



UNIVERSIDAD NACIONAL DEL LITORAL  
FACULTAD DE INGENIERÍA Y CIENCIAS HÍDRICAS

---

## Desarrollo de algoritmos de teselación adaptativa en GPU

---

Proyecto final de carrera  
Ingeniería en Informática

Alumno: Fernando Nellmedin

Director: Dr. Néstor Calvo

Santa Fe, 12 de Septiembre de 2014

# Índice general

<b>1. Introducción</b>	<b>1</b>
1.1. Motivación . . . . .	2
1.2. Objetivos . . . . .	2
1.3. Metodología . . . . .	3
<b>2. Teselación</b>	<b>4</b>
2.1. Estado del arte . . . . .	4
2.2. Nuevo pipeline gráfico . . . . .	5
2.3. Selección de API para realizar el proyecto . . . . .	7
2.4. Programación en GPU . . . . .	7
2.5. Teselación con OpenGL . . . . .	12
2.5.1. Niveles de teselación en triángulos . . . . .	16
2.5.2. Niveles de teselación en cuadriláteros . . . . .	19
2.5.3. Niveles de teselación en poligonales . . . . .	22
2.5.4. Posicionado de los nuevos puntos . . . . .	26
2.6. Ejemplos de shaders de teselación . . . . .	29
<b>3. Aplicaciones</b>	<b>33</b>
<b>4. Métricas</b>	<b>40</b>

<b>ÍNDICE GENERAL</b>	<b>II</b>
<b>4.1. Métricas implementadas . . . . .</b>	<b>43</b>
<b>4.1.1. Distancia a la cámara . . . . .</b>	<b>43</b>
<b>4.1.2. Curvatura . . . . .</b>	<b>45</b>
<b>4.1.3. Longitud de arista . . . . .</b>	<b>50</b>
<b>4.1.4. Posición en silueta . . . . .</b>	<b>53</b>
<b>4.2. Métricas descartadas . . . . .</b>	<b>56</b>
<b>4.2.1. Curvatura máxima como diferencia de normales . . . . .</b>	<b>56</b>
<b>4.2.2. Distancia entre modelos . . . . .</b>	<b>57</b>
<b>4.2.3. Propiedades físicas en movimiento de fluidos . . . . .</b>	<b>58</b>
<b>5. Implementación . . . . .</b>	<b>60</b>
<b>5.1. Herramientas y bibliotecas utilizadas . . . . .</b>	<b>60</b>
<b>5.2. Implementación del software final . . . . .</b>	<b>61</b>
<b>5.2.1. Estructuras de datos . . . . .</b>	<b>61</b>
<b>5.2.2. Diagrama de clases . . . . .</b>	<b>63</b>
<b>5.2.3. Funcionalidades . . . . .</b>	<b>64</b>
<b>5.3. Aplicaciones . . . . .</b>	<b>65</b>
<b>5.3.1. Aplicación de muestra de niveles de teselación . . . . .</b>	<b>65</b>
<b>5.3.2. Curva de Bézier de tercer grado . . . . .</b>	<b>68</b>
<b>5.3.3. Superficie de Bézier de tercer grado . . . . .</b>	<b>69</b>
<b>5.3.4. Campo vectorial . . . . .</b>	<b>71</b>
<b>5.3.5. Modelo de personaje . . . . .</b>	<b>73</b>
<b>5.3.6. Terreno . . . . .</b>	<b>77</b>
<b>6. Resultados . . . . .</b>	<b>82</b>
<b>7. Conclusiones . . . . .</b>	<b>86</b>

<i>ÍNDICE GENERAL</i>	III
7.1. Trabajo futuro . . . . .	86
7.1.1. Teselación en ingeniería . . . . .	87
7.1.2. Teselación en superficies de subdivisión . . . . .	87
<b>Apéndice: Propuesta de proyecto</b>	<b>90</b>

# Índice de figuras

2.1.	Comparación del pipeline gráfico de Direct3D 10 con el de Direct3D 11 y el pipeline de OpenGL 4.0. . . . .	6
2.2.	Ilustración del proceso de compilación de un programa de shader. . . . .	8
2.3.	Distintas interpretaciones para patches de 4 vértices. . . . .	13
2.4.	Ilustración de los tipos de subdivisión según el parámetro del TES. . . . .	14
2.5.	Generación de primitivas a partir de un patch. . . . .	15
2.6.	Parametrización de dominio para <code>triangles</code> . . . . .	16
2.7.	Distintos ejemplos de teselación en el modo <code>triangles</code> . . . . .	18
2.8.	Parametrización de dominio para <code>quads</code> . . . . .	19
2.9.	Distintos ejemplos de teselación en el modo <code>quads</code> . . . . .	21
2.10.	Parametrización de dominio para <code>isolines</code> . . . . .	22
2.11.	Ilustración de las poligonales generadas en <code>isolines</code> para cada valor del parámetro $v$ . . . . .	23
2.12.	Distintos ejemplos de teselación en el modo <code>isolines</code> . . . . .	25
2.13.	Diferencias entre los modos de espaciado. . . . .	28
3.1.	Ejemplo de dos aproximaciones distintas para renderizar un círculo. . . . .	34
3.2.	Ejemplo de distintos niveles de detalle para un modelo. . . . .	34
3.3.	Ejemplo de renderizado de un terreno. . . . .	36

3.4. Ejemplo de modificación de geometría utilizando teselación y un mapa de desplazamientos. . . . .	37
3.5. Ejemplo de visualización de campo vectorial con líneas de corriente. . . . .	38
4.1. Ilustración de cómo la corrección diádica reduce las uniones en T. . . . .	42
4.2. Comparación entre función lineal y función lineal inversa. . . . .	45
4.3. Ejemplos de patches de Bézier. . . . .	47
4.4. Ilustración de la noción de flecha de un arco circular y la aproximación de la flecha en una curva. . . . .	48
4.5. Ilustración del cálculo de la curvatura para curvas de Bézier de segundo grado. . . . .	49
4.6. Ilustración del cálculo de la curvatura diagonal. . . . .	50
4.7. Proyección de una arista como diámetro de esfera. . . . .	51
4.8. Pasos realizados para calcular la máxima longitud proyectada de una arista. . . . .	52
4.9. Ilustración de un patch que representa un triángulo y su vecindad. . . . .	54
4.10. Ejemplo de silueta. . . . .	55
4.11. Mapeo de $\hat{u}_i$ a $t_i$ en el interior de un triángulo. . . . .	56
4.12. Algoritmo propuesto para la métrica de intersecciones. . . . .	58
5.1. Diagrama de clases del software. . . . .	64
5.2. Capturas del software final. . . . .	66
5.2. Capturas del software final. . . . .	67
5.3. Capturas de la aplicación de muestra de niveles de teselación y métodos de espaciado. . . . .	68
5.4. Capturas de la métrica de curvatura para curvas de Bézier de tercer grado. . . . .	69
5.5. Capturas de la métrica de curvatura para superficies de Bézier de tercer grado. . . . .	70

## ÍNDICE DE FIGURAS

VI

5.6. Capturas de la métrica de distancia para superficies de Bézier de tercer grado. . . . .	71
5.7. Pipeline completo utilizado en el renderizado del campo vectorial. . . . .	73
5.8. Ilustración de la métrica de distancia aplicada a campos vectoriales. . . . .	74
5.9. Ilustración del cálculo de la posición en Phong Tessellation. . . . .	75
5.10. Ejemplo de teselación en silueta. . . . .	76
5.11. Muestra de Phong Tessellation para un octaedro con distintos grados de desplazamiento. . . . .	77
5.12. Aplicación de la métrica de distancia a un gran terreno. . . . .	79
5.13. Aplicación de la métrica de longitud proyectada a un terreno. . . . .	80
6.1. Relación entre FPS y nivel máximo de teselación para algunas de las aplicaciones desarrolladas. . . . .	84

## **Resumen**

En este proyecto se analizan los métodos de trabajo con la nueva tecnología de teselación en la unidad de procesamiento gráfico (GPU). El objetivo primordial consiste en investigar sus potencialidades y posible aprovechamiento académico en desarrollo y en investigación. Esta tecnología, lanzada al mercado en 2009, permite la generación y modificación dinámica de polígonos durante el flujo de trabajo en paralelo en la GPU. Al tener la posibilidad de generar nueva geometría a partir de la inicial, esta tecnología permite crear una escena compleja en la memoria de video a partir de una escena simple cargada en la memoria principal, o refinar una escena simple para obtener más detalle. En este proyecto se presentan métodos para utilizar esta tecnología y aplicaciones donde se emplea. El núcleo del proyecto lo conforma la definición de métricas específicas para decidir localmente si se debe crear nueva geometría y el nivel de refinamiento de la misma. Al finalizar, se realiza una evaluación sobre la utilidad y comportamiento de la tecnología.

# Capítulo 1

## Introducción

El crecimiento sostenido de la industria del hardware y del software ha dado a los especialistas en computación gráfica la oportunidad de construir mundos virtuales más grandes y densos de lo que se creía posible una década atrás. Además, esto se suma al aumento de la demanda de escenas más complejas por parte de los usuarios de estas mismas tecnologías. Es así como, en áreas como el cine, los videojuegos y la visualización de grandes volúmenes de datos científico-técnicos, la computación gráfica es uno de los principales pilares de los cuales depende la calidad del producto final.

Las GPU son dispositivos de hardware esenciales para poder generar mundos complejos. La responsabilidad principal de estos dispositivos es la de procesar geometría a través del *graphics pipeline* (pipeline gráfico). Éste está conformado por una serie de etapas por las que pasan los vértices y polígonos durante el proceso de renderizado. Un subconjunto de estas etapas son las que se denominan “programables”, porque le dan la libertad al programador de añadir o modificar las funcionalidades del pipeline. La arquitectura especial de las GPU permite que un gran volumen de datos se procese a mucha mayor velocidad que en el procesador central (CPU). Esto es debido a que la GPU tiene una gran cantidad de núcleos o procesadores individuales. Cada uno de estos núcleos ejecuta, en hilos de procesamiento separados, el pipeline completo.

La tecnología de teselación en GPU fue presentada por primera vez en 2008 [12] como respuesta a la demanda general ya mencionada. Esta tecnología añade nuevas etapas programables de teselación al pipeline gráfico y brinda la posibilidad de generar gran volumen de geometría nueva completamente en GPU. En el presente proyecto se busca realizar teselación en forma adaptativa. Al hablar de teselación adaptativa, se hace referencia a la capacidad que brindan las etapas de teselación de decidir la cantidad y forma de generar nueva geometría. Se debe destacar que esta decisión se hace en cada frame y en cada primitiva o elemento básico de la geometría, ‘adaptando’ el nivel de complejidad a los requerimientos. Además, estos requerimientos son locales tanto en el espacio como en el tiempo.

Por lo tanto, al realizarse enteramente en GPU, la teselación adaptativa presenta ventajas con respecto a métodos de teselación en CPU. Por un lado, es mucho más conveniente en el factor almacenamiento y transferencia de datos, ya que se decide, se genera, se usa y se descarta sin requerir (ni permitir) almacenamiento de la estructura refinada. Además, debido al paralelismo masivo que presenta la arquitectura de las GPU, este procesamiento se realiza en grandes bloques de datos a la vez, siendo mucho más rápido que cualquier implementación convencional en CPU.

## 1.1. Motivación

Las nuevas funcionalidades incorporadas en el hardware gráfico surgen como respuesta a la utilización masiva de determinados algoritmos en la industria, más notablemente en la del entretenimiento. El avance sostenido hace que aún para los especialistas en computación gráfica resulte difícil mantenerse al tanto de las especificaciones y alcances de los nuevos métodos disponibles.

En este proyecto se investigan los métodos, alcances y posibilidades de la teselación adaptativa en hardware y se genera un digesto ilustrativo para ser utilizado tanto en la cátedra de computación gráfica como en investigación y desarrollo.

## 1.2. Objetivos

### Objetivo general

El objetivo general de este trabajo consiste en investigar y utilizar la tecnología de teselación. A partir de la investigación, se pretende diseñar y desarrollar métricas de teselación que definan el nivel de refinamiento que se le debe dar a cada polígono de la escena. Estas métricas deberán ser implementadas como parte de algunas etapas del pipeline gráfico.

### Objetivos específicos

Los objetivos específicos propuestos en el proyecto son:

- Investigar la tecnología de teselación en placas aceleradoras de video.
- Analizar, desarrollar e implementar métricas para la utilización de la tecnología de teselación.
- Desarrollar e implementar modelos-ejemplo que utilicen la tecnología de teselación.

- Adquirir experiencia con el trabajo de mallas.
- Colaborar con la comunidad universitaria local al presentar tecnologías innovadoras del área de computación gráfica.

### 1.3. Metodología

La metodología seguida en el desarrollo del proyecto fue la metodología de trabajo del modelo en espiral. Las actividades de este modelo se conforman en una espiral donde cada bucle representa un conjunto de actividades que se repiten en cada iteración. El proyecto comenzó con una investigación del estado del arte en lo referente a computación gráfica y teselación. A continuación se seleccionó un conjunto de aplicaciones en las que utilizar teselación. Para cada una de estas aplicaciones se diseñaron una o más métricas que calculen el nivel de refinamiento que debe aplicarse a cada polígono.

El diseño, desarrollo e implementación de métricas a modelos se repartió en varios bucles del ciclo en espiral. En cada iteración se desarrollaron 2 o 3 de estas aplicaciones según el caso. Al terminar, se obtuvieron un total de 8 aplicaciones de muestra de teselación.

Se eligió esta metodología porque permite definir objetivos a corto plazo, en cada bucle de la espiral. Debido a que al iniciar el proyecto no se tuvo certeza de lo que se podía realizar con la tecnología, no se consideró conveniente utilizar una metodología que requiera de la definición temprana de todos los requerimientos, como por ejemplo el modelo en cascada.

## Resumen

El resto del informe se organiza de la siguiente manera. Se comienza con la exposición del nuevo pipeline gráfico. Se hace énfasis en las nuevas etapas programables y la capacidad de configuración que se dispone. Luego se realiza un breve resumen sobre las posibilidades de utilización de teselación en distintas áreas, tanto técnicas como de entretenimiento. A continuación se presentan las distintas métricas que fueron diseñadas, cuya función es calcular la cantidad de geometría nueva a generar. Una vez terminada la discusión sobre métricas, se detalla el software específico desarrollado en el proyecto. Este software es un compendio de ejemplos de distintos usos de teselación en GPU. En dicho software se utilizan las métricas para poder generar geometría nueva en tiempo real y adaptativamente. Hacia el final del documento, se presenta un capítulo de resultados que resume las características del producto obtenido y su utilidad. Por último, se presenta una breve conclusión y un comentario sobre el posible trabajo a realizar a futuro utilizando esta tecnología.

## Capítulo 2

# Teselación

En la generación de modelos computacionales se utiliza una representación discretizada de los objetos. Esta representación está compuesta por gran cantidad de primitivas geométricas, tales como triángulos, puntos y líneas. El término teselación, al hablar de polígonos en computación gráfica, hace referencia a la generación de varios polígonos más pequeños a partir de los iniciales, los cuales reemplazan a los originales. Por ejemplo, al teselar un triángulo, éste se subdivide en un conjunto de triángulos más pequeños.

El uso de teselación en computación gráfica no es reciente. La novedad consiste en su implementación volátil o descartable, en paralelo y en la GPU. En la primer etapa del proyecto se hizo un relevamiento sobre el estado del arte en esta temática. Si bien a partir del lanzamiento de la API Direct3D 11 fue cuando la tecnología de teselación estuvo disponible directamente como parte del pipeline gráfico, previamente los desarrolladores encontraron alternativas para realizar teselación de polígonos. La misma presión de estos desarrolladores por utilizar teselación fue la que llevó a los diseñadores de hardware a incorporarla como parte del flujo normal de trabajo de una GPU.

### 2.1. Estado del arte

En esta sección se presenta un breve resumen del estado del arte en teselación de polígonos. Esta investigación fue realizada al inicio del proyecto para tomar contacto con la tecnología; y poder definir y delimitar el contenido del proyecto final.

En el trabajo más antiguo consultado, Moule y McCool [14] propusieron la implementación de un dispositivo hardware que se encargaría de decidir si un triángulo debería ser subdividido. En el año de la publicación de este trabajo (2002), la capacidad de programar en la GPU era muy limitada, lo que llevó

a proponer soluciones de hardware que se encargarían de una única función: realizar teselación en triángulos.

Algunos años más tarde, los trabajos de Schwarz, Staginski y Stammerger [18], Dyken, Reimers y Seland [9] [10], y Boubekeur y Schlick [4] mostraron cuál sería la tendencia en teselación adaptativa en GPU. En todos estos trabajos, los autores propusieron el almacenamiento en la GPU de patrones de subdivisión para las primitivas. De esta manera, a la GPU sólo se le ingresaban los puntos de control de, por ejemplo, una superficie de Bézier, y en una etapa de la placa gráfica se realizaba el muestreo de la superficie. Para definir la densidad del muestreo, se utilizaban distintas métricas según la aplicación y el resultado deseado.

En otra línea de trabajo, Patney y Ownes [16], junto a Schwarz y Stammerger [19] propusieron realizar teselación en paralelo utilizando la GPU como máquina de procesamiento de propósito general (GPGPU) y programando con CUDA [5]. Debido a la gran carga de procesamiento que requiere subdividir los elementos, el uso de un esquema de programación en paralelo es de gran interés para lograr buenos resultados de teselación en tiempo real.

## 2.2. Nuevo pipeline gráfico

Para trabajar con cualquier hardware, es necesario comunicarse con éste y definir su comportamiento. Para ayudar a esta tarea existen las API (Interfaz de Programación de Aplicaciones, por sus siglas en inglés: *Application Programming Interface*). Una API es un conjunto de bibliotecas software que facilitan la comunicación entre dos elementos software o hardware. Por ejemplo, para guardar un archivo de texto en un disco duro, el usuario utiliza una API que convierte los datos de texto en señales a almacenar en el hardware del disco. En el caso de las placas aceleradoras de video, la utilización de una API es fundamental para poder definir el comportamiento de las distintas etapas del pipeline gráfico.

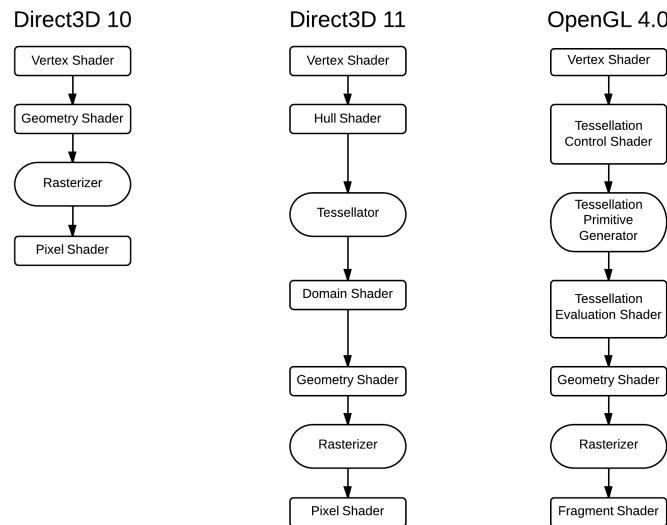
Como se dijo anteriormente, el pipeline gráfico es el conjunto de etapas por las que transita la geometría en la GPU [1]. Por ejemplo, utilizando una API gráfica, el programador puede definir qué color o qué posición se le asigna a cada vértice. Esto se hace posible mediante el uso del mecanismo presentado por la API llamado programa shader. Un programa shader es una porción o conjunto de códigos que definen el comportamiento de las etapas programables del pipeline gráfico. En un programa shader, se define un algoritmo para cada etapa programable, que es ejecutado por cada primitiva que transita el pipeline.

En la actualidad, las API gráficas más utilizadas son dos. La primera de éstas es Direct3D, creada y mantenida por Microsoft para todos los sistemas Windows [7]. La alternativa multiplataforma a Direct3D es OpenGL [15]. Esta API es desarrollada por el Khronos Group, un consorcio internacional dedicado a la creación de estándares abiertos para múltiples dispositivos y plataformas.

Cada placa gráfica, de acuerdo al hardware que tiene, es capaz de ejecutar hasta una cierta versión del software de las API Direct3D y OpenGL. Además, mantiene compatibilidad hacia atrás a las versiones anteriores de la misma API. Cada nueva versión incorpora nueva funcionalidad o actualización a la funcionalidad existente. Por lo general, estos cambios son a nivel software de la API y no hay grandes modificaciones en el pipeline gráfico. Realizar cambios en el pipeline conlleva a aplicar modificaciones en todo el hardware nuevo, lo que es una tarea difícil y costosa.

El lanzamiento de Direct3D 11 en 2009 presentó una revolución en la forma en que se utilizan las ideas de teselación. Por primera vez, las placas aceleradoras gráficas capaces de ejecutar Direct3D 11, dieron la oportunidad al programador de realizar teselación directamente en la GPU. Para ello, se modificó el pipeline gráfico para dar lugar a un grupo de etapas dedicadas exclusivamente a realizar teselación de polígonos.

En la Fig. 2.1 se ilustra la diferencia entre el pipeline gráfico de Direct3D 10 con el pipeline Direct3D 11 y de OpenGL 4.0. En rectángulos se muestran las etapas programables del pipeline, mientras que las etapas en círculos son las etapas fijas, de las que el programador sólo puede definir en forma limitada ciertos comportamientos. La mayor diferencia entre el pipeline de Direct3D 10 con el de Direct3D 11 se halla en la incorporación de las 3 etapas dedicadas a teselación: *Hull Shader*, *Tessellator* y *Domain Shader*. Por su parte, la versión OpenGL 4.0, lanzada al mercado en 2010, incorporó la comunicación con las nuevas etapas programables, a las que denomina: *Tessellation Control Shader*, *Tessellation Primitive Generator* y *Tessellation Evaluation Shader*. Estas etapas son análogas en funcionalidad a las etapas respectivas en Direct3D 11.



**Figura 2.1:** Comparación del pipeline gráfico de Direct3D 10 con el de Direct3D 11 y el pipeline de OpenGL 4.0.

### 2.3. Selección de API para realizar el proyecto

Luego del análisis del pipeline gráfico y de las API para interactuar con el mismo, se decidió realizar el proyecto utilizando OpenGL. Las razones de la elección se resumen en las siguientes:

- *Multiplataforma*: al ser una biblioteca multiplataforma, OpenGL se encuentra disponible para ser utilizada en todos los sistemas operativos convencionales. De esta manera, el software desarrollado no estará destinado a un sistema operativo en particular, y podrá ser ejecutado en cualquier sistema sin realizar cambios sustanciales. En contraste, Direct3D 11 tiene la limitación de que se requiere un sistema operativo Microsoft Windows 7 o superior para ser utilizado.
- *Material existente*: durante la investigación sobre teselación, se consultaron diversos trabajos que implementaban técnicas de teselación en GPU utilizando las últimas bibliotecas disponibles. Sin embargo, gran porcentaje de los trabajos utilizaban Direct3D 11 como API para interactuar con el hardware. Esto puede ser debido a que la versión de Direct3D capaz de ejecutar el shader de teselación, fue anunciada dos años antes que la correspondiente a OpenGL. Entonces, en el presente proyecto, se eligió la utilización de OpenGL para realizar un aporte diferente a los disponibles, y presentar la alternativa multiplataforma para realizar teselación en GPU.
- *Utilización en la universidad*: en los cursos universitarios de Computación Gráfica, tanto locales como nacionales, y posiblemente internacionales, se opta por dictarlos utilizando la biblioteca OpenGL. Por lo tanto, y como se pretende que el proyecto sea puntapié inicial para estimular el desarrollo local de nuevas tecnologías, el uso de una biblioteca que el alumno interesado ya conoce, facilita su inserción en la temática.

### 2.4. Programación en GPU

En esta sección se incluye un resumen de las responsabilidades de cada etapa programable del pipeline. Esta discusión tomará como objeto el pipeline de OpenGL 4.0, pero comentarios similares se aplican al pipeline implementado por Direct3D 11. Se recuerda aquí que una etapa programable es aquella que le permite al programador, a través de un programa de shader, dictar el comportamiento de dicha etapa. Cada etapa tiene un conjunto de entradas y un conjunto de salidas que se requiere que emita. Además, el programador puede definir un conjunto adicional de datos que se emiten de una etapa de shader a las siguientes.

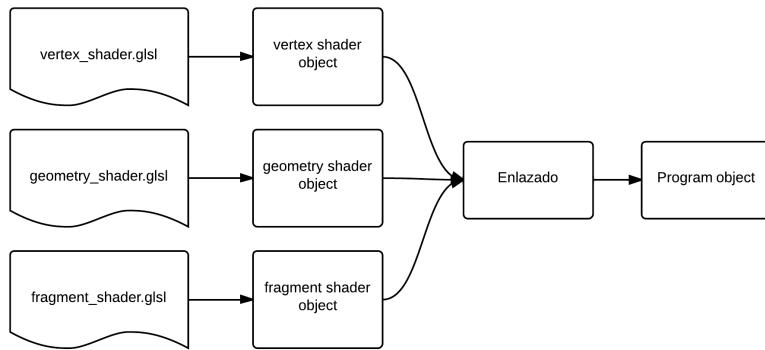
Además, a través de los *uniform*, el programador puede transferir datos a las etapas programables desde el procesador central (CPU). Los uniform son variables que se declaran en un programa de shader y se puede modificar su valor únicamente desde el programa cliente en CPU. Por ejemplo, estas variables, son

utilizadas para transferir a la GPU las matrices de transformación, las cuales están almacenadas en la memoria principal. Sin embargo, el valor de un uniform no cambia de una ejecución de un programa de shader a la siguiente. Su valor sólo puede ser modificado entre invocaciones de dibujado y antes de que se transfiera toda la geometría a la GPU.

Desde un programa ejecutado en CPU, se envían los datos al pipeline de la GPU. Estos datos contienen generalmente información de vértices como su posición o su normal. Además, por lo general se envían grupos de índices que definen cada uno de los elementos. En líneas generales, se tienen los siguientes tipos de primitivas: puntos, líneas y triángulos. Según el tipo de primitiva que se utilice, cada elemento debe estar formado por un grupo de índices que refieran a los vértices que lo componen, donde la cantidad de índices por elemento depende del tipo de primitiva. Por ejemplo, para un modelo formado por triángulos, se deben enviar conjuntos de 3 índices que identifican los vértices que constituyen cada elemento triangular.

Al emplear teselación, en contraposición, se utiliza el término genérico de *patch* para referirse a los grupos de datos que son enviados a la GPU. Un patch es un conjunto ordenado de  $n$  vértices, pero que por sí mismo no representa ningún tipo de las primitivas que se acostumbran a utilizar. En el programa de shader, el programador le da el sentido que requiera. Por ejemplo, un grupo de 6 vértices puede ser interpretado como un triángulo y sus vértices vecinos, o como un triángulo con 3 vértices por arista en lugar de 2.

Como se dijo anteriormente, un programa de shader es un grupo de algoritmos que describen el comportamiento de cada etapa programable del pipeline gráfico. El proceso de utilización de shaders comienza con la escritura del código para cada etapa programable. A continuación, cada archivo fuente se compila, dando lugar a un *shader object*. Finalmente, todos los shader object son enlazados, generando un *program object* (ver Fig. 2.2).



**Figura 2.2:** Ilustración del proceso de compilación de un programa de shader.

Cuando se utiliza el program object (también denominado programa de shader), el código generado se ejecuta con los patches como entrada. Un programa

de shader puede contener código para todas las etapas programables, o para un subconjunto de éstas.

El código fuente de los programas de shader está escrito en un lenguaje de programación similar a C, denominado GLSL (OpenGL Shading Language) [23]. En este lenguaje se declaran las entradas y salidas de cada etapa. Además se define una función principal `void main()` que es invocada con la información de entrada y cuya responsabilidad es definir datos para los valores de salida declarados.

A continuación se expone el resumen de las etapas programables de OpenGL 4.0. La discusión aquí está enfocada al dibujado de un modelo utilizando teselación. En el caso de no utilizar esta tecnología, el proceso es ligeramente diferente. Las etapas son:

- *Vertex Shader (VS)*: esta etapa describe las operaciones que se aplican a los valores de cada vértice y su información asociada (atributos de vértice, según la nomenclatura de OpenGL). En este proyecto no se le da uso extensivo a este shader, si no que se transfieren los atributos sin modificar a la siguiente etapa. En otras aplicaciones, se puede utilizar el VS para definir la posición final de los vértices mediante la aplicación de una transformación que lleve los vértices de posiciones de mundo a coordenadas de pantalla.
- *Tessellation Control Shader (TCS)*: Al terminar el VS, los vértices son agrupados en primitivas denominadas patches. Su principal responsabilidad es definir todos los niveles de teselación para cada patch que ingresa. Los niveles de teselación son los que definen el grado de refinamiento a efectuar en el patch. En esta etapa también es posible modificar la cantidad de vértices que componen el patch de entrada, de manera de agregar o quitar vértices del mismo.
- *Tessellation Evaluation Shader (TES)*: En la etapa previa no programable de Tessellation Primitive Generator (TPG) se interpolan las coordenadas de los nuevos vértices que se van a crear en el refinamiento. Cada uno de estos vértices, junto a sus coordenadas internas del patch original que lo generó, ingresa al TES. La responsabilidad del TES es definir todos los atributos de cada vértice nuevo. Por lo general, esta información es calculada por interpolación de los atributos de los vértices originales que dieron origen al vértice teselado. Las primitivas a rasterizar se crean en el TPG y, luego de definir la información particular para cada vértice en el TES, pasan a la siguiente etapa.
- *Geometry Shader (GS)*: Esta etapa opcional del pipeline gráfico es utilizada para modificar las primitivas a rasterizar. Se debe definir su entrada como un cierto tipo de primitiva, y su salida como otro tipo de primitiva (que puede ser o no ser igual a la entrada). Su salida pueden ser cero o más primitivas básicas: puntos, líneas o triángulos. Esta etapa también permite multiplicar la geometría, pero su capacidad es muy limitada con respecto a las etapas de teselación.

- *Fragment Shader (FS)*: Luego de pasar por la etapa no programable de rasterización, los fragmentos llegan al FS. Generalmente, este shader es utilizado para definir profundidad y color a los fragmentos.

En el siguiente apartado, se menciona la unidad de trabajo de cada etapa programable del pipeline:

- *Vertex Shader*: Se opera con cada uno de los vértices que componen cada patch y sus atributos.
- *Tessellation Control Shader*: Su unidad de trabajo es el patch completo, con la información asociada a cada uno de sus vértices.
- *Tessellation Evaluation Shader*: Se trabaja con todos los vértices que fueron creados en el proceso de teselación (en la etapa no programable).
- *Geometry Shader*: En esta etapa se procesan las primitivas básicas que se construyeron a partir de los vértices creados en la teselación. Estas primitivas son puntos, líneas o triángulos.
- *Fragment Shader*: Luego de la rasterización, cada primitiva del GS se discretiza en fragmentos que representan cada píxel de la primitiva a dibujar en pantalla. Un fragmento es la unidad de procesamiento de esta etapa.

En el listado 2.1 se muestra el código de un Vertex Shader cuya única función es enviar los datos de entrada a la siguiente etapa.

**Listado 2.1:** Ejemplo de Vertex Shader

```

1 #version 400
2 //Entradas
3 layout (location = 0) in vec3 in_Position;
4 layout (location = 1) in vec3 in_Normal;
5 //Salidas
6 out vec3 VPosition;
7 out vec3 VNormal;
8
9 void main() {
10     VPosition = in_Position;
11     VNormal = in_Normal;
12 }
```

La primer línea especifica la versión de GLSL que se utilizará. En este caso se utiliza la versión 4.00. Esta línea siempre debe ser la primera y debe tener el mismo valor en todos los códigos de un mismo programa de shader. Las siguientes dos líneas especifican los datos de entrada, los cuales deben estar en memoria principal del software que llama al programa de shader. Se toman dos entradas por vértice: posición y normal, ambas de dimensión 3. Estas entradas son llamados atributos del vértice. En este caso, el vértice tiene dos atributos: posición y normal.

La sentencia `layout` tiene distintos parámetros según la etapa de shader en la que se defina. En todos los casos, su función es determinar aspectos generales del pipeline gráfico. En este ejemplo en particular, se utiliza el parámetro `location` para configurar un índice. Este índice se emplea para definir un orden para los atributos de vértice, de manera de poder referenciarlos con claridad en el programa en CPU. En el ejemplo, el atributo posición tiene el índice 0, mientras que el atributo normal tiene el índice 1.

Las líneas que comienzan con `out vec3` identifican la salida de este shader, los cuales son los mismos datos ingresados. La función `void main()` se ejecuta por cada vértice. Aquí simplemente se copian los dos valores de entrada a los dos valores de salida. La siguiente etapa puede tomar los valores de salida del VS e interpretarlos como su propia entrada. La denominación `VPosition` hace referencia a que es la posición del vértice (`Position`) en la etapa de VS (V). Por lo tanto, en el siguiente ejemplo, `TEPosition` representa la posición del vértice en la etapa de TES.

**Listado 2.2:** Ejemplo de Geometry Shader

```

1 #version 400
2
3 layout( triangles ) in;
4 layout( triangle_strip , max_vertices = 3 ) out;
5
6 //Entrada al GS (salida del TES)
7 in vec3 TENormal[];
8 in vec4 TEPosition[];
9
10 //Salida del GS
11 out vec3 Normal;
12 out vec4 Position;
13
14 uniform mat4 MVPMatrix; //Projection*View*Model
15
16 void main() {
17     Normal = TENormal[0];
18     Position = MVPMatrix*TEPosition[0];
19     EmitVertex(); //fin vertice
20
21     Normal = TENormal[1];
22     Position = MVPMatrix*TEPosition[1];
23     EmitVertex(); //fin vertice
24
25     Normal = TENormal[2];
26     Position = MVPMatrix*TEPosition[2];
27     EmitVertex(); //fin vertice
28
29     EndPrimitive(); //fin primitiva
30 }
```

En el listado 2.2 se muestra un ejemplo de Geometry Shader. Este shader tiene como entrada 3 vértices en coordenadas locales y sus normales. Su función es transformar los vértices a coordenadas en el espacio normalizado. Para cada vértice, también copia la normal sin modificar. Con los 3 vértices forma un triángulo, el cual se pasa a la etapa de rasterización.

