



UNIVERSIDAD NACIONAL DEL LITORAL
FACULTAD DE INGENIERÍA Y CIENCIAS HÍDRICAS

Desarrollo de algoritmos de teselación adaptativa en GPU

Tercer informe de avance

Alumno: Fernando Nellmeldin
Director: Dr. Néstor Calvo

Santa Fe, Abril de 2014

Índice

| | |
|---|-----------|
| 1. Introducción | 2 |
| 2. Diseño general del software | 2 |
| 2.1. Flujo de trabajo | 3 |
| 2.1.1. Pipeline en CPU | 3 |
| 2.1.2. Pipeline en GPU | 5 |
| 3. Métricas de teselación y aplicaciones | 6 |
| 3.1. Distancia a la primitiva | 7 |
| 3.1.1. Superficies de Bézier | 8 |
| 3.1.2. Terreno | 9 |
| 3.1.3. Visualización de campo vectorial | 10 |
| 3.2. Longitud de arista | 10 |
| 3.2.1. Terreno | 12 |
| 3.3. Silueta | 14 |
| 3.3.1. Modelos tridimensionales | 15 |
| 3.3.2. Modelos y Phong Tessellation | 15 |
| 3.4. Curvatura | 16 |
| 3.4.1. Curva de Bézier de tercer grado | 17 |
| 3.5. Métricas descartadas | 19 |
| 4. Conclusión | 20 |

1. Introducción

En el anteproyecto y en los anteriores informes de avance, ya se había mencionado la intención del alumno en desarrollar e implementar técnicas de teselación adaptativa a utilizar en modelos tridimensionales. El desarrollo de estas técnicas se lleva a cabo diseñando, para un problema en particular, una métrica que defina el nivel de refinamiento que debe tener cada polígono original (por lo general un triángulo) del modelo a renderizar. Este nivel de refinamiento se dice que es adaptativo porque depende de la situación específica de la escena durante el renderizado, por lo que se generan distintos niveles según el tiempo y la situación general del observador y los polígonos.

De manera que una métrica de teselación sea efectiva, ésta debe calcularse de acuerdo a lo que se podría denominar la situación local y global del polígono. La situación local hace referencia a la ubicación del polígono con respecto al modelo en el que se encuentra. Una métrica de este tipo, por ejemplo, le da mayor importancia a polígonos que se encuentren en la silueta del modelo. Por otro lado, la situación global del polígono hace referencia, por ejemplo, a la distancia entre éste y la ubicación del observador. A las métricas locales también se las denomina *intrínsecas* y a las globales: *extrínsecas*.

En este tercer informe de avance, luego de haber elegido, diseñado, desarrollado e implementado algoritmos que se ejecutan en la placa aceleradora de video (GPU), se expondrán en forma resumida cada uno de los algoritmos desarrollados, el flujo de información entre el procesador central (CPU) y la GPU, y los resultados preliminares que se alcanzaron.

2. Diseño general del software

Como ya se ha expuesto en el segundo informe de avance, el diseño del software se realizó utilizando programación orientada a objetos. En C++ esto se traduce a un diseño de clases interrelacionadas. A cada clase se le asigna una responsabilidad general, y se interactúa con ésta a través de un conjunto de métodos que exponen sus características internas. El diseño que se había presentado en el segundo informe de avance ha sido modificado con la agregación de una nueva clase. Dicha clase, llamada `TextRenderer` tiene como responsabilidad la de realizar una abstracción para facilitar la presentación de texto en pantalla.

Con anterioridad, no se renderizaba texto en la ventana de la aplicación, sino que éste se mostraba en una consola de sistema. Durante el desarrollo, se vio la necesidad de mostrar texto en la misma ventana en que se renderiza el modelo para facilitar el *feedback* al usuario y mejorar la interfaz visual del software. Para cumplir con este requisito, se incorporó el motor de rasterización de fuentes *FreeType* [1]. La puesta a punto de dicho motor no es simple ya que para poder renderizar en pantalla, se debe inicializar una textura con el texto deseado, y luego enviar dicha textura a un programa de shader para que éste la escriba en la ventana del programa. Para facilitar esta tarea, se decidió agregar la clase `TextRenderer` que con una única llamada (que contiene el texto y la posición en ventana) se realiza el proceso de rasterización en pantalla. La clase `Example` y sus heredadas se relacionan con `TextRenderer` de manera que en la llamada de dibujado en `Example`, se convoca a la función de renderizado de texto de `TextRenderer`. En la Fig. 1 se muestra un *screenshot* del texto renderizado en pantalla.

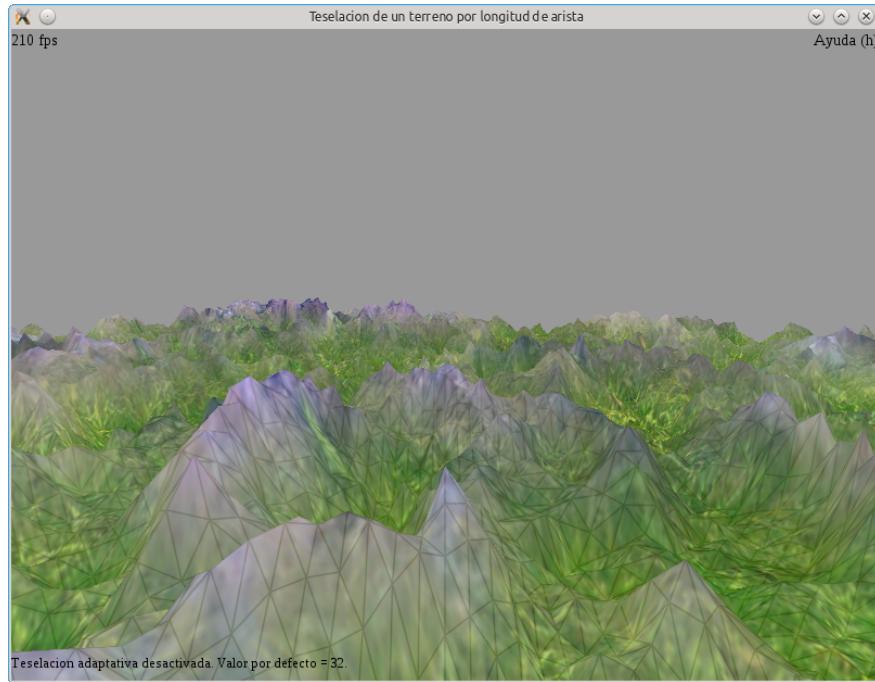


Figura 1: Ejemplo de renderizado de texto en pantalla.

En la Fig. 2 se presenta el diseño general de clases actualizado. Además, se presenta la totalidad de métodos que contienen dichas clases. Para simplificar el contenido del diagrama, no se incluyen los parámetros y el valor de retorno de las funciones. Además y como consecuencia de lo anterior, no se muestran las funciones sobrecargadas. Asimismo, en dicho diagrama no se incluyen los constructores ni destructores de las clases. En la mayoría de las clases, en donde el constructor no es vacío, los constructores inicializan los estados internos; por otro lado, los destructores liberan toda la memoria utilizada por dicha clase. Las clases que heredan de `Example` añaden métodos específicos que dependen de cada ejemplo en particular.

2.1. Flujo de trabajo

2.1.1. Pipeline en CPU

El flujo de trabajo que se realiza en cada modelo para renderizarlo en una ventana es similar entre todos los ejemplos. El proceso comienza al crear un objeto de una clase que en esta exposición se llamará `Example1`. Esta clase hereda de `Example` y define un conjunto de propiedades y métodos que dependen de cada ejemplo en particular. El constructor de `Example1` llama a la función heredada `Example1::InitVariables(...)` que se encarga de inicializar los valores de las propiedades utilizadas en todos los ejemplos (cámara, colores, matrices de transformación, ventana, etc.). Además, es responsabilidad de esta llamada la definición de los *callback* para OpenGL, que se utilizarán en cada ciclo del bucle general.

Luego de la invocación al constructor, se debe realizar una llamada a la función `Example1::Initialize()` que verifica que se posea la versión correcta de OpenGL y si se supera esta verificación, se construye el contexto y la ventana de OpenGL. Además, esta función llama a `Example1::CreateScene()` cuyo objetivo es inicializar todos los datos que

