Universidad Nacional del Litoral
Facultad de Ingeniería y Ciencias Hídricas

Desarrollo de algoritmos de teselación adaptativa en GPU

Segundo informe de avance

Alumno: Fernando Nellmeldin
Director: Dr. Néstor Calvo

Santa Fe, Noviembre de 2013

Índice

1. Introducción

De acuerdo a como se explica en el anteproyecto, la superficie de los objetos se representa subdividida en pequeños elementos, los cuales pueden ser triángulos o cuadriláteros. También se utilizan curvas subdivididas en situaciones tales como en la representación del cabello. La precisión de la representación será tanto mayor cuanto más pequeños sean los elementos. La utilización de técnicas de teselación tiene como objetivo lograr un mayor grado de precisión en las áreas de interés. La teselación de un polígono lo subdivide en polígonos más pequeños. Al generar más polígonos, se alcanza un nivel de detalle mayor en el modelo que estos representen. Por ejemplo, si se realizan cálculos de iluminación y sombreado por polígono, y se cuenta con más polígonos por unidad de área, se puede obtener más precisión en los cambios de luminosidad. Sin embargo, al generar más polígonos, aumenta la carga del procesador gráfico. Esto es debido a que deben realizar más cálculos por cuadro, lo que podría hacer más lento el renderizado, es decir: la generación de la imagen a mostrar en la pantalla.

Para llevar a cabo el proyecto, se propone desarrollar e implementar un conjunto de técnicas y algoritmos que logren ejemplificar todas las características de la tecnología de teselación en hardware. Mediante la utilización de teselación en diversas aplicaciones, se busca obtener un conocimiento más completo y preciso sobre la misma. La elección de cada aplicación se realizó con el objetivo de que sean diferentes entre sí y cubran el mayor rango de usos posible que se le pueda dar a la tecnología de teselación en GPU (Graphics Processing Unit). Entre los ejemplos elegidos, se encuentran aplicaciones puramente didácticas (superficies paramétricas), aplicaciones a videojuegos (modelos de personajes y colisiones) y aplicaciones científicas (terrenos y visualización de campos vectoriales).

En términos generales, la teselación se puede realizar en tres tipos de primitivas distintas: triángulos, cuadriláteros y líneas (*isolines* en terminología de OpenGL). Cada una de las aplicaciones elegidas utiliza un tipo de primitiva en particular. Se proponen 3 grupos de aplicaciones a desarrollar, donde cada grupo conformará un ciclo del modelo en espiral propuesto para realizar el proyecto. A continuación se mencionan los grupos propuestos y el tipo de primitiva que utiliza cada uno:

1. **Grupo 1**
   - 1.1 Curvas de Bézier: líneas
   - 1.2 Superficies paramétricas: cuadriláteros
   - 1.3 Terrenos: cuadriláteros
2. **Grupo 2**
   - 2.1 Modelos de personajes: triángulos
   - 2.2 Phong Tessellation: triángulos
3. **Grupo 3**
   - 3.1 Intersección de objetos: triángulos
   - 3.2 Visualización de campos vectoriales: cuadriláteros

Aplicaciones no seleccionadas

En la etapa de investigación, se identificaron muchas aplicaciones en las que se pueda utilizar teselación por hardware. Para continuar con el proyecto, fue necesario seleccionar un conjunto de ejemplos que permitieran cubrir las capacidades de teselación del hardware.

Algunas elecciones fueron hechas por interés o gusto personal, mientras que otras por el grado de dificultad trivial o excesivo. Del mismo modo, se debieron descartar algunas ideas. A continuación se menciona una lista de las aplicaciones que fueron descartadas y las razones correspondientes:

- **Teselación de pelaje**: consiste en generar pelaje (pelaje corto o cabellera larga) a partir de la teselación de líneas. Se descartó debido a que no se pudo adquirir un modelo adecuado en donde probar las técnicas. Normalmente los sistemas de modelado de pelaje o cabellos son sistemas complejos y muy completos que resultan difíciles de parcializar para ejemplificar lo que aquí pretende mostrarse. Sí se incluyó una aplicación simple de teselación de curvas de Bézier, la cual utiliza los mismos principios que la teselación de pelaje, pero con un modelo mucho más sencillo (curvas creadas individualmente).
- **Teselación por iluminación especular**: en esta aplicación, se decide teselar una primitiva si está afectada por iluminación especular variable en su superficie. Esta aplicación se descartó debido a que se trata más de una métrica a aplicar, que una aplicación en sí. Sin embargo, tampoco resulta una métrica adecuada porque sólo tiene en cuenta uno de los tantos elementos del modelo de iluminación. Al considerar los cambios de iluminación, otras métricas tales como la detección de curvatura, resultan más adecuadas.
- **Teselación de fluidos**: se trata de teselar una malla que representa un fluido en movimiento constante afectado por fuerzas tanto internas como externas. Esta es una aplicación muy interesante que involucra cálculos de mecánica computacional en fluidos, pero se descartó debido a que los tiempos de desarrollo son los propios de un proyecto de posgrado.
- **Teselación de superficies de subdivisión**: las superficies pueden definirse en forma paramétrica o mediante el límite de un modelo definido de subdivisiones sucesivas. Se busca teselar superficies de subdivisión, de manera de obtener más detalle (más iteraciones de subdivisión) en algunas superficies, y menos en otras. Las curvas de Bezier, ya elegidas en otra aplicación, utilizan el mismo principio de subdivisión y tienen mayor riqueza pues representan a ambos conjuntos. Esto último es debido a que las superficies de subdivisión se suelen convertir en superficies de Bézier para teselar. Por lo tanto, en este caso no habría un aporte distinto al ya incluído.

El resto de este informe se organiza de la siguiente manera:

- En la sección 2 se explicará qué es una métrica de teselación y qué impacto tiene definir cierto nivel de teselación en un elemento cualquiera.
- En la sección 3 se detallará la estructura de clases básica que se propone para el desarrollo del proyecto.
- En las secciones 4 a 10, se precisarán las aplicaciones a desarrollar y las métricas elegidas para cada una.
- En la sección 11 se realizará un breve comentario sobre los modelos en los que se van a probar los algoritmos.
- Por último, en la sección 12 se expone una breve conclusión.

2. Métricas y niveles de teselación

En la descripción de todas las técnicas, se mencionará el término genérico de *patch*. Un patch puede referirse a un triángulo, un cuadrilátero, o una superficie paramétrica, según cada técnica en particular.

