );
```

Una muestra del funcionamiento de este shader se puede encontrar en la Figura 5.12.

5.5.7. Line Integral Convolution

El caso del Line Integral Convolution presenta un problema a la hora de realizar los cálculos necesarios mediante un shader. Esto se debe a que, como hemos visto anteriormente, necesitamos, para calcular la línea de flujo, los valores del campo vectorial de los demás vértices, algo que no podemos, a priori, obtener en un shader. La solución para este problema consiste en codificar en forma de textura la información del campo vectorial asociado al problema.

Para ello, como se explica en Bailey and Cunningham [11] y como se puede observar en la Figura 5.13, se asocia a cada punto del campo vectorial un color dependiendo de los valores de las componentes vectoriales de dicho punto. A mayor valor de la coordenada x se le asocia un valor más alto del color rojo siguiendo el convenio RGB. Similarmente, a mayor valor de la coordenada y se le asocia un valor mayor del color verde.

Las texturas con estos valores así calculados tienen un aspecto similar al de las Figuras 5.14c y 5.14d, dependiendo del flujo vectorial subyacente.

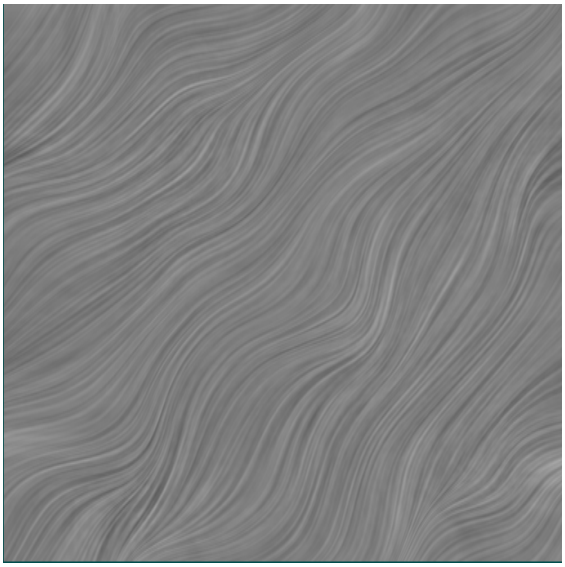
Para esta técnica necesitaremos solo un vertex shader y un fragment shader. El vertex shader toma de nuevo la posición de los vértices, que en este caso se trata de una malla uniforme de 256×256 vértices y las coordenadas de textura de cada uno de ellos. Este shader simplemente pasa las coordenadas de textura al fragment shader y escribe la posición del vértice.

El fragment shader toma las coordenadas de textura procedentes del vertex shader y realiza las operaciones explicadas en la sección 4.6. Primero realiza, para cada vértice, el cómputo de la línea de flujo mediante el método de Runge-Kutta de orden 4. Para este método se define el tamaño del paso mediante la variable `stp` y la longitud de la línea de flujo que queremos calcular mediante la variable `uniform uLength`.

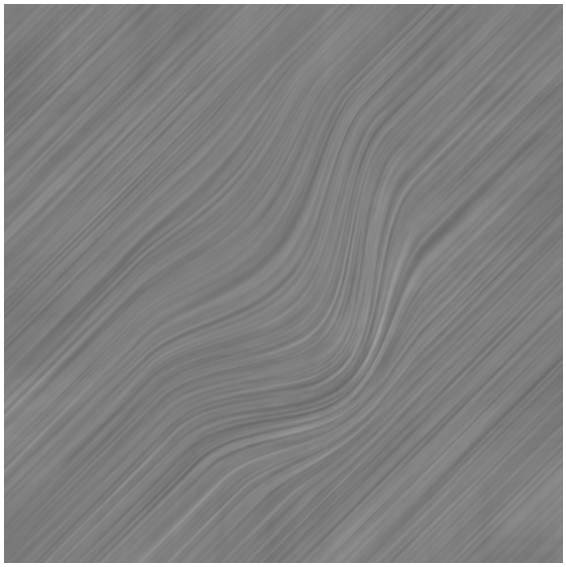
```
vec3 color = texture( texture1, vTexCoords ).rgb;
vec2 st = vTexCoords;
float stp = 0.00281;