DIAGRAMA DE CLASES COMPLETO

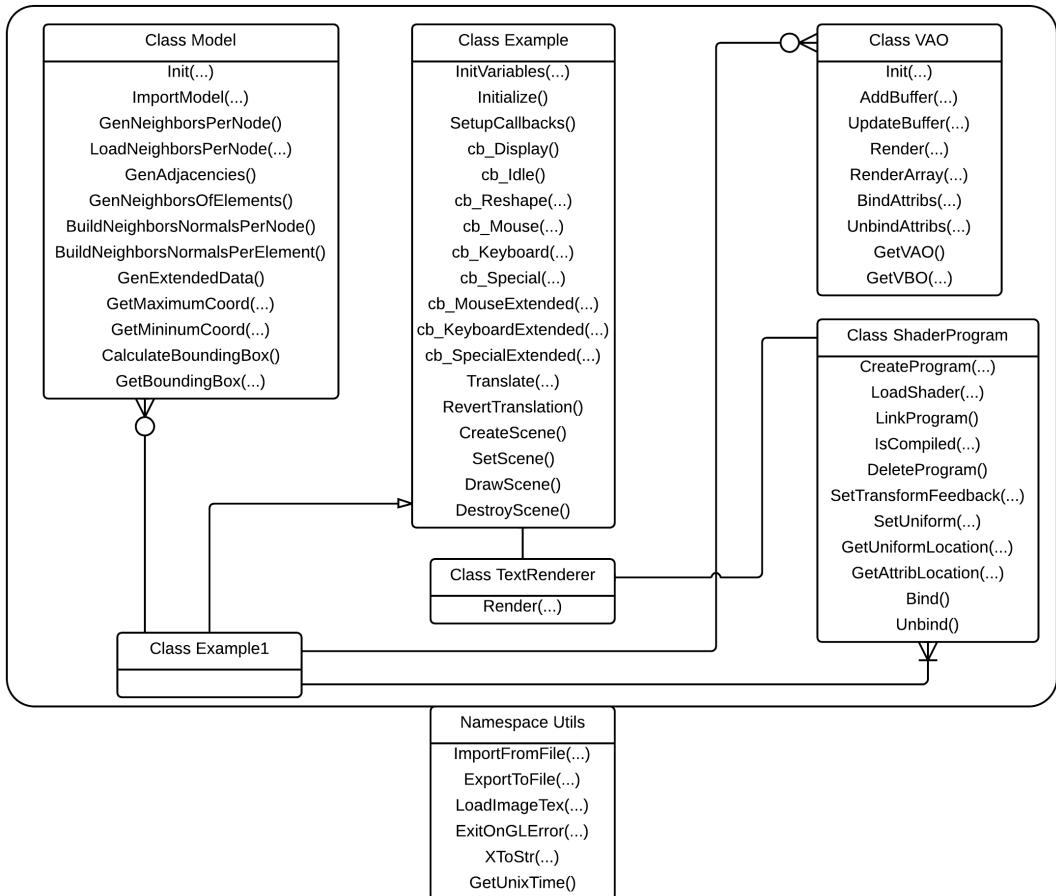


Figura 2: Diagrama de clases completo del proyecto.

se van a utilizar en el dibujado particular de este ejemplo. Entre estos datos se encuentra la información sobre los vértices y elementos (abstraídos en uno o varios objetos de la clase **VAO**). Asimismo, en esta misma función se construyen todos los programas de shader (instancias de la clase **ShaderProgram**). La última responsabilidad de **Example1::Initialize()** es llamar al constructor de **TextRenderer**.

La siguiente llamada a función que se realiza es a **Example1::SetScene()** que define el valor inicial de todas las propiedades específicas del ejemplo en particular. Notar que en **Example1::InitVariables(...)**, que es un método heredado, se inicializan los datos de propiedades que aparecen en todos los ejemplos y no en uno en particular.

Por último, se llama a la función general de GLUT **glutMainLoop()** que inicia el bucle general del software. Los callback que se definieron en el primer paso son invocados en cada bucle. El callback de dibujado a su vez llama a **Example1::DrawScene()** que se encarga de traer a contexto los programas de shader, transmitir las variables de la CPU a la GPU, enviar los vértices y elementos a la GPU, rasterizarlos y realizar la limpieza posterior. Dependiendo del problema, se utilizan uno o más programas de shader.

En la Fig. 3 se exhibe un diagrama de secuencia que muestra las interacciones entre los objetos. Por simplicidad, en este diagrama no se incluyó la clase **Model** porque su función es la de cargar un modelo al inicio del software y luego exponer los datos para la construcción

de un objeto de la clase VAO.

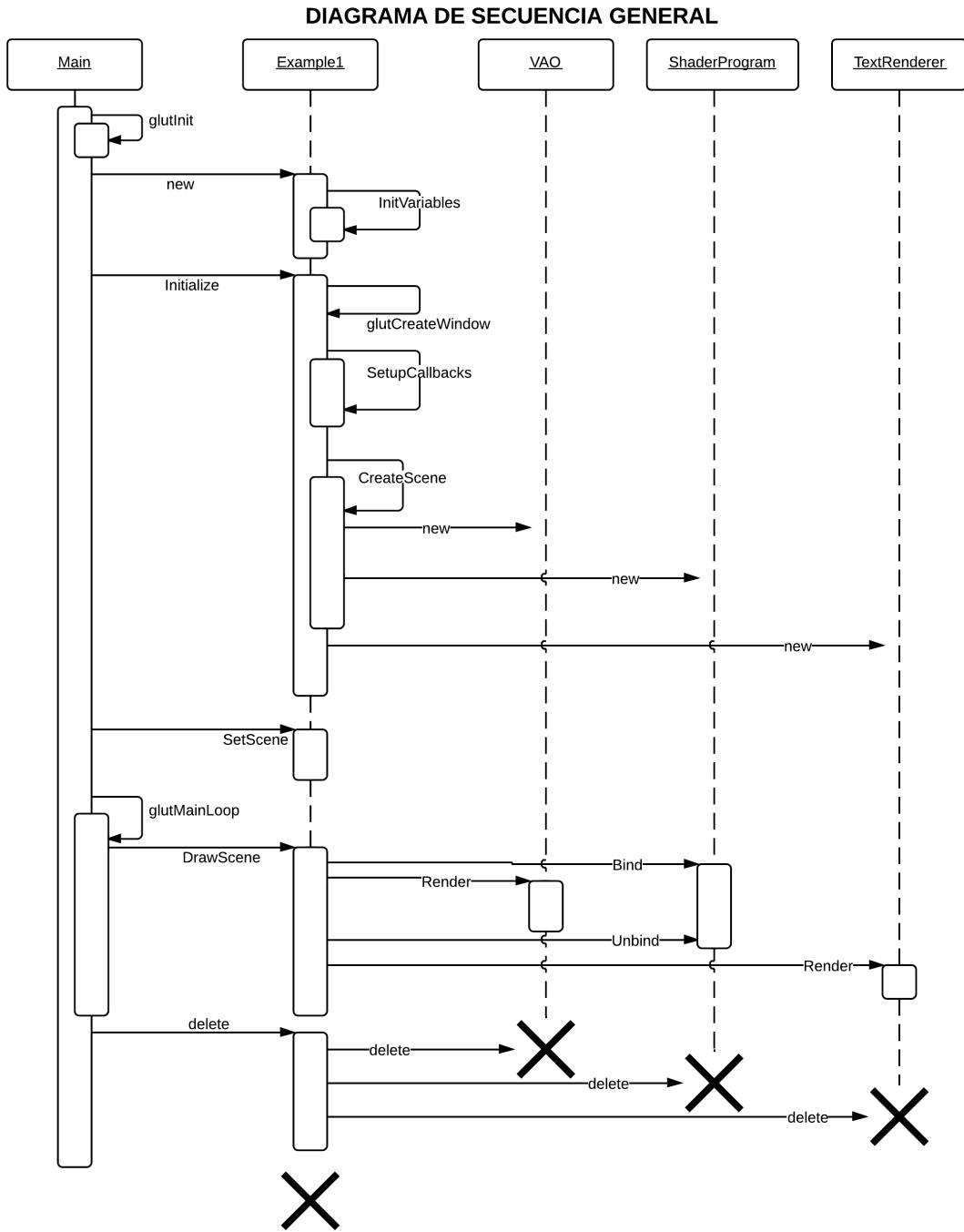


Figura 3: Diagrama de secuencia general simplificado.

2.1.2. Pipeline en GPU

En el *pipeline gráfico*, los vértices o elementos pasan por varias etapas. Un subconjunto de éstas son las que se denominan etapas programables, en las que se debe escribir código para poder completar el proceso de renderizado. Las etapas y sus funciones resumidas son:

1. *Vertex Shader (VS)*: Su entrada es un único vértice del modelo y por lo general su función es la de pasar este vértice a la siguiente etapa, igual y sin cambios a como lo

- recibe.
2. *Tessellation Control Shader (TCS)*: Cada primitiva entra a esta etapa con todos sus vértices (3 para un triángulo por ejemplo). La principal responsabilidad de esta etapa es definir todos los niveles de teselación para cada arista de la primitiva resultado.
 3. *Tessellation Evaluation Shader (TES)*: A esta etapa se le ingresa una primitiva junto a sus niveles de teselación y un grupo de coordenadas internas a la misma. Estas coordenadas, que varían según primitiva (3 para triángulos, 2 para cuadriláteros, 1 para isolíneas), se utilizan para interpolar la posición de los nuevos vértices obtenidos en la teselación.
 4. *Geometry Shader (GS)*: Su responsabilidad es la de crear las nuevas primitivas a partir de los vértices que se generaron en la etapa anterior. Estas primitivas son simples ya que sólo pueden ser triángulos, cuadriláteros o isolíneas.
 5. *Fragment Shader (FS)*: Su función es la de finalmente rasterizar la primitiva, que llega subdividida en fragmentos. En esta etapa también se aplican, si se lo requiere, los algoritmos de iluminación y texturizado.