Para cada aplicación, se deberá elegir una o más métricas. En el caso de teselación, una métrica es una forma de medir cuánto se debe teselar (subdividir) un patch, de manera de balancear el nivel de detalle y la carga del procesador. Una métrica simple es, por ejemplo, la distancia del patch a la cámara. Más adelante se explicará la forma de medir esta distancia. Si el patch está alejado de la cámara, no es necesario subdividirlo, ya que este aumento de precisión no se podrá apreciar. En cambio, si el patch está cerca del punto de vista, la subdivisión será necesaria para darle más detalle (cantidad de polígonos que lo representan).

La métrica puede generar un nivel de teselación único para todo el patch, o distinto para cada lado del mismo. El nivel de teselación define cómo subdividir un patch. Por ejemplo, un cuadrilátero tiene 4 niveles de teselación distintos, uno para cada lado. Un triángulo, por su parte, tiene 3 niveles. Además, para los cuadriláteros se puede definir dos niveles de teselación internos, uno vertical y uno horizontal. Los triángulos sólo tienen uno. A mayor nivel de teselación, se generan más polígonos por lado. Un nivel de teselación igual a 1, quiere decir que el polígono no se subdivide. El nivel de teselación máximo, según el hardware específico que se cuenta para el desarrollo del proyecto, es igual a 64. Por lo tanto, se generan 64 segmentos por lado y se crea un total del orden de 8000 triángulos. Sin embargo, esta cantidad depende específicamente del hardware. En la Fig. 1 se pueden apreciar distintos niveles de teselación para un cuadrilátero.

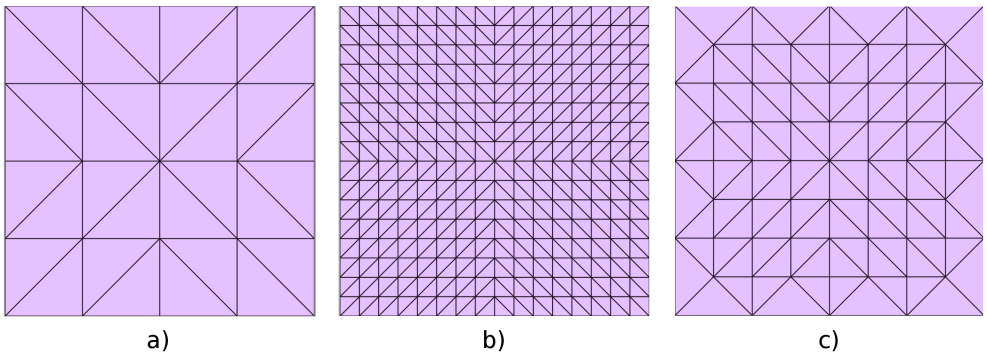

Figura 1: Niveles de teselación para un cuadrilátero: *a)* Niveles internos y externos igual a 4. *b)* Niveles internos y externos igual a 16. *c)* Niveles internos igual a 8. Niveles externos igual a 4.

Existe un tercer tipo de primitiva a la que se le puede aplicar teselación. Además de subdivisión en triángulos y cuadriláteros, la GPU tiene soporte para teselación de líneas a través de la primitiva *isoline*. En esta primitiva, se puede especificar en cuántos segmentos se desea que se subdivida una línea. Al dividir esta línea, se generará un conjunto de puntos con los que se podrá trabajar individualmente. Este tipo de primitiva es de utilidad, por ejemplo, para teselación de curvas paramétricas.

3. Diseño general del software

Para poder probar los algoritmos, se plantea crear un software interactivo básico que permita la apreciación de las características del uso de teselación. Este software se escribirá utilizando el lenguaje de programación C++ y la API OpenGL 4.0. El software permitirá elegir qué ejemplo cargar (modelos de personaje, terrenos, etc.) y a partir de allí se podrán controlar ciertos apartados específicos de cada ejemplo, tales como parámetros de dibujado o teselado, movimiento de cámara, etc. Para ello, se decidió utilizar la siguiente estructura de clases:

- **Clase Model**: se encarga de leer un modelo desde archivo y almacenar en memoria todos sus datos (vértices, caras, normales).
- **Clase ShaderProgram**: agrupa las funcionalidades para interactuar con la GPU. Se encarga de leer los archivos de shaders y transferir las variables desde el programa cliente OpenGL a la GPU.
- **Clase VAO**: su función es la de realizar una abstracción para facilitar la comunicación de conjuntos de datos entre un Modelo y un Shader. Recibe su nombre de *Vertex Array Object*, la estructura OpenGL que direcciona los datos.
- **Clase Example**: su función es agrupar todos los datos y funcionalidades necesarias para los ejemplos. Cada ejemplo en particular heredará de esta clase y agregará datos y funcionalidades apropiadas según corresponda. Además, se encarga de gestionar los *callback* de OpenGL, los cuales son funciones que se ejecutan automáticamente mientras el programa esté en ejecución.
- **Namespace Utils**: almacena funcionalidades generales tales como conversión de tipos, comprobación de errores, lectura de imágenes para texturas, etc.

En la Fig. 2 se puede ver la relación general entre clases. La clase más importante es Example, que se relaciona con el resto según como se indicó anteriormente. Cada ejemplo en particular copiará todas las funcionalidades de la clase Example utilizando el mecanismo de herencia de C++. Sin embargo, podrá redefinir los métodos según sea necesario, así como también agregar nuevas propiedades y funcionalidades. Por otro lado, Utils forma un *namespace*, el cual es un conjunto de funciones y constantes agrupados bajo un mismo nombre. Todas las clases pueden acceder a los datos internos del namespace y hacer uso de sus funciones. En este namespace se agruparán todas las funcionalidades que no se pueden identificar con una única clase.

Esta estructura fue creada a partir del análisis de los problemas que se querían resolver. Debido a que el proyecto consiste en un grupo de aplicaciones, se debe desarrollar un software flexible que permita adaptarse a cada aplicación. Además, se notó que al desarrollar habría mucha repetición de código ya que las aplicaciones tienen muchas cosas en común (interacción con teclado, movimiento de cámara, carga de modelos, etc.). Se necesitó contar con contenedores que realizaran una abstracción de los procesos repetitivos, de manera de agilizar el desarrollo. Clases como Model, VAO y ShaderProgram surgen de la necesidad de evitar la repetición de escritura de código. Estas clases se escribirán una única vez, realizando pocas modificaciones durante el resto del desarrollo. Por otra parte, la clase Example es la más volátil, debido a que depende de cada aplicación. Para evitar que las modificaciones en una aplicación en particular afectaran a las otras, se decidió utilizar herencia. De esta manera, en Example se pueden ubicar los métodos y propiedades generales, mientras que en sus clases heredadas se colocan los comportamientos especiales de cada aplicación.