// Cálculo de la línea de flujo en dirección positiva
for( int i = 0; i < uLength; i++ )
{
    vec2 k1 = texture( texture2, st ).xy;
    vec2 k2 = texture( texture2, st + k1*stp/2 ).xy;
```

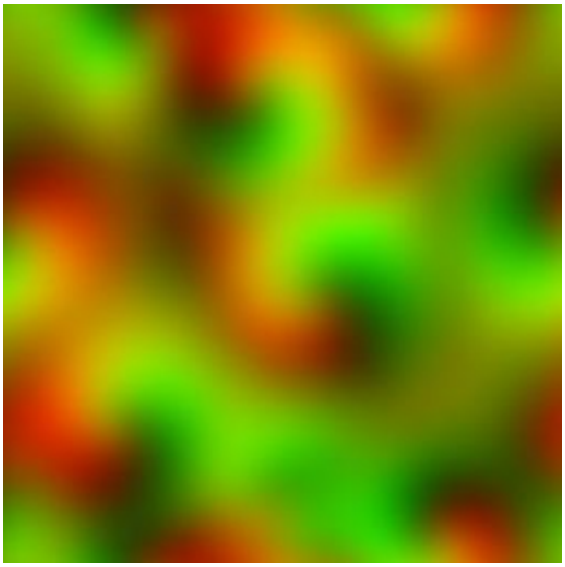
Figura 5.13: LIC - Información del campo vectorial codificado en una textura. Fuente: [11]



(a) Lic para el flujo vectorial de debajo.



(b) Lic para el flujo vectorial de debajo.



(c) Flujo vectorial codificado en textura.

(d) Flujo vectorial codificado en textura.

Figura 5.14: Line Integral Convolution.

```
        vec2 k3 = texture( texture2, st + k2*stp/2 ).xy;
        vec2 k4 = texture( texture2, st + k3*stp).xy;
        vec2 ks = k1 + 2*k2 + 2*k3 + k4;
        st += stp/6*ks;
        st = clamp( st, 0., 1. );
        color += vec3( texture( texture1, st ) );
    }

    // Cálculo de la línea de flujo en dirección negativa
    st = vTexCoords;
    for( int i = 0; i < uLength; i++ )
    {
        vec2 k1 = texture( texture2, st ).xy;
        vec2 k2 = texture( texture2, st - k1 * stp / 2 ).xy;
        vec2 k3 = texture( texture2, st - k2 * stp / 2 ).xy;
        vec2 k4 = texture( texture2, st - stp*k3).xy;
        vec2 ks = k1 + 2*k2 + 2*k3 + k4;
        st -= stp/6*ks;
        st = clamp( st, 0., 1. );
        color += vec3( texture( texture1, st ) );
    }
    color /= float(2*uLength + 1);
```

En este shader se ha omitido el cálculo de los pesos, tomando como valor para ellos 1 en todos los casos y dividiendo por el número de puntos en la línea de flujo. Con todo esto en cuenta, el resultado final se puede observar en la Figura 5.14.

5.5.8. Uso de la aplicación

Una vez visto el diseño de la aplicación y los shaders utilizados, se presenta el modo de utilización de la aplicación. Una vez conseguido el ejecutable, para esta explicación llamado `tfg` se debe ejecutar con alguno de los modos de visualización explicados anteriormente. Cada uno de estos modos permite unas opciones de visualización diferentes, que se explican a continuación.

Además, todos los modos comparten el movimiento de la cámara, que es el siguiente:

- `Esc` - Salir.
- `w` - Mover la cámara hacia delante.

- `s` - Mover la cámara hacia atrás.
- `d` - Mover la cámara hacia la derecha.
- `a` - Mover la cámara hacia la izquierda.

Junto con el movimiento del ratón, que cambia el objetivo al que apunta la cámara, y la ruleta del ratón, que permite hacer y deshacer zoom.

Visualización de Terrenos

```
$ ./tfg terrain
```

Opciones: No tiene opciones especiales.

Curva de Bézier