Las métricas de teselación se implementan en GPU, en la etapa de *Tessellation Control*. En la etapa de *Tessellation Evaluation*, según los niveles de teselación definidos en TCS, se cuenta con gran cantidad de vértices nuevos generados en el interior de una primitiva. Por lo tanto, el énfasis del presente proyecto, además de estudiar y comprender la tecnología, es el de experimentar con la etapa de Control y definir una métrica para cada problema que genere la densidad de muestreo deseada. En la Fig. 4 se muestra un diagrama que ilustra el flujo de la información en la GPU y los datos que se obtienen en cada etapa (y que también sirven de entrada a la siguiente). Este proceso se repite iterativamente en cada frame (cuadro de renderizado). Mientras más frames se ejecuten por segundo, el resultado es más fluido. Por lo tanto, la principal medida objetiva para evaluar el desempeño en el uso de teselación es la tasa de frames que se ejecutan por segundo, denominada FPS por su sigla en inglés: *frames per second*.

3. Métricas de teselación y aplicaciones

De las métricas que se habían considerado en un principio, se decidió implementar la mayoría de éstas. Esta sección se organiza de la siguiente manera: en cada apartado se expone una métrica y luego se presentan el o los modelos en los que se probó cada métrica.

Los niveles de teselación pueden variar desde 0 al máximo. Un valor igual a cero equivale a descartar la primitiva, mientras que un nivel igual a uno es lo mismo que desactivar teselación (no hay cambios en las primitivas). El máximo, en el hardware que se utilizó, es de 64. Pero definir un nivel de teselación igual a 64 puede ocasionar una alta sobrecarga en el renderizado, ya que por cada polígono inicial se pueden llegar a generar en el orden de 4000 polígonos internos (64 al cuadrado en el caso de cuadriláteros). Por lo tanto, el máximo nivel de teselación que un polígono puede tomar, según cada aplicación en particular, se definió de manera de lograr un renderizado fluido.

Sin embargo, en algunos casos se usó un nivel de teselación diádico. Esto implica que sólo se usaron potencias de 2, de manera de evitar saltos bruscos entre dos polígonos adyacentes que exponían distintos niveles de teselación en la interfaz. De esta manera, se reducen las conexiones “en T”. Los niveles de teselación que no coinciden con una potencia de dos, se

PROCESO DE RENDERIZADO POR FRAME

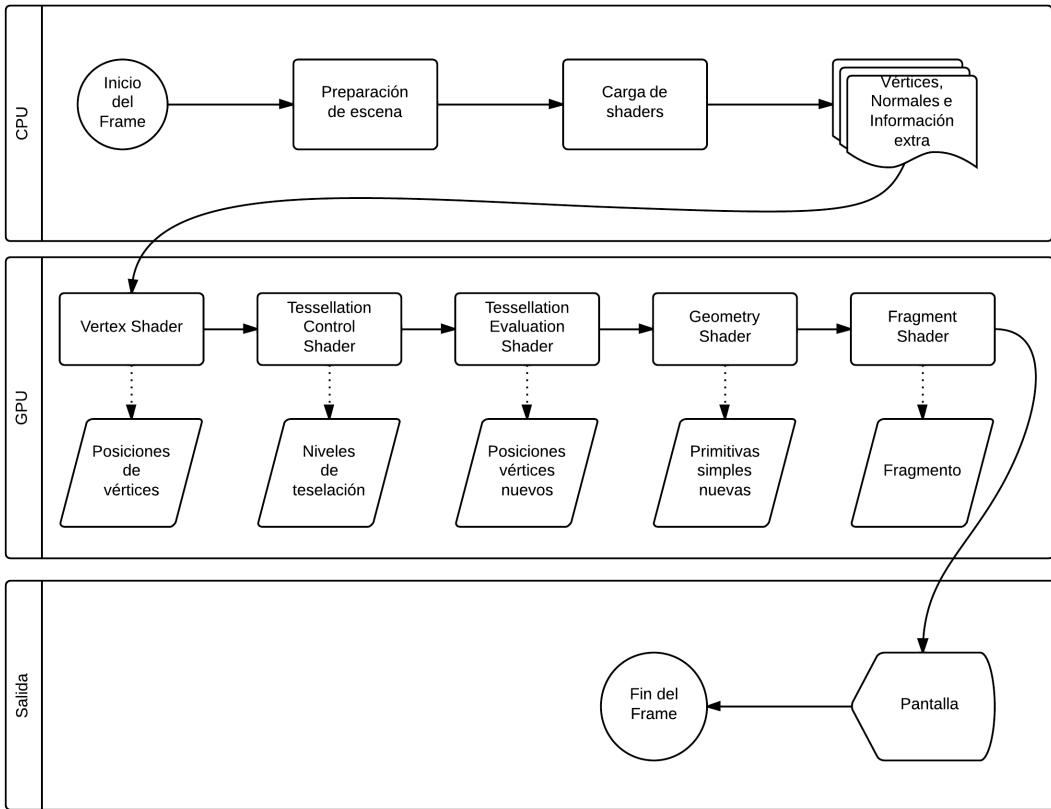


Figura 4: Diagrama de renderizado en cada frame.

redefinen a la potencia mayor más cercana.

Al diseñar una métrica, se buscó que posea ciertas propiedades:

- **Simple:** el cálculo del nivel de teselación se debe hacer por cada primitiva, por lo que se buscó que el cálculo sea lo suficientemente simple como para que no se convierta en un ‘cuello de botella’ en el proceso de renderizado.
- **Poca información:** debido a que la transferencia de datos de la CPU a GPU es lenta, se buscó que las métricas de teselación, que se deben ejecutar exclusivamente en GPU, no tengan dependencia de muchos datos. Esto es, si para cada primitiva se tiene que pasar gran cantidad de datos, el rendimiento general se verá afectado.
- **Adaptativa:** la métrica debe depender del estado específico de la escena en cada instante de renderización. Si la métrica se calcula una única vez y se aplica por el resto del renderizado, no es adaptativa.

3.1. Distancia a la primitiva

Esta métrica extrínseca es la más simple que se encontró. El nivel de teselación para un polígono se calcula en función de la distancia de éste, en coordenadas de mundo, al punto de vista. Esta métrica busca refinar elementos que estén cerca al punto de vista ya que, en la mayoría de los casos, se desea que tengan más detalles que aquellos que se encuentran

alejados. En una escena compleja en donde hay muchos polígonos, no es de utilidad renderizar un polígono refinado si éste se encuentra lejos del punto de vista, ya que la transformación proyectiva que lo lleva a pantalla lo convertirá en una primitiva que ocupa un reducido espacio en pantalla. El efecto contrario ocurre si el polígono está cerca del observador.

Esta métrica falla en el caso en que, aunque el polígono esté cerca del punto de vista, no sea visible. Este caso puede ocurrir cuando el polígono no está en el campo de visión del observador o está ocluido por otro polígono. Estos casos se consideran en el Fragment Shader en donde se descarta el fragmento si no es visible.

Para calcular la distancia, se experimentaron tres alternativas. La primera determina la distancia a un polígono como la distancia de su centroide al observador. Esto luego se modificó y se pasó a definir la distancia a un polígono como la distancia entre el observador y el vértice más cercano del polígono. La tercer 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 de la arista. Esta métrica tiene en especial consideración la característica de teselación por arista del hardware y permite asignar un nivel distinto para cada lado. Esto último no ocurre con las otras alternativas ya que sólo sirven para especificar un nivel de teselación general para todas las aristas de la primitiva.

Una vez obtenida la distancia, 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 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:

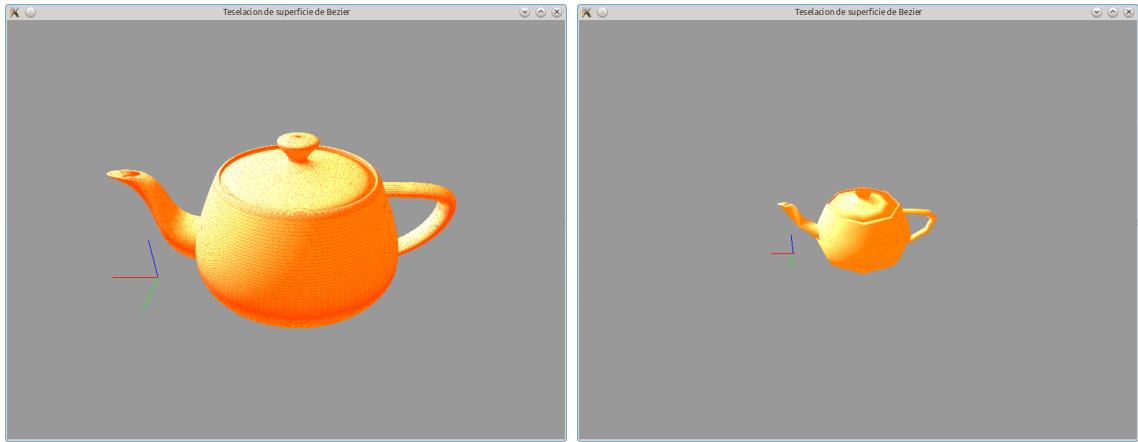
$$t_i = \begin{cases} -\frac{(t_{tessmax}-1)}{d_{max}}d + t_{tessmax}, & 0 \leq d < d_{max} \\ 1, & d \geq d_{max} \end{cases} \quad (1)$$

Según como se expuso antes, este nivel de teselación luego se puede modificar a una potencia de 2 si se está utilizando teselación diádica.