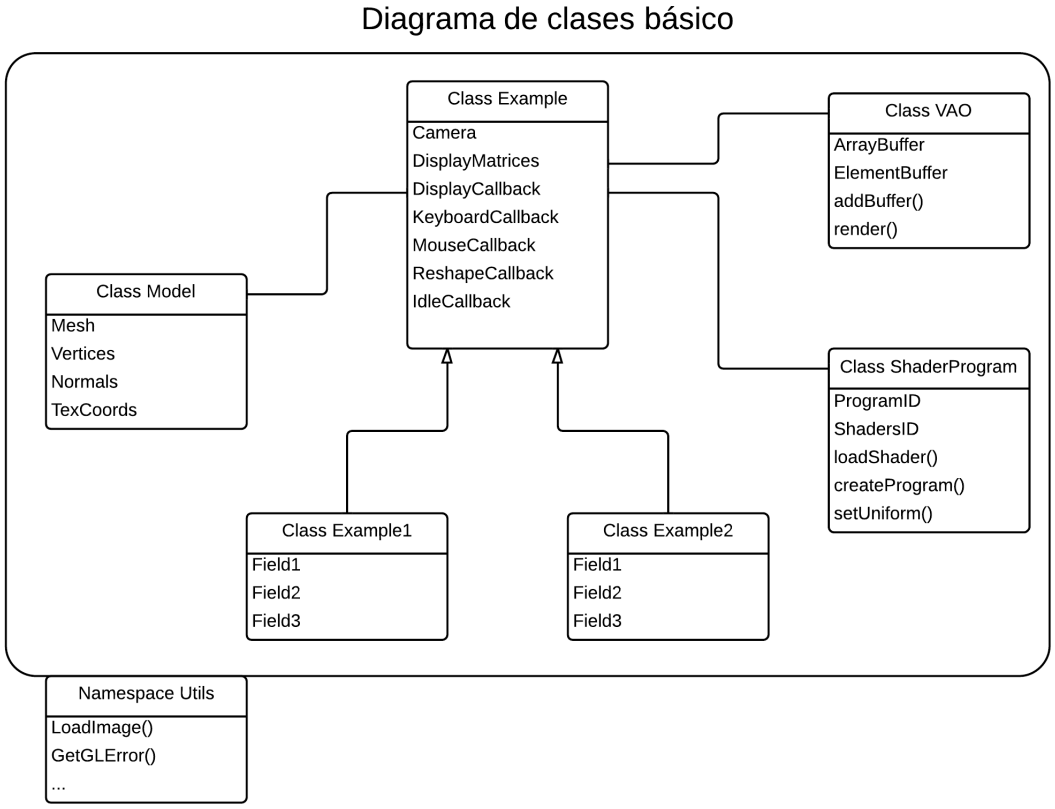

Figura 2: Diagrama de clases.

4. Curvas de Bézier

Se busca elaborar una demostración simple de cómo se utiliza la primitiva *isoline* para realizar teselación en líneas. Esta primitiva se puede utilizar para creación y animación de cabello o pelaje en modelos para videojuegos o películas.

Una curva de Bézier es una curva paramétrica con grado definido [1]. En el caso de una curva de Bézier de tercer grado, se trata de una curva definida por cuatro (3+1) puntos de control, y cuyo parámetro varía entre cero y uno. Cuanto más cantidad de valores del parámetro se muestree, más precisión visual tendrá la curva renderizada. En este caso, la "línea" a teselar es la curva, pero las subdivisiones se realizan en el dominio paramétrico, definiendo en cuántos y cuáles valores del parámetro se calculará la curva. Esta curva finalmente se representa en pantalla por una secuencia de segmentos rectos. En la Fig. 3 se expone un ejemplo de teselación de una curva de Bézier.

Métrica

Debido a que se trata de aplicación simple, se plantea también usar una métrica sencilla para realizar la demostración. En cada cuadro, se mide la distancia en coordenadas de mundo, del punto de vista (cámara) a cada uno de los (pocos) puntos de control de la curva. La distancia que interesa es la mínima de todas las calculadas. Si la distancia es grande, no se

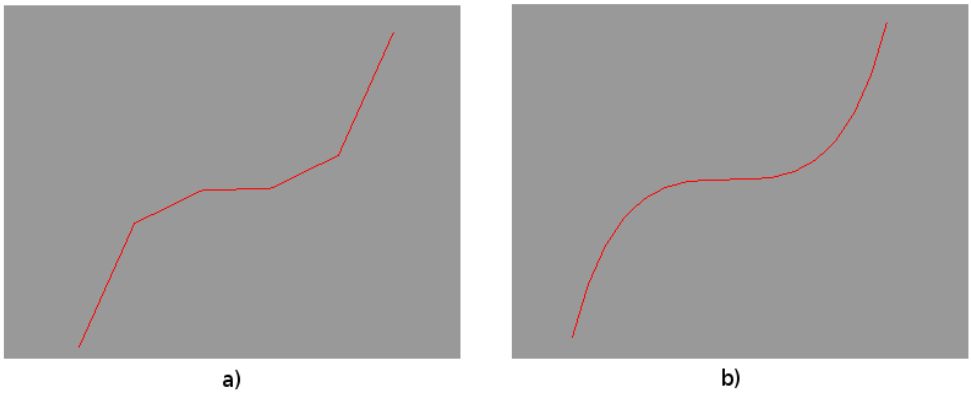

Figura 3: Teselación de una curva de Bézier de 4 puntos de control: *a)* 5 niveles de subdivision. *b)* 16 niveles de subdivisión.

debe teselar la curva. Por otra parte, si la cámara está cerca de la curva, se desea que se represente con más detalle. Para ello, se debe muestrear con más densidad el dominio de la curva, objetivo que se logra al especificar un nivel de teselación alto.

5. Superficies paramétricas

Se parte de un modelo formado por un grupo de superficies paramétricas tales como superficies de Bézier o NURBS. Para renderizar el modelo, se deben muestrear las superficies mediante un conjunto de puntos en el dominio paramétrico de las superficies. La cantidad de puntos en dónde muestrear se elige según la métrica.