```
$ ./tfg bezier
```

La aplicación permite aumentar y disminuir el número de vértices de la curva, así como mover los puntos de control. Opciones:

- `\textvisiblespace` - Seleccionar siguiente vértice.
- `\uparrow` - Aumentar número de segmentos.
- `\downarrow` - Disminuir número de segmentos.
- `m` - Cambiar modo cámara/selección.
- `Ratón izquierdo` - Seleccionar vértice. (Solo modo selección).
- `Ratón derecho` - Mover vértice seleccionado. (Solo modo selección).
- `x` - Mover el vértice seleccionado en la dirección $+x$.
- `y` - Mover el vértice seleccionado en la dirección $+y$.
- `z` - Mover el vértice seleccionado en la dirección $+z$.

- `Shift + x` - Mover el vértice seleccionado en la dirección $-x$.
- `Shift + y` - Mover el vértice seleccionado en la dirección $-y$.
- `Shift + z` - Mover el vértice seleccionado en la dirección $-z$.

Superficie de Bézier

```
$ ./tfg bsurface
```

La aplicación permite aumentar y disminuir el grado de teselación de la superficie, así como mover los puntos de control. Opciones:

- `\textvisiblespace` - Seleccionar siguiente vértice.
- `\uparrow` - Aumentar grado de teselación.
- `\downarrow` - Disminuir grado de teselación.
- `t` - Activar el modo wireframe
- `u` - Desactivar el modo wireframe
- `m` - Cambiar modo cámara/selección.
- `Ratón izquierdo` - Seleccionar vértice. (Solo modo selección).
- `Ratón derecho` - Mover vértice seleccionado. (Solo modo selección).
- `x` - Mover el vértice seleccionado en la dirección $+x$.
- `y` - Mover el vértice seleccionado en la dirección $+y$.
- `z` - Mover el vértice seleccionado en la dirección $+z$.
- `Shift + x` - Mover el vértice seleccionado en la dirección $-x$.
- `Shift + y` - Mover el vértice seleccionado en la dirección $-y$.
- `Shift + z` - Mover el vértice seleccionado en la dirección $-z$.

Nube de puntos

```
$ ./tfg cloud
```

Opciones:

- `\uparrow` - Aumentar límite para el escalar.
- `\downarrow` - Disminuir límite para el escalar.

Negativo

```
$ ./tfg negative
```

Opciones: No tiene opciones especiales

Sólido de revolución

```
$ ./tfg revolution
```

Opciones:

- `\uparrow` - Aumentar número de vértices de revolución.
- `\downarrow` - Disminuir número de vértices de revolución.
- `t` - Activar el modo wireframe
- `u` - Desactivar el modo wireframe

Line Integral Convolution