3.1.1. Superficies de Bézier

El modelo que se utilizó en esta aplicación está formado por superficies de Bézier de tercer grado. Cada superficie está definida por 16 puntos de control, los cuales se utilizan para calcular la posición final de cualquier punto sobre la misma. En este caso, como la noción de una ‘arista’ no es clara, no es conveniente utilizar teselación definida por arista. Por lo tanto, se uso la métrica de distancia al punto más cercano. Esta métrica se prefiere con respecto a la distancia al centroide porque el centroide de los puntos de control no necesariamente está en la superficie. En cambio, una superficie de Bézier siempre pasa a través de los 4 puntos de control extremos. Por lo tanto, se toma la mínima distancia del observador a los 4 puntos de control extremos.

Utilizar teselación aquí es de suma importancia para lograr una representación fiel de cada superficie de Bézier. La precisión de la representación de una superficie de Bézier depende de



(a) Superficies cerca de la cámara. (b) Superficies alejadas de la cámara.

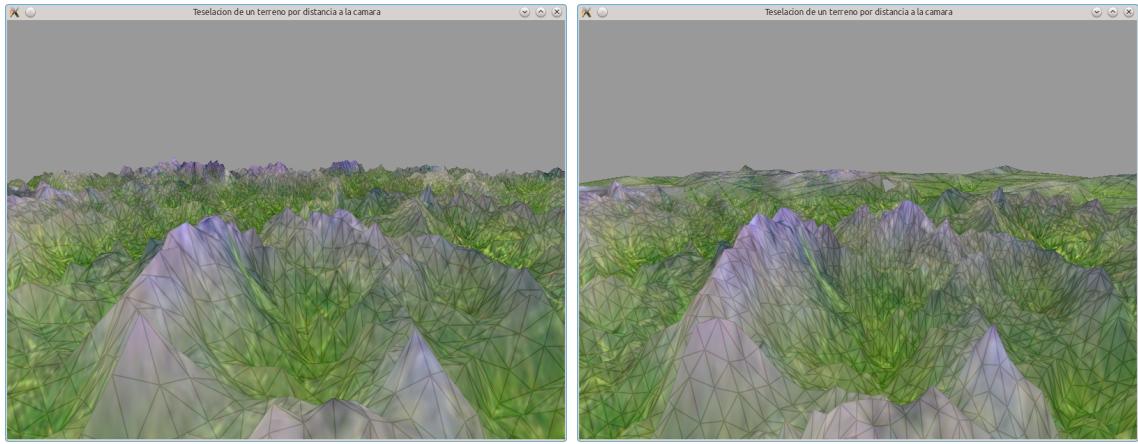
Figura 5: Teselación por distancia en superficies de Bézier.

la cantidad de puntos en la que ésta sea evaluada. Esta cantidad de puntos depende del nivel de teselación que se le asigne a la primitiva. Por lo tanto, mayor niveles de teselación darán como resultado una superficie más continua y precisa. En otro caso, al tener un nivel de teselación bajo, la superficie resultante puede llegar a no capturar toda la información dada por los puntos de control y por lo tanto generar una mala aproximación de la superficie. En la Fig. 5 se presenta el resultado de aplicar esta métrica a un modelo formado por superficies de Bézier de tercer grado. Cuando el observador se aleja, la representación de la superficie se hace cada vez más imprecisa.

3.1.2. Terreno

Se posee una malla regular en el plano de cuadriláteros, y una textura que representa un campo de desplazamientos. Esta textura se interpreta como una altura en cada píxel. Para renderizar el terreno, se mapea la posición de cada vértice de un cuadrilátero a una posición en la textura. El valor del píxel en esta posición identifica la coordenada vertical del vértice. Al utilizar teselación, se muestran más vértices dentro de cada elemento, lo que implica más muestreo de alturas, y por lo tanto un terreno con transiciones más suaves. La ventaja de renderizar un terreno de esta manera es que no se necesita una malla regular inicial muy grande. Partiendo de una malla poco densa, si se utiliza teselación, se pueden generar más vértices en dónde se necesite y allí muestrear las alturas.

En este caso, se utiliza la distancia del punto de vista al punto medio de cada arista. De esta manera, se define un nivel de teselación distinto para cada lado del cuadrilátero. Más aún, debido a que una arista es igual entre dos elementos adyacentes, se asegura que se calcule el mismo nivel de teselación en ambos elementos de la arista compartida, y así mantener la continuidad geométrica G0. En la Fig. 6 se expone el resultado de aplicar teselación por distancia a un terreno. Se puede ver que las primitivas más alejadas no tienen la misma densidad que las primitivas cercanas al observador.



(a) Teselación adaptativa desactivada. (b) Teselación adaptativa activada.

Figura 6: Teselación de un terreno por distancia al observador.

3.1.3. Visualización de campo vectorial

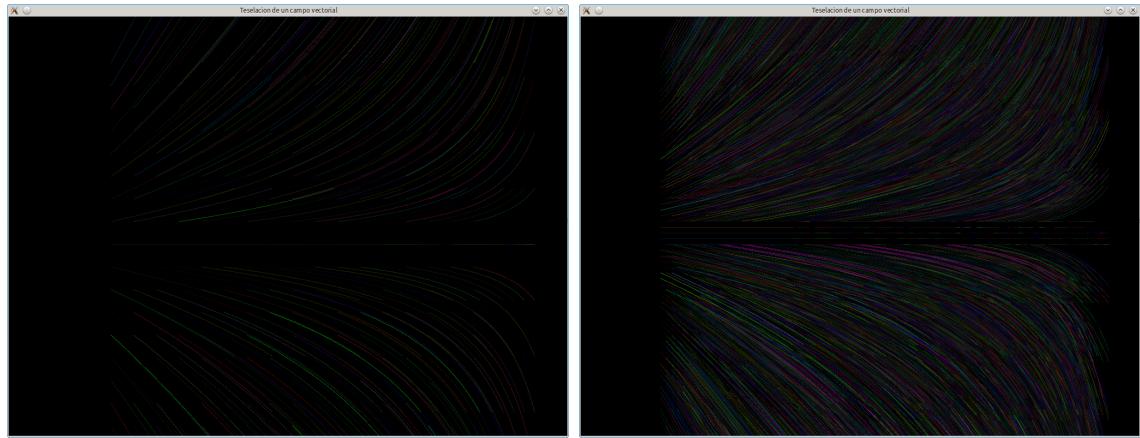
Se posee un conjunto de puntos afectados por un campo vectorial. Se desea generar una visualización del movimiento que sufren estos puntos al ser afectados por el campo vectorial. Utilizando ideas de la técnica de LIC (convolución de integral de línea, *line integral convolution* [2]), se aplica teselación para generar una mejor visualización.

Se parte de una malla regular de vértices y una imagen de ruido. La primitiva en este caso está conformada por un único vértice, aunque tiene información sobre el campo vectorial suyo y de sus vecinos. Cada vértice se relaciona con un color de la imagen de ruido. A continuación se simula el movimiento de los vértices al ser afectados por el campo vectorial. En cada paso, se mapea la posición del vértice a una textura de salida que contiene el camino que realizó cada vértice. Al finalizar lo que se obtiene es un conjunto de líneas de corriente para todos los vértices movidos por el campo.

Cuando se utiliza teselación, lo que se desea es generar más posiciones en donde calcular el movimiento de los vértices. Mediante interacción con el software, el usuario “mueve” la cámara al sector donde desea tener más detalle. Para decidir el nivel de teselación, se mide la distancia del observador al vértice y se mapea al rango $[0, t_{tessmax}]$, como ya fue explicado. Los vértices nuevos se generan en la vecindad del vértice original, formando un cuadrilátero con los nuevos, y donde el vértice original es una de las ‘puntas’ del cuadrilátero. En la Fig. 7 se muestra el resultado de aplicar teselación a un campo vectorial. Además se agrega un zoom en una zona del mismo y se puede apreciar la variación de densidad en la zona de interés.

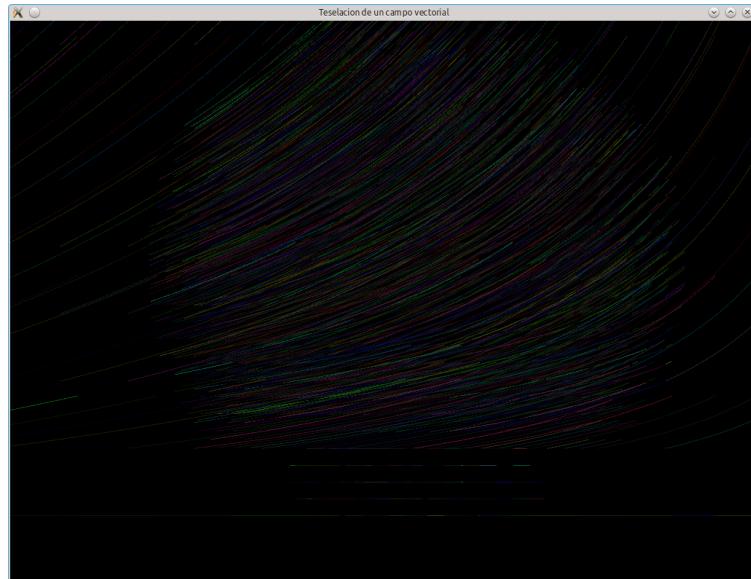
3.2. Longitud de arista

Un triángulo está formado por 3 aristas. Al rasterizar el triángulo, estas aristas se convierten a píxeles en la ventana de rasterizado. Mientras mayor longitud tenga la arista, o más cercana se encuentre con respecto al punto de vista, mayor cantidad de píxeles serán necesarios para representarla en el dispositivo de salida. La métrica intrínseca que se expone en esta sección hace uso de esta característica, de forma de dar mayor nivel de teselación a



(a) Campo vectorial general.