Métricas

Distancia al centroide

Se elige un único nivel de teselación para todos los lados del patch. Para cada patch, se calcula su centroide con respecto a los puntos de control. Se mide la distancia, en coordenadas de mundo, de la cámara a este centroide. Según esta distancia, el nivel de teselación T_{lineal} se decide según la fórmula:

$$T_{lineal} = \begin{cases} -\frac{(64-1)}{d_{max}}x + 64, & 0 < x < d_{max} \\ 1, & x \geq d_{max} \end{cases} \tag{1}$$

donde d_{max} es el valor crítico de distancia a la cámara desde donde el nivel de teselación siempre es 1. Este valor se estima de manera de asegurar un tamaño homogéneo de polígonos en pantalla. Por ejemplo, una misma cantidad de píxeles que representan cada lado de los polígonos.

Al aplicar esta fórmula, se obtiene un nivel de teselación que es máximo (64) para $x = 0$, y es mínimo (1) para valores igual o mayores a d_{max}. En este caso, x representa la distancia del centroide a la cámara.

Curvatura del patch

Se elige un nivel de teselación de acuerdo a la curvatura del patch. La curvatura de una superficie es variable en cada punto, por lo que para poder definirla con exactitud, se debería calcular en cada punto del dominio de la misma. Sin embargo, se busca precisar una métrica que no dependa de la evaluación de la superficie, si no que se pueda calcular con respecto a sus puntos de control.

Para realizar una aproximación a la curvatura, utilizando sólo los puntos de control, se propone realizar el siguiente proceso:

1. Considerar a los puntos de control como una malla, compuesta de nodos (los puntos de control) y elementos (cuadriláteros formados por estos puntos de control).
2. Obtener la normal para cada uno de estos elementos.
3. Calcular la desviación entre dos normales adyacentes. La desviación está dada por el ángulo entre las normales, obtenida a partir de su producto punto. Este proceso se repite para todas las normales del patch.
4. La curvatura total del patch se estima como la mayor desviación entre normales adyacentes.

Por lo tanto, el nivel de teselación será mayor mientras mayor sea la desviación (menor el producto punto). Cuando el desvío se acerca a 1 (curvatura nula), las normales tienen la misma dirección, la superficie está en un plano y la división del dominio debe ser mínima ya que no expone muchos detalles.

Se preveé que esta métrica no sea adecuada y aproxime una curvatura de la superficie no representativa de la cantidad de divisiones que se deban aplicar a la misma. Esto se produce al considerar que puede realizar sobreestimaciones debido a desviaciones altas presentes en un extremo del patch, y ausentes en el resto. Asimismo, se preveé que sólo será adecuada para superficies reducidas con pocos puntos de control, ya que a mayor cantidad de puntos de control, más incorrecta será la aproximación. A pesar de todas estas limitaciones, se decide perder un poco de precisión en la aproximación para ganar velocidad en los cálculos, ya que resulta ser una métrica simple para el cálculo de una característica muy compleja de una superficie.

6. Terrenos

Un terreno está formado por polígonos (triángulos o cuadriláteros) en el espacio. Para esta aplicación en particular, se va a utilizar una malla *gruesa* formada por cuadriláteros en el plano, y un mapa de desplazamientos para las alturas del terreno. El mapa de desplazamientos tiene una densidad mayor que la cantidad de vértices que tiene la malla. Se subdivide la malla según el nivel de teselación calculado para cada elemento, y se aplica el mapa de desplazamiento en cada vértice nuevo para moverlo verticalmente. El valor a utilizar en cada vértice se obtiene directamente del mapa de desplazamiento.

Métricas

En el caso de terrenos, los patch son cuadriláteros. Según la métrica, se elige el nivel de subdivisión del patch en cada uno de sus lados.

Cercanía

Se mide la distancia del patch a la cámara en coordenadas de mundo. Si éste se encuentra muy alejado, no se debe subdividir de la misma manera que si estuviera cerca de la cámara. Se plantea aquí una función lineal similar a la utilizada en la sección de superficies paramétricas a partir del centroide del patch.

Longitud de arista

Se calcula la longitud, dada en cantidad de píxeles, de una arista del patch una vez que se proyecta en pantalla. Se busca que todas las aristas de cada triángulo final de renderizado posean longitud similar L_{ref}. Por ejemplo, 10 píxeles por arista.

Para calcular la longitud proyectada de un segmento $\overline{e_i e_j}$, se mide la distancia d_{ij} entre los puntos e_i y e_j proyectados en el espacio de pantalla. El valor de teselación T a usar en esta arista es:

$$T = \begin{cases} 1 & d_{ij} \leq L_{ref} \\ \frac{d_{ij}}{L_{ref}} & \text{en otro caso} \end{cases} \tag{2}$$

7. Modelos de personajes

Se posee un modelo de un personaje a utilizar, por ejemplo, en un videojuego. El modelo está formado por elementos triangulares y cuyas normales de cada nodo vienen dadas. En el caso de no tener las normales, se calculan al iniciar la aplicación. Se busca subdividir los polígonos de este modelo, de manera de obtener más detalle en ciertas áreas de interés, y menos detalle en otras. Las áreas en las que se tiene interés en refinar son los contornos de los modelos, denominados siluetas. En la silueta, si no se cuenta con detalle, un mismo polígono puede representar todo el borde, situación que impide, por ejemplo, una correcta iluminación. Por lo tanto, al identificar secciones que pertenecen a la silueta, se puede generar una mayor densidad de polígonos que ayuden a mejorar los cálculos de iluminación.

Métrica

Se parte de una malla triangular que consiste en vértices $\mathbf{v}_i$ y normales en cada vértice $\mathbf{n}_i$. Para cada patch, se calcula su centroide $\mathbf{p}$. Dado el punto $\mathbf{c}$ donde se ubica la cámara de la escena, se procede de la siguiente manera para obtener el nivel de teselación para cada elemento [2].

Se calcula el vector unitario $\mathbf{u}$ que va de $\mathbf{c}$ a $\mathbf{p}$:

$$\mathbf{u} = \frac{\mathbf{p} - \mathbf{c}}{\| \mathbf{p} - \mathbf{c} \|} \tag{3}$$

Para cada uno de los vértices $\mathbf{v}_j$ del triángulo, se calcula el vector $\mathbf{v}_{cj}$ que conecta $\mathbf{c}$ con $\mathbf{v}_j$:

$$\mathbf{v}_{cj} = \mathbf{v}_j - \mathbf{c} \tag{4}$$

A continuación, se calcula el vector $\mathbf{v}_j^u$ que tiene la dirección de $\mathbf{u}$ y su módulo es la proyección de $\mathbf{v}_{cj}$ sobre la dirección de $\mathbf{u}$

$$\mathbf{v}_j^u = \langle \mathbf{v}_{cj} \cdot \mathbf{u} \rangle \mathbf{u} \tag{5}$$

El siguiente paso es calcular el vector diferencia entre $\mathbf{v}_j$ y $\mathbf{v}_j^u$. A este vector se lo identifica como $\mathbf{d}_{ij}$:

$$\mathbf{d}_{ij} = \mathbf{v}_j - \mathbf{v}_j^u \tag{6}$$

Este vector identifica la distancia, ortogonal al vector $\mathbf{u}$, del vértice $\mathbf{v}_j$ con el centroide del triángulo $\mathbf{p}$.