Aquí se puede ver la utilización de `layout` para definir la primitiva de entrada y de salida del GS. En este ejemplo, las entradas son triángulos (`triangles`) y las salidas son tiras de triángulos (`triangle_strip`). Además, se especifica que cada primitiva de salida tendrá un máximo de 3 vértices con `max_vertices = 3`. Aquí también se ilustra el uso de una variable uniform. En este caso, desde la CPU se envía a la GPU la matriz `MVPMatrix`. Esta matriz se utiliza para convertir los vértices de posición local a coordenadas normalizadas, y es el producto de la matriz `ModelView` por la matriz de Proyección. La matriz `ModelView` se utiliza para transformar los vértices de posición local a posición en el espacio de la cámara. Luego, al transformar con la matriz de Proyección, se obtienen las coordenadas normalizadas de cada vértice.

En la función `void main()` se construye cada primitiva de salida especificando los datos de cada vértice que la constituyen. Antes de cada llamada a `EmmitVertex()`, se asignan los valores de salida de cada vértice (normal y posición final). Notar que cada posición se multiplica por la matriz MVP para convertir las coordenadas locales a coordenadas normalizadas.

Al finalizar, la llamada a `EndPrimitive()` indica que se terminaron de definir los datos de la primitiva. En este caso, esta llamada no es necesaria porque es la última línea del shader y, de no estar presente, se invoca automáticamente. Sin embargo, puede ser necesario un procesamiento posterior en este shader y dicha función es utilizada para comunicar al shader que ya no se producirán más vértices para esta primitiva.

## 2.5. Teselación con OpenGL

Como se expuso anteriormente, tres etapas del pipeline gráfico son responsables de la configuración y uso de teselación en GPU.<sup>1</sup> La funcionalidad de teselación se considera activa si existe al menos un Tessellation Control Shader o un Tessellation Evaluation Shader contenido en el programa de shader activo. Si se da esta condición, la etapa de Tessellation Primitive Generator también es ejecutada. En otro caso, la información emitida por el Vertex Shader se transfiere al Geometry Shader (si existe) o al Fragment Shader.

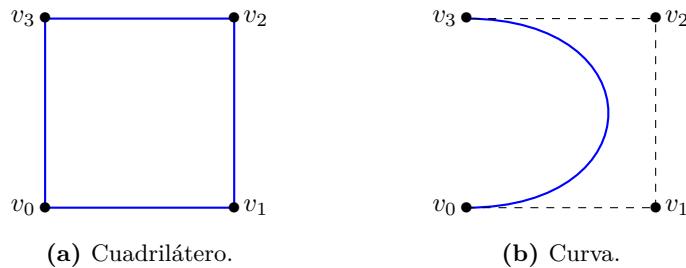
Para definir la cantidad  $n$  de vértices que constituyen un patch, en el programa cliente de OpenGL se debe realizar una llamada a la función `void glPatchParameteri(GLenum nombre, GLint valor)`. Los parámetros para esta función deben ser `GL_PATCH_VERTICES` en `nombre` y  $n$  en `valor`. El valor  $n$

---

<sup>1</sup>La discusión que se presenta aquí sigue el hilo conductor de la explicación presente en la especificación oficial de OpenGL 4.0 [20].

debe estar en el rango `[1, GL_MAX_PATCH_VERTICES]`. Este máximo depende de la implementación pero se garantiza que al menos sea de 32 vértices por patch.

Por ejemplo, con la llamada `glPatchParameteri(GL_PATCH_VERTICES, 4)` se define que se va a operar con patches de 4 vértices. En el programa en CPU no se tiene conocimiento sobre el tipo de primitivas finales que se van a renderizar, por lo que estos 4 vértices por sí mismos no tienen ningún sentido topológico. Dicho sentido, sobre si se trata de, por ejemplo, los vértices de un cuadrilátero o los puntos de control de una curva, se define en el TES. En la Fig. 2.3 se muestran dos patches distintos de 4 vértices y su interpretación dada por el TES. En la Fig. 2.3a, el patch se interpreta como un cuadrilátero, mientras que en la Fig. 2.3b, el patch se interpreta como los puntos de control de una curva de Bézier de tercer grado.



**Figura 2.3:** Distintas interpretaciones para patches de 4 vértices.

En las tres etapas de teselación, se convierte un patch en un grupo de muchas primitivas de acuerdo a la configuración de teselación definida. Debido a que la etapa de TPG no es programable, la configuración de teselación se realiza en el TCS y el TES.

Al ingresar al TCS, un patch es un conjunto de vértices ordenados pero sin estructura definida. En el TCS se configuran los niveles de teselación, mientras que en el TES se define la primitiva paramétrica que utiliza el TPG. Esta primitiva puede ser un triángulo o cuadrilátero. El TPG genera una primitiva paramétrica modelo de acuerdo a lo configurado en el TES y realiza la subdivisión en su interior según los niveles de teselación definidos en el TCS. Los nuevos vértices son emitidos al TES junto con sus coordenadas paramétricas. En el TES, utilizando la información de los vértices originales del patch, se debe definir todos los atributos de los vértices de salida (los originales y los nuevos vértices generados en la teselación).

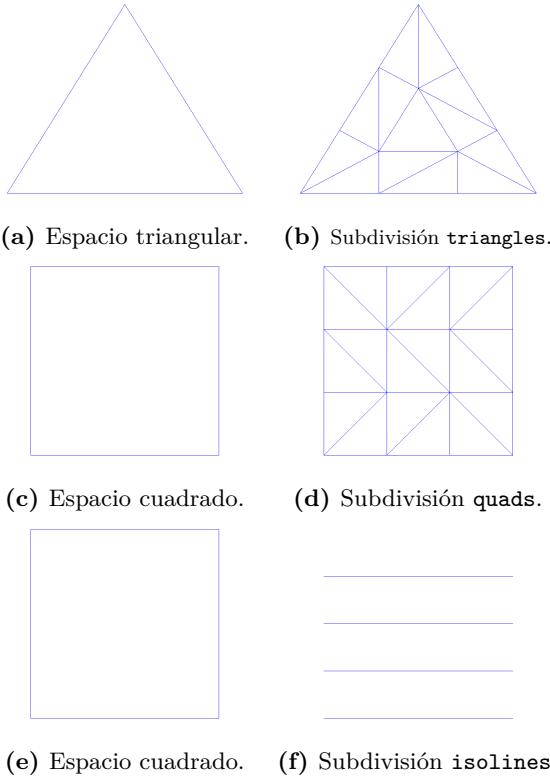
Para realizar este proceso, en el TCS se definen los niveles de teselación externos e internos en las variables `gl_TessLevelOuter[4]` y `gl_TessLevelInner[2]`. Dependiendo de la primitiva paramétrica modelo que se utilice en el TPG para realizar la teselación, todos o algunos de estos valores serán utilizados. En forma general, los valores exteriores definen la cantidad de segmentos en que se va a dividir una arista original del triángulo o cuadrilátero paramétrico. Por otra parte, los niveles interiores determinan la forma en que se interpolan las subdivisiones del área total dentro de la primitiva paramétrica. Es importante notar que al momento de escribir el programa shader para esta etapa, el pro-

gramador debe tener en consideración el tipo de subdivisión configurado en el TES, ya que es posible que algunos de los valores de las variables mencionadas sean ignorados.

El tipo de subdivisión efectuada depende de un valor definido en la sentencia `layout` del TES, cuyos valores pueden ser: `triangles`, `quads` o `isolines`. En el primer caso, el TPG subdivide un triángulo paramétrico en triángulos más pequeños. En el segundo caso, un cuadrado paramétrico se subdivide en triángulos más pequeños. Por último, en `isolines`, se subdivide un cuadrado paramétrico en un conjunto de poligonales.

Cada nuevo vértice generado tiene una posición  $(u, v, w)$  o  $(u, v)$  en un espacio paramétrico. En el caso de `triangles`, se trata de una coordenada baricéntrica que cumple la condición  $u + v + w = 1$  y cuyos valores indican el peso relativo de un vértice con respecto a los demás. Para `quads` e `isolines`, las coordenadas  $(u, v)$  indican la posición horizontal y vertical en un cuadrado paramétrico de lado 1. Vale notar que  $u, v, w \in [0, 1]$  en ambos espacios paramétricos.

En la Fig. 2.4 se muestran todos los tipos de subdivisión existentes. En la columna de la izquierda se presenta el espacio paramétrico. Por otra parte, en la columna de la derecha, se expone la forma de subdivisión realizada según el parámetro definido en la sentencia `layout` del TES.

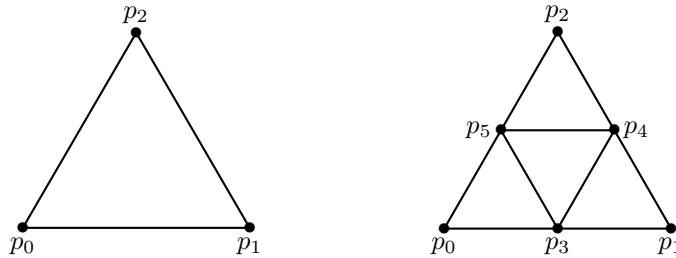


**Figura 2.4:** Ilustración de los tipos de subdivisión según el parámetro del TES.

Con respecto a los niveles de teselación, éstos son valores flotantes que tienen un mínimo de 0 y un máximo que está definido en la variable interna de OpenGL `GL_MAX_TESS_GEN_LEVEL`. Este valor depende de la implementación de la API, pero se garantiza de ser al menos 64. Cualquier nivel de teselación de los *outer* que se defina con valor igual a 0 ocasionará que la primitiva se descarte. Por otra parte, un valor igual a 1 implica que no se realice una subdivisión a través de esa arista.

El TCS se ejecuta una vez por cada patch, por lo que define niveles de teselación distintos para cada patch de los datos de entrada. Adicionalmente, el TCS debe emitir un patch como salida. Por lo general, este patch será el mismo que tuvo como entrada, pero es posible modificar la información del mismo (los atributos de sus vértices y la cantidad de vértices).

En la Fig. 2.5 se ilustra el proceso de teselación para un patch de 3 vértices y un conjunto de triángulos como primitiva de salida. En la primer fila de esta figura se muestra el procesamiento realizado en el espacio paramétrico del TPG. Luego, en la segunda fila, se muestra la correspondencia de la subdivisión en un patch cuando se transfiere del TCS al TES. En la Fig. 2.5a se muestra el triángulo paramétrico inicial y en la Fig. 2.5b el triángulo paramétrico una vez subdividido de acuerdo a los niveles de teselación. Este triángulo paramétrico está definido en el TPG. En la Fig. 2.5c se muestra un patch cualquiera de 3 vértices. Por último, en la Fig. 2.5d se muestran los triángulos que se generaron al mapear al patch la subdivisión efectuada en el triángulo paramétrico de la Fig. 2.5b. Vale la pena notar aquí que, según esta figura, el orden real de procesamiento es: c, a, b, d.

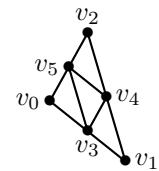


(a) Triángulo paramétrico (TPG).

(b) Triángulo subdividido (TPG).



(c) Patch de entrada (TCS).



(d) Triángulos de salida (TES).

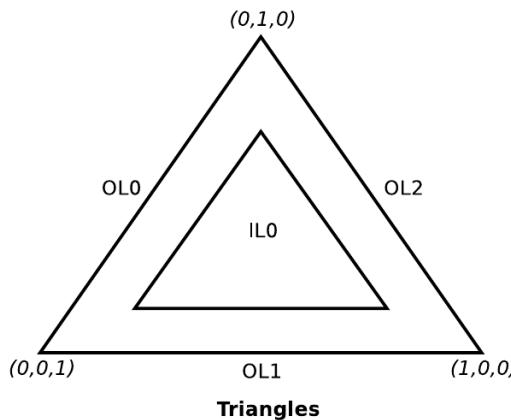
**Figura 2.5:** Generación de primitivas a partir de un patch.

Una vez definidos los niveles de teselación para un patch, las nuevas primitivas son generadas en el Tessellation Primitive Generator. Hay dos características a tener en cuenta sobre la generación de nuevas primitivas. Por un lado, la interpretación que realiza el TPG de los niveles de teselación. Por otro lado, la posición en la que se ubican los nuevos vértices (y por lo tanto, las nuevas primitivas) que se generen.

A continuación se expone la utilización de los niveles de teselación en los distintos tipos de primitivas. Como se dijo, la teselación se puede realizar según tres tipos de primitivas: triángulos, cuadriláteros y poligonales. La diferencia entre éstos se encuentra en la interpretación que se realiza de los niveles de teselación y cómo se generan las nuevas primitivas.

### 2.5.1. Niveles de teselación en triángulos

El triángulo paramétrico utilizado en el TPG tiene tres vértices y tres aristas. Para definir sus niveles de teselación, se utilizan los tres primeros valores de `gl_TessLevelOuter`. Éstos definen la cantidad de segmentos en que se divide cada arista. Por otra parte, se usa el primer valor de `gl_TessLevelInner` para configurar la transición, en el interior, de la subdivisión realizada en las aristas.



**Figura 2.6:** Parametrización de dominio para `triangles`.

En la Fig. 2.6, extraída de la especificación de OpenGL 4.0 [20], se muestra cómo se interpretan los niveles de teselación. En esta imagen, OL es la abreviatura de ‘Outer Level’ mientras que IL referencia a ‘Inner Level’. Se puede apreciar que se tiene un nivel externo por cada arista del triángulo paramétrico. Este valor especifica la cantidad de nuevos segmentos que se generan en esa arista. El valor OL0 corresponde a la arista opuesta al vértice 0 del triángulo (el vértice (1,0,0) en la imagen). Del mismo modo, OL1 se corresponde con la arista opuesta al vértice 1, y OL2 la opuesta la vértice 2. En las métricas, es necesario tener en cuenta esta correspondencia al calcular los niveles de teselación, ya que, por ejemplo, para cualquier cálculo que se realice sobre la arista que une

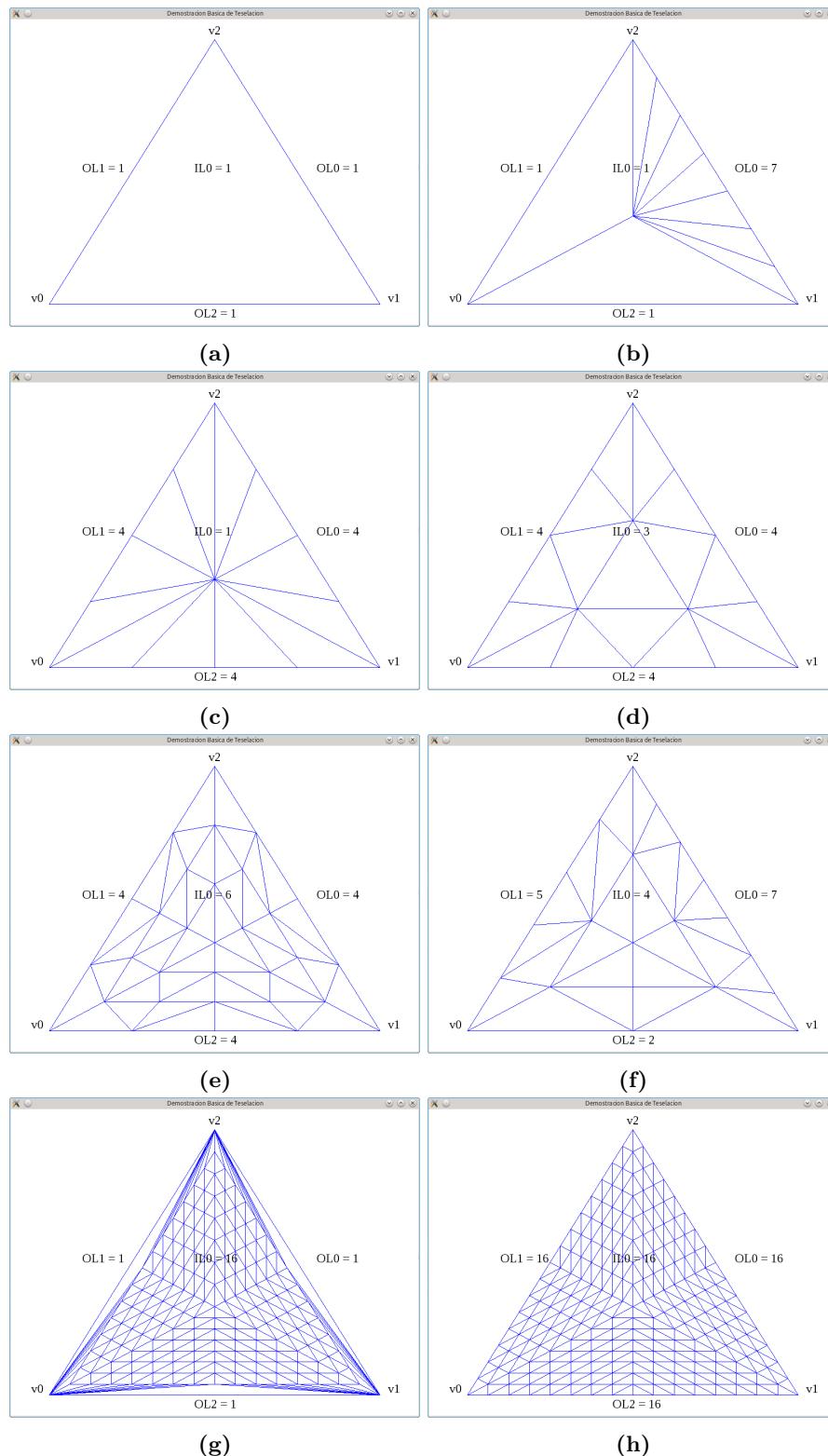
el vértice 0 con el 1, se debe definir el valor OL2 de teselación.

El comportamiento del IL es ligeramente distinto al de los niveles externos. Para un nivel IL igual a 0 o 1, no hay cambios. Para un valor igual a 2, se genera un único vértice en el interior y este vértice se une a todos los vértices de las aristas externas del triángulo paramétrico (incluidos los originales). Para  $IL > 2$ , se generan triángulos concéntricos. En dirección hacia el centro, cada triángulo concéntrico tiene dos segmentos menos que el triángulo inmediatamente anterior. De los triángulos concéntricos, utilizando el valor IL, sólo se tiene control sobre el más externo de éstos, el cual se une con las aristas exteriores del triángulo paramétrico. El triángulo del interior más externo será generado con  $IL-2$  segmentos por arista. Por ejemplo, si  $IL = 5$ , cada arista del triángulo interior está constituida por 4 vértices ( $IL-2 = 3$  segmentos). Estos 4 vértices se conectan a todos los vértices de las aristas externas. De acuerdo a los niveles externos e internos, el TPG se encarga de generar automáticamente el resto de los vértices paramétricos. Para una presentación visual de lo mencionado, ver las Fig. 2.7d y 2.7e.

En la Fig. 2.7, capturada del software creado para este proyecto, se presenta el resultado para distintos valores en los niveles de teselación de un triángulo paramétrico. Al lado de cada arista se presenta el nivel de teselación definido para dicha arista. Además, en el centro, se indica el nivel de teselación interno. En la Fig. 2.7a se muestra el triángulo original sin teselar. En la Fig. 2.7b se expone la modificación de un único nivel de teselación. Este nivel de teselación es igual a 7, por lo que se crean 7 segmentos a lo largo de la arista original (se insertan 6 nuevos puntos). Además, se agrega un vértice en el centro del triángulo y a éste se conectan los nuevos vértices, recuperando una malla conforme. En la Fig. 2.7c se muestra el caso en que se define este mismo nivel de teselación en todas las aristas. De la misma manera, estos vértices se unen al vértice central para formar nuevos elementos triangulares. Notar que el vértice central agregado no depende, en este caso, del nivel de teselación interno, si no que fue agregado automáticamente.

En las Fig. 2.7d y 2.7e se aumenta el nivel de teselación interno. Se puede ver que al incrementar este valor, se crea una serie de nuevos vértices en el interior del triángulo paramétrico. Estos vértices se alinean en forma paralela a las aristas del triángulo externo, formando triángulos concéntricos. En la Fig. 2.7f se presenta un ejemplo donde los niveles de teselación son todos diferentes. En este caso, las plantillas generadas por el TPG logran recuperar una malla conforme, uniendo en forma correcta todos los vértices generados.

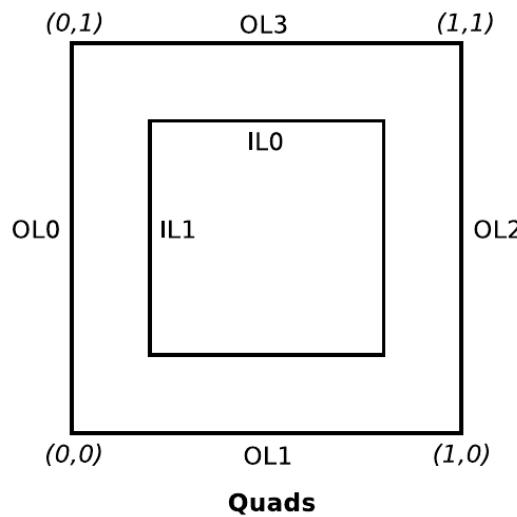
Por último, en las Fig. 2.7g y 2.7h se presentan casos extremos. En 2.7g, los niveles de teselación externos son iguales a 1 y por lo tanto no hay subdivisión a través de las aristas externas. En el interior se presenta un nivel de teselación alto, por lo que se genera la serie de triángulos concéntricos. En la interfaz, se unen todos los vértices a los vértices originales. Aquí se crean elementos estirados que no son deseables, pero se aprecia que el TPG se ajusta a los requerimientos impuestos por el programador. En la 2.7h se expone un ejemplo donde todos los niveles de teselación son iguales a 16. Aquí, todos los elementos son aproximadamente del mismo tamaño y la malla generada es regular.



**Figura 2.7:** Distintos ejemplos de teselación en el modo triangles.

### 2.5.2. Niveles de teselación en cuadriláteros

Al realizar teselación en cuadriláteros, las subdivisiones en el TPG se realizan en un cuadrado paramétrico. En el TES se toman las coordenadas  $(u, v)$  de todos los puntos y se generan los triángulos que finalmente serán renderizados. La primitiva `quads` hace uso de los 4 valores de `gl_TessOuterLevel` y los 2 de `gl_TessInnerLevel`. En la Fig. 2.8, también obtenida en la especificación de OpenGL 4.0, se muestra la correspondencia de los niveles con la subdivisión efectuada. En este caso, los dos niveles internos hacen referencia a la subdivisión en el interior del cuadrilátero paramétrico.



**Figura 2.8:** Parametrización de dominio para `quads`.

Los niveles externos de teselación se definen para cada arista del cuadrilátero paramétrico. La arista paramétrica  $u = 0$  (arista vertical izquierda) se corresponde con el nivel de teselación  $OL_0$ . La arista paramétrica  $v = 0$  (horizontal inferior) utiliza el valor  $OL_1$ . De igual forma, la arista  $u = 1$  (vertical derecha) hace uso de  $OL_2$ , y la arista  $v = 1$  (horizontal superior) utiliza  $OL_3$ .

Similar a lo que ocurre en triángulos, los niveles internos de teselación en cuadriláteros tienen un comportamiento especial. Para niveles iguales a 0 o 1, no hay cambios. Si al menos uno de  $IL_0$  o  $IL_1$  es 2, se inserta un vértice en el centro del dominio cuadrado. Este vértice se une a todos los vértices que se ubiquen en las aristas externas. A partir de  $IL_0 = 3$  o  $IL_1 = 3$ , se generan cuadriláteros concéntricos en el interior del dominio paramétrico. Si ambos niveles internos son iguales a 3, se crea un único cuadrilátero en el interior. Los 4 vértices de este cuadrilátero se unen a los vértices externos, siempre recuperando una malla conforme de triángulos y sin ocasionar superposiciones de elementos. Para niveles mayores, se genera una serie de cuadriláteros concéntricos. Al avanzar hacia el centro, cada cuadrilátero tiene dos segmentos menos que el cuadrilátero anterior. Luego, se conectan los vértices para obtener una malla de triángulos donde

las longitudes de todos los segmentos horizontales y verticales son similares.

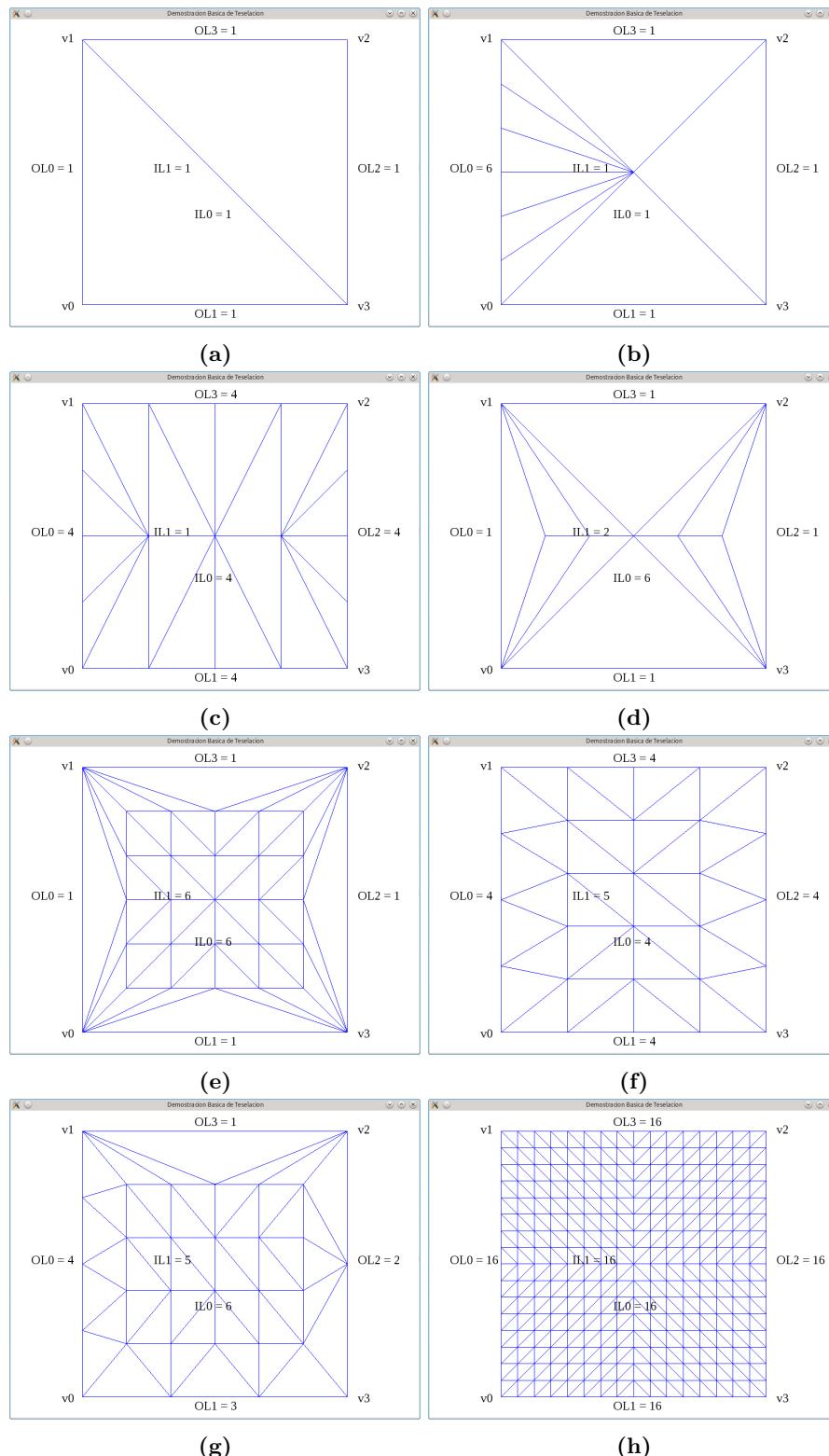
En el caso en que los niveles internos sean distintos entre sí, la disposición de los cuadriláteros es diferente. El nivel IL0 indica cuántos segmentos en su dirección horizontal tiene el cuadrilátero interior más externo. De igual forma, el IL1 indica lo propio para la dirección vertical. En el interior, se genera una malla regular de cuadriláteros concéntricos sujetos a esta restricción. En dicha malla, la longitud de los segmentos verticales es la misma, pero es distinta a las longitudes de los segmentos horizontales (que son iguales entre sí). Luego, para la rasterización, todos los cuadriláteros se descomponen en triángulos.

En la Fig. 2.9 se presentan distintos ejemplos del modo de subdivisión en teselación en cuadriláteros. En la Fig. 2.9a se muestra el cuadrilátero original. Como el hardware de video utiliza únicamente triángulos, este cuadrilátero se renderiza como dos triángulos. En la Fig. 2.9b se aumenta el nivel de teselación en una única arista. En este caso, se define un nivel de teselación OL0 = 6 para la arista  $u = 0$ . Se puede apreciar que se insertó un vértice en el centro al que se unen los vértices de la arista.

En 2.9c se muestra un ejemplo con nivel interno horizontal de teselación igual a 4. Aquí se generan 3 vértices internos y dos segmentos. Estos vértices corresponden al cuadrilátero interior. En este caso, debido a que el nivel interno vertical es igual a 1, el cuadrilátero interno está degenerado en una línea. Los vértices que están en las aristas externas se unen a los vértices del interior y se recupera una malla de triángulos. En la Fig. 2.9d se expone un caso donde los niveles externos son iguales a 1 y el interno horizontal es igual a 6. De la misma manera, se unen los vértices externos a los nuevos vértices internos. El IL1 no tiene ninguna influencia en esta subdivisión ya que su valor es igual a 2.

En la Fig. 2.9e se aumenta IL1, dando lugar a un mallado regular en el interior del cuadrilátero paramétrico. En este caso, se tienen dos cuadriláteros concéntricos de 4 y 2 segmentos por arista, respectivamente. La Fig. 2.9f ilustra el caso cuando los niveles internos son distintos entre sí. Se aprecia que el cuadrilátero interno tiene 2 segmentos por arista en la dirección horizontal y 3 en la vertical. Esta cantidad de segmentos está acorde a los niveles definidos: 4 en horizontal y 5 en vertical.

En la última fila de esta figura se muestran dos casos extremos. Por un lado, en la Fig. 2.9g se presenta el caso cuando todos los niveles de teselación son distintos. El generador de primitivas, uniendo los vértices correctamente, logra recuperar una malla de triángulos conforme. Por último, en 2.9h se expone un ejemplo cuando todos los niveles de teselación son iguales a 16. Se puede ver que la malla es regular y las aristas horizontales y verticales de todos los triángulos internos son iguales.

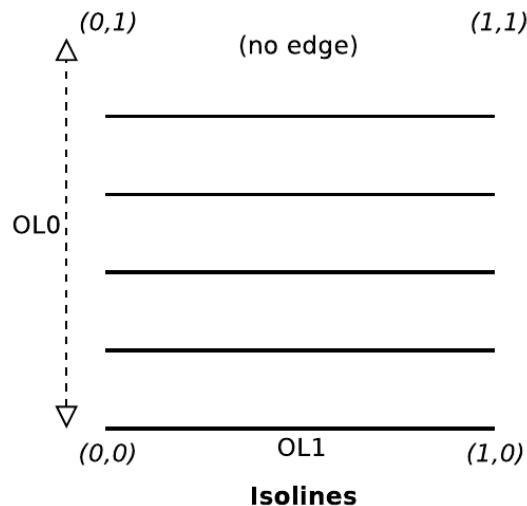


**Figura 2.9:** Distintos ejemplos de teselación en el modo quads.

### 2.5.3. Niveles de teselación en poligonales

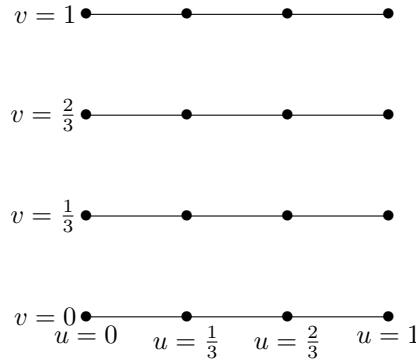
El tercer modo de primitiva en la que se puede realizar teselación es la denominada por OpenGL como **isolines**. Esta primitiva permite, a partir de un conjunto de vértices inicial, generar una o muchas poligonales y modificar los atributos de cada uno de los nuevos vértices generados. Una poligonal es un conjunto de segmentos conectados. Generalmente son utilizadas para aproximar una curva al dibujarla en pantalla.

En esta primitiva sólo se hace uso de OL0 y OL1. El primer valor, OL0 define la cantidad de líneas distintas que se generarán. El segundo, por su parte, configura la cantidad de veces que se subdividirá cada línea. En cada línea, se unen los vértices adyacentes por un segmento, pero no hay conexión entre vértices de distintas poligonales creadas a partir de un patch.



**Figura 2.10:** Parametrización de dominio para **isolines**.

En la Fig. 2.10 se ilustra la interpretación de estos valores. Se parte de un cuadrilátero paramétrico y según OL0 se genera una cierta cantidad de líneas muestreando la coordenada paramétrica vertical  $v$ . Por otro lado, mayores valores de OL1 ocasionan mayor cantidad de muestras en la coordenada horizontal  $u$ . Por ejemplo, para  $OL0 = 4$  y  $OL1 = 3$ , se obtienen 4 poligonales formadas por 3 segmentos y 4 vértices cada una. Por lo tanto, al TES llegan coordenadas paramétricas para 16 vértices. La posición final de cada uno de estos vértices debe ser definida por el programador utilizando la coordenadas paramétricas y los vértices del patch. En la Fig. 2.11 se ilustra este ejemplo y se incluyen, en los ejes, las coordenadas paramétricas correspondientes a cada posición vertical y horizontal.



**Figura 2.11:** Ilustración de las poligonales generadas en `isolines` para cada valor del parámetro  $v$ .

Como en los casos anteriores, en la Fig. 2.12 se muestran distintos ejemplos de la variación de los parámetros en este modo de teselación. Esta figura fue obtenida utilizando 4 vértices alineados en un cuadrilátero como en el ejemplo anterior, pero sólo se tiene interés en los dos inferiores. Los otros dos vértices son utilizados para interpolar, utilizando el parámetro  $v$ , las posiciones de las poligonales que se inserten cuando OL0 sea mayor a 1. Los cuatro vértices están dibujados con color negro en la figura. Por otra parte, todos los vértices generados en la teselación están dibujados con color azul.

Sin embargo, esto es sólo un ejemplo. En el caso general, se puede disponer cualquier cantidad de vértices en el patch, y el modo de teselación `isolines` simplemente se encarga de generar coordenadas en un cuadrilátero paramétrico. Las primitivas creadas como resultado son poligonales que unen vértices consecutivos de la misma coordenada  $v$ .