```
$ ./tfg lic
```

Opciones:

- `\uparrow` - Aumentar longitud de la línea de flujo.

- `\downarrow` - Disminuir longitud de la línea de flujo.

Capítulo 6

Conclusiones y Trabajo Futuro

6.0.1. Conclusiones

Con este trabajo he conseguido entender cómo funcionan la informática gráfica y cómo utilizarla para visualización científica. He podido observar como es un campo relativamente nuevo pero creciendo a una gran velocidad y ganando cada vez más importancia.

He podido observar la enorme utilidad que puede llegar a tener la visualización y la gran ayuda que puede suponer para los equipos que trabajan con grandes volúmenes de datos. También he podido ser consciente como equipos científicos enormemente importantes utilizan estas técnicas para realizar descubrimientos pioneros, como es el caso del EHT [56] con la obtención de la primera imagen de un agujero negro [57].

6.0.2. Posible Trabajo Futuro

Como trabajo futuro se plantean varios puntos:

- Seguir aprendiendo sobre el tema y alcanzar un mayor conocimiento que permita entender las últimas tendencias y descubrimientos en este área.
- Conocer más técnicas de visualización e incorporarlas a la aplicación.
- Mejorar la aplicación con la introducción de cuaterniones.
- Mejorar la aplicación con la posibilidad de realizar los tipos de visualización a partir

de modelos y ficheros especificados por el usuario.

- Profundizar en la visualización de fluidos y en los nuevos métodos tridimensionales.

Referencias

[1] Thomas A. Defanti and Maxine D. Brown. *Visualization in Scientific Computing*, volume 33 of *Advances in Computers*. Elsevier, 1991. doi: https://doi.org/10.1016/S0065-2458(08)60168-0. URL http://www.sciencedirect.com/science/article/pii/S0065245808601680.

[2] Matthew W. Rohrer. Seeing is believing: The importance of visualization in manufacturing simulation. In *Proceedings of the 32Nd Conference on Winter Simulation*, WSC '00, pages 1211–1216, San Diego, CA, USA, 2000. Society for Computer Simulation International. ISBN 0-7803-6582-8. URL http://dl.acm.org/citation.cfm?id=510378.510552.

[3] B. H. McCormick, T. A. DeFanti, and M. D. Brown. Visualization in scientific computing. *Computer Graphics*, 20(6), 1987.

[4] The Khronos Group Inc. OpenGL Website. https://www.opengl.org/, . Accessed: 23/07/2019.

[5] Shreiner, Dave, and The Khronos OpenGL ARB Working Group. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Versions 3.0 and 3.1*. Addison-Wesley Professional, 7th edition, 2009. ISBN 0321552628, 9780321552624.

[6] The Khronos Group Inc. About the OpenGL ARB. https://www.opengl.org/archives/about/arb/, . Accessed: 19/07/2019.

[7] The Khronos Group Inc. The Khronos Group Inc. https://www.khronos.org/. Accessed: 18/08/2019.

[8] The Khronos Group Inc. Vulkan Website. https://www.khronos.org/vulkan/, . Accessed: 18/08/2019.

[9] Inc Advanced Micro Devices. AMD Website. https://www.amd.com/es. Accessed: 18/08/2019.

[10] The Khronos Group Inc. SPIR Website. https://www.khronos.org/spir/, . Accessed: 18/08/2019.

[11] M. Bailey and S. Cunningham. *Graphic Shaders, Theory and Practice*, volume 2nd Ed. CRC Press, 2011.

[12] The Khronos Group Inc. Primitivas - OpenGL wiki. https://www.khronos.org/opengl/wiki/Primitive, . Accessed: 26/07/2019.

[13] Hobart and William Smith Colleges. Into 3D with OpenGL. http://math.hws.edu/eck/cs424/notes2013/06_Into_3D_OpenGL.html. Accessed: 20/08/2019.

[14] Song Ho Ahn. OpenGL Projection Matrix. http://www.songho.ca/opengl/gl_projectionmatrix.html. Accessed: 20/08/2019.

[15] Martin Kraus. Pixels Covered by a Triangle. https://en.wikibooks.org/wiki/GLSL_Programming/Rasterization#/media/File:Pixels_covered_by_a_triangle.png. Accessed: 20/08/2019.

[16] Microsoft. Microsoft Website. https://www.microsoft.com/es-es, . Accessed: 23/07/2019.

[17] Bedwyr. Proceso en la pipeline de gráficos. https://es.wikipedia.org/wiki/Direct3D#/media/Archivo:D3D_Pipeline_(es).png. Accessed: 20/08/2019.

[18] The Khronos Group Inc. GLSL Website. https://www.khronos.org/opengl/wiki/Core_Language_(GLSL), . Accessed: 24/07/2019.

[19] Microsoft. HLSL Website. https://docs.microsoft.com/en-us/windows/win32/direct3dhlsl/dx-graphics-hlsl, . Accessed: 18/08/2019.

[20] The Khronos Group Inc. Vertex Shader - OpenGL wiki. https://www.khronos.org/opengl/wiki/Vertex_Shader, . Accessed: 24/07/2019.

[21] The Khronos Group Inc. Tessellation Control Shader - OpenGL wiki. https://www.khronos.org/opengl/wiki/Tessellation_Control_Shader, . Accessed: 24/07/2019.

[22] Alfonse. Tessellation. https://www.khronos.org/opengl/wiki/Tessellation. Accessed: 20/08/2019.

[23] The Khronos Group Inc. Tessellation Primitive Generator - OpenGL wiki. https://www.khronos.org/opengl/wiki/Tessellation#Tessellation_primitive_generation, . Accessed: 26/07/2019.

[24] The Khronos Group Inc. Tessellation Evaluation Shader - OpenGL wiki. https://www.khronos.org/opengl/wiki/Tessellation_Evaluation_Shader, . Accessed:

25/07/2019.

[25] The Khronos Group Inc. Geometry Shader - OpenGL wiki. https://www.khronos.org/opengl/wiki/Geometry_Shader, . Accessed: 26/07/2019.

[26] The Khronos Group Inc. Fragment Shader - OpenGL wiki. https://www.khronos.org/opengl/wiki/Fragment_Shader, . Accessed: 26/07/2019.

[27] Donald Hearn and M. Pauline Baker. *Computer Graphics - C Version*, volume 2nd Ed. Pearson, 1997.