(b) Teselación máxima en todo el campo.



(c) Acercamiento en una zona.

Figura 7: Teselación en campo vectorial.

una arista que al rasterizarla ocupe mayor cantidad de píxeles. Lo que se busca es definir una cierta cantidad máxima de píxeles e_{max} que una arista puede ocupar en pantalla. Si excede este número, la arista se divide de manera de poder garantizar que las nuevas aristas generadas en su lugar tengan igual o menor longitud que e_{max} .

El cálculo de la longitud proyectada de un segmento (en este caso que representa una arista) se realiza como se explica a continuación. Dada una arista $\overline{e_1 e_2}$ y dada 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. El segundo punto, p_2 , surge de sumar a la coordenada y de p_1 , la distancia d . Notar que y , en coordenadas de mundo, representa el eje vertical positivo del mismo. Por lo tanto, p_2 es un desplazamiento del punto p_1 igual a la distancia d . Este desplazamiento sólo se hace en el eje y , dejando los otros dos ejes intactos.

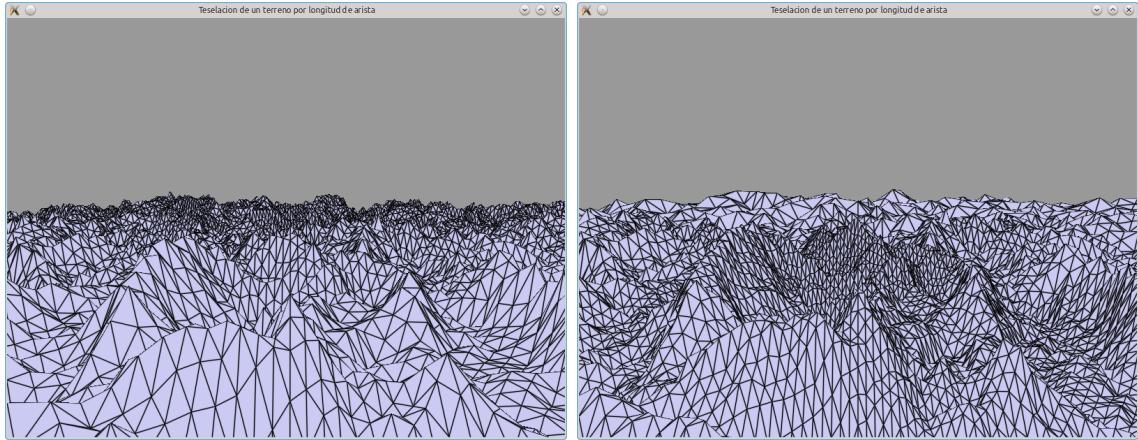
A continuación se utiliza la matriz de proyección, la cual depende de la posición de la vista, del modelo, y del tamaño de la ventana, para transformar los puntos p_1 y p_2 a coordenadas de dispositivos normalizadas (que están en el rango $[-1, 1]$ en cada uno de los ejes). Estas nuevas coordenadas se las representa con p_{p1} y p_{p2} . El paso siguiente consiste en calcular la distancia l en píxeles entre los puntos p_{p1} y p_{p2} . Esta distancia se obtiene al calcular el módulo del vector $\overline{p_{p1}p_{p2}}$ y multiplicándolo por la mitad del tamaño de la ventana. Este producto se hace tanto en el eje x (ancho) como en el eje y (alto). El eje z se desprecia porque ambos puntos están en la misma posición z . Si ambos puntos están en pantalla, su máxima distancia será el ancho (o alto) de las coordenadas de dispositivos normalizadas, es decir igual a 2. Por lo tanto, se multiplica por la mitad del tamaño de la ventana para convertir este valor que está en el rango $[0, 2]$, a un valor en el rango $[0, ancho]$ o $[0, alto]$.

Por último, para definir el nivel de teselación t_i para esta arista, se divide l 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 $\overline{p_1 p_2}$ de longitud 0,5 (un cuarto de ventana en coordenadas de dispositivo normalizadas), la distancia l en el eje x es igual a 160 ya que $d = 0,5 * 640 / 2 = 160$. Si se define e_{max} igual a 10, el valor de teselación es entonces $t_i = 160 / 10 = 16$. Por lo tanto, esta arista se va a dividir en 16 aristas nuevas, cada una de longitud de 10 píxeles.

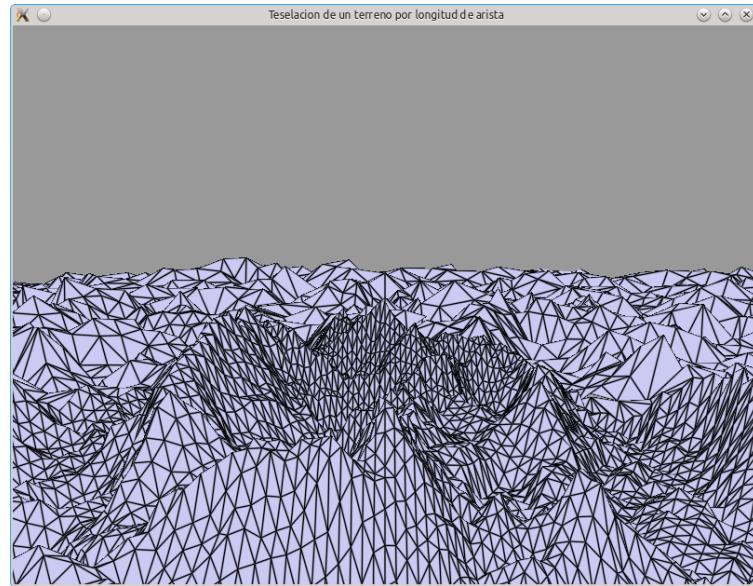
Según el procedimiento detallado, la longitud proyectada de una arista no depende de su posición relativa con respecto al observador, si no que sólo depende de su longitud absoluta. Por lo tanto, se obtiene la misma longitud para una arista que esté perpendicular al plano xy (pantalla) como si está perpendicular al plano xz (“piso”) o a cualquier otro plano. Esta característica puede ser deseable o no, ya que en ocasiones puede dar lugar a un nivel de teselación mayor del que se desee si, por ejemplo, la arista se “aleja” del observador.

3.2.1. Terreno

La métrica que se acaba de explicar es de especial utilidad para el caso de renderizado de terrenos. Según como se expresó, la longitud proyectada que se define para una arista no depende de su posición relativa con respecto a la cámara, si no de su longitud absoluta. Esto es importante al aplicar la métrica a terrenos cuya altura se calcula con respecto a un mapa de desplazamiento, ya que se le asigna un nivel de teselación sin importar su orientación. De este modo, se puede generar el terreno en forma correcta y que no presente “saltos”



(a) Teselación adaptativa desactivada. (b) Teselación adaptativa activada.



(c) Teselación adaptativa activada y corrección diádica desactivada.

Figura 8: Teselación de un terreno por longitud de aristas.

entre aristas de una misma primitiva que estén alineadas con la pantalla y otras aristas perpendiculares a las primeras.

De acuerdo a la métrica, se calcula la longitud proyectada de cada arista de un cuadrilátero y se le asigna un nivel de teselación según este valor. Así, la longitud proyectada en aristas adyacentes se garantiza ser la misma y por lo tanto el terreno se genera de forma continua. Experimentalmente se notó que al aplicar esta métrica y mover la posición del observador, la teselación de los polígonos variaba más de lo deseable a causa de errores de redondeo en todas las divisiones. Por lo tanto se optó por aplicar la corrección diádica previamente explicada. En la Fig. 8 se muestra el uso de esta métrica en un terreno. Se puede apreciar que las aristas son similares en longitud. Las aristas alejadas representan un tamaño de terreno más grande mientras que las cercanas una fracción pequeña del mismo. Por último, se muestra una imagen del mismo terreno sin la corrección diádica activada. Se pueden ver muchos cambios, ya que la corrección diádica tiende a dar niveles de teselación mayores. Pero la diferencia se percibe con más calidad cuando se ve el software en ejecución.

3.3. Silueta

Un modelo de personaje, utilizado por ejemplo en videojuegos, está formado por gran cantidad de polígonos. A menudo se necesita renderizar simultáneamente varios personajes en escena, por lo que se debe tener especial cuidado en el uso de teselación. Al utilizar esta tecnología, se incrementa la cantidad de polígonos que se deben renderizar y por lo tanto el rendimiento generalmente decrece. La métrica que se expone a continuación busca brindarle más importancia a polígonos que estén en la silueta del modelo, es decir a aquellos triángulos que están en su “borde”. Matemáticamente un triángulo se considera silueta si su normal tiene la dirección opuesta a alguna de las normales de sus elementos adyacentes. Tener una buena representación de la silueta es de importancia, por ejemplo, para cálculos de iluminación. En este caso, tener más polígonos a lo largo de un borde permite representar más y mejores cambios de luces y sombras en el modelo.