Por último, se define un peso w_{ij}:

$$w_{ij} = \frac{1}{\| \mathbf{d}_{ij} \|} \tag{7}$$

Este peso es más grande mientras menor sea la distancia proyectada de los vértices, y que por lo tanto pertezcan a la silueta. Para vértices más alejados, por ejemplo en polígonos que están de frente a la cámara, este peso tiende a cero.

Para finalizar, se define el parámetro de teselación t_i para el elemento i como:

$$t_i = \arg \max_j \{ w_{ij} \cdot \langle \mathbf{n}_i \cdot \mathbf{n}_j \rangle \} \tag{8}$$

En la ecuación anterior, $\mathbf{n}_i$ es la normal del elemento (promedio de sus normales) y $\mathbf{n}_j$ es la normal de cada uno de sus vértices.

Posteriormente, t_i debe mapearse al rango $[1, 64]$, de manera de dar un nivel de teselación acorde a los requerimientos de la GPU. Para ello, se hace un mapeo donde se define t_{min} para el cual la teselación será 64, y t_{max} en donde será 1. En el interior, se realiza un mapeo lineal de t_i al nivel de teselación T en $[1, 64]$. El mapeo es entonces:

$$T = \begin{cases} 64 & t_i < t_{min} \\ \frac{t_{min} - t_i}{t_{max} - t_{min}} 63 + 64 & t_{min} \leq t_i \leq t_{max} \\ 1 & t_i > t_{max} \end{cases} \tag{9}$$

Esta métrica tiene en cuenta la diferencia de normales, y por lo tanto la curvatura del modelo, como también la posición de la cámara con respecto al modelo.

8. Phong Tessellation

En esta aplicación, también se busca teselar un modelo de personaje, pero a partir de la métrica se crean nuevos elementos en posiciones desplazadas del original. En Phong Tessellation [3], se utilizan las ideas del modelo de iluminación de Phong para realizar teselación de triángulos. En el caso normal, al subdividir un triángulo, la teselación lineal ubica a los nuevos puntos en el interior del triángulo y en su mismo plano. 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 puntos nuevos ya no están en el mismo plano que el elemento original, si no que forman un triángulo curvado.

Para realizar Phong Tessellation, se parte de un triángulo con puntos en el espacio $\mathbf{v}_i, \mathbf{v}_j$ y $\mathbf{v}_k$, y normales en cada uno de estos puntos: $\mathbf{n}_i, \mathbf{n}_j$ y $\mathbf{n}_k$. La teselación genera un conjunto de puntos en el triángulo; cada uno está definido por sus tres coordenadas baricéntricas:

$$\mathbf{p}(u, v) = (u, v, w)(\mathbf{v}_i, \mathbf{v}_j, \mathbf{v}_k)^T \tag{10}$$

con $u, v \in [0, 1]$ y $w = 1 - u - v$.

Para un punto cualquiera $\mathbf{p}$, ubicado en el triángulo, se procede de la siguiente manera para calcular su nueva ubicación espacial. Se calcula el plano tangente a cada vértice, definido por el vértice y su normal. Se denomina a cada uno de estos planos como π_i π_j y π_k.

Se proyecta ortogonalmente el punto $\mathbf{p}$ en cada uno de los planos π_i. Esta proyección está dada por:

$$\pi_i(\mathbf{p}) = \mathbf{p} - ((\mathbf{p} - \mathbf{v}_i) \cdot \mathbf{n}_i)\, \mathbf{n}_i \tag{11}$$

A continuación, a partir de las 3 proyecciones en cada uno de los planos tangentes, se realiza una interpolación baricéntrica entre estos valores, utilizando las coordenadas baricéntricas del punto $\mathbf{p}$ en el triángulo original. De esta manera se obtiene la posición final $\mathbf{p}*$:

$$\mathbf{p}* = (u, v, w) \begin{pmatrix} \pi_i(\mathbf{p}) \\ \pi_j(\mathbf{p}) \\ \pi_k(\mathbf{p}) \end{pmatrix} \tag{12}$$

Este procedimiento se repite para todos los puntos generados en la etapa de teselación en todos los triángulos. El resultado logrado es que los triángulos creados en la subdivisión se adecuan mejor a la geometría descripta por las normales del triángulo inicial. En la Fig. 4 se muestra el principio básico de Phong Tessellation.

Métrica

En el trabajo de Boubekeur y Alexa [3], se utiliza una métrica cuyo objetivo es refinar los contornos de los modelos. En esta métrica, se tiene en cuenta la dirección de la normal de cada vértice del elemento con respecto a la cámara. Si la normal está apuntando en dirección a la cámara, la subdivisión no es necesaria ya que no se trata de un contorno. Por otra parte, si el triángulo se encuentra en la silueta, y la normal de alguno de sus vértices está en

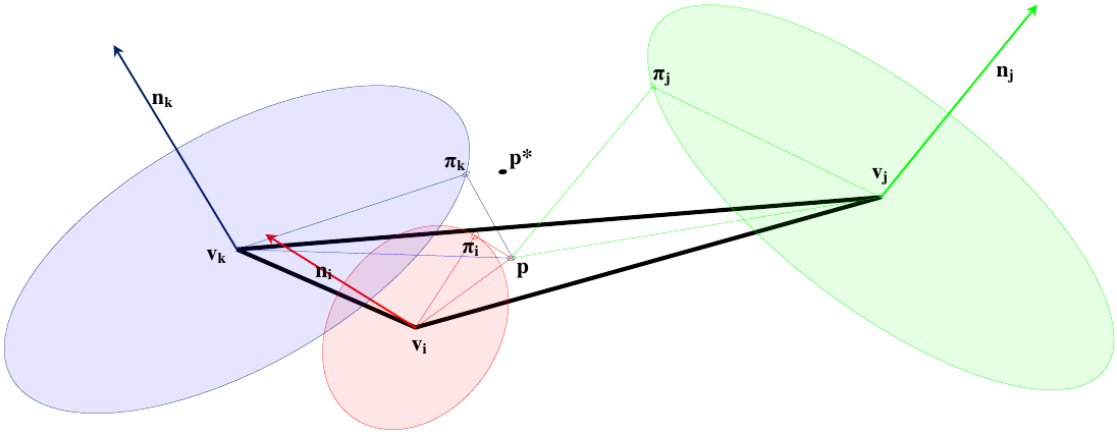

Figura 4: Principio básico de Phong Tessellation.

dirección perpendicular a la dirección de la cámara, se debe subdividir para poder obtener el triángulo curvado propio de la técnica de Phong Tessellation.