[28] Wojciech mula. Sample Bézier Surface. https://en.wikipedia.org/wiki/B%C3%A9zier_surface#/media/File:B%C3%A9zier_surface_example.svg. Accessed: 20/08/2019.

[29] Boston University. Scientific Visualization Techniques - Boston University. http://www.bu.edu/tech/support/research/training-consulting/online-tutorials/introduction-to-scientific-visualization-tutorial/techniques/. Accessed: 30/07/2019.

[30] WhiteboxDev. Choosing colour-ramp to use for elevation? https://gis.stackexchange.com/questions/25099/choosing-colour-ramp-to-use-for-elevation. Accessed: 20/08/2019.

[31] B. Cabral and L.C. Leedom. Imaging vector fields using line integral convolution. 3 1993.

[32] Alexander Petrov Petkov. Transparent line integral convolution: A new approach for visualizing vector fields in opendx. Master's thesis, University of Montana, 2005.

[33] Canonical Group Ltd (GB). Ubuntu. https://ubuntu.com/. Accessed: 08/08/2019.

[34] Bram Moolenaar et al. Vim - the ubiquitous text editor. https://www.vim.org/. Accessed: 08/08/2019.

[35] Linus Torvalds. git –distributed-even-if-your-workflow-isnt. https://git-scm.com/. Accessed: 08/08/2019.

[36] Inc. GitHub. Github. https://github.com/. Accessed: 08/08/2019.

[37] Inc. Free Software Foundation. GCC, the GNU Compiler Collection. https://gcc.gnu.org/, . Accessed: 08/08/2019.

[38] Inc. Free Software Foundation. GDB: The GNU Project Debugger. https://www.gnu.org/software/gdb/, . Accessed: 08/08/2019.

[39] Inc. Free Software Foundation. GNU Make. https://www.gnu.org/software/make/, . Accessed: 08/08/2019.

[40] Joey de Vries. *Learn OpenGL*, volume 3rd printing. Joey de Vries, 2017.

[41] Camilla Löwy. GLFW - An OpenGL Library. https://www.glfw.org/. Accessed: 08/08/2019.

[42] David Herberth. GLAD. https://glad.dav1d.de/. Accessed: 08/08/2019.

[43] Assimp Team. The Open-Asset-Importer-Lib. http://www.assimp.org/. Accessed: 08/08/2019.

[44] G-Truc Creation. GLM - OpenGL Mathematics. https://glm.g-truc.net/0.9.9/index.html. Accessed: 08/08/2019.

[45] Benjamín Bonell Navarro. Concepto de vector. http://agrega.educacion.es/repositorio/02122013/e7/es_2013120213_9134249/concepto_de_vector.html. Accessed: 20/08/2019.

[46] Wolfram Alpha LLC. Wolfram Alpha. https://www.wolframalpha.com/. Accessed: 20/08/2019.

[47] Etay Meiri. Perspective Projection. http://ogldev.atspace.co.uk/www/tutorial12/tutorial12.html. Accessed: 30/08/2019.

[48] Chua Hock-Chuan. 3D Graphics with OpenGL. https://www.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html. Accessed: 30/08/2019.

[49] Dainis. Vector Rotation. http://tm.spbstu.ru/%D0%A4%D0%B0%D0%B9%D0%BB:Vector_rotation.png. Accessed: 20/08/2019.

[50] John Vince. *Quaternions for Computer Graphics*. Springer Publishing Company, Incorporated, 1st edition, 2011. ISBN 0857297597, 9780857297594.

[51] Olinde Rodrigues. Des lois géometriques qui regissent les déplacements d' un systéme solide dans l' espace, et de la variation des coordonnées provenant de ces déplacement considérées indépendant des causes qui peuvent les produire. *Journal de Mathématiques Pures et Appliquées*, 5:380–440, 1840.

[52] Leandra Vicci. Quaternions and rotations in 3-space: The algebra and its geormetric interpretation. 4 2001.

[53] J. M. Arrieta, R. Ferreira, R. Pardo, and A. Rodríguez. *Análisis Numérico de Ecuaciones Diferenciales Ordinarias*. Universidad Complutense de Madrid, 2018.

[54] Brian Dunbar. NASA - National Aeronautics and Space Administration. https://www.nasa.gov/. Accessed: 16/08/2019.

[55] Cigdem Ozer Gokaslan, Ugur Toprak, Emin Demirel, Cagri Erdim, Aytul Hande Yardimci, and Ceyda Bektas Turan. Schwannomas of uncommon peripheral locations: Analysis of imaging findings of 21 cases. *Current Medical Imaging*, 15(6):578–584, 2019. ISSN 1573-4056/1875-6603. doi: 10.2174/1573405614666181005115631. URL http://www.eurekaselect.com/node/165982/article.

[56] Event Horizon Telescope. EHT - Event Horizon Telescope. https://eventhorizontelescope.org/. Accessed: 16/08/2019.

[57] The EHT Collaboration et al. First m87 event horizon telescope results. iv. imaging the central supermassive black hole. *ApJL*, 875:4, 2019. URL https://iopscience.iop.org/article/10.3847/2041-8213/ab0e85.

[58] The Khronos Group Inc. OpenGL Reference Pages. https://www.khronos.org/registry/OpenGL-Refpages/gl4/index.php. Accessed: 19/08/2019.

[59] A. Spitzbart. A generalization of hermite's interpolation formula. *The American Mathematical Monthly*, 67(1):42–46, 1960. ISSN 00029890, 19300972. URL http://www.jstor.org/stable/2308924.

[60] Encyclopedia of Mathematics. Linear interpolation. http://www.encyclopediaofmath.org/index.php?title=Linear_interpolation&oldid=16431. Accessed: 19/08/2019.

Apéndice A

El lenguaje GLSL

En este apéndice se tratan algunas de las características del lenguaje GLSL, sobre todo aquellas necesarias para la implementación de los shaders vistos en la sección 5.5. Para una guía completa sobre el lenguaje se puede consultar la documentación detallada [58].

A.1. Versión

Un programa GLSL ha de comenzar obligatoriamente con una declaración de la versión de GLSL que se va a utilizar. Esto se realiza mediante una directiva que se utiliza para saber la versión que se ha de utilizar para compilar el programa. Esta directiva se declara del siguiente modo:

```
#version 400
```

En este caso se ha de utilizar la versión GLSL 4.0.

A.2. Funciones y Estructuras de Control

Como estructuras de control, GLSL soporta bucles y saltos, con las instrucciones comunes del lenguaje C (if-else, switch, while, for...). Sin embargo, la recursión en este lenguaje no está permitida.

En cuanto a las funciones, este lenguaje soporta tanto funciones definidas por el usuario

como funciones proporcionadas por el propio lenguaje. Una lista de las funciones incluidas puede encontrarse en Inc. [58]. De entre estas funciones, se han utilizado las siguientes en los shaders desarrollados:

- `smoothstep` - Realiza una interpolación de Hermite entre dos valores [59].
- `mix` - Realiza una interpolación lineal entre dos valores [60].
- `normalize` - Calcula el vector unitario correspondiente a la dirección del vector original.
- `max` - Devuelve el máximo entre dos valores.
- `dot` - Calcula el producto escalar entre dos vectores.
- `reflect` - Calcula la dirección de reflexión del vector incidente.
- `pow` - Devuelve la potencia del primer parámetro elevado al segundo parámetro.
- `EmitVertex` - Explicado en la sección 3.3.
- `sin` - Devuelve el seno del parámetro.
- `cos` - Devuelve el coseno del parámetro.
- `texture` - Devuelve texels de una textura.
- `clamp` - Restringir un valor para que se encuentre entre otros dos valores.
- `discard` - Descarta el fragmento actual (Solo para el fragment shader).

A.3. Tipos de Datos

GLSL define una serie de tipos de datos. Algunos de estos son compartidos con los lenguajes C y C++, mientras que otros son completamente nuevos. En nuestro caso, los utilizados son los siguientes:

A.3.1. Escalares

- `bool` - Condicional. Valores `true` o `false`.
- `int` - Entero con signo de 32 bits.
- `float` - Número de coma flotante.

A.3.2. Vectoriales

GLSL soporta tipos de datos vectoriales de n componentes, con n siendo $2, 3$ o 4. Los tipos vectoriales soportan las mismas operaciones que los escalares. Estas operaciones se realizan componente a componente. Sin embargo, estas operaciones funcionarán solo si los dos vectores tienen el mismo número de componentes.

- `vec`n - Un vector de n componentes tipo `float`.

A.3.3. Swizzling

El swizzling consiste en la posibilidad que ofrece GLSL de acceder a las componentes de un vector de la siguiente manera:

```
vec4 someVec;
someVec.x + someVec.y;
```

Se pueden utilizar `x`, `y`, `z` y `w` para acceder a las componentes primera, segunda, tercera y cuarta respectivamente. Además se puede construir un nuevo vector a partir de otro especificando el orden de las coordenadas del vector antiguo en el nuevo. Con todo esto, algunos ejemplos más ilustrativos del swizzling son los siguientes:

```
vec2 someVec;
vec4 otherVec = someVec.xyxx;
vec3 thirdVec = otherVec.zyy;
vec4 someVec;
someVec.wzyx = vec4(1.0, 2.0, 3.0, 4.0); // Da la vuelta al vector.
someVec.zx = vec2(3.0, 5.0); // Pone la componente z a 3 y la x a 5.
```

Además, para facilitar el acceso a un vector que represente un color o a uno que re-

presente coordenadas de una textura, se pueden utilizar, juntos a las letras anteriores, las siguientes combinaciones:

- `rgba`
- `stpq`

Estas combinaciones no suponen ninguna diferencia con la principal, son simplemente azúcar sintáctico para ayudar al programador. Sin embargo, no se pueden realizar combinaciones que contengan letras de distintos tipos. Por ejemplo, la combinación:

```
vec3 aVec = someVec.xbg;
```

resultaría en un error de compilación.

A.3.4. Matrices

También soporta tipos matriciales. Para declarar una matriz se utiliza la siguiente sintaxis:

- `mat`n`x`m - Matriz $n \times m$
- `mat`n - Matriz $n \times n$

donde n puede ser $2, 3$ o 4. En los tipos matriciales no se puede utilizar swizzling, por lo que han de ser accedidos mediante una sintaxis de array propia de C o C++.

A.3.5. Uniforms

Los `uniform` son un tipo de variable de shader. Se utilizan como parámetros que el programador de un shader puede pasar al programa principal. Se llaman de esta manera porque su valor no cambia de una invocación de shader a otra durante la misma llamada de renderizado.

Estas variables han de ser declaradas de un modo global dentro del shader y pueden ser de cualquiera de los tipos permitidos en GALS. Son variables de solo lectura dentro del shader, y cualquier intento de cambio de su valor resultará en un error de compilación. Están pensados para utilizarse en el programa desde OpenGL y no desde el shader.

Para utilizar una variable `uniform` desde la aplicación, se ha de recabar la localización del shader mediante una llamada a la función `glGetUniformLocation`.

A.4. Variables y Entrada/Salida

Los shaders han de ser capaces de comunicarse entre si dentro del pipeline. Las entradas y salidas de cada uno de los tipos de shader se han explicado en la sección 3. En esta sección se expone la sintaxis necesarias para declarar estas variables de entrada y salida desde cada tipo de shader.

Asimismo, existe una serie de variables específicas construidas dentro del lenguaje GLSL que tienen un significado ya definido.

A.4.1. Entradas y salidas

De manera general, una variable de entrada a un shader ha de llevar el modificador `in`. De igual modo, una variable de salida dentro de un shader ha de llevar el modificador `out`. Por tanto, si queremos pasar un valor desde el vertex shader al fragment shader (suponiendo que no se utilizan tessellation ni geometry shaders) declararíamos lo siguiente en el vertex shader:

```
out vec3 vPos;
```

Y, de manera correspondiente, lo siguiente en el fragment shader:

```
in vec3 vPos;
```

Un caso particular proviene de las entradas del vertex shader. Al ser el primero en el pipeline, las entradas a este shader provienen de los datos contenidos en un VBO. Estos datos corresponden a los diferentes atributos de un vértice en particular, y están referenciados en un VAO. Así, el primer atributo de un vértice, especificado en el VAO, puede ser la posición, mientras que el segundo puede ser el color. Para pasar esta información al shader se declaran las siguientes líneas dentro del shader:

```
layout(location = 1) in vec3 aPos;
layout(location = 2) in vec3 aColor;
```

Esto quiere decir al shader que en el primer atributo ha de encontrar un tipo `vec3` que

representa la posición y en el segundo atributo ha de encontrar otro `vec3` que representa el color del vértice correspondiente.

De igual modo se pueden especificar salidas desde la última etapa del pipeline, el fragment shader.

Además de estos dos casos particulares, se dan los de aquellos shaders que necesitan instrucciones específicas para realizar su función. Son los siguientes:

TCS

Para utilizar el shader de teselación, se ha de habilitar una extensión ARB, que se realiza mediante la siguiente directiva.

```
#extension GL_ARB_tessellation_shader : enable
```

Además el TCS, acorde a lo expuesto en la seccion 3.2.1, ha de especificar el número de vértices en el parche de salida. Esto no se realiza mediante variables específicas, sino mediante la siguiente instrucción:

```
layout( vertices = 16 ) out;
```

TES

En este caso, al igual que en el TCS, se ha de habilitar la extensión ARB que permite la utilización de tessellation shaders, por lo que se ha de incluir la misma línea.

Además, en este caso se ha de especificar lo explicado en la sección 3.2.3 (espaciado, primitiva, orden), que se realiza mediante la siguiente instrucción:

```
layout( quads, equal_spacing, ccw ) in;
```

Geometry shader

Para habilitar la utilización de este shader se ha de habilitar otra extensión ARB:

```
#extension GL_EXT_geometry_shader4: enable
```

Además se ha de definir tanto la primitiva de entrada como la de salida y el número máximo de vértices que se pueden generar (Sección 3.3). Para ello hay que utilizar las siguientes líneas:

```
layout( lines_adjacency ) in;
layout( line_strip, max_vertices=256 ) out;
```

A.4.2. Variables Específicas

Además de las variables definidas por el usuario y de las instrucciones específicas necesarias anteriores, GLSL tiene algunas variables ya definidas que se utilizan para propósitos específicos, aunque no es necesario utilizarlas. Estas variables pueden ser específicas para cada tipo de shader o comunes a todos ellos. Una lista de estas variables puede encontrarse en Inc. [58]. Aquí se muestran las utilizadas en la aplicación.

Vertex Shader

- `out vec4 gl_Position` - La posición del vértice actual en el clip space.
- `out float gl_PointSize` - El tamaño del punto siendo rasterizado. Solo funciona con la primitiva `GL_POINTS`.

TCS

- `in int gl_InvocationID` - El índice de la invocación del TCS dentro del patch. El TCS escribe a las variables de salida vértice a vértice utilizando esta variable para indexarlos.
- `patch out float gl_TessLevelOuter[4]` - Niveles externos de teselación.
- `patch out float gl_TessLevelInner[2]` - Niveles internos de teselación.
- `gl_in[gl_MaxPatchVertices]` - Información de cada vértice proveniente del vertex shader.

TES

- `in vec3 gl_TessCoord` - La localización dentro del abstract patch para cada vértice particular.
- `gl_in[gl_MaxPatchVertices]` - Información de cada vértice proveniente del TCS.

Geometry Shader

- `gl_Position` - La posición del vértice actual dentro del clip space.