Para poder utilizar esta métrica, se necesita información de adyacencias, ya que para un elemento en particular se debe consultar las normales de sus vecinos. Sin embargo, el pipeline gráfico trabaja en múltiples hilos de ejecución y en cada hilo sólo se tiene información de una primitiva. Esta limitación se puede superar si se extiende el significado de “primitiva”. En teselación, una primitiva es un conjunto de puntos llamado *patch*. Este conjunto depende de la implementación y el hardware, pero al menos es de 32 vértices por patch como máximo. En el caso de trabajar con triángulos, el patch estará conformado por 3 vértices. Para cuadriláteros el valor será 4. Para las superficies de Bézier de tercer grado se utilizan 16 vértices (los puntos de control de la superficie). En definitiva, un patch es un conjunto de vértices y el programador es el que elige cómo se interpreta este conjunto.

De manera de tener información de adyacencias, se construyen primitivas “extendidas”. Estas primitivas están constituidas por 6 vértices: los 3 vértices del triángulo original, y los 3 vértices vecinos. Además, cada vértice tiene una normal. Con estos 6 vértices se puede obtener la normal del triángulo “central” y la normal de sus elementos vecinos (ya que comparten una arista con el triángulo central). Con la información de las normales y la posición del observador, se puede calcular un valor que ayude a decidir si un triángulo es silueta o no.

Se tiene un patch k , formado por 6 vértices:

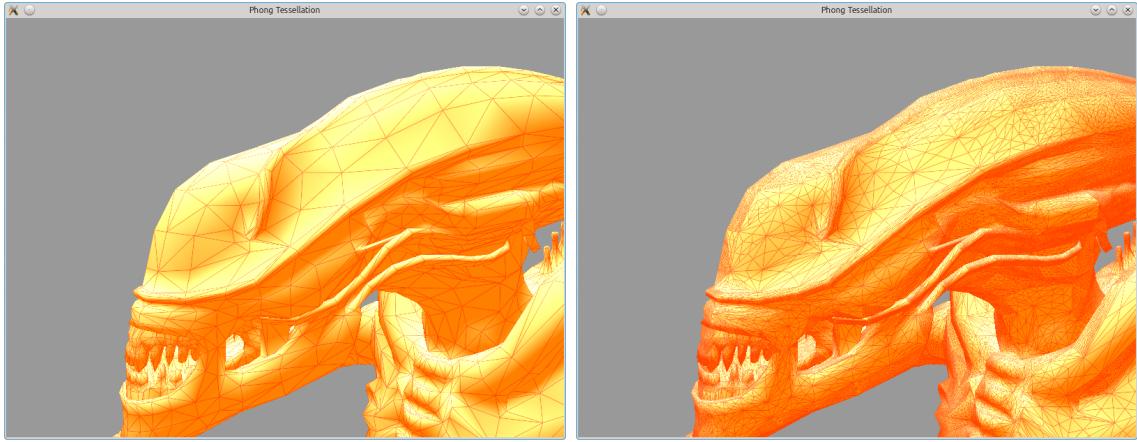
- v_0, v_1 y v_2 : los vértices del triángulo central.
- v_3, v_4 y v_5 : los vértices no compartidos de los triángulos vecinos.

Además, cada vértice tiene su normal n_i . Se obtiene la normal del triángulo central:

$$n_c = \frac{n_0 + n_1 + n_2}{\|n_0 + n_1 + n_2\|} \quad (2)$$

También se calculan las normales de los triángulos vecinos, que se identificarán con n_{vi} para el vecino por la arista i . Obtenidos estos valores, se procede al cálculo de s_i , el valor de silueta para la arista i del patch. Primero se computa el punto medio e_i de la arista i . En segundo lugar se obtiene el vector c normalizado que apunta desde la posición del observador a e_i . El siguiente paso consiste en calcular:

$$u_i = (n_c \cdot e_i)(n_{vi} \cdot e_i) \quad (3)$$



(a) Teselación adaptativa desactivada. (b) Teselación adaptativa activada.

Figura 9: Teselación de un modelo por silueta.

Este u_i se encuentra en el rango $[-1, 1]$. El número u_i será igual a -1 cuando las normales adyacentes sean totalmente opuestas. Si son iguales, este número será 1 . Para finalmente definir el valor s_i , se hace un mapeo de u_i en el rango $[0, 1]$. Este mapeo se hace ayudado de una constante β que identifica el límite para el que un triángulo sea considerado silueta. Para los casos en que se probó, se elige $\beta = 0,25$. De esta manera, s_i se define según u_i :

$$s_i = \begin{cases} 1, & u_i < 0 \\ 1 - \frac{u_i}{\beta}, & 0 \leq u_i \leq \beta \\ 0, & u_i > \beta \end{cases} \quad (4)$$

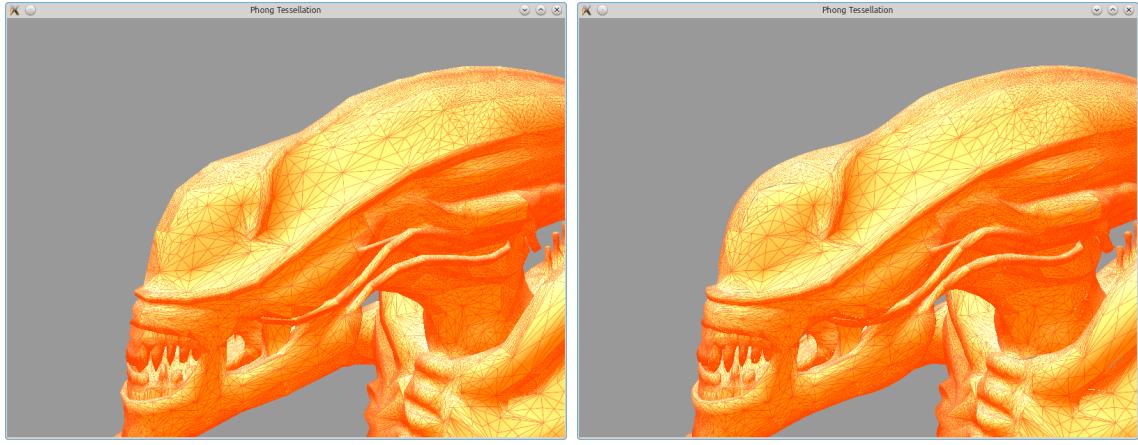
Para finalizar, este valor s_i se utiliza para definir el nivel de teselación para la arista i , realizando un mapeo lineal de $[0, 1]$ a $[0, t_{max}]$.

3.3.1. Modelos tridimensionales

Como ya se expuso, la definición de una métrica que tenga en consideración la silueta de un modelo es de especial importancia para los cálculos de iluminación. Por lo tanto, la aplicación directa de esta métrica es en el renderizado de un modelo tridimensional afectado por una fuente de luz. Los cálculos de la iluminación de la fuente de luz se realizan utilizando el modelo de Phong. Dada la posición de la cámara y las normales de todos los vértices, se calcula el nivel de teselación por arista como se explicó anteriormente. En la Fig. 9 se muestra el resultado de aplicar la métrica a un modelo complejo. Primero se muestra el modelo original y luego el resultado al agregar más elementos para mejorar la representación.

3.3.2. Modelos y Phong Tessellation

Adicionalmente se implementó un método especial que aprovecha el uso de teselación para mejorar la calidad general del modelo. Este método, denominado *Phong Tessellation* combina conceptos de teselación e iluminación de Phong para renderizar un modelo con perfiles más suaves [3]. Una vez que se definió el nivel de teselación para un triángulo y se generaron los nuevos vértices, éstos se desplazan fuera del plano del triángulo en una



(a) Modelo sin Phong Tessellation.

(b) Modelo con Phong Tessellation.

Figura 10: Teselación de un modelo por silueta y agregado de Phong Tessellation.

dirección y magnitud que depende de las normales del triángulo. Lo que se busca es generar triángulos curvados que representen con suavidad la frontera del modelo.

Este método es un agregado que se realiza en el Tessellation Evaluation Shader luego de haber calculado los niveles de teselación con respecto a la geometría original. Se implementó este método para evaluar la utilidad de teselación para algo distinto a generar nuevos vértices en el mismo plano que el triángulo original. El resultado obtenido demuestra que utilizar teselación puede ser un primer paso para crear geometría más fiel al modelo que se desea presentar. Este método presenta una ventaja con respecto al anterior ya que se le da más libertad a la posición donde se insertan los nuevos vértices. En la Fig. 10 se puede ver una comparación entre el resultado obtenido al utilizar Phong Tessellation o no. Se puede ver que los bordes quedan más curvos. Sin embargo, se aprecian “huecos” en el modelo, ocasionados por desplazamientos distintos en elementos adyacentes.

3.4. Curvatura

La curvatura es una propiedad de una curva que da una medida de cómo la misma se desvía, en cualquier punto, de una línea recta. Para una curva paramétrica $\mathbf{P}(t)$ la curvatura se define [4] como:

$$\kappa(t) = \frac{\mathbf{P}^t(t) \times \mathbf{P}^{tt}(t) \times \mathbf{P}^t(t)}{\|\mathbf{P}^t(t)\|^4} \quad (5)$$

donde $\mathbf{P}^t(t)$ es la primera derivada con respecto a t , y $\mathbf{P}^{tt}(t)$ es la segunda derivada, también con respecto al parámetro t . Según esta definición, la curvatura es un vector que apunta hacia donde la curva ‘gira’. El interés está en el módulo de este vector, que indica el valor de curvatura sin importar su dirección. Por lo tanto, cuando se hable de curvatura a continuación, se estará refiriendo al módulo del vector $\kappa(t)$.