Sea m el nivel máximo de teselación definido por el usuario, $\mathbf{c}$ la posición de la cámara, y d_i el nivel de teselación calculado para el vértice $\mathbf{v_i}$, se elige el mayor d_i tal que:

$$d_i = \left(1 - \left\| \mathbf{n}_i \cdot \frac{(\mathbf{c} - \mathbf{v}_i)}{\| \mathbf{c} - \mathbf{v}_i \|} \right\|\right) m \tag{13}$$

Para vértices cuyas normales estén en la dirección a la cámara, esta métrica dará un nivel de teselación 0 (que posteriormente se lleva a 1). Para normales perpendiculares a la dirección a la cámara, el nivel de teselación elegido será m. En esta métrica, se utiliza valor absoluto para considerar casos en que sean objetos abiertos, en donde a pesar de que la normal del elemento esté en dirección opuesta a la cámara, la cara de atrás del elemento es visible. Esta métrica permite una transición suave del grado de refinamiento en elementos vecinos cercanos a la silueta.

9. Visualización de campos vectoriales

La técnica de LIC (Convolución sobre Integral de Línea) se utiliza para la visualización de campos vectoriales. En particular, se posee un campo vectorial sobre una superficie discretizada. A partir de LIC, se obtiene una imagen resultado que muestra las líneas de corriente (*streamlines*) sobre la superficie [4]. Para obtener esta imagen, se parte de una malla regular de cuadriláteros y una imagen de ruido. Se mapea cada vértice de la malla a un valor de píxel de la imagen de ruido. A continuación se simula el movimiento de los vértices de la malla siendo afectados por el campo vectorial. El resultado es una imagen de ruido "movida" por efecto del campo vectorial. Este proceso se repite a lo largo de una serie de pasos, y se acumulan los resultados. El producto final es una imagen donde se ven las líneas de corriente que afectan a la superficie (ver Fig. 5).

La imagen resultado depende fuertemente de la discretización de la superficie. Mediante la utilización de técnicas de teselación, se busca poder generar un mallado denso en las regiones de interés, para así obtener una representación más precisa del mismo.

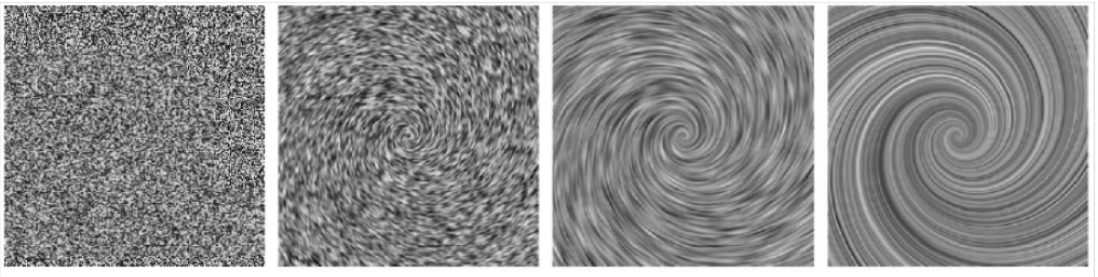

Figura 5: Movimiento de la imagen de ruido siendo afectada por un campo vectorial.

Métrica

La métrica utilizada en esta aplicación, debe tener en cuenta la variación del campo vectorial y la región de interés dada por el usuario. De esta manera, se debe identificar dónde este varía mucho y realizar allí una subdivisión más densa. En otra línea, el usuario puede elegir una zona de interés y realizar allí un acercamiento, de manera de adquirir más detalle.

A partir de la interacción con el software, el usuario puede realizar un acercamiento en una región en particular. Al momento de detectar esto, el software debe subdividir los elementos afectados por la vista actual. Los elementos deben ser subdivididos de manera de asegurar una discretización tal que se pueda ver con detalle el campo vectorial que está afectando la región de interés. En particular, se debe asegurar un tamaño de elemento adecuado que permita la correcta y precisa visualización.

Por ejemplo, se puede garantizar que cada lado del cuadrilátero tenga una longitud de 10 píxeles en pantalla. Si el lado del elemento tiene una longitud mayor, se debe subdividir ya que, en caso contrario, el campo vectorial mostrado no es "denso" en la región de interés. Continuando con el ejemplo, en casos donde el lado del elemento es menor a 10 píxeles, no se debe subdividir ya que el campo vectorial será denso por defecto.

10. Intersección de objetos

Se dispone de un grupo de objetos, representados por superficies paramétricas (por ejemplo NURBS o Bézier) o por mallas triangulares. Estos objetos están en movimiento y se desea calcular -si existe- el área de intersección entre dos objetos cualesquiera de este conjunto. Identificar el área de intersección es de interés en el cálculo de colisiones para actualización de mundos dinámicos en videojuegos. Por ejemplo, al renderizar dos vehículos en un videojuego simulador de carreras, se busca que el cálculo de la colisión entre los dos objetos se realice en tiempo real (cuadro a cuadro). De este modo, el motor de física podrá realizar eficazmente las modificaciones necesarias al estado de los objetos (posiciones, deformaciones, etc.). Sin embargo, en el caso en que los objetos estén formados por una cantidad elevada de polígonos, calcular las intersecciones puede ser una tarea muy costosa. Se busca utilizar teselación en hardware para ayudar a delimitar las secciones donde dichos modelos colisionan.

Cuando se detecta una colisión, el sistema de física debe reaccionar de cierta manera

definida por el desarrollador, labor que se puede realizar tanto en software como en hardware. Históricamente, estos cálculos se han realizado en CPU debido a que se tiene una perspectiva global del estado del mundo. Trabajos recientes [5] han demostrado que se pueden realizar los cómputos de colisión dentro de la GPU, en la nueva etapa de propósito general *Compute Shader* (CS). Sin embargo, el desarrollo de estos cálculos en el CS excede el alcance de este proyecto, por lo que sólo se limitará a identificar por hardware el área de la intersección.