En la Fig. 2.12a se muestra la poligonal original formada por únicamente dos vértices. En la Fig. 2.12b se incrementó OL0, dando lugar a la generación de 5 poligonales. La ubicación de estas nuevas poligonales la define el programador. En este caso, se desplazan en forma equitativa según su coordenada paramétrica  $v$  (utilizando los otros dos puntos ‘invisibles’ para interpolar la posición). Debido a que OL1 = 1, estas poligonales están constituidas por un único segmento que une dos vértices en cada coordenada  $v$ : uno en  $u = 0$  y el otro en  $u = 1$ . La Fig. 2.12c es similar a la anterior, pero se incrementó OL1. Cada poligonal está formada por 5 vértices unidos entre sí por 4 segmentos.

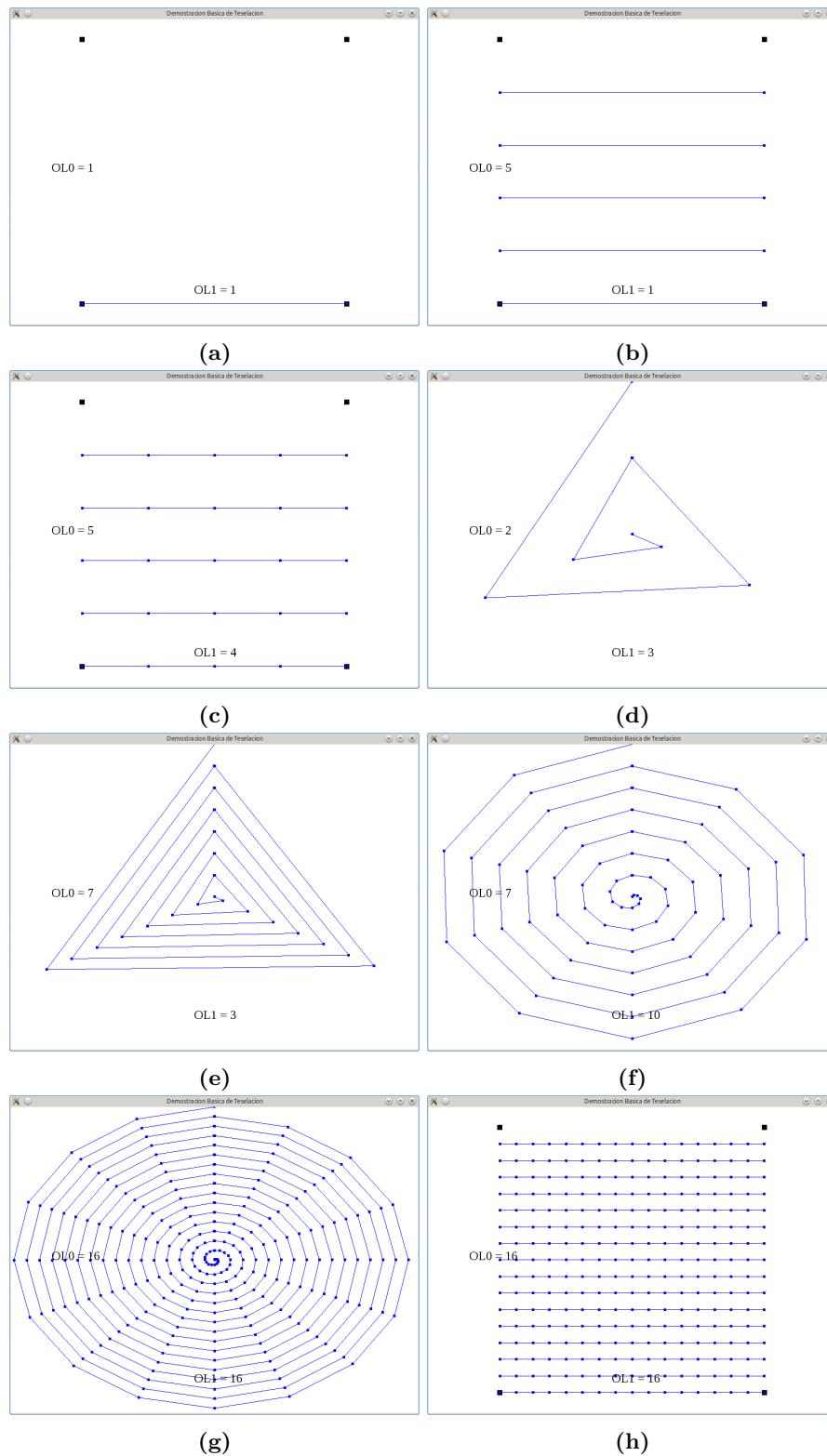
Para poder apreciar con más claridad la variación del parámetro OL1, se preparó un ejemplo en el que se renderiza una espiral. En este ejemplo, se utilizan las coordenadas paramétricas para ubicar los puntos alrededor de un círculo con distintos radios, dando lugar a una espiral. El radio que define la posición se calcula con respecto a las dos coordenadas paramétricas  $(u, v)$ , las cuales dependen de los dos niveles de teselación. Por otra parte, la posición angular se obtiene con respecto al parámetro  $u$ , dependiente de OL1. Aquí no se desea extenderse sobre cómo es calculada esta espiral, pero lo que interesa es ilustrar

cómo, al variar los niveles de teselación, se obtiene una mejor representación de la misma.

Se nota aquí que la espiral parece ser conexa, formada por una única poligonal. Sin embargo, este efecto se obtiene al ubicar el último punto de una poligonal (mismo parámetro  $v$ ) en la misma posición que el primer punto de la poligonal siguiente.

En un caso básico, en la Fig. 2.12d se muestra la espiral generada por 2 giros ( $OL_0 = 2$ ) y discretizada en 3 puntos equiespaciados por giro ( $OL_1 = 3$ ). La Fig. 2.12e expone la misma espiral pero con un total de 7 giros. La Fig. 2.12f, por su parte, muestra una espiral de 7 giros y discretizada en 10 puntos por giro.

Por último, la Fig. 2.12g presenta una espiral de 16 giros y 16 puntos por giro. En la Fig. 2.12h se muestra nuevamente el caso original, pero con 16 poligonales de 16 puntos cada una.



**Figura 2.12:** Distintos ejemplos de teselación en el modo **isolines**.

### 2.5.4. Posicionado de los nuevos puntos

Hasta aquí, no se hizo alusión a cómo se calculan los nuevos puntos en las primitivas generadas. OpenGL presenta tres modos distintos para ubicar los nuevos vértices una vez que se definieron los niveles externos e internos. Las plantillas que se explican a continuación son generadas en la etapa no programable de TPG. La posición exacta donde se ubican los nuevos vértices depende de la implementación de la especificación de OpenGL, y el programador no tiene control sobre esto. En este apartado se presenta lo que dice exclusivamente la especificación. Luego se incluyen ejemplos para apreciar la diferencia entre los distintos métodos. En la siguiente discusión, *max* representa el valor de `GL_MAX_TESS_GEN_LEVEL`.

Los distintos modos de espaciado presentados por OpenGL son:

- **`equal_spacing`**: el nivel de teselación se limita al rango  $[1, max]$ . Luego se redondea al entero mayor más cercano  $n$  y la arista se divide en  $n$  segmentos de igual longitud en el espacio  $(u, v)$  o  $(u, v, w)$ .
- **`fractional_even_spacing`**: el nivel de teselación se limita al rango  $[2, max]$  y luego se redondea al entero par  $n$  mayor más cercano.
- **`fractional_odd_spacing`**: Primero el nivel de teselación se limita al rango  $[1, max - 1]$  y luego se redondea al entero impar  $n$  mayor más cercano.

Tanto en el modo par (`even_spacing`) como el impar (`odd_spacing`), la longitud de los segmentos se comporta de manera diferente si el nivel  $t_i$  de teselación es par o no:

- *Si el modo es par y  $t_i$  es par*: la arista se subdivide en  $n = t_i$  segmentos de igual longitud.
- *Si el modo es par y  $t_i$  es impar*:  $n$  es igual al entero par mayor más cercano. La arista se subdivide en  $n - 2$  segmentos iguales y dos segmentos más de menor longitud. La posición y longitud exactas de estos segmentos dependen de la implementación de la API.
- *Si el modo es impar y  $t_i$  es par*: similar al caso anterior, pero  $n$  es el valor impar mayor más cercano.
- *Si el modo es impar y  $t_i$  es impar*: igual al primer caso, la arista se subdivide en  $n = t_i$  segmentos iguales.

La longitud de los dos segmentos más cortos en los modos par e impar dependen de la parte fraccionaria del nivel de teselación (que luego se pierde al realizar el redondeo). Nuevamente, los detalles particulares del cálculo de estas longitudes son dependientes de la implementación.

En el modo simétrico (`equal_spacing`), por otra parte, todos los segmentos de una misma arista son iguales, sin importar si el nivel de teselación es par o

impar. Los niveles interiores se comportan de la misma manera, modificando el nivel de teselación si corresponde.

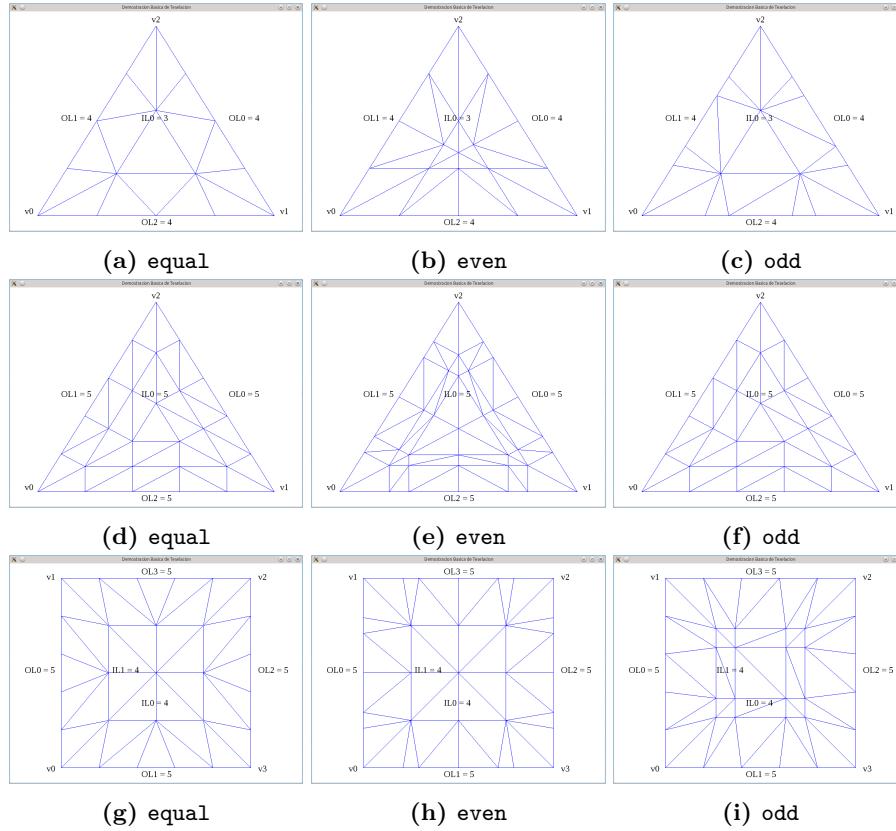
Todos los ejemplos que fueron presentados en la sección anterior, se utilizó `equal_spacing` para generarlo. En la Fig. 2.13 se exponen ejemplos de los distintos modos de espaciado. En la primer fila se tiene un triángulo con niveles de teselación externos iguales a 4. En la Fig. 2.13a, los segmentos son todos iguales ya que se usa el modo de espaciado simétrico. En la Fig. 2.13b se tiene el mismo triángulo pero con el modo par. Se puede ver que no hubo cambios en los niveles de teselación externos, ya que son pares. En el nivel interno, el cual es 3, se llevó a 4 para respetar el modo de espaciado par. Por lo tanto, en el modo simétrico se tiene un triángulo interno de un segmento por arista, mientras que en el modo par se tiene una arista conformada por dos segmentos. En la Fig. 2.13c se presenta el mismo triángulo pero con el modo impar. En este caso, no se hizo ningún cambio en el nivel interno pero sí en todos los externos, los cuales se llevaron de 4 a 5. Se puede apreciar que en cada arista se tienen 5 segmentos: 3 de igual longitud entre sí, y 2 más cortos pero iguales entre sí.

En la segunda fila de la Fig. 2.13 se presenta otro triángulo pero aquí todos los niveles de teselación son impares iguales a 5. En la Fig. 2.13d, las aristas se dividieron en 5 segmentos de igual longitud. La Fig. 2.13e muestra el caso par, donde todos los niveles de teselación se llevaron a 6. En último lugar de esta fila, la Fig. 2.13f expone el caso impar. Aquí no hay modificación con respecto al caso simétrico, ya que todos los niveles son originalmente impares.

La última fila de la Fig. 2.13 incluye un ejemplo para un cuadrilátero. En forma similar a los anteriores, se expone el caso simétrico, par e impar. Los niveles de teselación externos son iguales a 5 y los internos son iguales a 4. En el caso simétrico de la Fig. 2.13g, todos los segmentos, tanto del cuadrilátero exterior como del interior, tienen la misma longitud entre sí. El caso par de la Fig. 2.13h lleva los niveles de teselación externos al par más cercano, de  $t_i = 5$  a  $n = 6$ . Por último, la Fig. 2.13i expone el caso impar. Aquí no hay cambios en los niveles externos pero sí en los internos, los cuales son modificados y llevados de  $t_i = 4$  a  $n = 5$ .

Lo que definen los modos de espaciado es cómo se generan las coordenadas paramétricas de los nuevos vértices. El caso de poligonales es igual al de cuadriláteros. La diferencia está en cómo se construyen luego las primitivas: en cuadriláteros se generan triángulos, mientras que en poligonales se crean sucesiones de líneas.

Una vez definidas las posiciones paramétricas finales de los nuevos vértices, éstos son enviados al TES. En el TES, se tiene acceso a las coordenadas paramétricas  $(u, v, w = 1 - u - v)$  (triángulos) o  $(u, v)$  (cuadriláteros y poligonales). Con estas coordenadas y la información de los vértices iniciales del patch, el programador debe definir la posición, en coordenadas locales, de todos los vértices (tanto los nuevos como los iniciales). Luego de esta operación, todos los vértices continúan su paso por el pipeline gráfico. Adicionalmente, el programador debe transformar estas coordenadas locales a coordenadas normalizadas antes de la etapa de rasterización. En el caso de no utilizar teselación, este proceso se realiza



**Figura 2.13:** Diferencias entre los modos de espaciado.

normalmente en el Vertex Shader para los vértices originales. En contraposición, cuando se emplean las etapas de teselación, se pueden definir las coordenadas normalizadas de los nuevos vértices en el TES o en el GS.

Los vértices generados en cada muestra de las coordenadas paramétricas se unen formando primitivas. Estas primitivas son triángulos, líneas o puntos, y se configura su tipo desde la sentencia `layout` del TES. El programador no tiene control sobre cómo se generan estas primitivas (sus adyacencias). Sólo puede configurar, utilizando los niveles de teselación y los modos de espaciado, la forma en que se toman los puntos de muestra del espacio paramétrico. El generador de primitivas realiza toda la operación de generar las adyacencias y emitir los triángulos (para `quads` y `triangles`) o líneas (para `isolines`).

Como se puede apreciar, muchos de los temas aquí discutidos dependen de la implementación de la API. El comité encargado de regular la API escribe una especificación y ésta es enviada a los fabricantes. La responsabilidad de los fabricantes es desarrollar y distribuir el driver de video. El driver es un software que, entre otras cosas, implementa la API y permite la comunicación del sistema operativo con el hardware de video. Puede suceder que entre distintos fabricantes, la implementación sea distinta y se encuentren diferencias en el resultado

final. Esto se pudo verificar al probar el software final de este proyecto en dos computadoras distintas. Una computadora tenía una placa aceleradora de video de marca ATI, mientras que la otra poseía una GPU de marca NVIDIA. En el capítulo de resultados se extiende la discusión sobre la diferencia de implementaciones. Sin embargo, se desea remarcar aquí que el desarrollo con esta API es dependiente de la implementación específica según fabricante, por lo que es probable que el programador inexperimentado se encuentre con diferencias en el resultado final y no logre comprender su origen. En consiguiente, la fuente de estas diferencias puede que se halle en las implementaciones utilizadas.

## 2.6. Ejemplos de shaders de teselación

Para finalizar con este capítulo, se expone aquí un ejemplo simple de un Tessellation Control Shader y uno de Tessellation Evaluation Shader. En particular, estos códigos corresponden al software que renderiza los cuadriláteros que se mostraron en la Fig. 2.9.

El Listado 2.3 comienza con la definición de la versión de programa de shader que se utiliza. La segunda línea tiene la sentencia `layout` que en este caso define cuántos vértices hay en cada patch. Como el patch se corresponde con un cuadrilátero, el número es 4. Las siguientes líneas definen las entradas y salidas de este shader. Además se declaran algunos uniform que son enviados desde el programa en CPU y configuran todos los valores finales de teselación. El vector uniform `TessLevelInner` contiene los 2 niveles de teselación internos a definir, mientras que el vector `TessLevelOuter` determina los 4 niveles externos. Por su parte, el valor booleano `Tessellate` permite decidir si se utilizan los valores pasados por parámetro o un valor por defecto almacenado en `TessDefault`.

La función `void main()` es simple y sólo configura los niveles de teselación a partir de los valores definidos por el usuario en los uniform. Asimismo, transfiere la posición del vértice sin modificar a la siguiente etapa. La variable de OpenGL `gl_InvocationID` identifica el índice interno (dentro del patch) del vértice que está siendo procesado.

Las variables `gl_in` y `gl_out` son internas al shader y refieren a las entradas y salidas del shader actual. Si se asigna un valor a `gl_out` en una etapa programable, este valor estará disponible en `gl_in` de la etapa inmediatamente siguiente. Esto es utilizado como una forma básica de transferencia de datos entre etapas adyacentes en el pipeline gráfico. Sin embargo, al definir variables propias `out` (como `TCPosition[]`), su valor puede ser recuperado por *todas* las etapas siguientes que las declarén.

**Listado 2.3:** Ejemplo de Tessellation Control Shader

```

1 #version 400
2 //patch de 4 vertices
3 layout (vertices = 4) out;
4
5 in vec3 VPosition[]; //Entrada al TCS (viene de VS)
6 out vec3 TCPosition[]; //Salida del TCS
7
8 //Valores de teselacion pasados por parametro
9 uniform vec2 TessLevelsInner;
10 uniform vec4 TessLevelsOuter;
11
12 uniform bool Tessellate; //Si false, se usa TessDefault
13 uniform float TessDefault; //Valor por defecto
14
15 void main() {
16     //Copiado a la siguiente etapa
17     gl_out[gl_InvocationID].gl_Position
18         = gl_in[gl_InvocationID].gl_Position;
19     TCPosition[gl_InvocationID]
20         = VPosition[gl_InvocationID];
21     if (Tessellate) {
22         //Se copian los valores pasados por parametro
23         gl_TessLevelOuter[0] = TessLevelsOuter[0];
24         gl_TessLevelOuter[1] = TessLevelsOuter[1];
25         gl_TessLevelOuter[2] = TessLevelsOuter[2];
26         gl_TessLevelOuter[3] = TessLevelsOuter[3];
27         gl_TessLevelInner[0] = TessLevelsInner[0];
28         gl_TessLevelInner[1] = TessLevelsInner[1];
29     } else {
30         //Se utiliza el valor por defecto
31         gl_TessLevelOuter[0] = TessDefault;
32         gl_TessLevelOuter[1] = TessDefault;
33         gl_TessLevelOuter[2] = TessDefault;
34         gl_TessLevelOuter[3] = TessDefault;
35         gl_TessLevelInner[0] = TessDefault;
36         gl_TessLevelInner[1] = TessDefault;
37     }
38 }
```

En el listado 2.4 se muestra el Tessellation Evaluation Shader correspondiente al TCS anterior. En la sentencia `layout` se especifican tres parámetros distintos. Primero se configura el tipo de primitiva paramétrica utilizada en la generación de los nuevos vértices, que en este caso es `quads`, por lo que se tendrá un dominio paramétrico cuadrado con coordenadas  $(u, v)$ . El segundo parámetro de la sentencia `layout` es el modo de espaciado, que aquí es `fractional_even_spacing`. Estos dos valores son utilizados por el TPG para definir la posición de los nuevos vértices y crear las nuevas primitivas. Por último, el parámetro `cw` define el orden de los vértices en cada primitiva. En este

caso se utiliza un orden *clockwise* (en el orden de como giran las agujas de un reloj).

Este shader tiene una única entrada `TCPosition[]` (que se corresponde con la salida del anterior). Además se utiliza la matriz MVP que fue transferida desde la CPU. Se recuerda aquí que este shader es invocado por cada vértice del patch (tanto los originales como todos los nuevos que fueron creados en el TPG). En la función `void main()` se obtienen las coordenadas paramétricas del vértice actual. Estas coordenadas están almacenadas en la variable `gl_TessCoord`. En el caso de `quads` e `isolines`, esta variable tiene dimensión 2. Por otra parte, para `triangles`, su dimensión es 3 y la coordenada `w` se encuentra en `gl_TessCoord.z`.

En las últimas líneas, por medio de una interpolación bilineal, se define la posición del vértice. Para realizar dicha interpolación, se realizan 3 interpolaciones lineales. Primero se computan dos interpolaciones horizontales utilizando el parámetro `u` y los vértices extremos almacenados en `TCPosition[]`. Luego, a partir de estos resultados, se calcula la interpolación final en la dirección vertical, utilizando el parámetro `v`. Todas estas interpolaciones lineales se realizan empleando la función de OpenGL `mix(inicio, fin, peso)`. Para finalizar, se multiplica la posición interpolada del vértice con la matriz MVP. El resultado se guarda en `gl_Position`. Esta variable interna de OpenGL define la posición final del vértice en el espacio normalizado. Notar que este shader no tiene ninguna variable de salida `out` que enviar a la siguiente etapa, ya que, al utilizar `gl_Position`, ya se fija la posición final de los vértices a rasterizar.

**Listado 2.4:** Ejemplo de Tessellation Evaluation Shader

```

1 #version 400
2 //quads en clockwise con espaciado par
3 layout(quads, fractional_even_spacing, cw) in;
4
5 in vec3 TCPosition[]; //Entrada al TCS (viene del TCS)
6 uniform mat4 MVPMatrix; //Projection * View * Model
7
8 void main() {
9 //Coordenadas parametricas de este vertice
10    float u = gl_TessCoord.x;
11    float v = gl_TessCoord.y;
12
13 //Interpolacion bilineal de la posicion de este vertice
14    vec3 p1 = mix(TCPosition[0], TCPosition[3], u);
15    vec3 p2 = mix(TCPosition[1], TCPosition[2], u);
16    vec4 position = vec4(mix(p1, p2, v), 1.0);
17
18 //Posicion final en pantalla
19    gl_Position = MVPMatrix * position;
20 }
```

## Resumen

Como se puede ver, una vez que se comprende la utilización de los parámetros y la influencia que tienen en el resultado, la tecnología de teselación en GPU es simple de utilizar. La mayoría del trabajo lo realiza el Tessellation Primitive Generator, del que sólo se deben configurar algunos parámetros como el tipo de primitiva paramétrica (triángulos, cuadriláteros o poligonales) y el método de posicionamiento de vértices (simétrico, par o impar). Las principales tareas del programador son definir los niveles de teselación y ubicar los nuevos puntos generados según sus coordenadas paramétricas.

En el presente proyecto, el interés se concentró en desarrollar aplicaciones de teselación e ilustrar los distintos métodos en que se pueden calcular los niveles de teselación. Luego de adquirir destreza en el uso básico de teselación, el énfasis se dirigió a diseñar e implementar métricas que definan el nivel de teselación mediante métodos analíticos y heurísticos adecuados a cada caso particular.

## Capítulo 3

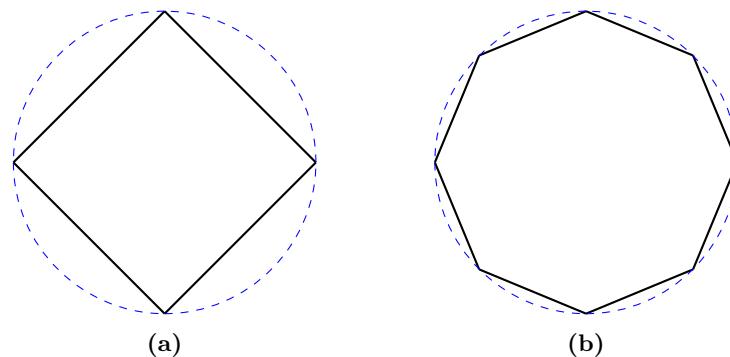
# Aplicaciones

En este capítulo se realiza una breve exposición sobre las posibles aplicaciones de la tecnología de teselación en GPU. Se discute en qué áreas se puede utilizar, en cuáles no, y cuáles son sus limitaciones.

El hardware de video sólo opera con primitivas lineales simples: puntos, líneas y triángulos, donde cada primitiva está en un mismo plano euclídeo. En el proceso llamado rasterización, un objeto geométrico se convierte en un conjunto de puntos ráster (fragmentos, y luego píxeles) que se utilizan para dibujar el objeto en pantalla. La representación fiel del objeto en pantalla depende de su representación lineal vectorial. Es decir, depende de cómo se represente su forma, la cual puede estar dada con cualquier tipo de representación computacional, con primitivas lineales. Para obtener una buena aproximación en pantalla, es necesario tener una buena representación geométrica. Sin embargo, debido a la naturaleza lineal de las primitivas, éstas no pueden representar -por sí mismas- objetos curvos.

Para reproducir un objeto curvo, por ejemplo un cilindro, se construye un modelo geométrico denominado malla. Una malla está formada por un conjunto de elementos que normalmente son triángulos. Con frecuencia, también se utilizan otras primitivas más complejas como cuadriláteros o polígonos, pero en un paso previo a la rasterización se descomponen en triángulos. En este caso, se habla de una discretización del objeto curvo, ya que la superficie curva es muestrada en un conjunto de puntos discretos con los que se forman los elementos de la malla. La fidelidad en la representación del objeto depende de la cantidad y disposición de las primitivas. En la Fig. 3.1 se ilustran dos aproximaciones distintas, a través de líneas, para un círculo. En la Fig. 3.1a, el círculo se renderiza con 4 líneas. En cambio, en la Fig. 3.1b, se utilizan 8 líneas para representar el mismo círculo. Se puede ver que la segunda aproximación, al tener más muestras equiespaciadas, logra una representación más fiel de la forma real del círculo.

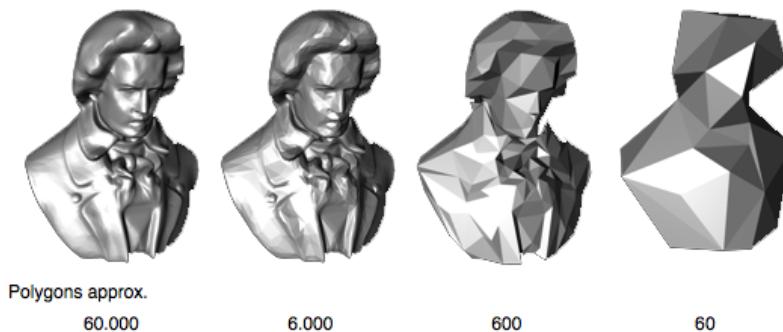
En algunos casos, es necesario tener un grupo de representaciones del mismo objeto, cada una con más detalle que la anterior. Esto se realiza para que, du-



**Figura 3.1:** Ejemplo de dos aproximaciones distintas para renderizar un círculo.

rante el renderizado, se pueda cambiar de representación según la demanda. En computación gráfica, esto es llamado *Nivel de detalle* (LoD por *Level of Detail*). Al tener en cuenta el LoD, se aumenta o disminuye la calidad de la representación de un objeto según su importancia relativa en escena. Por ejemplo, si el objeto está alejado del observador, no es necesario tener una representación densa (compuesta por gran cantidad de primitivas) ya que no podrá ser apreciada. En contraposición, si el objeto está en primer plano de la escena, se debe utilizar la mejor representación disponible del objeto.

Si no se utiliza teselación, el LoD se implementa cambiando el modelo según su importancia. Por ejemplo, si se tiene un modelo de 60000 triángulos para la representación más fiel, y otro modelo de 60 triángulos con una reproducción *gruesa*, el primer modelo se usará para los primeros planos mientras que el segundo será utilizado en los momentos que el objeto se encuentre en el fondo. Para lograr este efecto, el artista, que es quien construye los modelos, debe duplicar su esfuerzo. En la Fig. 3.2 se expone un ejemplo de un modelo construido con cuatro niveles de detalle diferentes. Debajo de cada modelo se presenta la cantidad de polígonos que lo constituyen.



**Figura 3.2:** Ejemplo de distintos niveles de detalle para un modelo.

La tecnología de teselación se utiliza para ayudar en la aplicación de LoD. Por un lado, reduce la labor del artista ya que el refinamiento del modelo (inserción de nuevas primitivas en su representación) se realiza en GPU dinámicamente.

Por otro lado, permite disponer de un gran conjunto de distintas representaciones del objeto (no sólo cuatro como en el ejemplo anterior). En el caso sin teselación, si se desea tener por ejemplo ocho representaciones con distinto detalle, el artista debe construir ocho modelos distintos del mismo objeto. En teselación, cualquier representación (y de cualquier calidad) se construye *al vuelo*. Entonces, la tecnología de teselación brinda la posibilidad de disponer de LoD continuo, es decir, permite cualquier grado de refinamiento (hasta un cierto límite impuesto por el hardware).

Según cada modelo en particular, se necesita una subdivisión ‘adecuada’ del mismo. El hecho de que una subdivisión de un modelo sea adecuada tiene un significado distinto según el contexto. Por ejemplo, puede ser requerido que se refine el modelo completo, o puede ser necesario refinar sólo una parte del mismo, dejando el resto sin modificaciones.

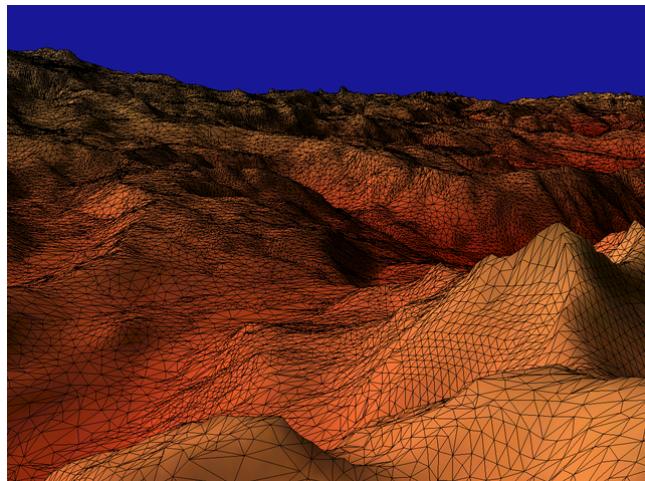
Además, el tipo y forma de subdivisión también dependen de la malla y del objeto geométrico que se intenta representar. Por ejemplo, un objeto puede estar representado directamente por una malla de triángulos, o puede estar formado por un grupo de superficies paramétricas que luego se deben muestrear y formar una malla. En el primer caso, la decisión de subdivisión se realiza sobre la malla de triángulos directamente, como por ejemplo en base a la orientación o tamaño de cada elemento. En el segundo caso, en cambio, la decisión debe hacerse en base a características propias de la superficie, como su curvatura. Una parte importante de este proyecto consistió en diseñar métricas que ayuden a decidir el grado y forma de refinamiento a utilizar en cada modelo en particular.

En las primeras etapas de este proyecto, se hizo un análisis sobre las áreas de aplicación de teselación en GPU. Lo que sigue es un breve repaso de las conclusiones obtenidas en este análisis.

La primera aplicación analizada fue la visualización de terrenos utilizando un modelo de elevaciones. Un modelo de elevación de terreno es un modelo geométrico de grandes dimensiones. En este contexto, se dice que el modelo es grande porque la variación en el plano es mucho mayor a la variación en altura ( $dx, dz$  mucho mayores a  $dy$ , siendo  $y$  la dirección vertical). Este modelo está formado por un gran conjunto de primitivas cuyos vértices están desplazados verticalmente en distinta medida. Los desplazamientos verticales pueden tener distinto significado. Por un lado, por ejemplo, en el caso de utilizar el terreno para estudios de suelo ante inundaciones, pueden representar directamente la altura del suelo. Por otro lado, se le puede atribuir un significado físico a los desplazamientos, por lo que se puede definir que diferentes alturas representen distintas temperaturas. En ambos casos, al refinar el modelo se obtiene una mejor discriminación del fenómeno analizado y, por lo tanto, se pueden apreciar mejor las variaciones locales. Un modelo de elevación de terreno, donde la elevación representa una altura, también es utilizado en simuladores de vuelo para entrenamiento de pilotos, y en videojuegos. En estos casos, el interés está en tener una representación de la escena lo más realista posible para ayudar a la inmersión del piloto o del jugador en la actividad.

En todos los casos mencionados, debido a que por lo general se trata de

un modelo muy grande, la subdivisión del terreno se debe realizar en donde esté concentrada la vista. Una métrica simple para este caso es la medida de la distancia de las primitivas al punto de visión. No se desea que la representación sea densa para lugares alejados del espectador, pues allí no está concentrada la vista. Por otra parte, para zonas cercanas, se debe utilizar una discretización mayor. En la Fig. 3.3 se presenta un ejemplo de renderizado de un terreno. Se puede apreciar que está formado por gran cantidad de polígonos, por lo que se hace necesario un manejo adecuado del nivel de detalle.



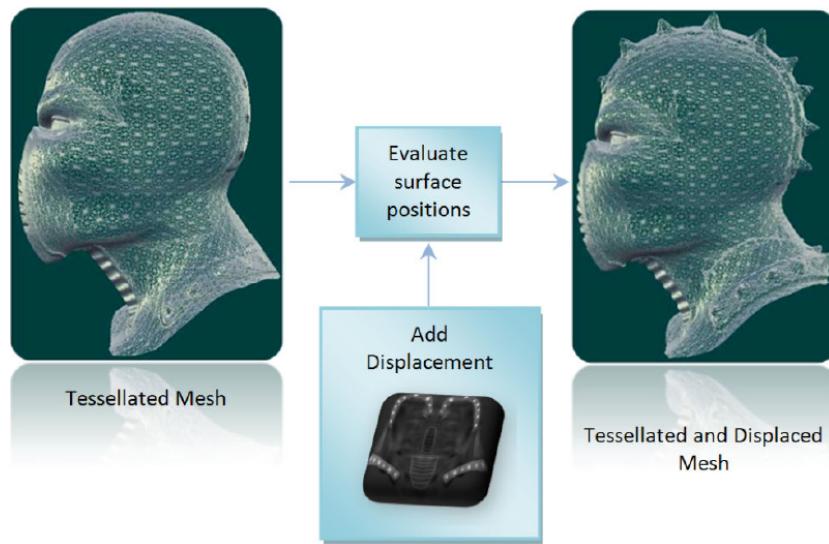
**Figura 3.3:** Ejemplo de renderizado de un terreno.