Una curva, al renderizarla en pantalla, se approxima por un conjunto ordenado y conectado de segmentos de recta interconectados entre sí llamada isolínea. Al utilizar teselación en esta curva, se puede decidir en cuántos segmentos aproximarla. Un mayor nivel de teselación implica que se tendrán más valores del parámetro t para evaluar la curva y luego aproximarla con segmentos entre los puntos de muestra obtenidos. Mientras más segmentos se tengan,

con mejor precisión se podrá aproximar su forma real.

Para este ejemplo se decidió utilizar curvas de Bézier de tercer grado por su facilidad de tratamiento matemático. En las curvas de Bézier, el parámetro varía como $t \in [0, 1]$. En principio la curvatura varía en cada punto distinto donde se muestree el parámetro. Se decidió muestrear el parámetro en más de un valor y obtener un promedio de la curvatura a lo largo de la curva. Esto se hizo así porque no se tuvieron buenos resultados con un sólo valor. Entonces, se eligió tomar 3 valores t_i de este rango y calcular allí la curvatura. Una vez obtenidos estos valores, se hace un promedio entre las curvaturas calculadas, para obtener un sólo valor representativo de la curvatura de la curva:

$$\kappa_{curva} = \frac{\kappa(t_0) + 2\kappa(t_1) + \kappa(t_2)}{4} \quad (6)$$

Se eligió tomar 3 valores ya que no se apreciaron grandes cambios para mayor cantidad de muestras. Los puntos específicos en que se evalúa la curva son configurables, pero en principio se seleccionaron $t_0 = 0,2$, $t_1 = 0,5$ y $t_2 = 0,8$. De esta manera, se le da mayor peso a la curvatura en la mitad de la curva y además se tiene información cerca de los extremos.

Una vez obtenida la curvatura media de la curva, se mapea este valor al rango $[1, t_{tessmax}]$. Para realizar el mapeo, se debe definir un valor de curvatura máxima. Experimentalmente se fijó este valor en $\kappa_{max} = 2$. Por lo tanto, el mapeo es:

$$t_{tesslevel} = \frac{t_{tessmax} - 1}{\kappa_{max}} \kappa_{curva} + 1 \quad (7)$$

3.4.1. Curva de Bézier de tercer grado

Como se puede apreciar, para poder calcular la curvatura es necesario tener información sobre las derivadas primera y segunda. La curva de Bézier es una curva paramétrica muy utilizada por su sencillo tratamiento matemático. Además, las derivadas son fácilmente calculables. Es por ello que se eligió este tipo de curva para aplicar la métrica. Una curva de Bézier de tercer grado tiene 4 puntos de control. Estos puntos de control son los que conforman la primitiva que ingresa al shader de teselación. Al definir un nivel de teselación, se evalúa la curva en varios puntos del parámetro y se forma el conjunto de isolíneas que la aproximan.

Las curvas de Bézier son muy utilizadas para simular cabello y vegetación. En este ejemplo sólo se renderiza una única curva para mostrar las bondades de teselación ya que no se pudo obtener ni generar un modelo más complejo. Sin embargo, las ideas aquí expresadas pueden ser extendidas sin dificultad a un grupo extenso de curvas, y sin realizar cambios en la métrica. En la Fig. 11 se puede apreciar cómo varía la teselación de una curva según la curvatura de ésta. Además se expone una imagen obtenida al usar nivel de teselación máximo para la misma curva, para comparar el resultado de utilizar la métrica o emplear teselación máxima sin cálculos adicionales.

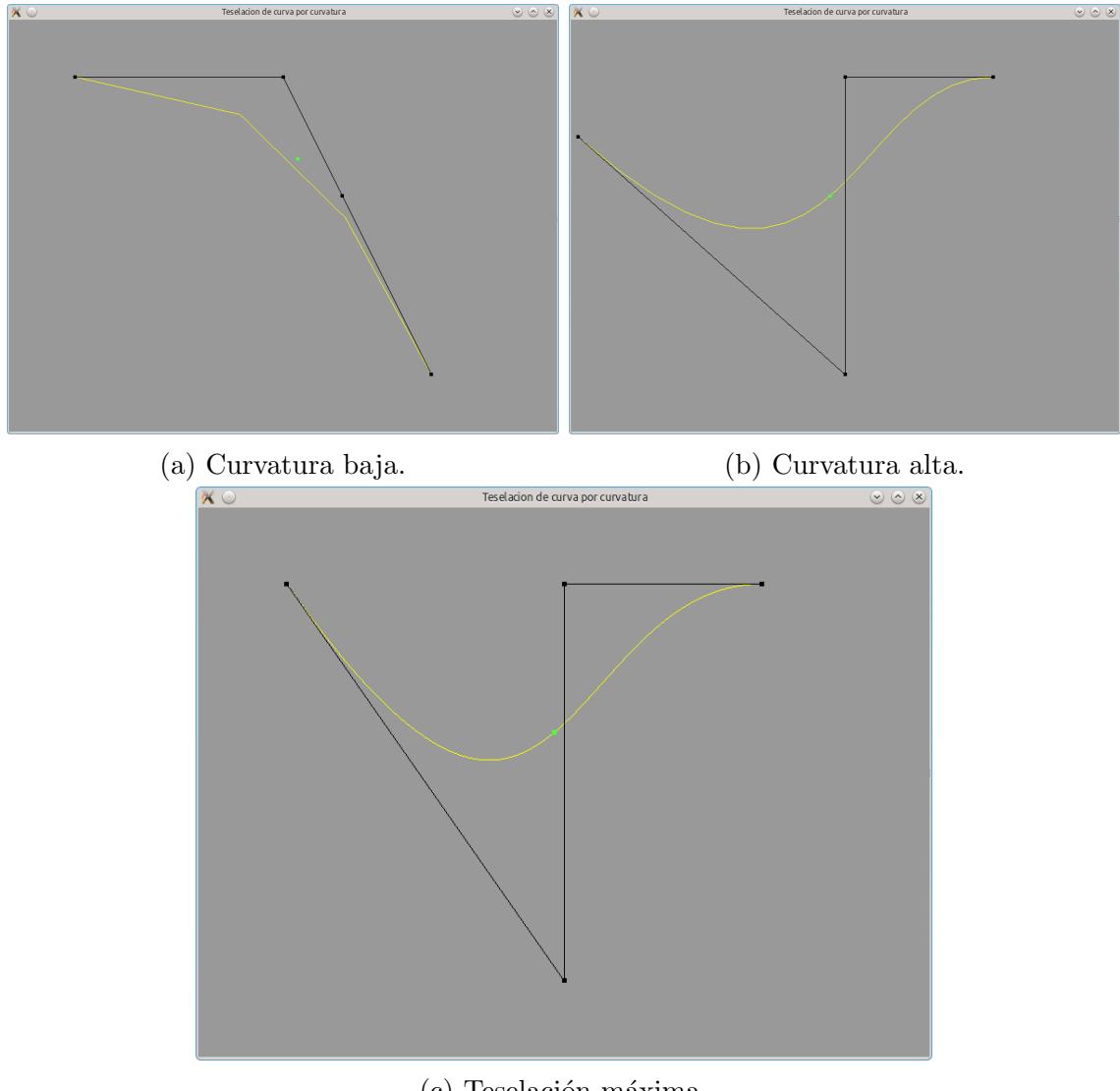


Figura 11: Teselación de una curva de Bézier por curvatura.

3.5. Métricas descartadas

Durante el diseño de métricas, algunas de éstas tuvieron que ser descartadas. En este apartado se menciona brevemente la idea original y por qué no se continuó con la métrica.

Curvatura en superficies

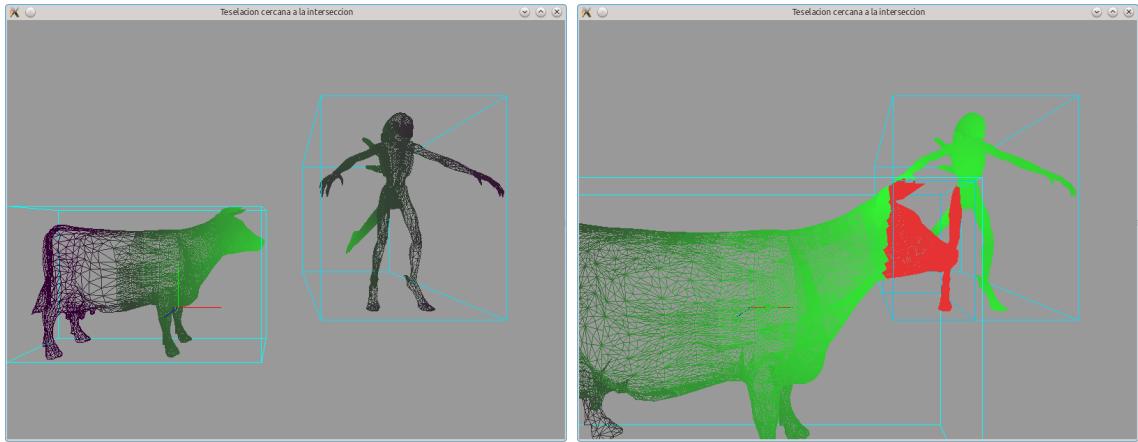
Tal y como se expresó, la curvatura de una curva es fácilmente calculable si se dispone de sus derivadas. Además, la curvatura de curvas tiene una única dirección. En contraposición, en superficies en el espacio, existen dos curvaturas. La primera es tangente a la superficie mientras que la segunda es normal a ésta. Sin embargo, en un punto de una superficie hay infinitos vectores tangentes. Es por esta razón que la curvatura no se puede calcular con facilidad. Diseñar una métrica que dependiera de la curvatura de la superficie conllevaría muchos cálculos y ni siquiera se obtendría una forma única de definir el nivel de teselación de la superficie.