Para simplificar el trabajo, y debido a que adquirir los conocimientos necesarios para realizar cálculos con el compute shader podrían extender los plazos de este proyecto, se plantea sólo identificar, por medio de un código de colores, las primitivas susceptibles de estar colisionando. Si un polígono está en colisión con otro polígono de otro objeto, se lo coloreará de color rojo. Si está cerca de un polígono que está en colisión, se lo renderizará con tonos naranjas según cercanía al área de colisión. De lo contrario, se lo dibujará con un color azul que represente que está libre de colisión.

Métrica

Para identificar si dos objetos colisionan, se puede hacer un descarte rápido previo mediante el cálculo de intersección entre sus *bounding boxes*. La bounding box (BB) de un objeto es la mínima caja virtual que contiene a todos los puntos del objeto en su interior (Ver Fig. 6). Para que dos objetos colisionen, es necesario que sus bounding boxes se intersecten. Debido a que identificar si dos BB se intersectan es un cálculo simple, puede ayudar a descartar rápidamente los casos en que los objetos estén alejados. Si las BB se intersectan, se crea una nueva bounding box que represente el área de intersección. Este cálculo se realiza en CPU, y se transfiere la BB de intersección a la GPU para poder proseguir con los cálculos.

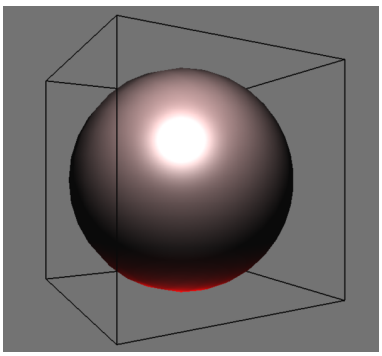

Figura 6: Bounding Box de una esfera [6].

A continuación y ya en dominio de shaders en GPU, todos los patch de ambos objetos se prueban si están contenidos en la BB de intersección. Si están contenidos, es muy probable que formen parte del área de colisión. En este caso, se los subdivide en gran cantidad de polígonos (por ejemplo, se les define un nivel de teselación de 32 por lado). A continuación, se los renderiza con un color rojo para identificar que pertenecen al área posible de colisión. Calcular la intersección de las BB no es suficiente para asegurar que los objetos colisionen, ya que a pesar de que las BB se intersecten, puede no haber contacto entre polígonos de los dos objetos. En un paso posterior, no implementado en este proyecto, se deberán calcular las intersecciones entre los polígonos de color rojo de los dos objetos. Si existe alguna intersección, el motor de física deberá realizar las modificaciones pertinentes.

Por otra parte, si el patch no está contenido en la BB de intersección, se calcula la distancia de éste a la misma. Se define un nivel de teselación entre 1 y 24 por lado del patch, según la distancia en coordenadas de mundo, del centroide al lado mas cercano de la bounding box. Se aplica un color naranja para identificar estos patch, cuya tonalidad variará de acuerdo a la distancia. Los patch que estén muy alejados, y que por lo tanto tengan un nivel de teselación igual a 1, se los renderizará con un color azul para señalar que no son candidatos a colisión, ya que están muy alejados del área de intersección.

11. Modelos a utilizar

Para poder probar el desempeño de todas las técnicas desarrolladas, será necesario contar con modelos tridimensionales en donde aplicarlas. En esta sección se mencionan los modelos que se han preseleccionado para probar estas técnicas. Estos modelos serán utilizados en las etapas de desarrollo y pruebas. Una vez que el desarrollo haya culminado, se pretende evaluar todos los algoritmos con otros modelos y realizar pruebas más elaboradas para poder obtener mejores conclusiones.

Para las pruebas en superficies paramétricas, se utilizará el modelo de la tetera de Utah, el cual es un modelo libre y clásico con el que se prueban los métodos de computación gráfica. Este modelo está conformado por superficies de Bézier de tercer grado y se muestra en la Fig. 7.

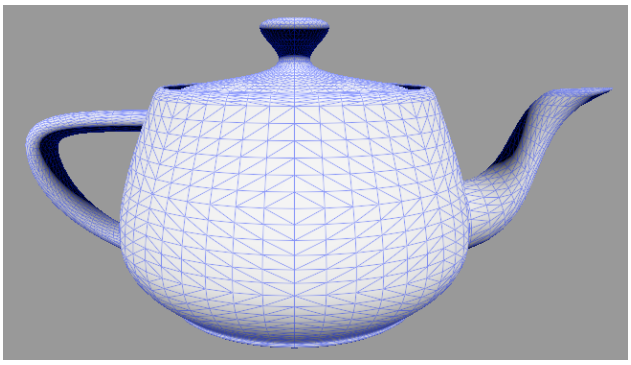

Figura 7: Modelo de la tetera de Utah.

Para las pruebas con modelos de personaje, se han seleccionado dos modelos. El primero, correspodiente a la representación de un perro, se ha elegido debido a que presenta muchas superficies curvadas. El segundo modelo representa a un personaje virtual de la película Alien (1979), dirigida por Ridley Scott. Este modelo se utilizará debido a que tiene muchos contornos con forma de punta, lo que marca una diferencia con respecto al modelo del perro. Ambos objetos están formados por gran cantidad de triángulos (Ver Fig. 8). Estos modelos se han obtenido gratuitamente del sitio web TF3DM [7].

En el caso de las pruebas con terrenos, se utilizará un terreno generado algorítmicamente (Ver Fig. 9). En particular, se construirá una malla en el plano y a continuación sus vértices se moverán afectados por el mapa de desplazamiento. Para generar el mapa de desplazamiento, se utilizará un algoritmo que a partir de ruido Perlin, cree la imagen de desplazamientos.

Para la técnica de intersección de objetos, se utilizarán los modelos ya detallados. Por

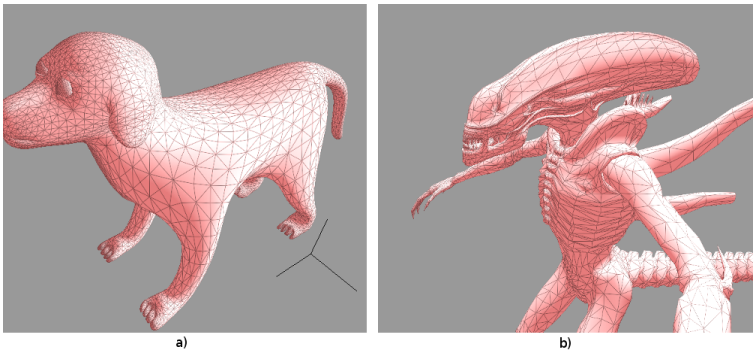

Figura 8: Modelos de personaje elegidos. a) Perro. b) Alien.