Otra área de interés para el uso de teselación se encuentra en los modelos tridimensionales utilizados en videojuegos. En un videojuego, estos modelos pueden representar personajes. El interés aquí está en obtener la mejor representación posible del personaje. Sin embargo, al refinar un elemento, los nuevos vértices generados en la teselación se insertan en el mismo plano del elemento. Esto no es de utilidad para mejorar la representación, pues no se obtiene ningún beneficio en la aproximación. Un paso posterior consiste en desplazar los nuevos vértices a una posición más adecuada para lograr una reproducción más precisa del personaje.

Aquí se identifican dos líneas para calcular los desplazamientos, las cuales dependen de la información disponible. En primer lugar se tienen los personajes que se representan por una malla gruesa y un mapa de desplazamientos. Normalmente, un mapa de desplazamientos es una textura bidimensional cuyo valor en cada posición representa un desplazamiento. Cada vértice de la malla tiene asignada una coordenada de textura que permite identificar el desplazamiento para ese vértice. Utilizando este valor y la dirección de la normal en dicho vértice, se modifica la posición geométrica del mismo. Por lo general, un mapa de desplazamientos tiene mucha más información de la que realmente se utiliza. Esto es debido a que las coordenadas de textura que se asignan a los vértices sólo toman un conjunto limitado de muestras del mapa. Si el modelo se refina, se obtienen más vértices donde muestrear el mapa de desplazamientos. Por lo tanto, se logran desplazamientos más precisos en el interior de las primitivas

iniciales y se pueden representar mejor los detalles del modelo.

En la Fig. 3.4 se exhibe un modelo ya teselado y el efecto que produce aplicarle un mapa de desplazamiento adecuado. En este caso, se puede apreciar cómo se logran detalles adicionales con la información de desplazamientos. En la parte inferior de la figura se muestra la textura que es utilizada para los desplazamientos. En esta textura, los colores cercanos al blanco indican desplazamientos grandes.

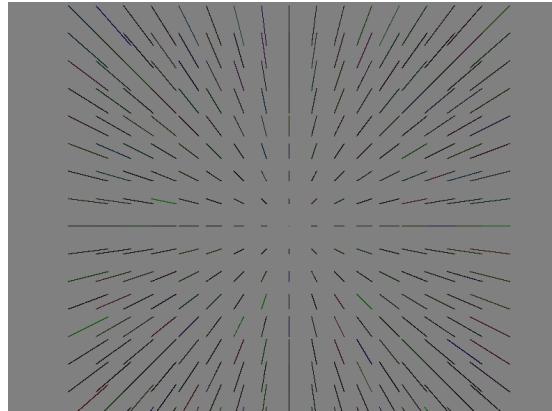


**Figura 3.4:** Ejemplo de modificación de geometría utilizando teselación y un mapa de desplazamientos.

La segunda línea identificada en el caso de personajes se utiliza cuando el modelo no puede representar todas las superficies curvas del personaje, y no se cuenta con un mapa de desplazamientos. En este caso, la discretización del modelo no alcanza a captar todos los detalles y es necesario un refinamiento para recuperar la curvatura. Por ejemplo, si se piensa en una esfera discretizada en triángulos, se puede aumentar la densidad de elementos para así poder representar con más precisión la superficie de la misma. Una vez que se ha refinado, los vértices nuevos deben desplazarse en dirección y módulo radial para adecuarse a la curvatura de la esfera. En casos generales, el dato del radio no está disponible. En estas situaciones, un enfoque posible es calcular una superficie paramétrica en un vecindario, y mover los vértices nuevos para que estén sobre la superficie. Se espera que esta superficie sea representativa de la curvatura del objeto en dicho vecindario.

Otra área de interés es la visualización de campos escalares y vectoriales. Ya se mencionó la visualización de campos escalares en una representación de terrenos. Otra posibilidad es representarlo por un mapa de colores. En el caso de campos vectoriales, se desea obtener una visualización que permita apreciar el campo que afecta a un área. Por ejemplo, dibujando las líneas de corriente o flechas que muestren el sentido del campo en cada punto. En estos casos, la

teselación se puede utilizar para generar una representación más precisa en un área de interés. Ejemplos posibles de esto último son los lugares donde la vista esté concentrada, o donde el campo tenga cierta característica de interés que se quiera resaltar, tales como las áreas donde el gradiente es alto, o presenta valores altos del rotacional. En la Fig. 3.5 se presenta un ejemplo de este tipo de visualizaciones. Cada línea ilustra el recorrido que realiza una partícula afectada por el campo vectorial que la incide.



**Figura 3.5:** Ejemplo de visualización de campo vectorial con líneas de corriente.

En la concepción del proyecto, una de las principales aplicaciones que se pensó fue la utilización de teselación para mejorar la precisión en cálculos aplicados a ingeniería. Por ejemplo, aprovechar el refinamiento para tener más detalle en donde un fluido tiene más variación. Sin embargo, un análisis completo de la pipeline gráfica durante el desarrollo del proyecto indicó que, en principio, no se puede utilizar teselación para estos cálculos. La principal limitación que se tiene en teselación es que la nueva geometría generada se descarta en cada paso por el pipeline, y no es posible realimentar esas primitivas al proceso. Esto último es necesario en el caso de cálculo y simulación, ya que las interacciones en un instante de tiempo dependen del resultado de las interacciones en el instante anterior.

Más allá de lo antedicho, no se puede decir que la teselación no sea de utilidad en estos casos. En las últimas modificaciones del pipeline gráfico también se incluyó una etapa de programación de propósito general llamada *Compute Shader (CS)*. Esta etapa es utilizada en GPGPU y está ubicada luego de la etapa de TES. Sin embargo, no es el interés de este proyecto el entender y utilizar el CS en GPGPU, por lo que esta alternativa fue dejada de lado en el proyecto. No se tiene certeza sobre las capacidades de esta etapa, pero al ubicarse luego de la teselación, se espera que tenga acceso a toda la información nueva generada. Trabajos futuros podrán retomar esta línea y utilizar teselación para cálculo.

Otra aplicación que se consideró al iniciar el proyecto fue la utilización de teselación en superficies de subdivisión. Una superficie de subdivisión es un método para representar una superficie suave a partir de una representación gruesa de la misma. Esta representación es afectada por un conjunto de subdivisiones,

donde en cada paso se agregan y mueven vértices según un algoritmo específico. Por ejemplo, con este método se logra transformar, luego de pocos pasos de subdivisión, un cubo en una esfera. El objetivo aquí es decidir cuántos pasos de subdivisión efectuar y ubicar los nuevos vértices generados en la teselación de las primitivas originales.

Sin embargo, en los algoritmos de subdivisión, la posición de un vértice en un paso depende de la posición de un grupo de vértices vecinos en el paso anterior. En la pipeline gráfica, como ya se dijo, no se puede disponer de información previa. Además, cada primitiva se procesa en paralelo y no se tiene información sobre el resultado del resto de las primitivas. Estas limitaciones impiden el uso de teselación aplicada a superficies de subdivisión. Una forma posible de poder utilizar teselación en este área sería si se podría calcular la posición final de todos los vértices generados luego de cierta cantidad de pasos. En un análisis rápido, se concluyó que este no es el caso, pues al menos se necesita información sobre la subdivisión de las primitivas vecinas en un gran vecindario, y esta información no está disponible. De todos modos, se deja esto como un posible trabajo futuro, siempre que se encuentren métodos para poder generar la superficie final (luego de muchos pasos de subdivisión) a partir de la inicial en un sólo paso.

La tecnología de teselación, desde su misma definición, sólo permite aumentar la geometría existente. En la actualidad, no existe ningún mecanismo en GPU para reducir la geometría. Esta reducción, también llamada desrefinamiento, es utilizada para disminuir la carga de procesamiento. Por lo tanto, si se tendría disponible la capacidad de desrefinamiento en GPU, se tendría todo el control necesario para regular el nivel de detalle de un modelo. Sin embargo, no es el caso y por lo tanto se debe utilizar una malla básica a partir de la cual se aumenta la geometría, pero nunca se puede reducir.

La única reducción posible es el descarte completo de primitivas. Se puede pensar en un algoritmo de teselación que tome un vecindario de triángulos y los reemplace por un único triángulo según cierta métrica (por ejemplo, una medida de la superficie ocupada), pero dicho algoritmo sería muy básico y no tendría la misma flexibilidad que la teselación para creación de geometría.

---

Origen de las imágenes externas utilizadas:

1. OpenSG, “*Node cores*”. Disponible en <http://www.opensg.org/projects/opensg/wiki/Tutorial/OpenSG2/NodeCores> [Consultado el 09-07-2014].
2. Cignoni, P., Ganovelli, F., Gobbetti, E., Marton, F., Ponchio, F. y Scopigno, R., “*BDAM! Batched Dynamic Adaptive Meshes for high performance terrain visualization*”. Computer Graphics Forum, 22(3). 2003. Disponible en <http://vcg.isti.cnr.it/activities/surfacegrevis/bdam/bdam.htm> [Consultado el 09-07-2014]
3. Tatarchuk, N., Barczak, J., y Bilodeau, B., “*Programming for real-time tessellation on GPU*”. AMD whitepaper 5 (2010).

## Capítulo 4

# Métricas

Como fue explicado en el capítulo 2, la tecnología de teselación en GPU permite subdividir las primitivas originales en primitivas más pequeñas, y operar con éstas por separado. En este capítulo se presentan los métodos diseñados para definir el grado de refinamiento que se aplica a cada patch de la escena.

En el contexto de este trabajo, una *métrica* es una función que, dada cierta información interna y externa de un patch, calcula el grado de refinamiento a aplicar en el patch. Con información interna se refiere a datos que son parte del patch, como la posición de los vértices o sus normales. En contraposición, información externa es aquella que depende de la escena o de la entrada del usuario. El principal dato externo que se utiliza en el presente trabajo es la posición de la cámara.

El grado de refinamiento se traduce en los niveles internos y externos de teselación que se deben configurar en el TCS. Se recuerda aquí que los niveles de teselación son valores flotantes en el rango  $[1, \text{GL\_MAX\_TESS\_GEN\_LEVEL}]$  y, dependiendo de la primitiva paramétrica utilizada, tienen distinta interpretación.

Cada métrica se diseña teniendo en cuenta el modelo tridimensional en el que se quiere aplicar y las características que se desean resaltar. En este proyecto se desarrollaron distintos modelos tales como curvas y superficies paramétricas, modelos de personajes de videojuegos y terrenos discretizados en triángulos. Todos estos modelos están constituidos por patches de distinta cantidad de vértices, según como se mencionará más adelante. Cada patch de estos modelos, al pasar por el pipeline gráfico, es refinado según los niveles de teselación calculados en la métrica aplicada.

En este proyecto se diseñaron 4 métricas que toman en consideración distintos aspectos del modelo. Estas métricas son: distancia al ojo, grado de curvatura, longitud de arista y ubicación en silueta. En una sección posterior se explican exhaustivamente cada una de estas métricas.

A nivel general, se puede realizar una clasificación de métricas en dos grupos:

- *Intrínsecas*: Se pueden calcular para cada patch sin tener en cuenta información externa al mismo. Por ejemplo, el grado de curvatura que tiene una curva.
- *Extrínsecas*: Para poder calcularlas, se necesita información del contexto de la escena. Ejemplo de esto es la métrica de distancia al ojo, ya que además de la posición del patch, se necesita la posición del punto de vista (cámara).

En la búsqueda y diseño de métricas, se deseó respetar tres propiedades fundamentales:

- *Simplicidad*: Debido a que el cálculo del nivel de teselación se debe realizar para cada primitiva de cada modelo de la escena, se buscó que este cálculo sea lo suficientemente simple para que no se convierta en un ‘cuello de botella’ en el proceso de renderizado. Esto es, que el cálculo no sea complejo para que la GPU lo pueda hacer en pocos ciclos de procesamiento.
- *Poca Información*: Mientras que la capacidad de cálculo en paralelo de la GPU es una de sus principales características, la transferencia de datos de la CPU a la GPU es más lenta. Con esta característica se pretende que la métrica no tenga dependencia de muchos datos y que pueda operar solo con la información local y/o de contexto.
- *Adaptatividad*: Esta última característica hace referencia a que la métrica debe depender del estado específico de la escena en cada cuadro de dibujado. Si la métrica se puede calcular una única vez al comenzar el renderizado, no es adaptativa y por lo tanto no se está tomando ventaja del nuevo pipeline gráfico, ya que se podría hacer en CPU.

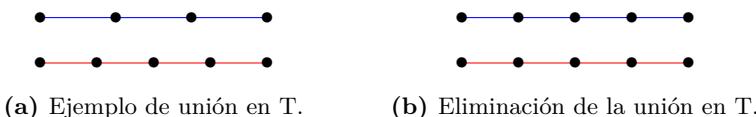
La tercer característica es la más importante para este proyecto. Esta característica es la principal razón por la que se emplea teselación en GPU. Al utilizar el TCS, se puede realizar teselación adaptativa en tiempo real. Esto quiere decir que al ejecutar el software y mostrar la escena, en cada cuadro de renderizado se calculan los niveles de teselación para cada patch y, por lo tanto, se ‘adapta’ según las características de la escena en cada instante de tiempo. Esto se hace posible por el gran paralelismo con que trabaja la GPU. Antes de disponer de la tecnología en GPU, realizar teselación y calcular métricas en tiempo real era un proceso más lento porque se debía realizar en CPU (que en general es más rápido pero tiene un limitado paralelismo). Además, todos los datos nuevos generados debían ser almacenados en la memoria principal, lo que también aumentaba la carga general del software.

Al subdividir un polígono en primitivas más pequeñas, se obtiene una discretización más fina del modelo. Al tener una mejor discretización, se puede discriminar con más detalle los cálculos a realizar en cada vértice. Por ejemplo, se puede modificar la posición de todos los vértices originados en la teselación y proyectarlos a una esfera, de manera de generar una superficie más curva.

Si bien la implementación de OpenGL tiene un límite superior para los niveles de teselación, con frecuencia es necesario limitar este valor en la métrica. La generación de nuevas primitivas aumenta la carga del procesador y por lo tanto puede reducir el rendimiento general del software. Por ejemplo, con niveles de teselación iguales a 64, un patch que representa un cuadrilátero se subdivide en más de 8000 triángulos. Por lo tanto, lo que antes era una única primitiva, luego serán más de 8000 primitivas, donde cada una debe transitar por el pipeline gráfico y ser dibujada en pantalla. Según la potencia de la placa aceleradora de video, este aumento de carga puede afectar el rendimiento final ya que se tiene que procesar mucha mas información. En definitiva, con frecuencia es necesario poner un límite superior  $t_{tessmax}$  a los niveles de teselación calculados. Este límite debe tener en cuenta el hardware específico en que se ejecuta el software y las necesidades del problema en particular.

Otra corrección que se puede aplicar en la métrica es el uso de una corrección diádica. Esta corrección modifica el nivel de teselación para que sea una potencia de 2. Para ello, dado un nivel de teselación  $n$ , éste se lleva a la potencia de 2 más cercana. Esta corrección se realiza para reducir las conexiones ‘en T’. Recordar que los niveles de teselación externos se calculan por aristas. Si dos primitivas adyacentes que comparten una arista, definen niveles de teselación distintos en la arista compartida, los nuevos vértices no coincidirán. Esto produce una conexión ‘en T’: un vértice generado en la teselación de un patch pero que no fue creado en la teselación del otro patch. Esta situación no es deseable porque produce discontinuidades en el modelo que afectan a la calidad percibida. Al utilizar corrección diádica, las conexiones en T son reducidas porque se limita la cantidad de valores distintos que se puede tomar. Por ejemplo, si dos primitivas adyacentes tenían niveles de teselación 18 y 20, la corrección modifica estos valores para que sean iguales a 32. Si no se realizaría este cambio, se encontrarán conexiones en T en todos los vértices de la arista compartida por ambas primitivas.

En la Fig. 4.1 se presenta un ejemplo de esta corrección. Se tienen dos aristas subdivididas de patches adyacentes. En la Fig. 4.1a, se define un nivel de teselación igual a 4 para la arista inferior, mientras que el nivel de la arista superior es igual a 3. Se puede ver que existen uniones en T ya que los vértices internos de ambas aristas no coinciden. En la Fig. 4.1b se aplica la corrección diádica y se modifica el nivel de teselación de la arista superior, llevándolo de 3 a 4. Con esta modificación ya no se tienen uniones en T en la interfaz entre las aristas.



**Figura 4.1:** Ilustración de cómo la corrección diádica reduce las uniones en T.

Notar, sin embargo, que si la corrección se hace hacia arriba, se configurará un nivel de teselación mayor al calculado, aumentando la carga general. En otro caso, si la corrección se hace hacia la potencia de 2 inmediatamente menor, se

reduce la carga pero también se puede afectar la calidad, ya que se generan menos primitivas que las calculadas en la métrica.

En general, toda métrica que se diseñe debe intentar, en su definición, no generar uniones en  $T$ . Esto se logra al realizar los cálculos de los niveles de teselación externos con respecto a las aristas del patch. Por ejemplo, para evitar las uniones en  $T$  en una malla de cuadriláteros, cada uno de los 4 niveles de teselación externos (un por cada arista) del cuadrilátero se debe calcular con respecto a la información contenida en los dos vértices que representan la arista. Si la malla es conexa, el cuadrilátero adyacente tendrá la misma arista como uno de sus lados, y por lo tanto se calculará el mismo nivel de teselación en la arista compartida.

## 4.1. Métricas implementadas

En esta sección se explican las métricas que fueron diseñadas, desarrolladas e implementadas como algoritmos dentro de un programa de shader. Cada métrica fue aplicada a uno o más modelos tridimensionales. En la siguiente discusión, los términos posición del ojo, cámara y punto de vista, se utilizan indistintamente para referirse a la posición de la cámara en una proyección perspectiva. De las métricas diseñadas, sólo la métrica de curvatura es una métrica intrínseca. El resto de las métricas dependen de la posición del observador, por lo que esto las convierte en extrínsecas.

### 4.1.1. Distancia a la cámara

La primer métrica que se diseñó es también la más simple. Se trata de una métrica extrínseca que le da mayor importancia a patches que estén cercanos al ojo (posición de la cámara). Por otra parte, los patches que están alejados del ojo son afectados por un menor o nulo refinamiento. Se supone que si una primitiva está alejada del ojo, subdividirla en primitivas más pequeñas no generará una variación en el resultado final percibido, ya que el cambio no será apreciado por el usuario. Por otra parte, subdividir polígonos cercanos a la cámara es de utilidad para tener una mejor discretización del modelo y permite realizar cálculos más exactos de iluminación u obtener mejores interpolaciones para texturas, oclusión, etc.

En general, no se realiza descarte de primitivas en el TCS (configurando un nivel de teselación igual a 0), por lo que los niveles de teselación siempre serán mayores o iguales a 1. Sin embargo, puede pasar que una primitiva esté cerca del ojo pero no sea visible y sea necesario descartarla. Este tipo de descarte se puede realizar en el Geometry Shader para cada primitiva a rasterizar, o en el Fragment Shader para cada fragmento.

Para calcular la distancia, se experimentaron tres alternativas. La primera

determina la distancia a un polígono como la distancia del observador a su centroide. Esto luego fue modificado y se pasó a definir la distancia a un polígono como la distancia entre el observador y el vértice más cercano del polígono. La tercera alternativa, considerando que los niveles de teselación se configuran por arista, consiste en calcular el nivel de teselación de cada arista en función de la distancia del observador al punto medio, o al más cercano, de la arista. Esta métrica, la cual tiene en consideración la característica de teselación por arista del hardware, permite asignar un nivel distinto para cada lado. Esto último no ocurre con las otras alternativas, ya que sólo especifican un nivel de teselación general para todas las aristas de la primitiva. Además, se garantiza que no habrá uniones en T, pues las primitivas que comparten aristas tendrán el mismo nivel de teselación asignado a través de ese lado. Esta última es la finalmente implementada, pues evita desde raíz las uniones en T en las interfaces entre elementos. En el interior de la primitiva, el nivel de teselación se calcula en función de la distancia a su centroide.

Una vez obtenida la distancia  $d$ , ésta se debe mapear a un nivel de teselación. Intuitivamente, se quiere que para una distancia mayor a  $d_{max}$ , el nivel de teselación sea igual a 1. Para una distancia  $d_{min}$  cercana a 0, se desea que el nivel de teselación sea  $t_{tessmax}$ , el máximo. En el interior de este rango, el nivel de teselación decrece linealmente. Por lo tanto, se calcula el nivel de teselación  $t_i$  para una distancia  $d$  como:

$$t_i = 1 + \frac{(t_{tessmax} - 1)(d_{max} - d)}{d_{max} - d_{min}} \quad (4.1)$$

La función que define la métrica, y los valores  $d_{max}$ ,  $d_{min}$  y  $t_{tessmax}$  son elecciones del programador. Por ejemplo, en lugar de una función lineal, se podría utilizar una función cuadrática. Asimismo, los límites se deben definir según las características que el programador desee resaltar y dependen de cada modelo o aplicación en particular.

Una alternativa a este cálculo de distancia es utilizar una función lineal inversa decreciente. Esta función tiene la forma  $y = -b + \frac{a}{x}$ , donde  $a$  y  $b$  dependen de  $d_{max}$  y  $d_{min}$ . Lo que se busca con esta función es imitar el comportamiento del z-buffer de video, el cual tiene más resolución (más valores distintos para asignar) para fragmentos cercanos a la cámara (cercaos a  $d_{min}$ ), y menos para los alejados (cercaos a  $d_{max}$ ). En este caso,  $d_{min}$  se asigna igual al valor de z-near. El valor z-near mide la distancia de la cámara al plano donde se proyecta la escena. Para distancias menores a z-near, no se renderizan polígonos en pantalla.

La función que utiliza el z-buffer [1] tiene la forma:

$$f_1 = -\frac{d_{max} + d_{min}}{d_{max} - d_{min}} + \frac{1}{d} \frac{2 d_{max} d_{min}}{d_{max} - d_{min}} \quad (4.2)$$

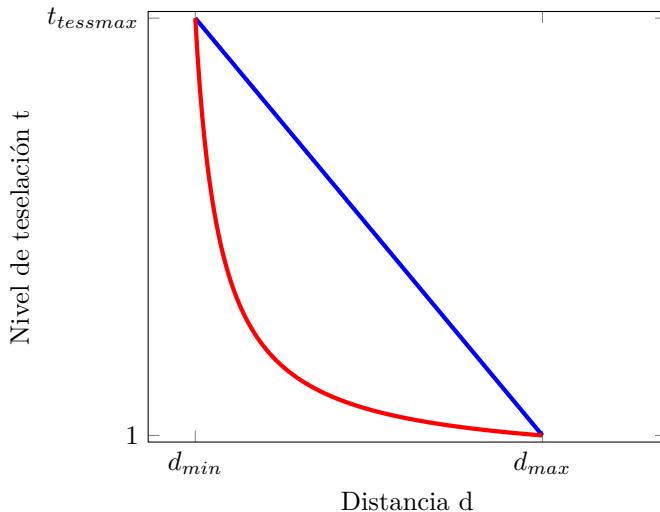
En el caso del z-buffer, cuando la proyección es perspectiva central, el valor  $d_{min}$  se corresponde con el z-near y el valor  $d_{max}$  con el z-far.

A continuación se desarrolla el método de obtención de una función lineal inversa decreciente para realizar el mapeo de una distancia a un nivel de teselación.

Primero se obtiene el valor de  $f_1$  para una distancia  $d$  según la Ec. 4.2. Luego se debe mapear este valor  $f_1 \in [-1, 1]$  a niveles de teselación  $t \in [1, t_{tessmax}]$ . Este mapeo es lineal y se realiza de la siguiente manera:

$$\begin{aligned} t &= \frac{t_{tessmax} - 1}{2} f_1 + \frac{t_{tessmax} + 1}{2} \\ \Rightarrow t &= -\frac{[(t_{tessmax} - 1) d_{max} - d t_{tessmax}] d_{min} + d d_{max}}{d (d_{min} - d_{max})} \end{aligned} \quad (4.3)$$

En la Fig. 4.2 se muestra la comparación entre ambas funciones. Se puede ver que mientras la función lineal (azul) provee una transición lineal en todo el dominio, la función inversa (roja) brinda más rango de niveles de teselación para distancias menores. En un amplio rango de distancias mayores, se define el mismo nivel de teselación.



**Figura 4.2:** Comparación entre función lineal y función lineal inversa.

#### 4.1.2. Curvatura

Debido a que el hardware gráfico sólo rasteriza primitivas lineales (puntos, segmentos y triángulos), es necesario linealizar el objeto de modo que la rasterización genere una representación adecuada del mismo. En esta sección se expone un método original para definir un nivel de refinamiento adecuado, basado en la curvatura del objeto. Se pretende que una superficie cilíndrica, por ejemplo, defina una métrica para subdivisión que sea independiente del radio, es decir, que sólo importe fijar cuántos segmentos periféricos se deben usar para definir una circunferencia. Por lo tanto, no se utilizará la curvatura en forma cruda, si no dividida por alguna longitud normalizadora relacionada con el radio en el caso del cilindro. Más allá de lo antedicho, siempre que se mencione el término de curvatura en el texto, se debe entender que se refiere a este valor específico calculado.

En particular, este método se aplica a curvas y superficies paramétricas que se definen a partir de un conjunto de puntos de control y uno o más parámetros. Primero se expone el caso para curvas y luego la extensión a superficies, pero el método aplicado en las curvas está diseñado de modo que su extensión a superficies sea directo.

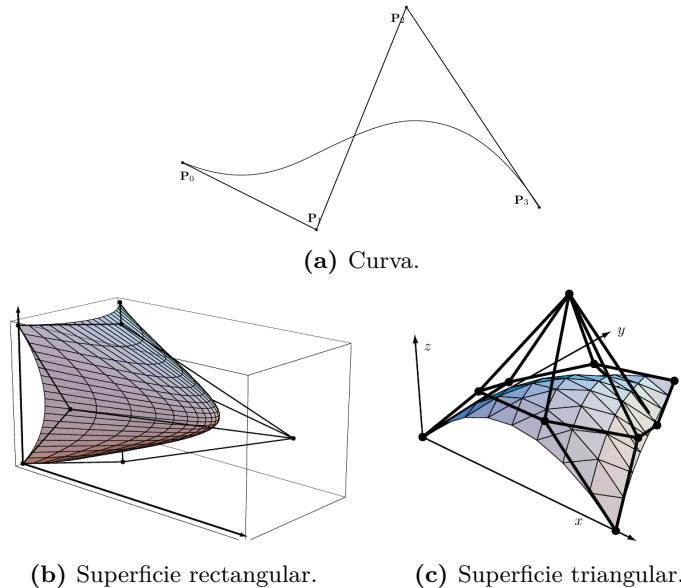
Para una curva, la linealización consiste en aproximarla mediante una poligonal, buscando que el error de aproximación sea a) acotado y, en lo posible, b) constante. La imposición a) refiere a que se desea que la diferencia (error) entre la curva real y la aproximación tenga una cota conocida de antemano. Con respecto a b), se busca que esta cota sea constante a lo largo de toda la curva, de manera de no producir un refinamiento inadecuado tanto por exceso como por defecto.

Sin embargo, el requerimiento de error acotado es el que domina en la métrica diseñada. En este sentido, si la curva tiene mucha curvatura en un intervalo, se desea refinirla para acotar el error en el mismo, aunque en el resto de la curva el error sea despreciable. Por ello, el método presentado a continuación tiende a refinar por exceso una curva que sólo presenta un valor alto de curvatura en un tramo pequeño de la misma. La causa de lo anterior es que a pesar de que la métrica de curvatura se calcula por tramos, el refinamiento definido se realiza a lo largo de todo el intervalo. Finalmente, la idea es obtener un nivel de teselación que logre estos objetivos en forma aproximada, diseñando una métrica que defina la cantidad de puntos a evaluar.

En una curva paramétrica  $\mathbf{P}(t) = \{x(t), y(t), z(t)\}$ , las coordenadas se definen a partir de un único parámetro  $t$ . Uno de los principales tipos de curvas paramétricas son aquellas que se definen a partir de un conjunto de puntos de control formando un polígono de control. En este tipo se incluyen las curvas de Bézier y su generalización, las curvas racionales y las curvas NURBS [17].

Por otra parte, una superficie paramétrica  $\mathbf{P}(u, v)$  se define a partir de dos parámetros  $u$  y  $v$ , y un conjunto de puntos de control organizados en una grilla o malla regular. Por ejemplo, para un patch cuadrilátero de Bézier de tercer grado, se tienen 16 puntos de control organizados en una malla de 4 filas y 4 columnas, con 4 puntos por lado. Cada fila o columna de puntos de control define una isocurva paramétrica sobre la superficie. La discusión que sigue está enfocada a curvas y superficies de Bézier, y sus generalizaciones: curvas y superficies NURBS. De forma similar, en los patches triangulares, se tiene un conjunto de 3 coordenadas paramétricas  $u, v, w$  donde  $w = 1 - u - v$  y los puntos de control forman una grilla regular triangular.

En la Fig. 4.3 se presentan ejemplos de curvas y superficies paramétricas. Estas imágenes fueron extraídas de [17]. En la Fig. 4.3a se muestra una curva de Bézier de tercer grado con cuatro puntos de control. En la Fig. 4.3b se presenta una superficie racional de Bézier rectangular de segundo grado, con 3 puntos de control en cada dimensión y un total de 9 puntos en su malla de control. Por último, en la Fig. 4.3c se expone un ejemplo de patch triangular de Bézier de grado 3 con 4 puntos de control en cada una de las tres direcciones principales.



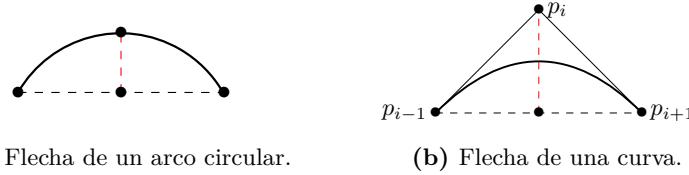
**Figura 4.3:** Ejemplos de patches de Bézier.

La característica que interesa para poder decidir la cantidad de muestras es la curvatura. Esta es una medida que indica el grado de desviación de la superficie respecto a un plano, o de la curva respecto a una recta. La curvatura de una curva paramétrica es un vector y se define para cualquier punto del parámetro. Asimismo, se define la curvatura máxima como el mayor valor de curvatura a lo largo de toda la curva. La expresión para el cálculo de curvatura máxima no es sencillo ya que requiere la obtención de las raíces de un polinomio de grado alto, lo que es computacionalmente costoso. Para superficies, la curvatura es un tensor y lo que interesa son las dos curvaturas principales máximas. En este caso, el cálculo es aún más costoso.

Para evitar uniones en T en el caso de superficies, la métrica que defina el nivel de teselación de la frontera debe depender sólo de la frontera y no de lo que suceda en el interior de la superficie. Para ello se tomarán las medidas de curvatura de las isolíneas extremas.

Tanto para las curvas como para definir el nivel de teselación interior de las superficies, se utilizará una medida que aproxime la desviación de la planaridad. En el caso 1D, la desviación (de la linealidad) se mide como la flecha de la curva. La flecha, o sagita, de un arco circular, se define como la distancia del centro del arco al punto medio de su base (cuerda). Para aproximar la flecha de la curva, se mide la flecha del polígono de control. En la Fig. 4.4 se ilustra la noción de flecha en ambos casos. En la Fig. 4.4a se presenta la definición de flecha para un arco de círculo. Por otro lado, en la Fig. 4.4b se expone el ejemplo de la aproximación de la flecha para una curva a partir de su polígono de control. En ambas figuras, la longitud vertical en rojo representa la flecha de la curva.

En el caso 2D, las superficies bilineales indican que puede haber mucha



**Figura 4.4:** Ilustración de la noción de flecha de un arco circular y la aproximación de la flecha en una curva.

curvatura aún cuando las isolíneas tengan curvatura nula. Esto es porque la curvatura se produce en la diagonal de la superficie bilineal y no en forma paralela a la frontera recta. Para tener en cuenta ambos casos, en superficies se realizan dos medidas: la curvatura diagonal y la curvatura en las isocurvas.

La aproximación elegida para curvas consiste en medir la distancia entre cada punto de control y el punto medio entre el anterior y el posterior. Este valor se divide por la distancia entre el punto anterior y el posterior. El numerador es una medida sencilla y aproximada del orden de magnitud de la flecha que tendrá la curva, y el denominador es el factor de normalización que evita que se considere la curvatura en forma cruda. Esto último tiene la ventaja de que permite definir la curvatura en forma general y ésta no depende de las dimensiones de la curva. De este modo, si las distancias relativas entre los puntos de control varían por un mismo factor mientras que su posición relativa no, la curvatura no cambia, aún aunque se haya modificado el tamaño de la curva.

Se denomina a dicho valor de curvatura  $\kappa^+$  y se define como:

$$\kappa_i^+ = \frac{\left\| p_i - \frac{(p_{i-1} + p_{i+1})}{2} \right\|}{\| p_{i+1} - p_{i-1} \|} \quad (4.4)$$

Este cálculo se repite para todos los puntos intermedios y se mantiene el valor máximo calculado  $\kappa_{max} = \max_{\forall i} (\kappa_i^+)$ .

Resta especificar cómo se traduce la curvatura calculada en niveles de teselación. En primer lugar, no se desea que ninguna curva sea submuestreada, por lo tanto se impone un mínimo de dos puntos por cada intervalo de control en curvas y cuatro interiores para superficies. A partir de ahí, la curvatura actúa linealmente. Intuitivamente si se desea que la desviación no sea mayor a  $\frac{1}{10}$  del ancho, se tendría que agregar un punto por cada décimo de altura. Es decir que  $\kappa_{max}$  quedaría multiplicada por 10. Además, se impone un máximo  $t_{tessmax} = 32$ , pues una curva con excesiva curvatura concentra toda la curvatura en una pequeña porción y no se ganaría demasiado teselando de más. Asimismo, hay un límite impuesto por el hardware.