Diferencia de normales en superficies

Se intentó buscar una alternativa al cálculo de la curvatura de una superficie. La misma consistía en calcular la desviación de los vectores normales a la superficie en puntos específicos. Pero para calcular estos vectores, se debería evaluar la superficie en varios puntos. Lo que no sería bueno para el rendimiento general. Por lo tanto, se intentó aproximar esta desviación midiendo la variación de los vectores normales de la malla de control de la misma. La malla de control de una superficie, al igual que el polígono de control de una curva, es un conjunto de puntos que son utilizados para calcular la posición en la superficie para distintos valores de los parámetros. Sin embargo, al implementar este método, no se obtuvieron buenos resultados ya que la desviación entre normales era grande aún para superficies no muy curvadas, lo que daba lugar a niveles altos de teselación en lugares donde no se necesitaban.

Distancia entre modelos

Como una variación a la métrica de distancia, se pensó en calcular distancias entre modelos tridimensionales. La idea bajo esta métrica era la de utilizar la distancia para generar representaciones más finas en los lugares cercanos a colisión, para poder calcular las intersecciones con mayor precisión. Se elige un nivel de teselación mayor o menor según la cercanía entre los objetos. Para calcular las distancias, se calcula la distancia del centroide de un triángulo de un modelo al bounding box del otro modelo. No hubo dificultades en la implementación de esta técnica, pero como no agrega nada distinto a lo ya expresado, se optó por descartarla de la selección final. En la Fig. 12 se puede apreciar el resultado de utilizar esta métrica. Se usa un código de colores para identificar las primitivas susceptibles de estar colisionando y las que no.



(a) Modelos alejados.

(b) Modelos intersectándose.

Figura 12: Teselación por distancias entre modelos.

4. Conclusión

Durante el desarrollo del proyecto se encontró con la dificultad de diseñar métricas de teselación, ya que el principal problema radica en respetar las propiedades que se mencionaron al inicio. Al iniciar el proyecto, se tenía la esperanza de poder encontrar un mayor conjunto de métricas. Sin embargo, durante el desarrollo se notó que si se desea definir algunas métricas, éstas pueden ser complicadas de evaluar y pueden afectar al rendimiento del software. Por lo tanto, en el proyecto se priorizaron métricas simples y fáciles de implementar y evaluar.

Para el resto del proyecto, faltan pulir detalles con respecto a las métricas implementadas. Además, se debe conformar el software final que combinará todos los ejemplos y permita cambiar entre éstos con facilidad. Por último, se realizarán pruebas para evaluar el desempeño de los algoritmos de teselación, tanto en rendimiento computacional como en calidad visual percibida.

Bibliografía

- [1] FreeType, “*Motor gratuito, portable y de alta calidad para el renderizado de fuentes en pantalla*”. Disponible en <http://www.freetype.org>. [Consultado el 3 de Abril de 2014].
- [2] Dorsch, J., “*Visualización de campos vectoriales sobre superficies mediante la técnica de Convolución sobre Integral de Línea*”, Proyecto Final de Carrera de Ingeniería en Informática, Facultad de Ingenería y Ciencias Hídricas, Universidad Nacional del Litoral, 2012.
- [3] Boubekeur, T. y Alexa, M., “*Phong tessellation.*”, ACM Transactions on Graphics (TOG). Vol. 27. No. 5. ACM, 2008.
- [4] Salomon, D., “*Curves and Surfaces for Computer Graphics*”, Springer Editorial, 2006.

| | |
|---|---------------------------------------|
| Proyecto Final de Carrera Ingeniería Informática | Informe de estado del proyecto |
| | FICH UNL |

| REALIZADO POR | FECHA | FIRMA |
|---------------------|------------|-------|
| Fernando Nellmeldin | 10/05/2014 | |
| REVISADO POR | FECHA | FIRMA |
| Dr. Néstor Calvo | 16/05/2014 | |
| APROBADO POR | FECHA | FIRMA |
| | | |

Nombre del Proyecto: Desarrollo de algoritmos de teselación adaptativa en GPU

Periodo del Informe: Diciembre de 2013-Abril de 2014

Alcance: Etapas 3 y 4 (parciales)

| Informe de estado del proyecto | | FICH | UNL |
|--------------------------------|--|------|-----|
| | | | |

| Estado del proyecto: | | | |
|---|------------------|-------------------|---|
| | | | |
| Etapa 3: Análisis, selección y diseño de técnicas y modelos | Actividad | Fecha realización | Resultados obtenidos |
| 3.3 Análisis de cada una de las técnicas | Estimada | Real | Se realizó el análisis y diseño de las técnicas correspondientes al tercer ciclo del modelo en espiral. |
| 3.4 Diseño de cada una de las técnicas | 20 horas | 50 horas | <i>Nota:</i> El detalle del análisis y selección de modelos, inicialmente planeado para este tercer informe de avance, finalmente fue incluido en el segundo informe de avance ya presentado. Sin embargo, por completitud, se incluyen aquí los tiempos empleados en esas actividades. |
| 3.6 Análisis de modelos tridimensionales para la evaluación de las técnicas | 7 horas | 20 horas | |
| 3.7 Selección de modelos | 20 horas | 15 horas | |
| | 10 horas | 10 horas | |
| Cronograma | | | |
| Etapa 4: Desarrollo de técnicas | Actividad | Fecha realización | Resultados obtenidos |
| 4.1 Desarrollo de cada una de las técnicas de teselación seleccionadas | Estimada | Real | Se realizó el desarrollo de las técnicas de teselación correspondientes al tercer ciclo (y último) del modelo en espiral. Este desarrollo tuvo muchas complicaciones pero al final se lograron buenos resultados. Las aplicaciones nuevas desarrolladas en esta etapa fueron: Intersección de objetos, Visualización de campos vectoriales y Curvas de Bézier (cuya métrica fue cambiada de: distancia a cámara, a: curvatura de la curva). Además se modificaron las ya desarrolladas. |
| 4.2 Pruebas de funcionalidad unitarias de cada técnica desarrollada | 40 horas | 100 horas | |
| 4.3 Redacción del tercer informe de avance | 10 horas | 20 horas | |
| | 15 horas | 40 horas | |

| Proyecto Final de Carrera Ingeniería Informática | Informe de estado del proyecto | | |
|---|--------------------------------|------|-----|
| | | FICH | UNL |

| Riesgos | Riesgo | Se efectivizó | Impacto | Mitigación |
|---------|--|---------------|-------------|---|
| | Pérdida de recursos hardware | Sí | No | |
| | Pérdida del desarrollo o investigación realizados | Sí | No | |
| | Cambio de tecnologías con las que se realiza el desarrollo | Sí | No | |
| | Funcionamiento de bibliotecas de software que no sea el esperado | Sí | Ñø Muy alto | Fue necesario rehacer el análisis de la técnica de teselación en campos vectoriales y cambiar el enfoque del algoritmo para lograr el resultado perseguido. |
| | Riesgos futuros | | | |
| | | Probabilidad | Impacto | |

| Proyecto Final de Carrera Ingeniería Informática | Informe de estado del proyecto <table border="1" data-bbox="223 361 954 631"> <thead> <tr> <th data-bbox="223 361 319 631">FICH</th><th data-bbox="319 361 954 631">UNL</th></tr> </thead> </table> | FICH | UNL |
|---|--|------|-----|
| FICH | UNL | | |
| <p>Notas</p> <p>Los tiempos inicialmente planeados para la culminación de este hito fueron totalmente superados. Las causas que llevaron a este atraso son varias: por un lado, el alumno dedicó menor cantidad de tiempo al desarrollo del proyecto del planeado inicialmente; por otro lado, dificultades en el desarrollo de la aplicación de teselación a campos vectoriales llevaron al alumno a volver atrás en el trabajo y volver a realizar el diseño y desarrollo de la misma, ya que no se podían obtener los resultados esperados. Esto fue ocasionado no por dificultades en la métrica de teselación aplicada, si no por problemas en el desarrollo de la visualización del resultado al aplicar la métrica. Se tuvo que realizar un algoritmo complejo que incluye múltiples pasadas por la GPU para lograr el resultado final. Este algoritmo de múltiples pasadas fue difícil de implementar ya que no se tenía el conocimiento sobre cómo realizarlo y los detalles que se debían tener en cuenta. Finalmente se obtuvieron los resultados perseguidos, pero a costa de retrasar el proyecto.</p> <p>Como aspecto positivo de este retraso, se pudo adquirir práctica en el desarrollo de estos algoritmos complejos que incluyen temas como <i>transform feedback</i> y <i>rasterización a textura</i>, útiles en cualquier tipo de algoritmo de computación gráfica.</p> <p>La redacción de este informe de avance también tuvo retrasos dado que se debía ordenar en un sólo documento todo el desarrollo realizado en las etapas 3 y 4 del proyecto. Este informe se retrasó porque constantemente se cambiaron detalles tanto del código como de las métricas implementadas, hasta que se lograron obtener los resultados buscados en todos los ejemplos.</p> | | | |