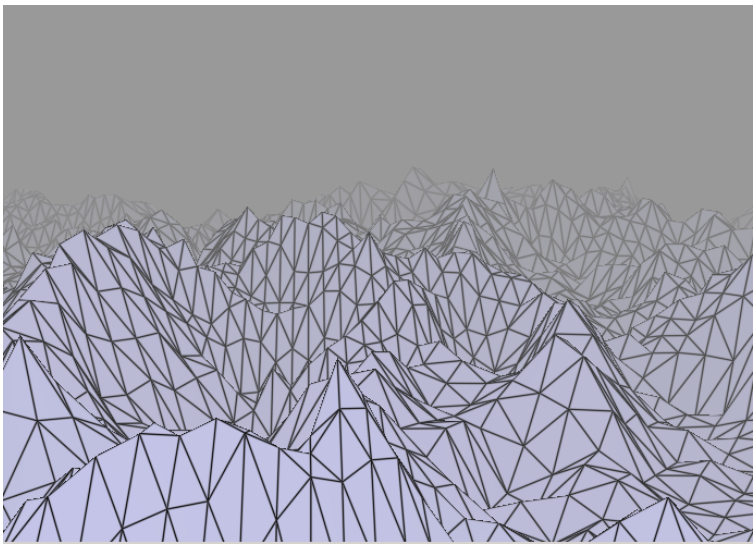

Figura 9: Terreno generado para probar los algoritmos.

otra parte, para la visualización de campos vectoriales, se escribirán los algoritmos para generar tanto la malla como la imagen de ruido.

12. Conclusión

Como se puede apreciar, las aplicaciones elegidas son diferentes entre sí y cubren un gran espectro de las capacidades de teselación de las placas aceleradoras de video de la actualidad. Si bien el desarrollo se realizará utilizando la API OpenGL 4.0, cualquiera de estos ejemplos se puede convertir a DirectX 11, ya que sus shaders de teselación poseen las mismas características. En todos los ejemplos, se busca hacer hincapié en las características de teselación del hardware, por lo que se crearán aplicaciones simples que sirvan principalmente para demostración de las técnicas de teselación desarrolladas.

Por último, en la mayoría de los ejemplos elegidos, el trabajo realizado dará lugar a desarrollos más extensos en el área. Se deja planteado esto para futuros proyectos y actividades realizadas por el tesista, o por cualquier persona interesada en la temática.

Bibliografía

[1] Salomon, D., *"Curves and Surfaces for Computer Graphics"*, Springer Editorial, 2006.

[2] Boubekeur, T.. *"A View-Dependent Adaptivity Metric for Real Time Mesh Tessellation."* Image Processing (ICIP), 2010 17th IEEE International Conference on. IEEE, 2010.

[3] Boubekeur, T. y Alexa, M., *"Phong tessellation."*, ACM Transactions on Graphics (TOG). Vol. 27. No. 5. ACM, 2008.

[4] 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.

[5] Nießner, M., Siegl, C., Schäfer, H. y Loop, C., *"Real-time Collision Detection for Dynamic Hardware Tessellated Objects"*, Eurographics, 2013.

[6] Wikibooks.org, *"Bounding Box"*. Disponible en `http://en.wikibooks.org/wiki/OpenGL_Programming/Bounding_box`. [Consultado el 8 de Noviembre de 2013].

[7] TF3DM.com, *"3d Models for Free"*. Disponible en `http://tf3dm.com/`. [Consultado el 20 de Octubre de 2013].

REALIZADO POR	FECHA	FIRMA
Fernando Nellmeldin	11/11/2013	
REVISADO POR	**FECHA**	**FIRMA**
Dr. Néstor Calvo	18/11/2013	
APROBADO POR	**FECHA**	**FIRMA**
Gastón Martin		

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

Periodo del Informe: Septiembre-Noviembre de 2013

Alcance: Etapas 3 y 4 (parciales)

Estado del proyecto:

Cronograma

Etapa 3: Análisis, selección y diseño de técnicas y modelos

Actividad	Fecha realización	
	Estimada	Real
3.1 Análisis general de técnicas	20 horas	15 horas
3.2 Selección de técnicas a desarrollar	10 horas	10 horas
3.3. Análisis de cada una de las técnicas	Por iteración: Primera: 10 hs Segunda: 10 hs	Por iteración: Primera: 10 hs. Segunda: 10 hs
3.4 Diseño de cada una de las técnicas	Por iteración: Primera: 7 hs. Segunda: 7 hs.	Por iteración: Primera: 7 hs. Segunda: 8 hs.
3.5 Redacción del segundo informe	15 horas	25 horas

Resultados obtenidos

A partir de los resultados de la investigación en las etapas anteriores, se analizaron y seleccionaron técnicas de teselación que permitan un estudio completo y abarcativo de la tecnología.

Etapa 4: Desarrollo de técnicas

Actividad	Fecha realización	
	Estimada	Real
4.1 Desarrollo de cada una de las técnicas	Por iteración: Primera: 30 hs. Segunda: 30 hs.	Por iteración: Primera: 70 hs. Segunda: 40 hs.
4.2 Pruebas de funcionalidad unitarias de cada técnica	Por iteración: Primera: 10 hs. Segunda: 10 hs.	Por iteración: Primera: 15 hs. Segunda: 10 hs.

Resultados obtenidos

Se comenzó con el desarrollo de las técnicas de teselación seleccionadas. Si bien hubo dificultades, se lograron resultados aceptables al aplicar las técnicas a modelos de prueba.

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		

Riesgos futuros	Probabilidad	Impacto
Funcionamiento de bibliotecas de software que no sea el esperado	Media (50%)	Medio

Notas

Se tuvieron dificultades en la segunda iteración correspodiente al desarrollo de las técnicas. En particular, surgieron muchos problemas al intentar utilizar una técnica de teselación aplicada a un modelo de un terreno formado por cuadriláteros en el espacio. Las dificultades surgieron debido a que para poder generar el terreno, se debían utilizar ciertas funcionalidades de OpenGL con las que no se tenía experiencia. A partir de investigación, prueba y error, se logró el objetivo, pero tomó más del doble del tiempo planificado.