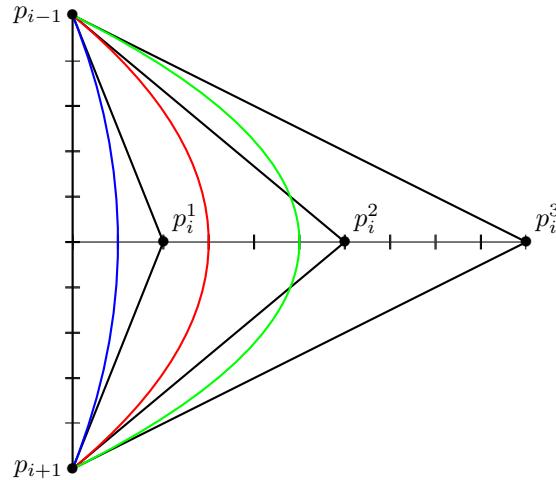
Entonces, el nivel de teselación para una curva es igual a:

$$t_{tesslevel} = \max(2(n - 1), \min(t_{tessmax}, 10\kappa_{max})) \quad (4.5)$$

En esta expresión,  $n$  es la cantidad de puntos de control en la curva. Para  $n$  puntos, hay  $n - 1$  intervalos de control, por lo que la primer parte de esta

expresión indica el mínimo mencionado. Por otro lado, se calcula el mínimo entre el nivel máximo de teselación, y el producto entre la desviación máxima definida y el nivel de curvatura máxima calculado. Para finalizar, el calcula el máximo entre los dos valores calculados, el cual se asigna al nivel de teselación para la curva.

En la Fig. 4.5 se presenta un ejemplo para diferentes curvas de Bézier de segundo grado. El segmento  $\overline{p_{i-1}p_{i+1}}$  se dividió en 10 tramos iguales. Con  $p_i^1, p_i^2$  y  $p_i^3$  se identifican los puntos de control centrales de las distintas curvas dibujadas. Para la curva 1 en azul, se utilizarán dos puntos internos, ya que la curvatura es  $\kappa_{max}^1 = \frac{2}{10}$ . Para la segunda curva (rojo), el valor de curvatura es  $\kappa_{max}^2 = \frac{6}{10}$  por lo que se utilizarán 6 puntos internos. Por último, se utilizarán 10 puntos para representar la curva 3 (verde) ya que  $\kappa_{max}^3 = \frac{10}{10}$ .



**Figura 4.5:** Ilustración del cálculo de la curvatura para curvas de Bézier de segundo grado.

En el caso de superficies paramétricas, el cálculo es similar pero se debe realizar para todas las isocurvas verticales y horizontales, manteniendo dos valores máximos, uno en cada dirección, para la teselación interior. Además, se define un método similar para las diagonales, que en caso de ser mayor que los valores anteriores, reemplaza a ambos.

Primero se calcula  $\kappa_{max}$  para cada una de las curvas de la frontera de la malla de control y se utiliza ese valor para definir el nivel de teselación en los extremos.

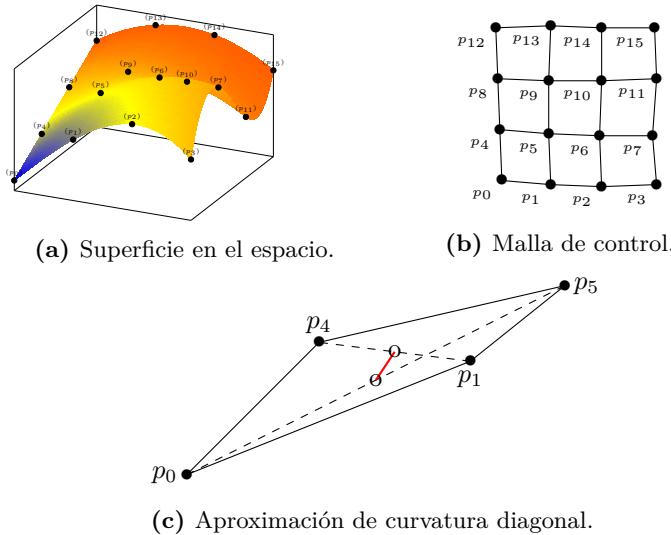
Para aproximar la curvatura diagonal se mide un cociente entre dos valores: la distancia entre los puntos medios de las diagonales dividida por la longitud de la menor (el resultado se puede relacionar con la norma  $L^\infty$  del Hessiano). Se representa a este valor como  $\kappa^\times$  y se calcula como:

$$\kappa_{i,j}^\times = \frac{1}{2} \frac{\|p_{i+1,j+1} + p_{i,j} - p_{i+1,j} - p_{i,j+1}\|}{\min(\|p_{i+1,j+1} - p_{i,j}\|, \|p_{i,j+1} - p_{i+1,j}\|)} \quad (4.6)$$

Cabe aclarar que para los patches triangulares no se requiere computar los valores diagonales  $\kappa_{i,j}^X$  pues están implícitos y son provistos por los valores calculados en las isolíneas  $w = cte$ .

Por ejemplo, para un patch cuadrilátero de superficies con  $m \times n$  puntos de control, en la dirección  $m$  se impone para cada extremo  $\max(2(m-1), \min(t_{\text{tessmax}}, 10 \kappa_{\text{max}}))$  puntos de muestra, incluyendo los extremos, donde  $\kappa_{\text{max}}$  es el máximo calculado en la curva del borde. Del mismo modo, se procede en la otra dirección. Respecto a los valores de teselación en el interior, se considerará el máximo en la dirección  $m$  de todos los interiores, y lo mismo para la otra dirección, o únicamente el mayor diagonal si es un valor superior.

En la Fig. 4.6 se ilustra el cálculo de la curvatura diagonal. En la Fig. 4.6a se muestra una superficie de Bézier de tercer grado. En la Fig. 4.6b se expone la malla de control de esta superficie. Por último, en la Fig. 4.6c se ilustra el cálculo de uno de los valores de curvatura diagonal. La distancia en rojo se divide por la mínima de las dos distancias marcadas con líneas de puntos, que en este ejemplo es la distancia entre los puntos  $p_1$  y  $p_4$ .



**Figura 4.6:** Ilustración del cálculo de la curvatura diagonal.

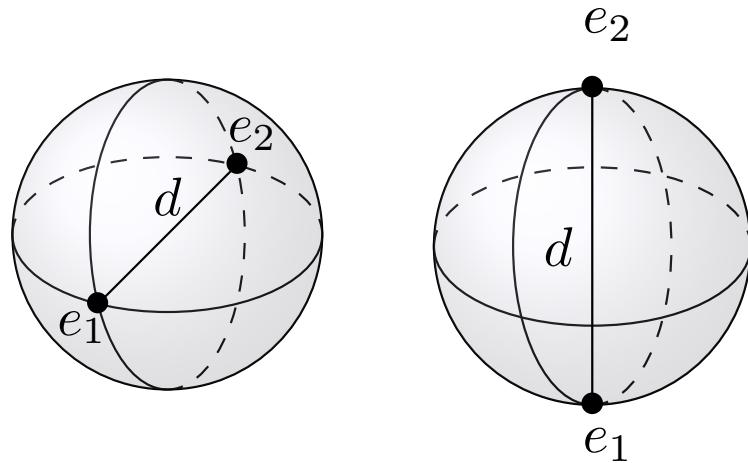
#### 4.1.3. Longitud de arista

Un triángulo está constituido por 3 vértices unidos por aristas. Al rasterizar el triángulo, las aristas se convierten en un conjunto de píxeles en la ventana de rasterizado. Mientras mayor longitud tenga la arista, o más cercana se encuentre al punto de vista, mayor cantidad de píxeles serán necesarios para representarla en el dispositivo de salida. La métrica que se expone a continuación tiene en cuenta esta característica y define un mayor nivel de teselación a una arista que al rasterizarla ocupe gran cantidad de píxeles. Esta es una métrica extrínseca porque, además de medir la distancia entre los vértices de la arista, debe

considerar la proyección y por lo tanto, la distancia del objeto al ojo. Por lo tanto, esta métrica es superior a la métrica de distancia porque tiene en cuenta, además de la distancia, la longitud de la arista.

Se busca definir un valor  $e_{max}$  que represente la cantidad máxima permitida de píxeles a utilizar para representar una arista original. Si se excede este valor, la arista es subdividida de manera de garantizar que todas las nuevas aristas generadas en su lugar respeten el límite de  $e_{max}$ . De esta manera, aristas cercanas recibirán un mayor grado de refinamiento que las alejadas. Debido a que el cálculo de los niveles de teselación de esta métrica se realiza por cada arista por separado, se garantiza que no se generen uniones en T ya que la arista entre dos triángulos adyacentes es la misma (tiene los mismos dos vértices). Para un triángulo se calculan los tres niveles de teselación externos OL por separado. El nivel interno restante IL se define como el máximo de los externos.

Una forma posible de obtener la longitud de una arista una vez rasterizada es calcular la distancia entre sus vértices proyectados, esto es: multiplicados por la matriz MVP. Sin embargo, este cálculo de proyección depende de la orientación de la arista con respecto a la cámara. En cambio, se desea obtener la longitud de la arista proyectada como si ésta estuviese alineada al plano de imagen. Esta *máxima longitud de arista proyectada* no depende de la orientación de ésta, si no que siempre es igual para una arista de la misma longitud a la misma distancia. Para lograr este resultado, se calcula la proyección de una esfera cuyo diámetro es igual a la longitud de la arista. En la Fig. 4.7 se ilustra el fundamento de este cálculo.



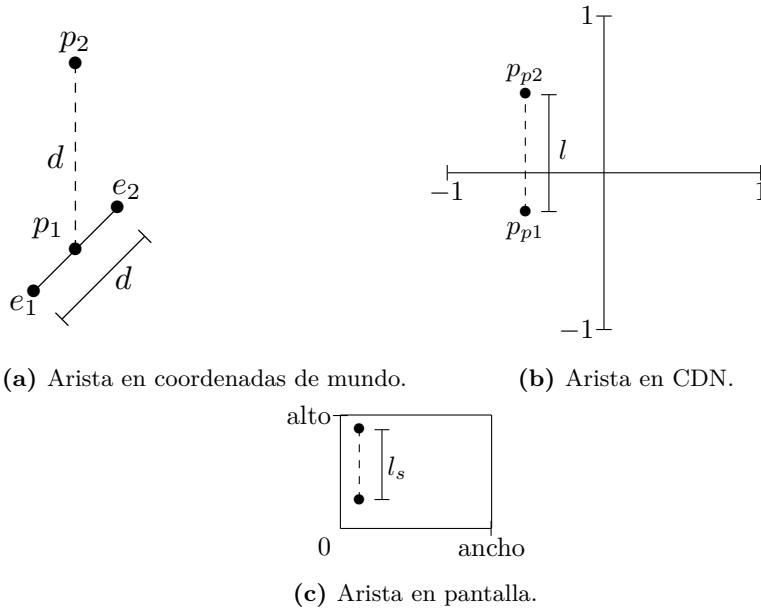
(a) Orientación original de la arista. (b) Orientación corregida de la arista.

**Figura 4.7:** Proyección de una arista como diámetro de esfera.

A continuación se explica cómo se realiza el cálculo de la longitud proyectada para una arista. Para apoyar la explicación, se incluye la Fig. 4.8, donde se muestra el procedimiento completo para poder obtener dicho valor. Dadas una arista  $\overline{e_1e_2}$  y  $d$ , la distancia entre  $e_1$  y  $e_2$ , se calculan dos nuevos puntos. El

primer punto  $p_1$  se define como el punto medio de esta arista. Como se desea calcular la distancia en el plano de imagen, sólo se está interesado en la longitud en el plano XY, descartando la coordenada Z. Por lo tanto, el segundo punto  $p_2$  se ubicará en este plano. Se elige posicionar este punto desplazado verticalmente del primero (coordenada Y). Por lo tanto, el segundo punto  $p_2$  surge de sumar a la coordenada y de  $p_1$ , la distancia  $d$ .

A continuación se utiliza la matriz MVP, la cual depende de la posición de la vista, del modelo, y del tamaño de la ventana, para transformar los puntos  $p_1$  y  $p_2$  a coordenadas de dispositivo normalizado (CDN). Las coordenadas de este espacio normalizado, como su nombre lo indica, están en el rango  $[-1, 1]$  en cada uno de los ejes. Estas nuevas coordenadas se las representa con  $p_{p1}$  y  $p_{p2}$ . El paso siguiente consiste en obtener la distancia  $l$  entre los puntos  $p_{p1}$  y  $p_{p2}$ . Esta distancia se adquiere de la longitud del segmento  $\overline{p_{p1}p_{p2}}$  y se calcula como la resta de las coordenadas y de ambos puntos (en este paso las coordenadas x y z son idénticas). Si ambos puntos están en pantalla, su máxima distancia será el alto de las coordenadas de dispositivo normalizado, es decir igual a 2. Para obtener la distancia final en pantalla  $l_s$  se multiplica  $l$  por la mitad del tamaño de la ventana en píxeles. De esta manera, se convierte  $l \in [0, 2]$ , a un valor  $l_s \in [0, \text{alto}]$ .



**Figura 4.8:** Pasos realizados para calcular la máxima longitud proyectada de una arista.

Por último, para definir el nivel de teselación  $t_i$  para esta arista, se divide  $l_s$  por un valor  $e_{max}$  definido por el usuario y que identifica la cantidad máxima de píxeles que puede ocupar una arista en pantalla. Por ejemplo, para una ventana de  $640 * 480$  y una arista  $\overline{p_{p1}p_{p2}}$  de longitud 0,5 (un cuarto de ventana en coordenadas de dispositivo normalizadas), la distancia  $l_s$  en el eje Y es igual a 120 ya que  $d = 0,5 * 480/2 = 120$ . Si se define  $e_{max}$  igual a 10, el nivel de

teselación es entonces  $t_i = 120/10 = 12$ . Por lo tanto, esta arista será dividida en 12 aristas nuevas, cada una de longitud de 10 píxeles.

Según el procedimiento detallado, la longitud proyectada de una arista no depende de su orientación con respecto al observador, si no que sólo depende de su longitud absoluta (razón de que se la denomine máxima longitud proyectada). Por lo tanto, se obtiene la misma longitud para una arista que esté perpendicular al plano XY (pantalla) como si está perpendicular al plano XZ (“piso”) o a cualquier otro plano.

#### 4.1.4. Posición en silueta

En un videojuego moderno, normalmente se utilizan modelos tridimensionales que están compuestos por triángulos y que representan los personajes y los entornos. En el caso particular de personajes, presentar un modelo de alta calidad es fundamental para lograr una mejor apreciación visual del jugador. Sin embargo, aumentar la cantidad de polígonos conlleva a un aumento en la carga de procesamiento. Con frecuencia es necesario seleccionar con atención las áreas donde se requiere un mayor refinamiento, para así poder balancear la carga de procesamiento con la calidad del modelo.

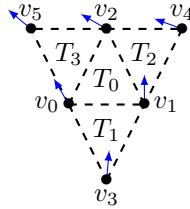
Un área de interés para refinar es la silueta. La silueta marca la frontera visual del modelo, donde cambia abruptamente la profundidad visual. El caso típico son los bordes visuales del modelo contra el fondo de la escena, pero también pueden ser bordes locales de partes del modelo con otras partes del mismo modelo como fondo. Una arista se considera silueta si las normales de los dos triángulos de los que es frontera tienen orientaciones distintas (uno es visible y el otro no).

En cálculos de iluminación, la interpolación del sombreado en el interior de los triángulos provee una transición suave de los colores calculados en los vértices. Esto genera la ilusión de una superficie suave y curva en el interior. Pero esta ilusión se quiebra en la silueta, donde un borde poligonal, de aristas relativamente grandes, se hace muy evidente. Para aminorar este defecto, se propone refinar la silueta. Una vez que la silueta está refinada, en un paso posterior se puede mover cada uno de los nuevos vértices para adquirir una representación más suave de la frontera y así reducir los bordes poligonales.

En esta métrica extrínseca se busca refinar la silueta de un modelo tridimensional y luego mover los nuevos vértices creados. En la presente sección se explica exclusivamente la métrica para decidir si una arista de un triángulo pertenece a la silueta o no. En el capítulo de implementación se expone el método para realizar el movimiento de los vértices fuera del plano del triángulo original.

Para poder implementar esta métrica, se necesita información de adyacencias. En este caso, el patch no se define como un único triángulo, si no como un triángulo y sus vértices vecinos. Dado un triángulo  $T_0$  con vértices  $v_0, v_1$  y  $v_2$ , un vértice  $v_3$  es vecino de  $T_0$  si pertenece a un triángulo  $T_1$ , adyacente a  $T_0$ ,

el cual está formado por dos vértices de  $T_0$  (por ejemplo  $v_0$  y  $v_1$ ) y el vértice vecino  $v_3$ . Cada uno de estos vértices (los del triángulo y los vértices vecinos) tiene una normal definida en el modelo. De esta manera, se puede calcular la normal del elemento central y la normal de los tres elementos adyacentes. Por lo tanto, cada patch estará formado por 6 vértices y sus normales, según como se muestra en la Fig. 4.9. En azul se muestran las normales por cada vértice.



**Figura 4.9:** Ilustración de un patch que representa un triángulo y su vecindad.

Esta métrica está basada en el trabajo de Dyken, Reimers y Seland [9]. En dicho trabajo, a cada arista  $i$  se le asigna un valor de silueta  $s_i$  que los autores denominan *silhouetteness*. Según el valor de  $s_i$ , se calcula el nivel de teselación a asignar a la arista.

En el presente trabajo, el enfoque es ligeramente distinto. La pertenencia de una arista a la silueta es un valor binario:  $s_i = 0$  si la arista no es silueta, o  $s_i = 1$  si la arista es silueta. El nivel de teselación a configurar en una arista silueta es  $t_{tessmax}$ .

A continuación se explica cómo se define el nivel de teselación para una arista  $i$  del triángulo central  $T_0$  del vecindario mostrado en la Fig. 4.9. La arista  $i$  es la que une el vértice  $v_i$  al vértice  $v_{i+1}$ , con  $i \in \{0, 1, 2\}$ . Notar que para la arista con  $i = 2$ , los vértices utilizados son  $v_2$  y  $v_0$ .

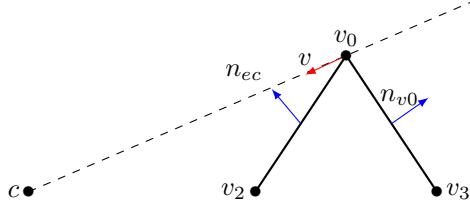
Se comienza por calcular las normales de los elementos, identificando a  $n_{ec}$  como la normal del elemento central y  $n_{vi}$  como la normal del elemento vecino por la arista  $i$ . Se indica  $e_i$  como el punto medio de la arista  $i$ , y  $c$  como la posición de la cámara. Sea  $v = c - e_i$  el vector que une la arista con la cámara, se define un valor  $u_i$  según:

$$u_i = (v \cdot n_{ec}) (v \cdot n_{vi}) \quad (4.7)$$

El vector  $v$  representa el ángulo de visión actual. Este vector es de utilidad para definir el ‘horizonte’ de la vista. Si se realiza el producto punto de la normal de un triángulo con  $v$  y el resultado es positivo, significa que el triángulo es visible, ya que su normal está orientada hacia la cámara. En cambio, si el producto punto es negativo, el triángulo no es visible, ya que su normal tiene orientación opuesta a  $v$ . Esto último se hace bajo la suposición de que el modelo geométrico es cerrado y sólo es visible la cara positiva de un triángulo. Entonces, una arista es silueta si el producto punto (con  $v$ ) de uno de sus triángulos tiene signo distinto al producto punto del otro de sus triángulos. Por lo tanto, en el

caso en que  $u_i$  sea negativo, la arista  $i$  es silueta. Si  $u_i$  es positivo, significa que ambas normales tienen la misma orientación y por lo tanto no es una silueta.

En la Fig. 4.10 se expone un ejemplo para una silueta. Aquí se ve de perfil la arista  $v_0v_1$  (sólo se dibuja el vértice  $v_0$ ) y los dos vértices adyacentes a esta arista:  $v_2$  por el triángulo  $T_0$  y  $v_3$  por el triángulo  $T_1$ . Además se muestra el horizonte con la línea punteada, y las normales de los dos elementos con azul. Por último, en rojo se muestra el vector  $v$ . Si el punto  $v_3$  se ubica por encima de la línea del horizonte, el producto punto  $v \cdot n_{v_0}$  será positivo y la arista no será silueta.



**Figura 4.10:** Ejemplo de silueta.

Aún quedan por definir los niveles de teselación restantes para un triángulo que tiene una o dos aristas silueta. En el interior de un triángulo, si no se define un nivel de teselación distinto de 1, la transición será muy abrupta, ya que se pasará de un nivel  $t_i = t_{tessmax}$  en una arista, a un nivel  $t_{inner} = 1$  en el interior. Por lo tanto, se propone definir un nivel de teselación intermedio en estos casos. Este nivel debe ir en relación a qué tan silueta resultó ser la arista. Se utiliza aquí una expresión similar a la Ec. 4.7, pero ahora se está interesado en el ángulo que forman  $n_{ec}$  y  $n_{vi}$  con respecto a  $v$ , por lo que se utiliza el vector  $v$  normalizado:

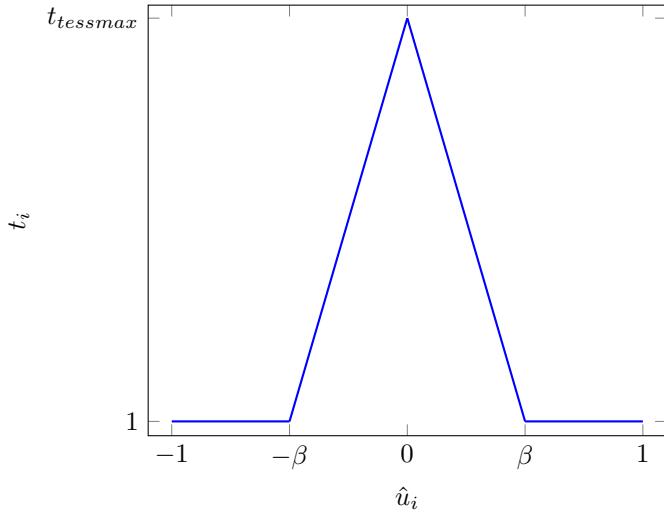
$$\hat{u}_i = \left( \frac{v}{\|v\|} \cdot n_{ec} \right) \left( \frac{v}{\|v\|} \cdot n_{vi} \right) \quad (4.8)$$

Para valores  $\hat{u}_i$  cercanos a 0, el nivel de teselación debe aumentar, pero no se hace distinción aquí en su signo. Por lo tanto, se utiliza el valor absoluto de  $\hat{u}_i$  para aplicar el mismo refinamiento en el interior a ambos triángulos de la silueta, tanto el que es visible como el que no, y evitar los cambios abruptos. Se emplea una constante  $\beta$  que identifica el valor máximo de  $|\hat{u}_i|$  para el cual el nivel de teselación es 1. Esta constante ayuda a definir la pendiente de la recta de mapeo, como se muestra en la Fig. 4.11.

Por otro lado, para el resto de las aristas de un triángulo que no son silueta, se debe definir un nivel de teselación igual a 1 para poder garantizar que no se generen uniones en T.

Por lo tanto, el nivel de teselación de una arista es:

$$t_i = \begin{cases} t_{tessmax}, & u_i < 0 \\ 1, & u_i \geq 0 \end{cases} \quad (4.9)$$



**Figura 4.11:** Mapeo de  $\hat{u}_i$  a  $t_i$  en el interior de un triángulo.

En forma similar, siendo  $j$  la arista que resultó ser silueta de este triángulo, el nivel de teselación interior es:

$$t_{inner} = \begin{cases} -\frac{t_{tessmax}-1}{\beta} |\hat{u}_j| + t_{tessmax} & u_j \leq 0, |\hat{u}_j| \leq \beta \\ 1, & \text{en otro caso} \end{cases} \quad (4.10)$$

En la última expresión, se utiliza la función lineal mostrada en la Fig. 4.11 para definir un valor de teselación intermedio en el interior del triángulo.

## 4.2. Métricas descartadas

Además de las métricas ya descritas, se consideraron otras que finalmente fueron descartadas. Algunas fueron eliminadas por ser muy complejas y no cumplir las características de simplicidad o poca información, mientras que otras se descartaron porque no dieron buenos resultados. En esta sección se explica brevemente en qué consisten y por qué fueron descartadas.

### 4.2.1. Curvatura máxima como diferencia de normales

Esta es una alternativa considerada para el cálculo de la curvatura en superficies paramétricas. En una superficie paramétrica de Bézier, se tiene una malla de control que es la equivalente al polígono de control de la curva de Bézier. Con esta métrica se buscó obtener la mayor curvatura al medir la diferencia de normales de la malla de control.

A partir de los puntos de control se genera una malla de cuadriláteros. A

cada cuadrilátero se le calcula su normal como el producto cruz de sus dos diagonales cruzadas. A continuación, por cada punto de la curva se mide la máxima diferencia angular entre las normales de los elementos que lo contienen. La máxima curvatura entonces es proporcional a esta desviación angular.

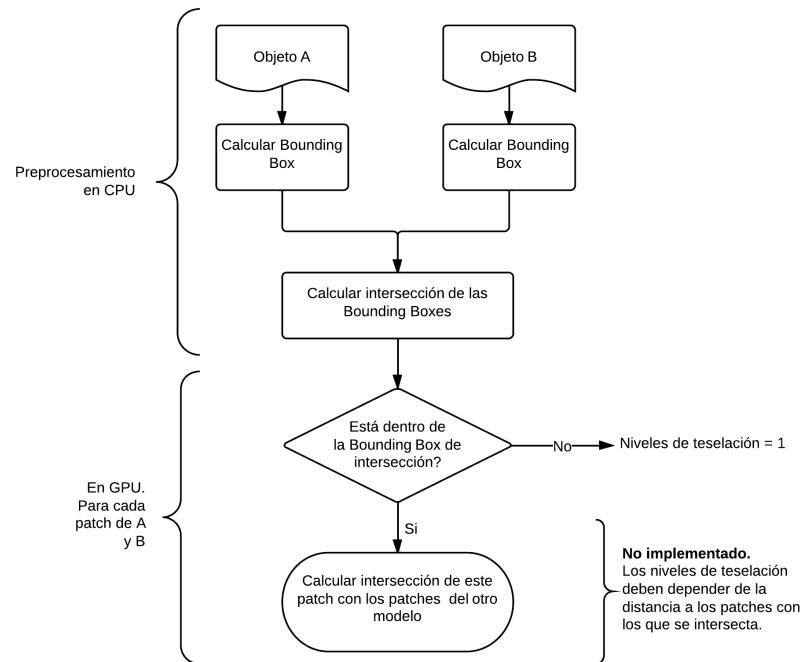
Esta métrica fue implementada y probada en superficies de Bézier de tercer grado, pero no se obtuvieron buenos resultados. Para generar todos los detalles de un modelo, la malla de control está muy ‘curvada’ y las normales adyacentes son mayoritariamente opuestas. Por lo tanto, el nivel de teselación finalmente asignado era muy cercano al máximo.

#### 4.2.2. Distancia entre modelos

En esta métrica, en lugar de medir distancias de los vértices a la posición de la cámara, se mide la distancia de los vértices de un modelo a los vértices de otro modelo. La idea subyacente de esta métrica es la de refinar los polígonos de un modelo que están más próximos al otro modelo. Esto puede ser de utilidad para por ejemplo, realizar cálculos de intersecciones más exactos, ya que al tener más densidad de elementos, el área de intersección se puede aislar con más exactitud.

Fue desarrollado un ejemplo que utiliza esta métrica parcialmente. En la Fig. 4.12 se incluye un diagrama de flujo que ilustra el algoritmo propuesto para esta métrica. Primero se realiza un descarte rápido al evaluar si un vértice del primer objeto está en la caja envolvente (*bounding box*) del segundo objeto. Si no está dentro, está garantizado que no pertenece al área de intersección. En otro caso, está en la caja envolvente y por lo tanto puede ser posible que esté intersectándose. Sin embargo, los cálculos de intersección no pudieron ser realizados en este proyecto porque se evaluó que la implementación de ese apartado no sería sencilla e iba a requerir mucho tiempo de diseño y desarrollo. Además, realizar un algoritmo de intersección polígono a polígono escapa del alcance de este proyecto. Por último, para poder realizar los cálculos de intersección en detalle, esta métrica requiere una estructura de ordenamiento espacial que pueda ser consultada en la GPU. Dicho enfoque no parece, a priori, utilizable en programación en paralelo en GPU.

Por lo tanto, esta métrica fue descartada del producto final ya que, por un lado, la métrica en sí no es muy diferente a aquella de distancia a la cámara, y por el otro, la utilización de la métrica no quedó completa por su complejidad de implementación.



**Figura 4.12:** Algoritmo propuesto para la métrica de intersecciones.

#### 4.2.3. Propiedades físicas en movimiento de fluidos

Se analizó la posibilidad de aplicar teselación al problema del movimiento de fluidos, donde refinar la malla podría ayudar a mejorar los cálculos. Este refinamiento debería hacerse en lugares donde se detecte un mayor gradiente de las variables que se estudian, como en la cercanía de un vórtice. Las métricas en este problema, por ejemplo, podrían tener en cuenta la divergencia o el rotacional del fluido en un pequeño vecindario, y allí refinar según su variación.

Sin embargo, el alumno no cuenta con una formación extensiva en el área y se consideró que este tema, por su complejidad física, es adecuado para realizar un proyecto completo. Por lo tanto, se descartó debido a la gran cantidad de conocimiento que el alumno debería adquirir en poco tiempo. De todas maneras, este tema se deja como proyecto futuro ya que es un área de interés, tanto para el alumno como para el director de este proyecto, en donde se puede aplicar teselación en métodos ingenieriles. Además, se deberá evaluar si efectivamente se puede utilizar teselación en este problema.

## Conclusión

Como se puede apreciar por el contenido de este capítulo, las métricas de teselación son variadas. El contenido expresado en este capítulo surgió de diseño propio e investigación de trabajos anteriores. En algunos casos, fue necesaria una larga etapa de prueba y error hasta dar con el mejor método para calcular el nivel de teselación según cada métrica.

En el capítulo siguiente se expone el software final y los modelos en los que fueron aplicados estas métricas. Si bien las métricas fueron presentadas primero, esto no significa que primero se hizo el diseño total de las mismas y luego los modelos de aplicaciones. Algunas métricas surgieron como necesidad de aplicar teselación a un tipo de modelo en particular. Ejemplo de esto último es la métrica de máxima longitud proyectada de arista, la cual surge del modelo en que se probó (un modelo de terreno muy poco denso).

# Capítulo 5

## Implementación

En este capítulo se muestra el software final desarrollado. La finalidad de este software es presentar ejemplos del uso de las métricas de teselación explícadas en el capítulo anterior. Este software inicia con una pantalla de selección de aplicaciones. A partir de esta pantalla, el usuario puede elegir qué ejemplo ejecutar. Cada aplicación implementa una de las métricas y la usa en un modelo en particular.

Se comienza con una descripción de todas las bibliotecas utilizadas en el desarrollo. Luego se presentan las estructuras de datos (clases) y el diagrama de clases que ilustra las relaciones entre éstas. Por último, se detalla cada uno de los ejemplos y la métrica que se utiliza en cada uno.

### 5.1. Herramientas y bibliotecas utilizadas

El software fue escrito utilizando el lenguaje de programación C++ y desarrollado completamente bajo el sistema operativo Kubuntu Linux 13.04. Como compilador se utilizó GCC 4.8.1 y como IDE de desarrollo se utilizó el editor de texto Vim. Además, para evitar posibles contratiempos y tener una copia de recuperación, se utilizó el sistema de versionado Git.

Como API para la comunicación con la GPU se utilizó la biblioteca OpenGL 4.2. El equipo principal de desarrollo contiene una placa aceleradora de video ATI Radeon 5770 y el driver de video ejecutado en este equipo es ATI Catalyst 13.10.10, lanzado el 23 de Mayo de 2013.

Para facilitar la interacción con el software, se utilizó la biblioteca GLUT [22]. Esta biblioteca brinda una interfaz para construir ventanas en el dispositivo de salida e interactuar con éstas a través de *callbacks*. Los callbacks son funciones que son invocadas por GLUT automáticamente en cada ciclo. Entre

los callbacks, se encuentra aquel que registra las entradas de teclado y el que se encarga de llamar a la función de dibujado en pantalla.

En el software es necesario el uso y mantenimiento de matrices (modelo, vista, proyección, etc.) y vectores (cámara, luces, etc.). Además, estos valores deben ser transferidos constantemente a la GPU. Para facilitar el tratamiento de estas estructuras, se utilizó la biblioteca GLM [13]. Esta biblioteca permite la definición de vectores y matrices, y expone una serie de métodos para permitir la operatoria con estos datos (suma, producto, normas, creación de la matriz de proyección, etc.).

Se utilizó la biblioteca Assimp [2] para facilitar la lectura y utilización de los modelos. Esta biblioteca permite interpretar y cargar un modelo a partir de un archivo OBJ, generado en programas de diseño tales como Blender. El modelo es luego asignado a un vector en memoria, desde donde se puede transferir a la GPU.

Por otra parte, para la lectura de imágenes, se usó la biblioteca DevIL [6]. Esta biblioteca facilita la creación de texturas a partir de la lectura de imágenes en archivos JPG y PNG. La información de color en estos formatos no se puede acceder directamente y es por esto que se necesita una biblioteca que sirva de interfaz y convierta la imagen en una estructura de datos que OpenGL pueda interpretar.

Por último, para el texto en pantalla, se utilizó la biblioteca FreeType [11]. Esta biblioteca ayuda a rasterizar una cadena de texto en pantalla y se incorporó debido a que OpenGL no tiene un mecanismo interno para esta tarea.

Como se puede ver, se utiliza una amplia gama de bibliotecas donde cada una tiene una funcionalidad especial. Todas las bibliotecas utilizadas son de código abierto y multiplataforma. Además se encuentran disponibles para todos los sistemas operativos convencionales. Por lo tanto, con pequeñas modificaciones, el software se puede portar a cualquier otro sistema operativo que tenga soporte para OpenGL 4. Sólo se deben incluir las bibliotecas utilizadas e indicar al compilador dónde encontrarlas.

## 5.2. Implementación del software final

### 5.2.1. Estructuras de datos

Para el desarrollo del software se utilizaron las capacidades de programación orientada a objetos de C++. Se diseñó y desarrolló un grupo de estructuras de datos, contenidas en clases, que agrupan la información y brindan una interfaz para interactuar con la misma.

Las clases utilizadas y sus responsabilidades son:

- *Example:* Esta es la clase principal que almacena el estado general del software. Entre sus propiedades se encuentran todas las matrices de transformación utilizadas y los datos sobre la posición y orientación de la cámara. Además agrupa toda la funcionalidad de los callbacks de GLUT. Estos callbacks, a su vez, llaman a funciones adicionales dentro de Example. Asimismo, esta clase es heredada utilizando el mecanismo de herencia de C++. Todas sus propiedades y métodos son copiadas a sus clases ‘hijas’. Las clases hijas son cada una de las aplicaciones que se detallan luego. Al utilizar el mecanismo de herencia, se evita repetir información y por lo tanto se reduce la posibilidad de errores al desarrollar. La función más importante que tiene Example es `Example::DrawScene()`, la cual es llamada por el callback de dibujado. En esta función se cargan los programas de shader en memoria, se le transfieren los datos, y se realiza el renderizado, invocando el pipeline gráfico.
- *ShaderProgram:* Su responsabilidad es brindar una interfaz simple para la utilización de un programa de shader. A partir de sus métodos, se crean, compilan y enlazan todos los objetos de shader que están presentes en el programa de shader. Sus propiedades internas son variables que utiliza OpenGL para identificar el programa y cada uno de sus objetos de shader. Además, presenta métodos para definir el valor de todos los uniform que se utilizan en el shader. Antes de hacer una llamada a renderizado, se debe traer a contexto un programa de shader. Esto se realiza llamando a la función `ShaderProgram::Bind()`. Al finalizar, para liberar la memoria o traer a contexto un programa diferente, se ejecuta `ShaderProgram::Unbind()`.
- *VAO:* El nombre de esta clase hace referencia a *Vertex Array Object*, ya que su responsabilidad es dar una interfaz simple para el trabajo con los datos. Un VAO está compuesto por un grupo de buffers. Cada buffer es un grupo de datos que debe ser enviado a la GPU. Ejemplos de buffer son todas las posiciones de los vértices, todas las normales, y los índices de los vértices que componen cada patch. En la etapa de inicialización, se crea un objeto de esta clase y se le enlaza cada uno de los buffers. En la función de dibujado, luego de traer a contexto un programa de shader y actualizar el valor de los uniform, se llama a `VAO::Render()`. Este método envía el VAO a la GPU para que transite por el pipeline gráfico.
- *Model:* Esta clase utiliza la biblioteca Assimp para leer modelos tridimensionales desde archivos OBJ y convertirlos en datos interpretables por un VAO. Luego de leer el archivo, cada tipo de dato (posición, normal, índices de elementos) es almacenado en un vector. Este vector se envía al VAO como un buffer.
- *TextRenderer:* Esta clase ayuda al renderizado de texto en pantalla exhibiendo una interfaz simplificada para realizar esta tarea. Dibujar un texto en pantalla requiere de un programa de shader simple al que se envía una textura transparente con la cadena a rasterizar. En el constructor de esta clase se crea un programa de shader interno, cuya única función es dibujar una textura en pantalla. Luego, en su función `TextRenderer::Render()`, se construye la textura con la cadena de texto pasada como parámetro, y se envía la textura al programa de shader interno para que sea dibujada.

- *Utils*: Esta no es una clase, si no un conjunto de funciones agrupadas en un *namespace*. Un *namespace* (espacio de nombres) es un mecanismo de C++ para definir un ámbito en donde se concentra un grupo de variables y funciones, y cuya falta de relación justifica no incluirlas en una clase. Estas funciones son generales y son llamadas desde diferentes lugares del software. Ejemplos de estas funciones son:

- `utils::ExitOnGLError()`: verificación de errores en ejecución;
- `utils::LoadImageTex()`: lectura de una imagen en memoria utilizando la biblioteca DevIL;
- `utils::XToString()`: conversión de cualquier tipo de dato a una cadena de texto que TextRenderer pueda interpretar y dibujar en pantalla.

### 5.2.2. Diagrama de clases

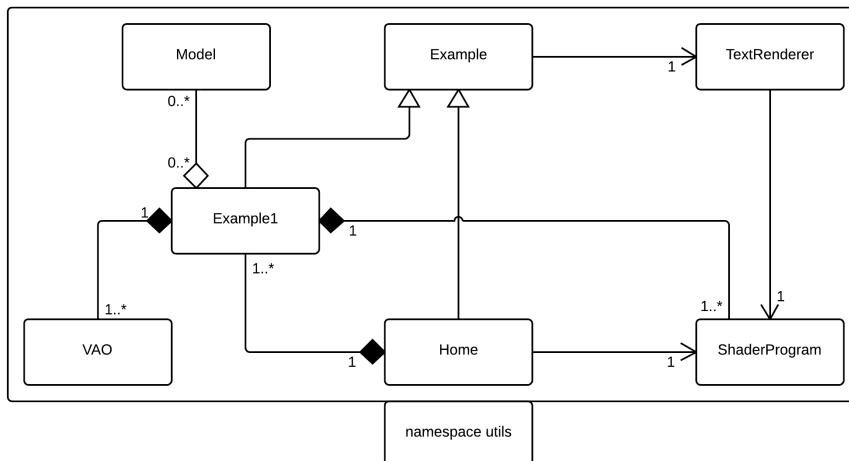
En la Fig. 5.1 se muestra el diagrama de clases del software. Se incluye la clase Home, la cual hereda de Example y es la primer aplicación que se ejecuta (y quien controla los callbacks). La función `Home::DrawScene()` dibuja el texto de bienvenida en pantalla. Como se dijo, cada aplicación desarrollada está contenida en una clase separada que hereda de Example. En el diagrama, esta clase se la representa con el nombre de Example1. Cuando el usuario selecciona una aplicación, el objeto de la clase Home construye un objeto de la clase Example1 y guarda un puntero a dicho objeto en una variable llamada `Example* Home::loadedExample`.

Cuando existe una aplicación cargada, Home redirecciona la interacción a la aplicación. Por ejemplo, Example1 define una función `Example1::DrawScene()` que es llamada desde `Home::DrawScene()`, la cual a su vez es llamada desde el callback de dibujado de GLUT. Cuando se regresa a la pantalla principal, el objeto Example1 es destruido y se deja de realizar la redirección, retornando el control a Home. Esta misma redirección es realizada en los callbacks que controlan la interacción por el teclado y el mouse.

A continuación se explica brevemente cada una de las relaciones presentes en el diagrama de clases:

- Home-Example: Representa la herencia de Home de la clase Example.
- Home-ShaderProgram: Corresponde al programa de shader que se utiliza en la clase Home para dibujar la posición de la cámara virtual en la escena (cuando se la utiliza). Se dibuja para ayudar al usuario a conocer la ubicación de esta cámara. Esto se incluye aquí para reutilizar el código, ya que se repite en la mayoría de los ejemplos.
- Home-Example1: Representa la aplicación ejemplo cargada. Home mantiene un puntero a Example1 para redireccionar las interacciones. Por su parte, Example1 tiene un puntero a Home para retornar el control a Home en el destructor de Example1.

- Example-TextRenderer: Corresponde a la instancia de TextRenderer cuya función es dibujar texto en la ventana actual. Esta instancia es única y puede ser accedida por cualquier clase que herede de Example.
- TextRenderer-ShaderProgram: Representa el programa de shader que la clase TextRenderer utiliza para dibujar una textura con un texto.
- Example1-Example: Define la herencia de Example1 de la clase Example.
- Example1-Model: Representa los objetos de la clase Model que Example1 puede tener cargados para dibujar. Puede no haber un modelo cargado si no es necesario (los datos vienen dados en estructuras más simples como vectores).
- Example1-VAO: Representa uno o más buffers que deben ser cargados en memoria y transferidos a un programa de shader para ser dibujados.
- Example1-ShaderProgram: Representa uno o más programas de shader utilizados por Example1 para dibujar en pantalla. Se necesita al menos un programa de shader para dibujar el contenido de un VAO.



**Figura 5.1:** Diagrama de clases del software.

### 5.2.3. Funcionalidades

En la Fig. 5.2 se muestran distintas capturas del software final. Al ejecutar el software, se presenta una pantalla inicial de selección de ejemplo de aplicación. A partir de aquí, el usuario selecciona la aplicación que desea ejecutar, pulsando la tecla conveniente (ver Fig. 5.2a).

Cada aplicación tiene sus funcionalidades propias. En forma general, se presenta un modelo en escena en una posición elegida por defecto. El usuario luego

puede interactuar con el software a través del teclado y mouse. Además, el usuario puede moverse por la escena a discreción. Asimismo, el usuario puede elegir cómo se dibuja la escena, intercambiando entre dibujado de polígonos rellenos o *wireframe*, donde sólo se muestran las aristas de los polígonos (ver Fig. 5.2b). Se puede activar o desactivar la iluminación en ejemplos donde es utilizada, y mover libremente la posición de la luz. También se incluye un modo de visualización del modelo utilizando un mapa de colores. En este modo, un tono naranja denota niveles de teselación altos en el interior de un patch. En otro caso, el color es un tono celeste claro. De esta manera el usuario puede apreciar, en forma gráfica e intuitiva, la variación de niveles de teselación que tienen los distintos patches del modelo (ver Fig. 5.2c).

En todos los ejemplos se pueden modificar todos los parámetros de la métrica que se aplica. Por ejemplo, para la métrica de distancia se puede modificar la distancia máxima  $d_{max}$ . Por su parte, para la métrica de longitud de arista, se puede cambiar el valor de la constante  $e_{max}$ .

Dado que la mayoría de las métricas implementadas (todas excepto curvatura) dependen de la posición de la cámara, se incorporó una funcionalidad que permite separar la posición de la cámara del punto de vista, generando así una posición ‘virtual’. De esta manera, se puede apreciar el funcionamiento del software con una posición distinta de la cámara actual. Esto es muy útil para ver, de cerca, los niveles y forma de teselación definida para los patches.

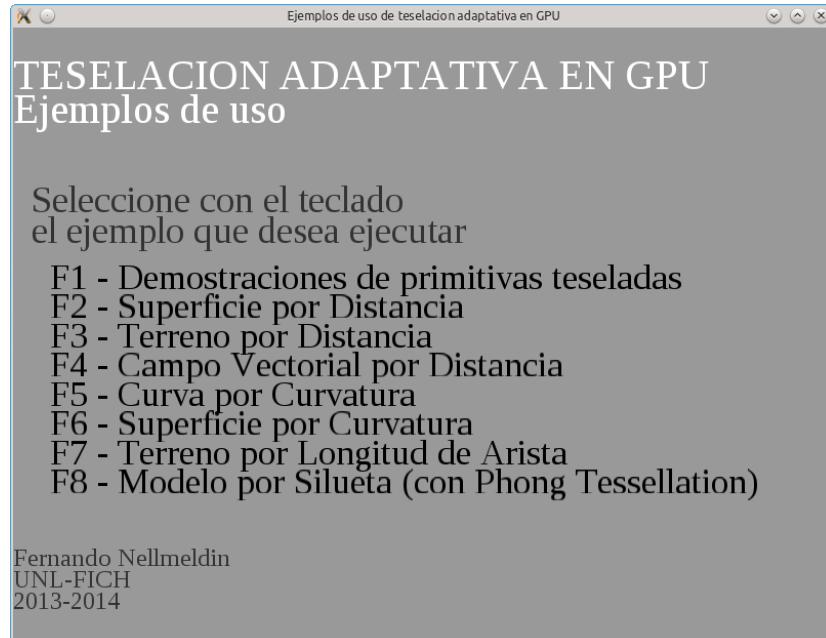
Todos los comandos de interacción son detallados en el software al apretar la tecla ‘H’ de *Help* o Ayuda. Al pulsar esta tecla, se muestra en pantalla el texto de ayuda con la lista y tecla asignada de todos los comandos (ver Fig. 5.2d). Esta lista varía en cada ejemplo en particular, pues cada aplicación tiene sus distintas opciones de interacción. Este texto desaparece de la pantalla luego de unos segundos, o se puede eliminar apretando la tecla Backspace.

### 5.3. Aplicaciones

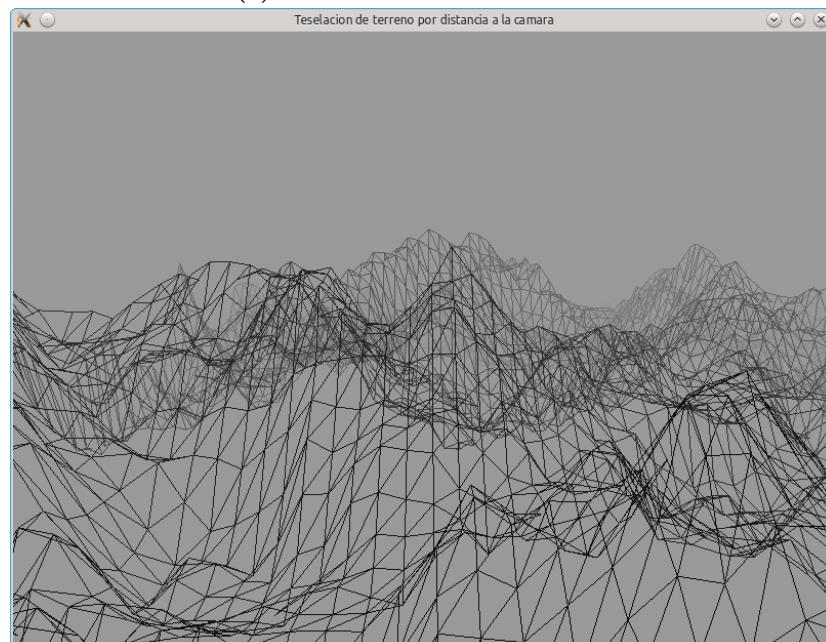
Esta sección expone cada una de las aplicaciones desarrolladas en el software final. Como ya se expresó, cada una de estas aplicaciones está contenida en una clase por separado. Por ejemplo, la clase *CurveCurvature* incluye los métodos de inicialización, dibujado e interacción del uso de la métrica de curvatura a una curva. Esta clase lee los programas de shader, inicia los buffers con la posición de las primitivas, y finalmente, dentro del ciclo de dibujado, invoca al pipeline gráfico.

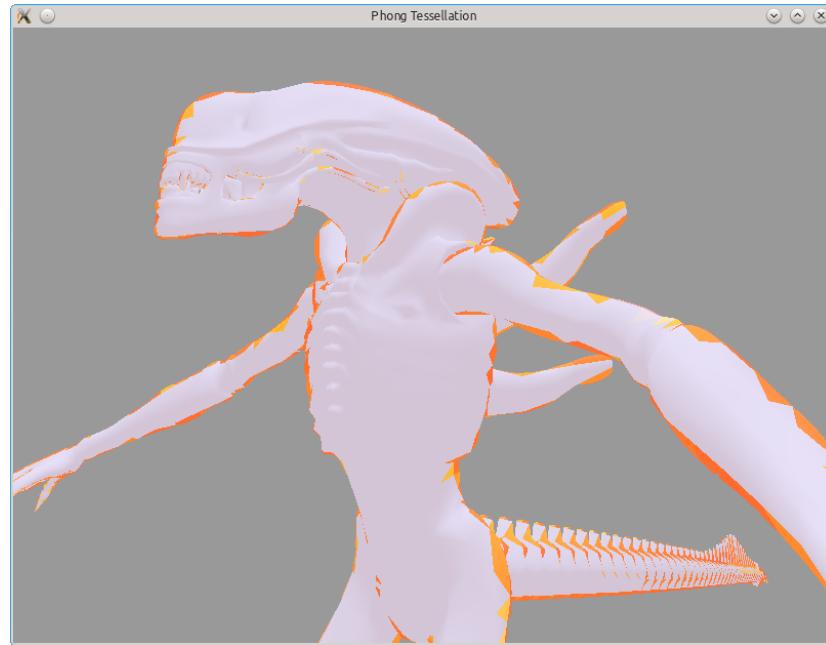
#### 5.3.1. Aplicación de muestra de niveles de teselación

Esta aplicación está enfocada a mostrar cómo funciona el Tessellation Primitive Generator. Aquí no hay métrica, si no que el usuario elige por teclado cada

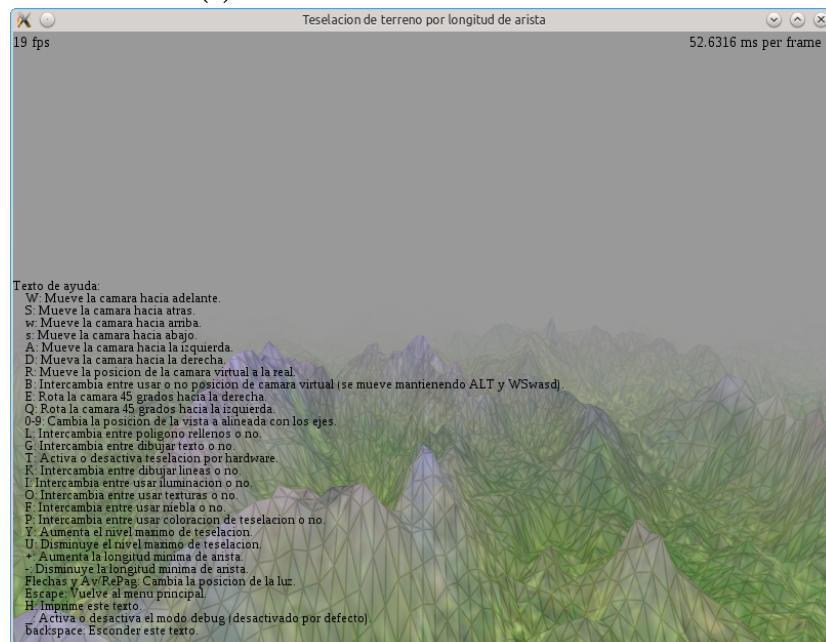


(a) Pantalla de inicio del software.

(b) Polígonos en *wireframe*.**Figura 5.2:** Capturas del software final.



(c) Visualización de niveles de teselación.

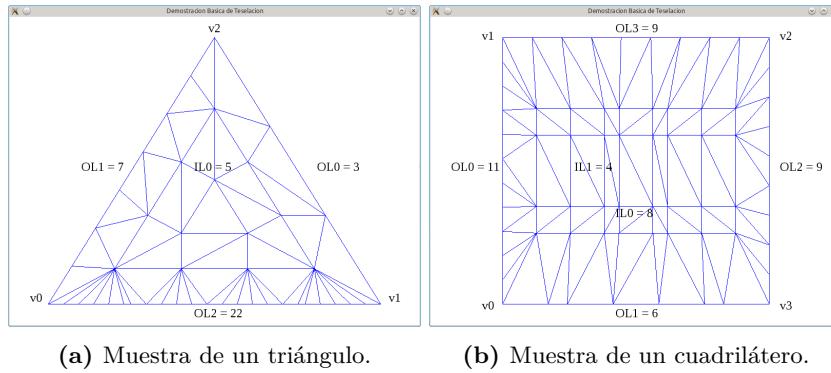


(d) Texto de ayuda para uno de los ejemplos.

**Figura 5.2:** Capturas del software final.

parámetro de teselación (niveles internos y externos), y la primitiva se dibuja en pantalla. Además se puede pasar de un tipo de primitiva a otra (triángulos, cuadriláteros y poligonales), y modificar el algoritmo de espaciado (par, impar o simétrico).

La clase que controla este ejemplo contiene un grupo de programas de shader. Se utiliza un programa de shader para cada combinación de tipo de primitiva y tipo de espaciado. En la Fig. 5.3 se muestran algunas capturas adicionales de esta aplicación instructiva.



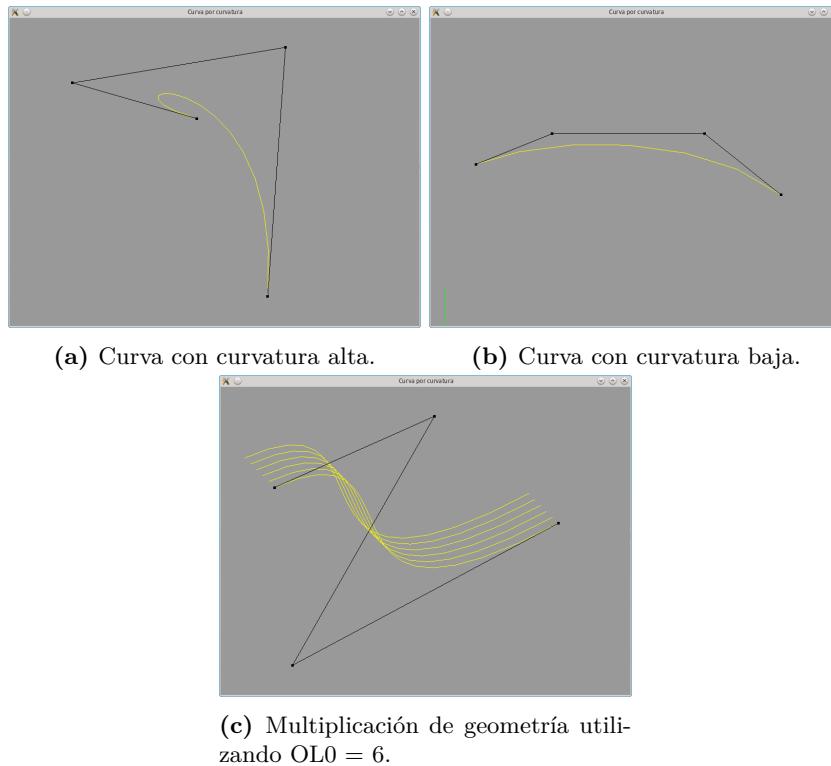
**Figura 5.3:** Capturas de la aplicación de muestra de niveles de teselación y métodos de espaciado.

### 5.3.2. Curva de Bézier de tercer grado

Una curva paramétrica de Bézier de tercer grado está definida a partir de 4 puntos de control. Estos puntos de control son utilizados para calcular el nivel de teselación en función de la curvatura, según como se detalló en el capítulo sobre métricas. Al tener 4 puntos de control, es necesario calcular dos curvaturas y elegir la mayor de éstas. Aquí se utiliza la primitiva `isolines`, por lo que hay que definir dos niveles de teselación. El primer nivel de teselación indica la cantidad de poligonales distintas que se van a crear. La métrica no define este valor, sino que simplemente es transferido como un parámetro que es modificable por el usuario. Estas poligonales son copias desplazadas de la poligonal original y pueden ser utilizadas, por ejemplo, para multiplicar la geometría original al renderizar cabello. El otro nivel de teselación define en cuántos segmentos se va a dividir cada una de estas poligonales, las cuales aproximan la curva (y sus copias) al rasterizarlas.

En la Fig. 5.4 se presentan capturas de esta aplicación. En la Fig. 5.4a se presenta una única curva con un alto nivel de teselación, ya que la curvatura es alta. Por otra parte, en la Fig. 5.4b se muestra una curva con una discretización reducida debido a que presenta curvatura casi nula (el polígono de control se acerca a una única línea recta). Por último, en la Fig. 5.4c se muestra la multiplicación de geometría que se puede realizar utilizando valores mayores a 1

en OL0. En este caso, a partir de un único polígono de control, se dibujaron 6 curvas desplazadas entre sí.



**Figura 5.4:** Capturas de la métrica de curvatura para curvas de Bézier de tercer grado.

### 5.3.3. Superficie de Bézier de tercer grado

Una superficie de Bézier  $\mathbf{P}(u, v)$  es la extensión de una curva paramétrica de Bézier a dos parámetros y un conjunto mayor de puntos de control. Un patch que representa una superficie de Bézier de tercer grado está formado por una malla de 16 puntos de control. Estos puntos de control están alineados en una grilla estructurada como una malla de cuadriláteros regular, con 4 nodos por lado. Para superficies de Bézier de tercer grado se aplicaron dos métricas. La primera es la métrica de curvatura, mientras que la segunda es la distancia de la cámara a los patches.

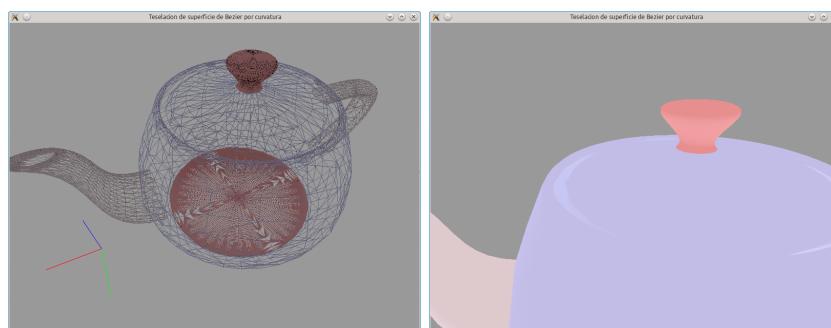
#### Métrica de curvatura

Esta métrica mide la extensión a superficies de la aproximación de curvatura presentada en el capítulo anterior. En este caso, se tienen 16 puntos de control

en 4 filas y 4 columnas. Se deben calcular todos los valores de curvatura para conjuntos de 3 vértices en las direcciones verticales y horizontales, y también las diagonales. En total, se calculan dos aproximaciones por fila y dos por columna. Dando lugar a un total de 16 valores de curvatura. Además se deben calcular las curvaturas diagonales. Con 16 puntos de control, se tienen 9 curvaturas diagonales.

El dominio paramétrico es un cuadrilátero, por lo que se deben definir 4 niveles externos y 2 niveles internos de teselación. De los valores obtenidos, se utilizan los correspondientes a los extremos para definir los niveles de teselación en las fronteras. El resto de los valores se utiliza para configurar los niveles internos. Si el mayor diagonal es mayor a cualquiera de los verticales u horizontales, se utiliza directamente el mayor diagonal. En otro caso, en el nivel de teselación interno horizontal, se utiliza el mayor horizontal. De igual manera, en el vertical, se utiliza el mayor valor obtenido en la dirección vertical .

En la Fig. 5.5 se presentan dos capturas de esta aplicación. El modelo ejecutado es el de la famosa tetera de Utah, el cual está conformado por 32 patches de superficies de Bézier de tercer grado. Se puede apreciar que los patches que presentan más curvatura tienen un valor mayor de teselación (colores cercanos al rojo). En la Fig. 5.5a se muestra el modelo en wireframe. Se puede ver que los mayores niveles de teselación se dieron en la agarradera de la tapa y en la base. Aunque en la imagen no se puede apreciar, las superficies que conforman la base tienen una curvatura alta porque se ‘doblan’ en uno de sus extremos. En la Fig. 5.5b se presenta un acercamiento y se pueden ver los distintos tonos del mapa de color para los diferentes niveles de teselación utilizados.



(a) Coloración de los distintos niveles de teselación en *wireframe*. (b) Posición más cercana donde se aprecian mejor los distintos niveles.

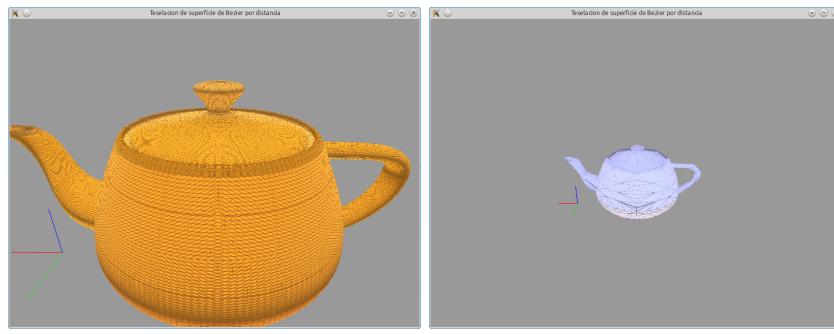
**Figura 5.5:** Capturas de la métrica de curvatura para superficies de Bézier de tercer grado.

### Métrica de distancia

La métrica de distancia mide la distancia de la cámara a cada patch y computa su nivel de teselación como una función lineal o lineal inversa de esta distancia. Se define un nivel de teselación distinto para cada frontera del cuadrilátero

paramétrico. Los dos niveles interiores, por otra parte, se calculan en base al centroide de todos los puntos del patch. Los niveles de teselación en cada frontera se obtienen con respecto al punto más cercano entre todos los puntos del polígono de control de esa frontera. Por lo tanto, se calculan 4 distancias por frontera del cuadrilátero paramétrico. Para definir el nivel de teselación a través de dicha frontera, interesa la menor de estas distancias. Se espera que si dos patches son adyacentes, los puntos de control en su interfaz sean los mismos y se calcule la misma distancia menor. Por lo tanto, se obtiene el mismo nivel de teselación y se evitan las uniones en T.

En la Fig. 5.6 se muestran capturas sobre esta aplicación. Se puede ver la diferencia del mapa de color en la Fig. 5.6b con respecto a la Fig. 5.6a. Esta diferencia se da porque la cámara se ubica en una posición más alejada a cada patch.



(a) Posición cercana, todos los niveles de teselación son altos.  
(b) Posición alejada, los niveles de teselación son bajos.

**Figura 5.6:** Capturas de la métrica de distancia para superficies de Bézier de tercer grado.

### 5.3.4. Campo vectorial

Un campo vectorial es una función vectorial que se aplica a cada punto del dominio. Para la visualización de un campo vectorial existen distintos métodos. El método que aquí se detalla está basado en los fundamentos de la técnica de LIC (*Line Integral Convolution*) [8] y se aplica a campos vectoriales que no cambian a través del tiempo.

La aplicación implementada consiste en tomar puntos de muestra de la función vectorial y graficarlos con un color sobre una imagen. Dicho color se obtiene de una textura de ruido que se envía al programa de shader. Luego, los puntos son movidos según el campo vectorial que los está afectando y se dibujan en la misma imagen anterior, acumulando el resultado. Una vez que se realizaron varios ciclos de este proceso, el resultado obtenido es que para cada punto original, se tiene una línea de flujo (*streamline*) que ilustra el recorrido realizado por cada punto.

En este modelo, los patches tienen dimensión 1 y están formados por los puntos originales en donde calcular el campo vectorial. Estos puntos están organizados en una grilla cuadrada y regular para facilitar la utilización de la información de los vecinos. Para poder implementar este modelo, fue necesario el desarrollo de un algoritmo de doble pasada por la GPU. En la primer pasada, se mueven los puntos según el campo vectorial que los afecta. Para este movimiento se utiliza el método de Euler, que tiene la forma:

$$x(t + h) = x(t) + hf(t, x) \quad (5.1)$$

donde  $x(t)$  es la posición del punto en el paso  $t$ ,  $x(t + h)$  es la posición en el paso siguiente y  $f(t, x)$  es el campo vectorial que afecta al punto. En esta primer pasada no se utiliza teselación.

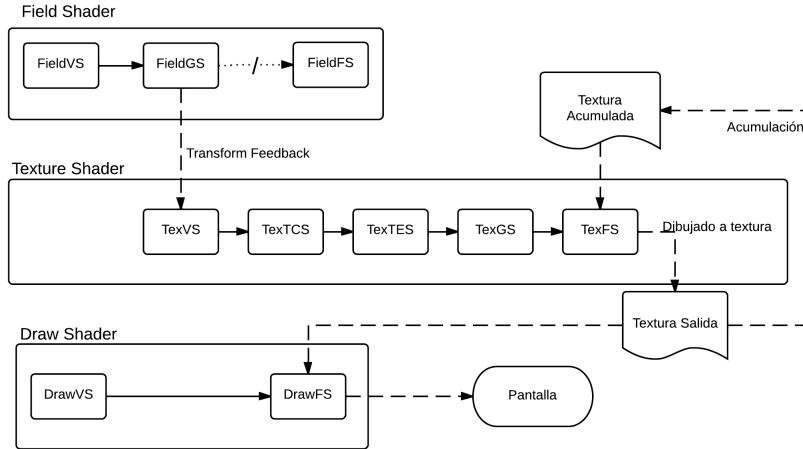
Una vez que se realiza el movimiento de todos los puntos originales, son reinsertados al pipeline gráfico utilizando la funcionalidad de *Transform Feedback*. Dicha funcionalidad permite que la salida de un programa de shader pueda ser realimentada a otro programa de shader, pues almacena, en una estructura de datos auxiliar, los vértices de salida del Geometry Shader.

En la segunda pasada se ejecuta un programa de shader que incluye teselación y cuya entrada son los puntos movidos, sus posiciones originales y el campo vectorial que afecta a sus vecinos. Con esta información se aplica la métrica de teselación y se subdivide la malla regular original. Los nuevos puntos son insertados en el interior de la malla original. Luego, estos puntos son desplazados según el campo vectorial que los afecta. Este campo vectorial es calculado por interpolación del campo vectorial de los vecinos del punto original. Los vecinos que se consideran son, según la malla regular original, el vecino por derecha, el vecino por abajo, y el vecino por abajo y derecha. El espacio paramétrico utilizado en esta aplicación es el de cuadriláteros, donde sus esquinas las constituyen el vértice original y los 3 vecinos mencionados.

Para finalizar, el resultado del movimiento es almacenado en una textura. En pasos sucesivos, las dos texturas resultado se suman para obtener la acumulación antes descrita.

En la Fig. 5.7 se presenta el procesamiento completo realizado en esta aplicación. Se tienen tres programas de shader distintos. El primero es *FieldShader*, el cual se encarga de calcular la nueva posición de cada punto afectado por el campo vectorial. Este shader no utiliza teselación ni ejecuta el FS. Después de pasar por el GS, se utiliza transform feedback para alimentar el vértice movido al siguiente shader: *TextureShader*. Este programa tiene como entrada el vértice movido, su posición original, el campo vectorial que lo afecta, y el campo vectorial que incide en sus vecinos. Toda esta información está encapsulada en un patch de dimensión 1. Con esta información se calculan los niveles de teselación para cada patch y se realiza la subdivisión. Luego, cada vértice nuevo se mueve utilizando los valores del campo vectorial que afecta a sus vecinos, interpolando sus valores. En el FS de este shader se escribe la salida a una textura. Esta textura se obtiene a partir de una acumulación entre la textura anteriormente calculada y la actual. Además, esta textura será realimentada a la siguiente ejecución del pipeline completo, para continuar con la acumulación.

Después, esta textura pasa al siguiente programa de shader: *DrawShader*. La única responsabilidad de este último shader es dibujar en pantalla la textura acumulada.



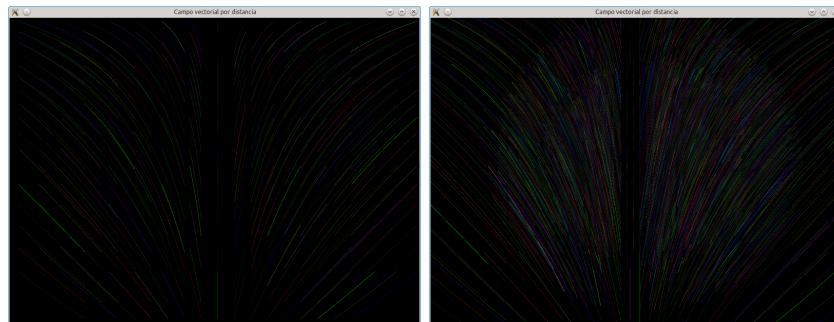
**Figura 5.7:** Pipeline completo utilizado en el renderizado del campo vectorial.

La métrica utilizada en este modelo es la métrica de distancia. Para cada punto original se mide su distancia con respecto al ojo. En su vecindario se generan nuevos puntos según esta distancia, usando la función lineal descrita con anterioridad. La métrica de distancia es empleada aquí para permitir al usuario acercarse al campo vectorial y tener más detalle en el lugar donde está enfocado. El resultado obtenido se muestra en la Fig. 5.8. En la Fig. 5.8a se presenta el campo vectorial original, mientras que en las Fig. 5.8b-c se presentan distintos acercamientos. Estos acercamientos muestran la distinta densidad que se genera en el campo respecto del punto de vista.

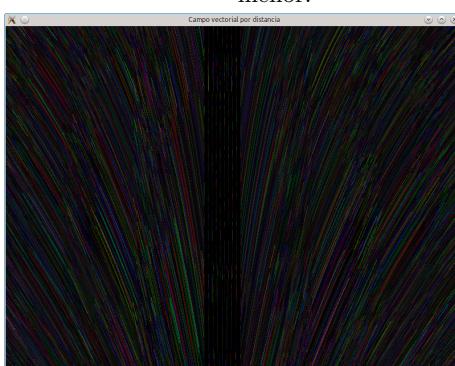
### 5.3.5. Modelo de personaje

En esta aplicación se utilizó un modelo de personaje compuesto por elementos triangulares. Este modelo fue obtenido gratuitamente del sitio TF3DM [21]. Según como se expuso anteriormente, los patches de esta aplicación son extendidos y contienen los tres vértices vecinos a los elementos. Por lo tanto, un patch está formado por 6 vértices y sus normales.

Para este modelo se utilizó la métrica de silueta. Además se incorporó una técnica de movimiento de vértices llamada Phong Tessellation [3]. Al aplicar Phong Tessellation, los nuevos vértices son desplazados fuera del plano del elemento original, generando una superficie curva que intenta aproximar al objeto original.



(a) Campo vectorial a una distancia máxima.  
 (b) Campo vectorial a una distancia menor.



(c) Campo vectorial a distancia mínima.

**Figura 5.8:** Ilustración de la métrica de distancia aplicada a campos vectoriales.

### Métrica de silueta

La métrica de silueta se implementó de la misma manera a como fue descrita en el capítulo 4. Debido a que refinar la silueta no tiene ninguna utilidad en sí misma, se utilizó un método de desplazamiento de los puntos teselados. Entonces, se aplicó el método de Phong Tessellation para mover los vértices y obtener una silueta más suave. En el apartado siguiente se expone este método y el resultado obtenido.

### Phong Tessellation

En el caso normal, al subdividir un triángulo, los nuevos vértices se ubican en el mismo plano del triángulo. En contraposición, en Phong Tessellation, los nuevos puntos se ubican en el espacio según información dada por los vértices del triángulo y sus normales. De este modo, los nuevos triángulos ya no están en el mismo plano que el elemento original, si no que forman un triángulo curvado fuera de su plano. Phong Tessellation recibe su nombre debido a que es un método basado en el modelo de iluminación de Phong. En este modelo, las normales de los vértices ayudan a determinar la cantidad de iluminación que

recibe un vértice.

El procedimiento que se detalla a continuación define la manera de ubicar los vértices que se crearon en el paso de teselación. Estos cálculos son realizados en el Tessellation Evaluation Shader.

Se parte de un triángulo con vértices  $\mathbf{v}_i$ ,  $\mathbf{v}_j$  y  $\mathbf{v}_k$ . Además se tienen las normales de estos puntos:  $\mathbf{n}_i$ ,  $\mathbf{n}_j$  y  $\mathbf{n}_k$ . Se denomina como  $\mathbf{p}$  a un punto obtenido en la teselación como interpolación de los vértices originales, con coordenadas paramétricas  $(u, v, w)$ . Se está interesado en calcular la nueva posición espacial del punto  $\mathbf{p}$ .

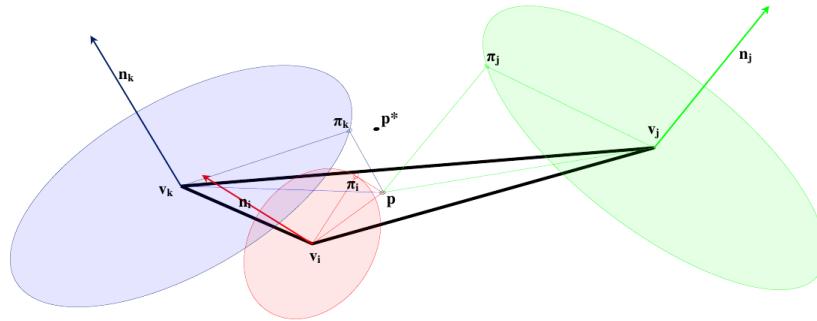
Primero se calcula cada plano tangente a cada vértice. Dicho plano, denominado  $\pi_i$  para el vértice  $\mathbf{v}_i$ , está definido por la posición del vértice y su normal. Se proyecta el punto  $\mathbf{p}$  en cada uno de estos planos. La proyección de  $\mathbf{p}$  en el plano  $\pi_i$  se identifica como  $\pi_i(\mathbf{p})$  y se calcula como:

$$\pi_i(\mathbf{p}) = \mathbf{p} - ((\mathbf{p} - \mathbf{v}_i) \cdot \mathbf{n}_i)\mathbf{n}_i \quad (5.2)$$

Por último, se realiza una interpolación entre la proyección del punto en los tres planos. Los pesos de la interpolación se obtienen de las coordenadas paramétricas del punto  $(u, v, w)$  en el triángulo original. Por lo tanto, la posición desplazada  $\mathbf{p}^*$  es:

$$\mathbf{p}^* = (u, v, w) \begin{pmatrix} \pi_i(\mathbf{p}) \\ \pi_j(\mathbf{p}) \\ \pi_k(\mathbf{p}) \end{pmatrix} \quad (5.3)$$

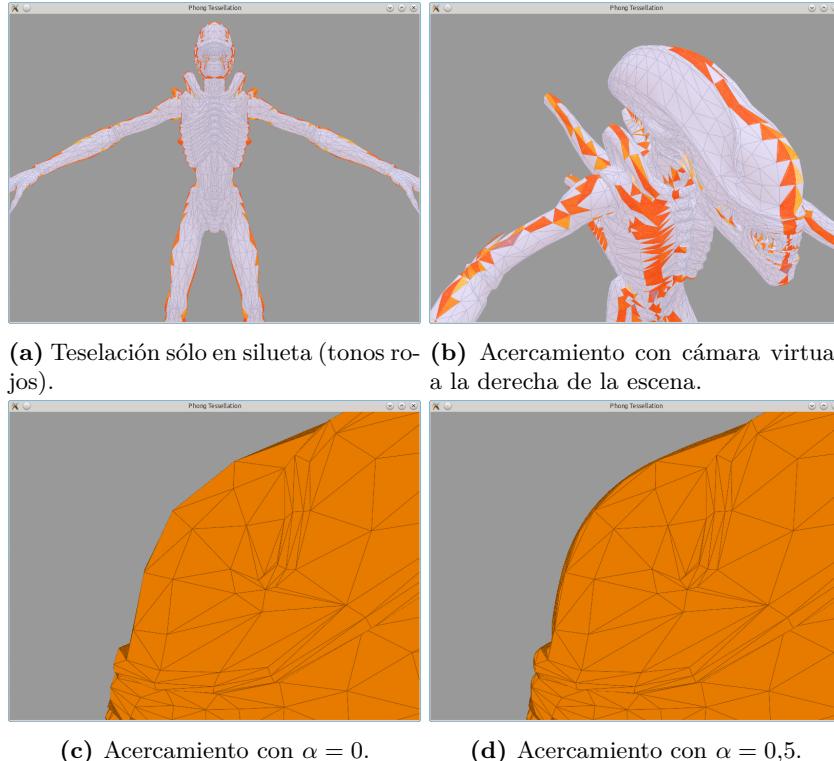
En la Fig. 5.9 se ilustra el modo de cálculo de Phong Tessellation. Se muestran los vértices del triángulo original y sus normales. Los círculos identifican cada uno de los planos tangente. Por su parte, las líneas identifican las proyecciones realizadas.



**Figura 5.9:** Ilustración del cálculo de la posición en Phong Tessellation.

El resultado que se obtiene de este procedimiento es que los nuevos elementos se adecuan mejor a la geometría descripta por las normales de los triángulos.

En la Fig. 5.10 se muestra la aplicación de este método. Se utiliza un parámetro  $\alpha \in [0, 1]$  para definir la cantidad de desplazamiento que sufren los vértices. Este parámetro interpola la posición del vértice en el plano  $\mathbf{p}$  y el vértice desplazado  $\mathbf{p}^*$ . Por lo tanto, para  $\alpha = 0$  no hay desplazamiento, mientras que si  $\alpha = 1$ , la posición final es  $\mathbf{p}^*$ . Este parámetro se agregó ya que un desplazamiento máximo no es conveniente porque deforma mucho la geometría. De esta manera se puede regular el resultado.

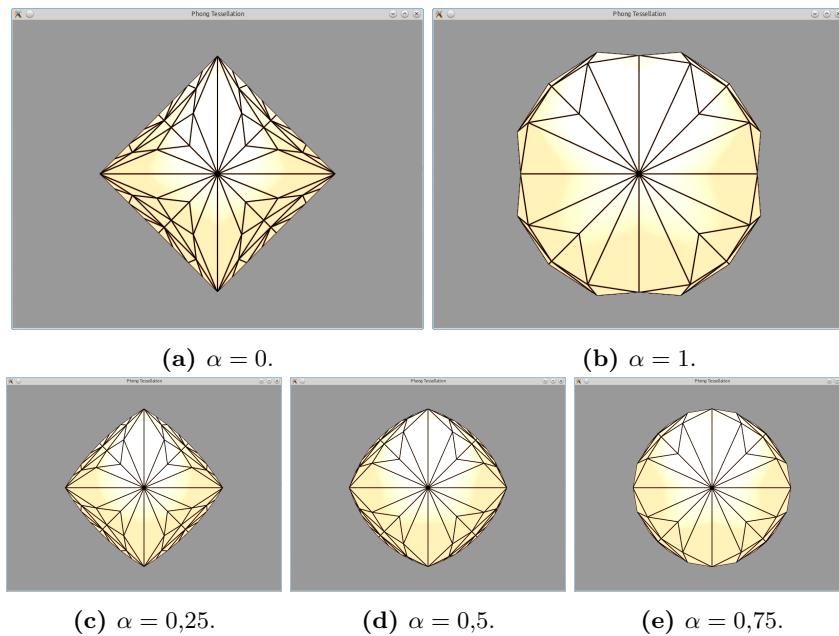


**Figura 5.10:** Ejemplo de teselación en silueta.

En primer lugar, la Fig. 5.10a muestra el modelo original visto de frente. Aquí se utiliza el mapa de colores para mostrar que la teselación se realizó en toda la silueta del modelo (en naranja). Este método no genera uniones en T, ya que cada nivel de teselación se calcula con respecto a información de cada arista. En la Fig. 5.10b se muestra un acercamiento. En este caso, se utiliza la posición virtual de cámara y se la ubica a la derecha de la escena (izquierda del modelo). Se puede apreciar que no se generan uniones en T y el único refinamiento efectuado se da en la silueta del modelo (con respecto a la posición de la cámara virtual).

En las Fig. 5.10c y 5.10d se expone con más detalle el resultado de utilizar Phong Tessellation. En 5.10c no hay desplazamiento en la silueta. En la Fig. 5.10d se realiza el desplazamiento. Se puede ver que se logra el resultado perseguido, ya que se generan bordes más suaves.

Para poder apreciar mejor el efecto logrado con Phong Tessellation, se presenta un ejemplo con un octaedro en la Fig. 5.11. En la primer fila se muestran los casos extremos. La Fig. 5.11a muestra el modelo original, cuando  $\alpha = 0$ . Por otra parte, en la Fig. 5.11b se presenta el caso con  $\alpha = 1$ . Aquí se puede ver cómo la geometría se deforma, dando un resultado poco agradable. En la segunda fila de esta figura, se exponen ejemplos intermedios. El caso que dio mejor resultado fue con  $\alpha = 0,5$ , ya que esto permite un desplazamiento suficiente para generar geometría curva, pero no exagera el perfil del modelo.



**Figura 5.11:** Muestra de Phong Tessellation para un octaedro con distintos grados de desplazamiento.

### 5.3.6. Terreno

El último modelo en que se probaron métricas de teselación fue el de un modelo que representa un gran terreno. Este terreno tiene la característica de que todas las posiciones se calculan dinámicamente según el nivel de detalle requerido.

Se parte de una malla gruesa de patches de cuadriláteros en el plano, compuesta de unos pocos elementos. En el ejemplo que se muestra luego, esta malla está compuesta de 16 elementos ordenados en una grilla de 4 elementos por lado. Además se tiene una textura de alturas donde cada píxel indica una altura. Al dibujar el terreno, cada punto es mapeado a la textura para obtener su altura. Luego, este punto es desplazado verticalmente según esta altura para obtener su posición final. El tamaño de la textura utilizada es de 1024 píxeles por lado.

La ventaja de generar el terreno de esta manera es que se reduce la infor-

mación a transferir a la GPU y sólo se construye y dibuja la información que se necesita.

Se pretende visualizar con más detalle el terreno más cercano al ojo. Para ello se aplicaron dos métricas. La primera es la de distancia y la segunda es la de arista de máxima longitud proyectada. Para calcular el nivel de teselación de un patch, primero se calculan los desplazamientos verticales de sus vértices. Luego se aplica la métrica y en el TES se calculan las posiciones finales desplazadas verticalmente de todos los nuevos vértices generados.

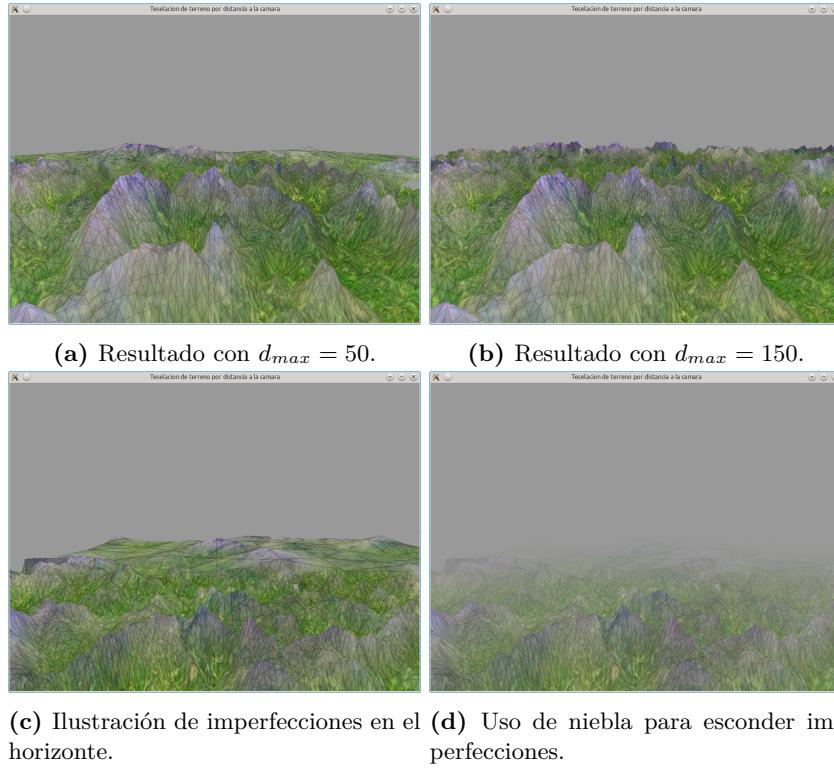
Cabe aclarar que ambas técnicas adolecen de un defecto. La ‘skyline’, la cual es la línea que divide terreno y cielo, es muy cambiante entre cuadro y cuadro, pues los elementos más alejados están muy pocos teselados. Reconocer esta situación y teselar más las ‘montañas’ que tapan el cielo es imposible sin una teselación previa. Por lo tanto, se hace necesario la aplicación de niebla o alguna técnica que minimice ese defecto indeseable. Otro método pensable sería guardar el cuadro actual como textura para interpolarlo con el cuadro siguiente. Pero como aquí sólo se quiere mostrar el efecto de la teselación por hardware, esta última técnica no fue implementada.

### Distancia

En el caso de la aplicación de esta métrica, primero se mide la distancia del punto de vista al centroide del patch. Además, se calcula la distancia a los extremos y a los puntos medios de las aristas. La distancia final que se utiliza en cada frontera es la menor de las obtenidas con respecto a dicha frontera. A partir de esta distancia, se realiza un mapeo al nivel de teselación.

En la Fig. 5.12 se exponen capturas sobre la aplicación de esta métrica. Se utiliza un conjunto de texturas para darle mayor realismo a la escena. En las Fig. 5.12a y Fig. 5.12b se muestran dos capturas distintas del terreno con distintos niveles de  $d_{max}$ . Se puede ver que a medida que  $d_{max}$  aumenta, se pasa a aplicar el refinamiento en patches alejados de la cámara.

En la Fig. 5.12c se presenta una escena distinta. En este caso, se notan imperfecciones en los elementos alejados, ya que su nivel de teselación es muy bajo. Para contrarrestar este problema, se aplica una función de niebla que oculta hasta una cierta distancia los elementos. En la Fig. 5.12d se expone la misma escena pero con el agregado de una función de niebla. Como ya se dijo, la niebla se emplea para ocultar las imperfecciones del fondo. Para distancias grandes, donde la niebla es muy densa, las primitivas se descartan, reduciendo la carga de procesamiento. El uso de este tipo de funciones es muy común en videojuegos para evitar que se genere un corte abrupto de los objetos en el horizonte de la escena.



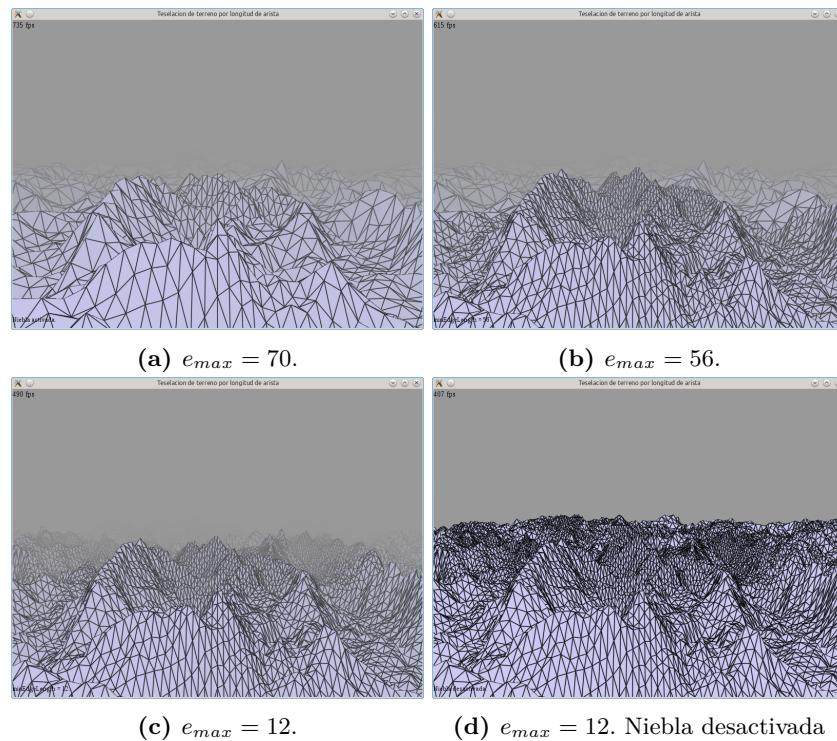
**Figura 5.12:** Aplicación de la métrica de distancia a un gran terreno.

### Arista proyectada

Al modelo de terreno también fue aplicada la métrica de máxima longitud de arista proyectada. En la Fig. 5.13 se presenta el resultado de la aplicación de esta métrica. Se incluyen distintos ejemplos con la variación del parámetro  $e_{max}$ , el cual identifica la cantidad máxima de píxeles que puede ocupar una arista en pantalla. Se puede apreciar que mientras más chico sea este valor, mayores niveles de teselación son generados. En la Fig. 5.13a se muestra un valor grande de  $e_{max}$ . En las dos figuras sucesivas, las Fig. 5.13b y 5.13c, se reduce el valor de  $e_{max}$ . Por último, en la Fig. 5.13d se muestra el mismo resultado pero con la niebla desactivada. De esta manera se puede apreciar con claridad el efecto que produce el uso de niebla en este modelo.

En principio, esta métrica debería dar triángulos con longitudes de arista uniformes, ya sea que estén en el fondo o en el frente. Pero en la práctica, los triángulos muy cercanos teselados al máximo dan aristas visuales más grandes que los grandes triángulos sin teselar en el fondo. Esto es causado por la cantidad limitada de elementos que tiene la malla plana que se utiliza. Como se dijo, esta malla está compuesta por sólo 16 elementos (4 aristas por lado), y es en esta malla donde se realizan los cálculos de longitud proyectada. Por lo tanto, se toman pocas medidas de longitud de arista y se tiende a sobreestimar o subestimar el valor de teselación a definir en cada una. Una alternativa posible

es aumentar el tamaño de la malla plana, de manera de tener más aristas para calcular su longitud y realizar teselación adaptativa con mayor precisión. Sin embargo, aumentar la cantidad de elementos en la malla plana implica tener que almacenar y transferir más información desde la CPU a la GPU. Por lo tanto, el programador debe evaluar cuál opción es preferible según sus necesidades y los recursos disponibles.



**Figura 5.13:** Aplicación de la métrica de longitud proyectada a un terreno.

## Conclusión

Como se puede apreciar, se logró desarrollar un software completo que permite la apreciación de la tecnología de teselación en GPU. Para entender cómo se generan los nuevos puntos, se hace esencial consultar el primer ejemplo, el cual muestra los distintos métodos de espaciado para los distintos niveles de teselación. El resto de los ejemplos presentan modelos y métricas aplicadas a éstos.

Debido a cómo está organizado el software, la incorporación de nuevos ejemplos es muy simple, ya que sólo se debe crear una clase que herede de `Example` e implemente su propia versión de `Example::DrawScene()`. Esta función debe cargar los shaders en el contexto de OpenGL y enviar los datos a renderizar.

Las métricas aplicadas a los modelos pueden ser utilizadas en videojuegos para aumentar la calidad general de la escena. Sin embargo, al aumentar el refinamiento de los polígonos, se aumenta la carga de procesamiento, por lo que es importante balancear la búsqueda de mayor calidad con los costes de cálculo del hardware.

En algunas aplicaciones, puede ser necesario aplicar una combinación de métricas para obtener el resultado requerido. Por ejemplo, en el modelo de terreno se podría aplicar una métrica de silueta para los bordes, y una de longitud de arista para el resto del modelo.

# Capítulo 6

## Resultados

Como se expresó en la introducción, el objetivo principal de este proyecto fue el de estudiar la tecnología de teselación en GPU. Luego de un estudio minucioso de la tecnología, se desarrollaron e implementaron distintos modelos-ejemplo para apreciar su utilidad. Los modelos se diseñaron para que sean simples y no desvíen la atención del proyecto. Sin embargo, fueron de suma utilidad para tener una perspectiva general de las capacidades de teselación en GPU.

En la primera fase de investigación de este proyecto se pudo reconocer la necesidad de los desarrolladores de tener una etapa programable dedicada a teselación. Dichos desarrolladores pertenecen principalmente al área de computación gráfica, donde siempre se busca crear mundos más realistas y complejos minimizando el esfuerzo de trabajo tanto humano como computacional. El uso de teselación reduce el esfuerzo al permitir la generación dinámica de nueva geometría en forma automática. De todos modos, se debe tener cuidado con el refinamiento desmedido, ya que más geometría se traduce en más carga computacional. Además de las originales, cada primitiva que se agregue en el TES deberá pasar por las etapas (programables o no) que siguen al TES en el pipeline gráfico.

Ademas, puede ocurrir que la nueva geometría perjudique la calidad visual en lugar de beneficiarla. Debido a que los niveles de teselación se deben especificar por arista de polígono, con frecuencia ocurre que, en polígonos adyacentes, la arista tiene niveles de teselación distintos. Esto genera las llamadas uniones ‘en T’. Dependiendo de la aplicación, se puede tolerar o no la presencia de estas imperfecciones. Todas las métricas desarrolladas evitan las uniones en T al calcular el grado de refinamiento por arista. Sin embargo, estas métricas se aplican bajo el supuesto de que el modelo es correcto y que las uniones entre partes son conformes. En el ejemplo de Phong Tessellation es donde más perjudican las uniones en T. En este caso, como los vértices son desplazados, si a través de una arista se tienen niveles de teselación distintos, el desplazamiento de los vértices será diferente y se generará un hueco. En general, es conveniente evitar las uniones en T, aún a costa de complejizar la métrica.

No todas las métricas son útiles a todos los modelos. Por ejemplo, la métrica de distancia no es de utilidad para el modelo de personaje en un videojuego, ya que hay poca variación de posición entre polígonos de un vecindario chico. En este caso, la métrica no es adaptativa para cada polígono del modelo, si no que se puede calcular una única distancia (en CPU) y definir los niveles de teselación para todos los polígonos del modelo. En cambio, esta métrica fue aplicada al modelo de superficies de Bézier porque en este caso los patches son grandes y, a partir de pocos puntos ubicados en un vecindario no muy reducido, se generan superficies curvas de gran detalle. De todas maneras, se encontró que la métrica de curvatura es más adecuada para el caso de curvas y superficies.

En el modelo de terrenos fue necesario el uso de la corrección diádica. Debido a que todo el terreno es generado dinámicamente en cada cuadro, con frecuencia sucede que una variación pequeña de la posición de la cámara ocasionaba variaciones pequeñas en los niveles de teselación. Estas variaciones pequeñas en los niveles de teselación se convertían en variaciones grandes en la posición de los vértices, ya que se muestreaba un valor distinto de la textura de desplazamientos verticales. Por lo tanto, se generaba un efecto de ‘ola’ al moverse por la escena. Para evitar este problema se utilizó la corrección diádica. De este modo, se anula el efecto de ‘ola’ al reducir los niveles posibles de teselación a únicamente potencias de 2. Además, si se usan correctamente el parámetro  $d_{max}$  y la niebla, el cambio de nivel de teselación se puede realizar a una distancia donde la niebla oculta (parcialmente) el ‘salto’, reduciendo el efecto de ola.

La teselación en GPU presenta tres modos de espaciado de puntos: simétrico, par o impar. Sin embargo, no se identificaron casos puntuales donde conviene utilizar uno sobre el otro. En general, todos los ejemplos presentados se realizaron utilizando espaciado simétrico. Notar que si se utiliza teselación diádica, todos los niveles de teselación son inherentemente pares.

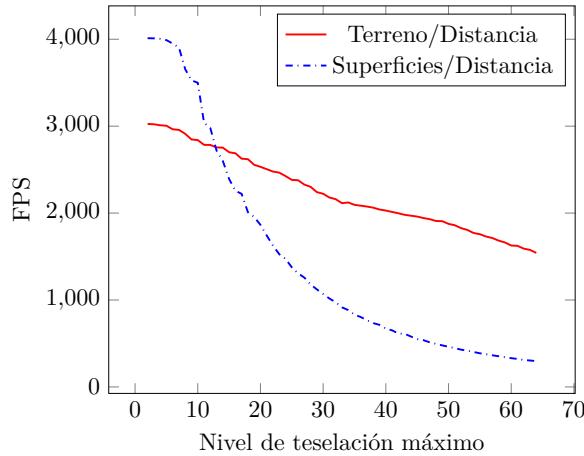
Con respecto al rendimiento, la generación de nueva geometría tiene un impacto directo en el desempeño del software. Una placa aceleradora de video tiene una gran cantidad de núcleos (en el orden de los cientos o miles). Cada núcleo procesa en paralelo la información a lo largo de todo el pipeline. Si la GPU no tiene suficientes núcleos, la geometría transita en grupos por todo el pipeline. Toda la geometría nueva debe añadirse a la cola de procesamiento y por lo tanto se incrementa la cantidad de grupos a procesar. Mientras más información se tenga, más lento será el procesamiento por cuadro. Por consiguiente, un aumento excesivo en la carga por el uso de altos niveles de teselación influye directamente en el desempeño del software. En los ejemplos implementados fue necesario definir un tope máximo de niveles de teselación, para evitar sobrecargar la GPU. Sin embargo, este tope depende exclusivamente del hardware de video que posea el equipo.

En una aplicación comercial, este tope debería calcularse en función de la potencia del hardware en que se ejecute. Debido a que no se tiene a disposición distintos tipos de hardware gráfico, no se pudo ensayar un método para calcular el tope de teselación, pero se estima que dependerá de la capacidad de parallelización que tenga la placa gráfica. Por lo tanto, una GPU que tenga mayor cantidad de hilos de ejecución, deberá definir un tope de teselación mayor.

En la Fig. 6.1 se presenta un gráfico donde se ilustra la relación entre el nivel máximo de teselación y el rendimiento de la aplicación. El rendimiento está calculado como un promedio de los FPS (*frames per second*) a lo largo de 10 segundos de ejecución de una escena estática. Sin embargo, notar que lo que se mide aquí son los cuadros *ejecutados* por segundo pero no se miden los *dibujados*. El hardware de la pantalla no puede dibujar más de una cierta cantidad de cuadros por segundo. Este límite está impuesto por la velocidad de refresco vertical del mismo. Según la tecnología del monitor, esta velocidad de refresco permite dibujar hasta, por ejemplo, 60 cuadros por segundo. Por lo tanto, lo que se está midiendo es la cantidad de veces que se ejecuta el callback de dibujado de GLUT, esta medida también se denomina *frame rate*.

Se expone el resultado para dos ejemplos distintos. El primer ejemplo (color rojo) corresponde al renderizado de un terreno con la métrica de longitud de arista. El segundo ejemplo (color azul) pertenece al renderizado de una escena compuesta por tres modelos de superficies de Bézier, y también se emplea la métrica de distancia. En el eje horizontal se representa el nivel máximo de teselación disponible. Por su parte, el eje vertical indica los FPS obtenidos para cada muestra. Se puede ver que, para ambos ejemplos, a medida que aumenta el nivel máximo de teselación, el rendimiento decrece. Esto es debido a que aumenta la carga de datos a rasterizar luego de la teselación.

Sin embargo, vale notar que el rendimiento depende no sólo de la escena que se esté dibujando, si no de los parámetros que gobiernan la métrica de teselación utilizada. Por ejemplo, si se modifica la distancia máxima en la métrica de distancia, el rendimiento se verá afectado porque es posible que se generen distintos niveles de teselación para cada patch. De todas maneras, esta figura es de utilidad para ver el impacto en el desempeño que tiene el utilizar niveles altos de teselación, aún con adaptatividad. Resultados similares se obtuvieron para el resto de los ejemplos.



**Figura 6.1:** Relación entre FPS y nivel máximo de teselación para algunas de las aplicaciones desarrolladas.

Con respecto al hardware, el software desarrollado se probó en dos equipos

distintos. En la Tabla 6.1 se exponen las características de ambos equipos. Por otra parte, en la Tabla 6.2 se exponen las características de las placas aceleradoras de video que tiene cada uno. Estas características fueron obtenidas de la especificación oficial de los fabricantes de las GPU.

El Equipo1 (E1) es la computadora personal del autor del proyecto. Por otro lado, el Equipo2 (E2) es una computadora ubicada en el Aula FICH-CIMNE de la Facultad de Ingeniería y Ciencias Hídricas, donde tiene su oficina el director del proyecto.

Característica	Equipo1 (Personal)	Equipo2 (FICH-CIMNE)
<i>Procesador central</i>	Intel Core 2 Duo e8400	Intel Core i7-870
<i>Núcleos del CPU</i>	2	4
<i>Hilos del CPU</i>	2	8
<i>Velocidad CPU</i>	3 GHz.	2.93 GHz.
<i>Memoria principal</i>	2 GB.	8 GB.
<i>GPU</i>	ATI Radeon HD 5770	NVIDIA GeForce GTX 470
<i>Sistema operativo</i>	Kubuntu Linux 13.04	Ubuntu Linux 12.04 LTS

**Tabla 6.1:** Características generales de los equipos.

Característica	Equipo1	Equipo2
<i>Cantidad de núcleos</i>	800	448
<i>Memoria</i>	1024 MB.	1280 MB.
<i>Velocidad procesador</i>	850 MHz.	607 MHz.
<i>Velocidad de memoria</i>	2400 MHz.	1674 MHz.
<i>Ancho de banda de memoria</i>	76.8 GB/s	133.9 GB/s

**Tabla 6.2:** Características de las GPU de los equipos.

En el E1 se escribió y probó todo el software. Una vez avanzado el proyecto, se lo probó en el E2. Si bien la GPU del E1 es más moderna que la del E2, el rendimiento del software es mejor en el segundo equipo. La potencia de una GPU no es lo único que se debe tener en cuenta al analizar el rendimiento del software en ejecución. El E2 tiene un CPU más potente, el cual que puede procesar sin problemas el software desarrollado mientras ejecuta los servicios del sistema operativo. Por otro lado, el CPU del E1 es menos potente y tiene menos capacidad de paralelización, lo que reduce la dedicación (en ciclos de procesamiento) al software desarrollado. Por lo tanto, a pesar de tener una GPU de menor potencia, el E2 presentó un mejor desempeño ya que cuenta, en general, con un hardware más moderno y potente.

Fue necesario realizar cambios menores para poder ejecutar el proyecto en el Equipo2. Según lo experimentado, los drivers de video de la marca ATI son más permisivos con respecto a los de la marca NVIDIA. Esto fue puesto en evidencia al notar que algunos ejemplos daban errores de ejecución y se cerraban sin aviso. Dichos errores no fueron detectados en el Equipo1, donde el software se ejecutaba con normalidad. De todas maneras, una vez detectados los errores, su corrección fue sencilla y se pudo obtener un software capaz de ejecutarse en el hardware de ambos fabricantes.

## Capítulo 7

# Conclusiones

El uso de la tecnología de teselación abre la puerta para una nueva forma de controlar el nivel de detalle de los entornos tridimensionales. Esta tecnología ayuda a aumentar la calidad general del escenario sin la necesidad de tener almacenados modelos con distinto nivel de detalle. El mayor peso recae en el correcto diseño de las métricas que definen el nivel de teselación para los polígonos. En este proyecto se partió desde un nulo conocimiento sobre el uso de teselación en GPU y se logró diseñar, desarrollar e implementar distintos modelos donde se aplican las métricas de teselación. En el diseño de estas métricas se buscó que se respetaran las características de simplicidad, poca información y adaptatividad. Utilizando las métricas se obtuvieron buenos resultados y se logró controlar el nivel de detalle de los modelos.

Los parámetros que gobiernan el control de cada métrica fueron aproximados experimentalmente según la calidad que se quería lograr y el hardware que se contaba. Sin embargo, si se quiere utilizar teselación en una aplicación comercial, es necesario encontrar un método para definir los parámetros automáticamente, ya que el hardware en donde se ejecute la aplicación puede variar drásticamente.

Además, se adquirió una invaluable experiencia en el desarrollo de software con la versión más moderna de la API OpenGL. Al terminar el proyecto, el alumno cuenta con un amplio conocimiento sobre el uso y comportamiento de la API, lo que facilitará el desarrollo de nuevas aplicaciones utilizando OpenGL.

### 7.1. Trabajo futuro

Para un futuro, se plantea continuar con dos líneas de trabajo que fueron dejadas de lado en este proyecto.

### 7.1.1. Teselación en ingeniería

La primer línea de trabajo consiste en el uso de teselación adaptativa aplicada a problemas reales de ingeniería. Los problemas de ingeniería hacen uso de métodos de simulación tales como el método de los elementos finitos (MEF). En este método se tienen mallas bidimensionales o tridimensionales afectadas por algún fenómeno en particular. El fenómeno, tal como una fuerza o una variación de temperatura, puede tener distinto efecto en distintas posiciones de la malla. El cálculo de elementos finitos consiste en simular varios pasos de tiempo donde la malla es afectada por dicho fenómeno. Lo que interesa entonces es ver cómo la geometría (posición de los vértices de la malla) se deforma al ser afectada por el fenómeno. Por ejemplo, si se simula el efecto del viento o la marea en una masa de agua, lo que interesa es ver cómo se generan olas y remolinos en dicho fluido.

Esta malla puede ser alterada utilizando teselación y así tener un mejor detalle en las áreas de interés. De este modo, al utilizar teselación se pueden refinar áreas específicas del objeto a simular, dando lugar a un cálculo más exacto del MEF.

### 7.1.2. Teselación en superficies de subdivisión

Como se dijo anteriormente, se analizó la posibilidad de utilizar teselación en superficies de subdivisión, pero en un análisis rápido se concluyó que se necesita un método para calcular las posiciones finales de los vértices a partir de las iniciales. Se espera que una investigación más extensa podría dar con el método de obtención de estas posiciones.

Durante el estudio del estado del arte en la temática, se consultaron algunos trabajos donde se aproximaba la forma de una superficie de subdivisión a partir de descomponerla en superficies paramétricas. Por lo tanto, se tiene evidencia en que existe un método para calcular dichas posiciones, aunque sólo sean aproximaciones.

# Bibliografía

- [1] T. Akenine-Möller, T. Haines, y N. Hoffman. *Real-time rendering*. A K Peters Ltd., third edition, 2008.
- [2] (Open Asset Import Library) Assimp. Biblioteca portable para la importación de modelos tridimensionales. <http://assimp.sourceforge.net/>, 2008–2012. [Online; consultado el 07 de Junio de 2014].
- [3] T. Boubekeur y M Alexa. Phong tessellation. *ACM Transactions on Graphics*, 27(5), 2008.
- [4] T. Boubekeur y C. Schlick. A flexible kernel for adaptive mesh refinement on GPU. *Computer Graphics Forum*, 27(1), 2008.
- [5] NVIDIA Corporation. *NVIDIA CUDA programming guide 2.0*. 2008.
- [6] DevIL. Biblioteca multiplataforma para lectura en memoria de imágenes. <http://openil.sourceforge.net/>, 2000–2010. [Online; consultado el 07 de Junio de 2014].
- [7] Direct3D. API para facilitar el manejo de recursos multimedia. <http://windows.microsoft.com/es-AR/windows7/products/features/directx-11>. [Online; consultado el 07 de Junio de 2014].
- [8] J. P. Dorsch. Visualización de campos vectoriales sobre superficies mediante la técnica de convolución sobre integral de línea. Master's thesis, Facultad de Ingeniería y Ciencias Hídricas, Universidad Nacional del Litoral, 2012.
- [9] C. Dyken, M. Reimers, y J. Seland. Real-time GPU silhouette refinement using adaptively blended Bézier patches. *Computer Graphics Forum*, 27(1), 2007.
- [10] C. Dyken, M. Reimers, y J. Seland. Semi-uniform adaptive patch tessellation. *Computer Graphics Forum*, 28(8), 2009.
- [11] FreeType. Motor gratuito, portable y de alta calidad para el renderizado de textos en pantalla. <http://www.freetype.org>. [Online; consultado el 07 de Junio de 2014].
- [12] K. Gee. DirectX 11 tessellation. 2008.

- [13] GLM. Biblioteca multiplataforma para programación gráfica. <http://glm.g-truc.net/>, 2005–2013. [Online; consultado el 07 de Junio de 2014].
- [14] K. Moule y M. D. McCool. Efficient bounded adaptive tessellation of displacement maps. *Graphics Interface*, 2002.
- [15] OpenGL. API para el renderizado de gráficos 3D. [www.opengl.org/](http://www.opengl.org/). [Online; consultado el 07 de Junio de 2014].
- [16] A. Patney y J. D. Owens. Real-time Reyes-style adaptive surface subdivision. *ACM Transactions on Graphics*, 27(5), 2008.
- [17] D. Salomon. *Curves and surfaces for computer graphics*. Springer, 2006.
- [18] M. Schwarz, M. Staginski, y M. Stamminger. GPU-based rendering of PN triangle meshes with adaptive tessellation. *Vision, Modeling, and Visualization*, pages 161–168, 2006.
- [19] M. Schwarz y M. Stamminger. Fast GPU-based adaptive tessellation with CUDA. *Computer Graphics Forum*, 28(2), 2009.
- [20] M. Segal y K. Akeley. *The OpenGL graphics system: A specification*. 2010.
- [21] TF3DM.com. 3d models for free. <http://tf3dm.com/>. [Online; consultado el 07 de Junio de 2014].
- [22] GLUT (OpenGL Utility Toolkit). Biblioteca de herramientas para escribir programas con OpenGL. <http://www.opengl.org/resources/libraries/glut/>. [Online; consultado el 07 de Junio de 2014].
- [23] D. Wolff. *OpenGL 4.0 Shading Language Cookbook*. Packt, 2011.

# **Apéndice: Propuesta de proyecto**

A continuación, se incluye la versión original de la propuesta de proyecto, la cual fue confeccionada antes de iniciar el mismo.



UNIVERSIDAD NACIONAL DEL LITORAL  
FACULTAD DE INGENIERÍA Y CIENCIAS HÍDRICAS

PROPUESTA DE PROYECTO FINAL DE CARRERA  
INGENIERÍA EN INFORMÁTICA

---

**Desarrollo de algoritmos de teselación  
adaptativa en GPU**

Alumno: Fernando Nellmeldin

Director: Dr. Néstor Calvo

*Palabras clave:* refinamiento dinámico, programación en GPU, nivel de detalle

---

Santa Fe, Septiembre de 2013

# Índice

<b>1. Introducción</b>	<b>2</b>
<b>2. Justificación</b>	<b>3</b>
<b>3. Objetivos perseguidos</b>	<b>4</b>
3.1. Objetivo general . . . . .	4
3.2. Objetivos específicos . . . . .	4
<b>4. Alcances</b>	<b>4</b>
<b>5. Metodología</b>	<b>5</b>
<b>6. Plan de tareas</b>	<b>5</b>
<b>7. Cronograma</b>	<b>7</b>
<b>8. Puntos de control y entregables</b>	<b>7</b>
<b>9. Riesgos y estrategias de mitigación</b>	<b>8</b>
<b>10. Recursos disponibles y necesarios</b>	<b>9</b>
<b>11. Presupuesto</b>	<b>10</b>
<b>12. Bibliografía</b>	<b>13</b>

## 1. Introducción

Las tecnologías de hardware han tenido un crecimiento exponencial desde la concepción misma de la informática, y en particular desde su aceptación por el público no especializado. En los primeros años, el objetivo era minimizar el espacio ocupado por las máquinas; en esta etapa se sitúa el lanzamiento de la primera PC de IBM en el año 1981 [1]. Más tarde la industria del hardware se concentró en dotarlas de más potencia, principalmente en sus microprocesadores; este paradigma se mantuvo durante más de 20 años [2]. Desde ese momento se bifurca la tendencia, por un lado hacia máquinas más potentes, con sus problemas de peso y refrigeración; y por el otro en dispositivos livianos, portátiles y agradables al usuario hogareño.

Durante la década de 1980, y en paralelo con el avance en la tecnología de los microprocesadores, las placas aceleradoras de video evolucionaron de manera de brindarles una distintiva capacidad para realizar cálculos geométricos a gran escala y velocidad [3]. Fue en aquella época cuando los videojuegos comenzaron a mostrar un gran nivel de detalle en entornos tridimensionales. A diferencia de los microprocesadores para la CPU, los coprocesadores gráficos o GPU, pudieron evolucionar con arquitecturas totalmente innovadoras, debido a que no tenían el requisito de compatibilidad hacia atrás.

Los entornos 3D se generan a partir de una gran cantidad de polígonos conectados entre sí formando mallas. Los polígonos elegidos son triángulos y cuadriláteros, debido a su simplicidad. Así, cada escenario y personaje de una animación cinematográfica o de un videojuego, está representado por un conjunto de polígonos que le dan forma.

La cantidad de elementos utilizados para mostrar mundos virtuales crece año tras año, siguiendo las exigencias del consumidor. Cuanto más polígonos contenga un modelo en particular, se apreciará un mejor nivel de detalle. En la misma línea, los ingenieros de hardware se enfrentaron a la tendencia de crear mundos de gran tamaño y complejidad para la industria de videojuegos. En ésta, se necesita renderizar simultáneamente un gran volumen de polígonos que den vida a toda la escena con un nivel de detalle adecuado. Por otra parte, todo lo anterior se refiere al renderizado de una sola imagen en 3D; en un videojuego, se renderizan varias imágenes por segundo, con la geometría cambiante en forma fluída, de manera que la experiencia visual del usuario sea placentera.

Para aliviar esos inconvenientes, surge la tecnología de teselación adaptativa, una solución de hardware y software que permite generar mundos grandes (muchos polígonos) y con gran detalle. Esta tecnología de teselación comenzó a utilizarse ya desde su implementación en la API multimedia Direct3D 11, presentada en 2008 [4].

La teselación adaptativa consiste en actualizar dinámicamente el nivel de detalle de una región en particular de la escena. Se parte de una malla gruesa (de poca densidad) que se refina a medida que se requiere. Para los objetos que estén próximos a la cámara, y por lo tanto deban verse de cerca, se genera un mallado denso que los representa con gran cantidad de polígonos. Por otra parte, a los objetos alejados de la cámara y que no requieran una alta precisión, se los representa con una malla de pocos elementos. Cuando la cámara se mueve, el mallado se regenera dinámicamente para obtener más detalles de los nuevos objetos centrados en la pantalla y reducir la carga de los que ya no están en foco.

En la GPU, la flexibilidad atenta contra la velocidad de procesamiento que tiene el software rígido implementado directamente en el hardware. Para permitir programar teselación en forma dinámica, fue necesario introducir una etapa programable específica en el rígido *pipeline* gráfico de la GPU, y modificar el software para que lo interprete.

En la actualidad, la necesidad de generar mundos de gran tamaño, no sólo se halla en los videojuegos. Los entornos tridimensionales también se encuentran, por ejemplo, en los simuladores de vuelo necesarios para el entrenamiento de pilotos y otros productos de similares características. Además, distintas áreas de ingeniería hacen uso de simulaciones para el estudio de fenómenos específicos; ya sea para la visualización o para el cálculo numérico de sistemas de ecuaciones diferenciales. En todos estos casos, la posibilidad de adaptar la geometría de la malla, refinando y desrefinando sectores según el interés variable, presenta una notable y deseable mejora en el procesamiento de la escena y/o el cálculo numérico.

## 2. Justificación

La vertiginosa velocidad de desarrollo de las tecnologías dificulta que un profesional se mantenga actualizado en la totalidad de los últimos avances de su área. En el caso concreto de computación gráfica, y si bien se cuenta con un curso de grado en nuestra facultad sobre el tema, los contenidos allí brindados abarcan los fundamentos del área. Estos fundamentos son de importancia primaria para el desarrollo del alumno que desee dedicarse a computación gráfica, ya que sin un manejo adecuado de éstos, no se podrían abordar los temas más avanzados. Sin embargo, poco espacio queda en el planeamiento general del curso y de la carrera en sí, para la adición de tópicos más avanzados de computación gráfica, ya que éstos deben convivir con cursos en otras áreas.

El principal propósito del presente proyecto es el de realizar un estudio minucioso sobre las técnicas de teselación para la generación y actualización dinámica de mallas. La presentación de esta tecnología, a partir del análisis, desarrollo e implementación de algunos modelos ejemplificadores, será la clave para analizar sus ventajas y desventajas. Adicionalmente, y como subproducto, se busca acercar a la comunidad universitaria los últimos conocimientos desarrollados en el área. A partir de la presentación cuidadosamente estudiada y comprendida de tecnologías innovadoras, se pretende estimular a la comunidad local en el desarrollo e investigación en el área de computación gráfica, mostrando cuáles son las posibilidades de las tecnologías recientes.

Previamente al agregado de teselación en GPU, los desarrolladores trabajaron en diseñar técnicas que generaran nivel de detalle dinámico con algoritmos puramente en CPU, o utilizando las capacidades reducidas de las GPU antiguas. Una vez disponible la tecnología de teselación, se espera que estos algoritmos hayan quedado en desventaja frente a una implementación pura y directa en la GPU. Con este proyecto, se pretende analizar la aceptación y utilización de esta tecnología en la industria, y desarrollar e implementar pruebas propias para obtener una conclusión referente a sus ventajas y desventajas.

Este trabajo permitirá cumplir con los objetivos, demandando muy poco dinero para su desarrollo, pues se dispone del hardware requerido. Por otra parte, los objetivos están plenamente justificados en lo individual, por la ganancia en conocimientos aplicables en la profesión, y en lo institucional, por la trasmisión de dichos conocimientos a la comunidad uni-

versitaria, tanto en herramientas didácticas como en puntas para iniciar futuros desarrollos productivos o de investigación.

Por último, y a nivel personal, el interés del alumno en el estudio y desarrollo de tecnologías para videojuegos, y la predilección por el desarrollo de videojuegos en general, resulta como principal motivación para la realización del proyecto. La oportunidad de comprender cómo los profesionales solucionan problemas que surgen de las exigencias de la misma industria, resulta en un gran atractivo para el alumno, quien siempre se ha planteado soluciones propias ante los problemas que ha sabido identificar.

### **3. Objetivos perseguidos**

#### **3.1. Objetivo general**

Investigar, desarrollar e implementar un conjunto de técnicas de teselación para la actualización de mallas y aplicarlas en modelos-ejemplo que requieran un nivel de detalle variable.

#### **3.2. Objetivos específicos**

Los objetivos específicos propuestos son:

- Investigar bibliotecas que permitan la teselación en placas aceleradoras de video.
- Analizar, desarrollar e implementar técnicas de teselación.
- Investigar, desarrollar e implementar aplicaciones de teselación.
- Adquirir experiencia con el trabajo de mallas y con teselación adaptativa.
- Colaborar con la comunidad universitaria local al presentar tecnologías innovadoras del área de computación gráfica.

### **4. Alcances**

El presente trabajo tiene como objetivo investigar, desarrollar e implementar algoritmos de teselación en placas aceleradoras de video de última generación. Por lo tanto y a pesar de que existen técnicas de teselación ya desarrolladas que no hacen uso de las nuevas tecnologías, se excluye del trabajo planificado el desarrollo de algoritmos que no utilicen exclusivamente las capacidades de teselación de la GPU.

Para probar dichos algoritmos, se propone el uso de modelos tridimensionales simples que sean suficientes para mostrar las características de las técnicas desarrolladas. La complejidad de estos modelos se decidirá acorde a las técnicas que se desarrollen, y se intentará que se

mantengan lo más simple posible para no desviar así la atención en el desarrollo. Adicionalmente, se evaluará la posibilidad de realizar pruebas de teselación en escenas complejas propias de un videojuego, de manera de analizar sus características en esta industria.

Para finalizar, se propone realizar un análisis cuantitativo y cualitativo con respecto a la velocidad y complejidad de renderizado de estas escenas. De este modo, se busca analizar las ventajas y desventajas de utilizar teselación en diversidad de aplicaciones.

## 5. Metodología

Para el desarrollo del presente proyecto se utilizará la metodología de trabajo del modelo en espiral. Este modelo propone agrupar las actividades en fases que se repiten en forma de ciclos a medida que se avanza. Al inicio de cada iteración, se propone un conjunto de objetivos a corto plazo a cumplimentar. Al finalizar cada bucle, se realiza un contraste entre los objetivos propuestos y los alcanzados y se efectúa un análisis de riesgos, de manera de plantear los objetivos para la próxima iteración.

El proceso comienza con el análisis del problema, por lo que el alumno debe realizar una búsqueda bibliográfica sobre el estado del arte en la temática. A partir de la comprensión del problema, las etapas 2 y 3 consisten en el análisis y selección tanto de las herramientas a utilizar (etapa 2), como de las técnicas de teselación propiamente dichas (etapa 3). Luego de esta selección, se procederá al desarrollo de técnicas de teselación específicas en la etapa 4. En la quinta etapa, al finalizar el desarrollo, se realizará un estudio comparativo para evaluar las características del método estudiado. Pero todo este proceso se realiza en forma espiral con pequeños avances en cada ciclo. De esta manera, se realizan varias iteraciones abarcando ciertas actividades de las etapas 3 y 4, donde el alumno debe repetir el ciclo: análisis-diseño-desarrollo de cada técnica, realizar las pruebas correspondientes y llevar a cabo la integración con el trabajo ya realizado. Además, antes de avanzar, el alumno debe adquirir nuevos conocimientos que le sean necesarios para el desarrollo de la próxima iteración.

El modelo de trabajo en espiral es elegido para el presente proyecto, ya que su flexibilidad brinda oportunidades de elegir los objetivos en base al avance que se realiza. Además, en cada iteración se tiene un prototipo funcional, de manera de poder evaluar el desempeño al finalizar cada ciclo, sin esperar a la finalización del proyecto. Si durante el trabajo se encontraran nuevas técnicas de teselación, se evaluará el tiempo y dificultad de desarrollarlas según el estado de avance.

## 6. Plan de tareas

Para la realización del proyecto, se propone el siguiente conjunto de etapas, discriminado por las tareas que se detallan en cada uno. Las etapas 1 y 2 se realizarán en forma secuencial. En la actividad 3.2 se definirá cuántas iteraciones del modelo en espiral se van a realizar, y cuáles son las técnicas de teselación que se van a implementar en cada etapa. Se utilizarán las iteraciones del modelo en espiral para la realización de las etapas 3.3 (análisis), 3.4 (diseño), 4.1 (desarrollo) y 4.2 (pruebas unitarias), donde se iterará por cada técnica o grupo de

técnicas a desarrollar. En este grupo de tareas es en donde se debe iterar, ya que allí se encuentra la mayor carga de desarrollo de software para el alumno.

**1. Lectura de bibliografía [60 horas]**

- 1.1 Investigación sobre el estado del arte en teselación. [40 horas]
- 1.2 Investigación sobre representación de modelos tridimensionales. [20 horas]

**2. Análisis y selección de herramientas [105 horas]**

- 2.1 Búsqueda bibliográfica sobre bibliotecas y herramientas de desarrollo de software. [20 horas]
- 2.2 Análisis de bibliotecas y herramientas a utilizar. [20 horas]
- 2.3 Selección de bibliotecas y herramientas. [10 horas]
- 2.4 Adquisición de experiencia en el desarrollo utilizando las bibliotecas y herramientas seleccionadas. [40 horas]
- 2.5 Redacción de primer informe de avance. [15 horas]

**3. Análisis, selección y diseño de técnicas y modelos [135 horas]**

- 3.1 Análisis general de técnicas. [20 horas]
- 3.2 Selección de las técnicas a desarrollar. [10 horas]
- 3.3 Análisis de cada una de las técnicas. [40 horas]
- 3.4 Diseño de cada una de las técnicas. [20 horas]
- 3.5 Redacción de segundo informe de avance. [15 horas]
- 3.6 Análisis de modelos tridimensionales para la evaluación de las técnicas. [20 horas]
- 3.7 Selección de modelos. [10 horas]

**4. Desarrollo de técnicas [255 horas]**

- 4.1 Desarrollo de cada una de las técnicas de teselación seleccionadas. [100 horas]
- 4.2 Pruebas de funcionalidad unitarias de cada técnica desarrollada. [30 horas]
- 4.3 Redacción de tercer informe de avance. [15 horas]
- 4.4 Desarrollo de los modelos tridimensionales. [50 horas]
- 4.5 Desarrollo de interfaz para la aplicación de las técnicas a los modelos. [20 horas]
- 4.6 Pruebas integrales del software desarrollado. [40 horas]

**5. Análisis comparativo [45 horas]**

- 5.1 Comparación de las técnicas según complejidad algorítmica, tiempos de ejecución y apreciación subjetiva de la calidad. [30 horas]
- 5.2 Redacción de cuarto informe de avance. [15 horas]

**6. Presentación [100 horas]**

- 6.1 Redacción de informe final. [60 horas]
- 6.2 Presentación de los desarrollos en la universidad y en congresos. [20 horas]
- 6.3 Publicación de resultados que lo justifiquen. [20 horas]

## 7. Cronograma

En la siguiente figura, se detalla el cronograma propuesto para la realización del proyecto. Se fija como fecha de inicio el día primero de Julio de 2013. Se estima que para la ejecución del proyecto serán necesarias 36 semanas, por lo que la fecha esperada de finalización es el 14 de Marzo de 2014.

Act	2013						2014		
	Julio	Agosto	Septiembre	Octubre	Noviembre	Diciembre	Enero	Febrero	Marzo
1.1	■								
1.2		■							
2.1		■							
2.2			■						
2.3			■						
2.4			■						
2.5			■						
Hito 1			■	●					
3.1			■						
3.2			■						
3.3			■						
3.4			■						
3.5			■						
Hito 2			■	●		■			
3.6			■			■			
3.7			■			■			
4.1			■	■	■	■			
4.2			■	■	■	■			
4.3			■	■	■	■			
Hito 3			■	■	■	■	●	■	■
4.4			■	■	■	■	■	■	
4.5			■	■	■	■	■	■	
4.6			■	■	■	■	■	■	
5.1			■	■	■	■	■	■	
5.2			■	■	■	■	■	■	
Hito 4			■	■	■	■	■	■	■
6.1			■	■	■	■	■	■	
6.2			■	■	■	■	■	■	
6.3			■	■	■	■	■	■	

Figura 1: Cronograma por semanas propuesto para llevar a cabo el proyecto.

## 8. Puntos de control y entregables

En el desarrollo del proyecto, se fijan un conjunto de hitos donde se presentarán informes de avances. A continuación se detalla la lista de los mismos, junto a las etapas que abarca cada uno y la fecha aproximada de realización.

### Hito 1: Resumen de la investigación – Etapas 1 y 2 – Semana 9 – 5 de Septiembre de 2013

En este documento se incluirá un resumen de toda la información recopilada con respecto al tema en general, el estado del arte, y las herramientas a utilizar para desarrollar el proyecto. Se propone realizar un análisis comparativo sobre la elección de una u otra biblioteca para el desarrollo del software, exponiendo las razones correspondientes.

### Hito 2: Análisis y diseño de técnicas de teselación – Actividades 3.1 a 3.4 –

## **Semana 15 – 24 de Octubre de 2013**

En este hito, se hará entrega de un informe en el que se detalle, con ayuda de los diagramas correspondientes, las técnicas de teselación que se van a desarrollar y sobre qué modelos se van a probar. Se realizará una justificación sobre por qué se elige cierto modelo o técnica. Además se especificarán las iteraciones que se van a realizar y qué técnicas pertenecen a cada ciclo del modelo en espiral. Por último, se entregarán los documentos correspondientes al análisis y diseño de las primeras 2 iteraciones del modelo en espiral.

### **Hito 3: Desarrollo de técnicas y modelos – Actividades 3.6 a 4.2 – Semana 23 – 12 de Diciembre de 2013**

En este hito, se hará demostración de todas las técnicas de teselación adaptativas implementadas. Se hará entrega de un pseudocódigo apoyado con gráficos, donde se explique en qué consiste cada técnica. Asimismo, se hará demostración de los modelos tridimensionales implementados para la aplicación de las técnicas.

### **Hito 4: Integración y comparación de técnicas – Actividades 4.2 a 5.1 – Semana 31 – 20 de Febrero de 2014**

En este último hito, se hará demostración del software final que aplica las diversas técnicas de teselación a los modelos tridimensionales previamente seleccionados. Se hará entrega de diagramas que detallen el flujo que realiza el renderizado para cada modelo. Además, se realizará un análisis comparativo sobre las características de los métodos de teselación y se los comparará con el desempeño del mismo hardware en renderizado sin teselación. Se tomarán ciertos parámetros tanto objetivos como subjetivos y se realizará una tabla comparativa.

## **9. Riesgos y estrategias de mitigación**

En esta sección, se detallan los riesgos identificados al inicio del proyecto. Si durante la ejecución del mismo, resultara la aparición de nuevos riesgos, se los agregará en este apartado exponiendo las razones correspondientes.

### **• Pérdida de recursos de hardware: *Probabilidad: Baja – Impacto: Alto***

Para la realización del proyecto, se necesita disponer de una placa aceleradora de video de última generación, de manera que pueda soportar la tecnología de teselación en GPU. Específicamente, se necesita una placa de video compatible con DirectX 11 [5] y OpenGL 4.0 [6].

*Estrategia ante el riesgo:* Mitigar

Tanto el alumno (en su hogar) como el director del proyecto (en su oficina en el aula FICH-CIMNE de la Facultad) cuentan con el hardware necesario para la realización del proyecto. Si se presentan inconvenientes en uno de los dos recursos, el desarrollo se puede trasladar al otro. Por otra parte, si se imposibilita la utilización de ambos recursos, se deberá analizar la posibilidad de adquirir uno nuevo.

- **Pérdida del desarrollo o investigación realizados:** *Probabilidad: Baja – Impacto: Medio*

Los recursos relacionados a la investigación y el desarrollo estarán principalmente almacenados en computadoras. Si ocurriese un problema en el que se perdiera la información (rotura de disco rígido, robo, virus informático, etc.), el proyecto se vería comprometido.

*Estrategia ante el riesgo:* Mitigar

Se propone la utilización del sistema de versionado Git [7] para almacenar el software en servidores online en Internet, y poder recuperarlo ante cualquier eventualidad. Además, se propone hacer un backup semanal en un disco externo o DVD, de manera de tener un punto de control semanal para recuperarse ante el imprevisto.

- **Cambio de tecnologías con las que se realiza el desarrollo:** *Probabilidad: Media – Impacto: Bajo*

Las bibliotecas de desarrollo de software son muy volátiles y se actualizan con frecuencia. Sin embargo, en el caso de APIs y bibliotecas de interfaces gráficas (OpenGL o DirectX por ejemplo), a pesar de que se actualizan frecuentemente, mantienen una buena compatibilidad hacia atrás.

*Estrategia ante el riesgo:* Evitar

Se propone limitarse a versiones específicas de las bibliotecas y APIs utilizadas durante el proyecto, y no actualizar el software para evitar inconvenientes que escapan al desarrollo propio del proyecto.

## 10. Recursos disponibles y necesarios

Todos los recursos que el alumno necesitará para llevar a cabo el proyecto, se encuentran disponibles al inicio del mismo.

1. Recursos humanos
  - 1.1 Alumno: desarrollo del proyecto.
  - 1.2 Director del Proyecto: supervisión del proyecto.
2. Hardware
  - 2.1 PC de escritorio con procesador multicore, monitor LCD y placa aceleradora de video compatible con DirectX 11 y OpenGL 4.0.
  - 2.2 Impresora para la impresión de papers, apuntes y entregables.
3. Software de disponibilidad gratuita e inmediata
  - 3.1 Sistema Operativo: Kubuntu Linux 13.04.

3.2 Entorno de desarrollo: Vim Editor + compilador GCC. Lenguaje C++ y bibliotecas OpenGL o DirectX 11.

#### 4. Equipamiento e insumos

4.1 Ubicación con escritorio y comodidades para el desarrollo.

4.2 Conexión a Internet para consulta de bibliografía y para la realización de copias de seguridad.

4.3 Acceso a publicaciones científicas.

4.4 Insumos varios tales como hojas, lapiceras, cartuchos de tinta para impresora, pendrive para la transferencia de datos, etc.

## 11. Presupuesto

A continuación se detalla el presupuesto estimado para la realización del proyecto, discriminado por actividad. Primero se detallan unas notas con respecto a la forma de estimación y los detalles de qué significa cada entrada del presupuesto.

- El alumno trabajará 4 horas por día en el aula FICH-CIMNE, ubicada en la Facultad de Ingeniería y Ciencias Hídricas. El coste unitario por hora de la remuneración ficticia al trabajo del alumno se estimó en \$50.
- Se propone una remuneración ficticia al Director de Proyecto de \$90 por hora, estimando que se le consultará dos horas por semana sobre los pasos a seguir en el desarrollo.
- El coste de impresiones se refiere a la impresión de libros o publicaciones para el estudio y formación del alumno. Además, se refiere a la impresión de todos los informes, tanto los reportes de avance, como el informe final a presentar a los evaluadores. Por último, en este apartado se incluyen las impresiones de paneles para la presentación en congresos si correspondiera
- El transporte se calcula como si cada jornada de trabajo, que consta de 4 horas, se toman dos colectivos de ida y vuelta a \$3,50 cada pasaje. Este coste se divide por la cantidad de horas diarias, dando un coste de transporte de \$1,75 por hora de trabajo.
- El costo asociado a “Viajes a Congresos” toma como referencia el coste del viaje a Buenos Aires y hotelería, en el caso de que se desarrollen conocimientos adecuados para su presentación en congresos.
- El coste de hardware se realizó en base a la estimación del precio amortizado al día de realización del proyecto, de las dos computadoras que se van a utilizar en el desarrollo: la del alumno en su hogar estimada en \$2000, y la computadora ubicada en el aula FICH-CIMNE de \$3000.
- En el apartado “Servicios y otros insumos”, se incluye el coste de energía eléctrica e insumos de oficina tales como papeles, tinta de impresión o lapiceras, entre otros.

<b>Actividad</b>	<b>Recurso</b>	<b>Hs. hombre</b>	<b>Total (\$)</b>
1.1	Alumno Impresiones Transporte	40 — —	2000 100 70
1.2	Alumno Impresiones Transporte	20 — —	1000 50 35
2.1	Alumno Impresiones Transporte	20 — —	1000 100 35
2.2	Alumno Transporte	20 —	1000 35
2.3	Alumno Transporte	10 —	500 18
2.4	Alumno Transporte	40 —	2000 70
2.5	Alumno Transporte Impresiones	15 — —	750 26 30
3.1	Alumno Transporte	20 —	1000 35
3.2	Alumno Transporte	10 —	500 18
3.3	Alumno Transporte	40 —	2000 70
3.4	Alumno Impresiones Transporte	20 — —	1000 100 35
3.5	Alumno Transporte Impresiones	15 — —	750 26 30
3.6	Alumno Transporte	20 —	1000 35
3.7	Alumno Transporte	10 —	500 18
4.1	Alumno Transporte	100 —	5000 175
4.2	Alumno Transporte	30 —	1500 53
4.3	Alumno Transporte Impresiones	15 — —	750 26 30
4.4	Alumno Transporte	50 —	2500 88

<b>Actividad</b>	<b>Recurso</b>	<b>Hs. hombre</b>	<b>Total (\$)</b>
4.5	Alumno Transporte	20 —	1000 35
4.6	Alumno Transporte	40 —	2000 70
5.1	Alumno Impresiones Transporte	30 — —	1500 50 53
5.2	Alumno Impresiones Transporte	15 — —	750 50 26
6.1	Alumno Impresiones Transporte	60 — —	3000 500 105
6.2	Alumno Impresiones Viajes a congresos	20 — —	1000 300 2000
6.3	Alumno Impresiones Transporte	20 — —	1000 200 35
Costos indirectos	Hardware Conexión a Internet 5M, 8 meses Servicios y otros in-sumos	— — —	5000 1000 500
	Total	—	6500
Proyecto	Alumno Director de Proyecto Impresiones Transporte Viajes a congresos Costos indirectos	700 70 — — — —	35000 6300 1540 1192 2000 6500
	TOTAL		\$ 52532

## 12. Bibliografía

- [1] IBM, “*Press Release*”, Information Systems Division, Entry Systems Business. 12 de Agosto, 1981.
- [2] Radhakrishna H., “*From Moore’s Law to Intel Innovation: Prediction to Reality*”. Technical Marketing Engineer. Intel Corporation. 2005.
- [3] Blythe, D., “*Rise of the graphics processor*”. Proceedings of the IEEE Vol. 96, No. 5, pp. 761-778. Mayo de 2008.
- [4] Gee, K., “*DirectX 11 Tessellation*”. Microsoft GameFest, 2008.
- [5] DirectX: API para facilitar el manejo de recursos multimedia. 1995-2013. Disponible en <http://windows.microsoft.com/es-AR/windows7/products/features/directx-11>. [Consultado el 10 de Junio de 2013].
- [6] OpenGL: API para el renderizado de gráficos de computadora en 3D. 1992-2013. Disponible en <http://www.opengl.org>. [Consultado el 10 de Junio de 2013].
- [7] Git: sistema distribuído de control de versión. 2005-2013. Disponible en <http://git-scm.com>. [Consultado el 10 de Junio de 2013]
- [8] CPlusPlus.com Reference. 1983-2013. Disponible en <http://www.cplusplus.com/reference>. [Consultado el 10 de Junio de 2013].
- [9] De Berg, M., Cheong, O., Van Kreveld, M., & Overmars, M., “*Computational Geometry: Algorithms and Applications*”, Springer Editorial, Third Edition, 2008.
- [10] Salomon, D., “*Curves and Surfaces for Computer Graphics*”, Springer Editorial, 2006.
- [11] Taylor & Francis Group, “*OpenGL Insights*”, CRC Press, 2012.
- [12] Rost, R., “*OpenGL Shading Language*”, Second Edition, 2006.
- [13] McKesson, J., “*Learning Modern 3D Graphics Programming*”, 2012.