Asociación Argentina
de Mecánica Computacional

Mecánica Computacional Vol XXXI, págs. 3111-3126 (artículo completo)
Alberto Cardona, Paul H. Kohan, Ricardo D. Quinteros, Mario A. Storti (Eds.)
Salta, Argentina, 13-16 Noviembre 2012

VISUALIZACION DE TERRENOS EN TIEMPO REAL PARA SIMULADORES DE VUELO

Pablo Sebastián Rojas Fredini, Marina Hebe Murillo y Alejandro César Limache

International Center of Computational Methods in Engineering (CIMEC) INTEC-CONICET. Santa Fe, Argentina., http://www.cimec.gov.ar/

Palabras Clave: Terrenos, CLOD, Teselación, GPU.

Resumen. Uno de los mayores desafíos para la implementación de simuladores de vuelo es la generación de un sistema de visualización de terrenos que permita representar de manera realista la superficie del planeta. Fundamentalmente se pueden observar dos dificultades: el manejo de mallas de millones de vértices y el almacenamiento de texturas de imágenes aéreas para brindar mayor realismo. En el presente trabajo se desarrolla un sistema de visualización de terrenos a gran escala íntegramente en hardware de video explotando las ventajas de los GPU de última generación. Para ello se proponen algoritmos para manejar el nivel de detalle de la superficie y la generación de texturas a partir de diccionarios utilizando máscaras que permiten optimizar el espacio requerido para su almacenamiento. Se dan detalles de la implementación y se muestran algunos resultados obtenidos con la presente técnica.

1. INTRODUCCION

La visualización de terrenos en tiempo real es un campo de gran interés para distintas ramas de la ingeniería y el arte. El problema consiste en la visualización de terrenos de gran tamaño para usarlos en entornos virtuales como simuladores y videojuegos. Para la representación gráfica se usan una mallas de miles e incluso millones de triángulos las cuales son deformadas para obtener la geometría deseada.

En particular en los simuladores de vuelo los terrenos deben ser representaciones fiables de locaciones reales. Es por ello que para la construcción de los mismos se deben utilizar parámetros e información relevada mediante fotografías aéreas, datos satelitales, etc. La elección de éstas fuentes de datos impactarán directamente sobre los algoritmos y técnicas empleadas a la hora de visualizar los terrenos.

Para poder representar un terreno de forma realista se necesitan conocer al menos dos cosas. Por un lado se necesita información de la forma del terreno (si hay valles, montañas, llanuras, etc.) y por otro se necesita información sobre la cobertura del mismo, es decir si es un desierto, una ciudad, un bosque, etc. Por lo tanto el problema de visualización de un terreno puede ser descompuesto en dos partes: un problema geométrico y un problema de procesamiento de imágenes.

El problema geométrico radica en cómo representar mallas de muchos triángulos de manera eficiente. En un simulador de vuelo el terreno es un elemento importante de la visualización pero no es el único, por lo que debemos utilizar algoritmos que dejen potencia de cálculo para dibujar el resto de los elementos y ejecutar los cálculos físicos. Debido a la necesidad de dibujar mallas de grandes extensiones con gran cantidad de triángulos el problema de dibujar terrenos es netamente un problema de nivel de detalle de mallas. Históricamente la mayoría de algoritmos busca implementar algortimos que refinen adaptativamente la malla a medida que la simulación transcurre, de manera de conservar una representación aceptable cerca de la cámara y disminuir la cantidad de triángulos lejos o fuera de la misma. Estos algoritmos pueden clasificarse en algoritmos de nivel de detalle discretos (Discrete Level of Detail -DLOD-) o algoritmos de nivel de detalle continuo (Continuos Level of Detail -CLOD-). La diferencia básica radica en que en los algoritmos de DLOD los saltos entre los niveles de detalle suelen ser discretos mientra que en los algoritmos de CLOD la variación del nivel de detalle es continua. Los algoritmos clásicos más populares están basados en el algoritmo ROAM (Duchaineau et al., 1997; Hwa et al., 2005) o en triangulaciones restringidas (Lindstrom et al., 1996; Pajarola, 1998; Lindstrom y Pascucci, 2002; Pajarola y Gobbetti, 2007). Ambos son algoritmos de CLOD. ROAM propone explotar la coherencia temporal de la malla suponiendo que de un frame a otro son pocos los cambios en la misma y genera incrementalmente mallas que tienden a una óptima. Por otro lado los algoritmos basados en quadtrees restringidos (Restricted Quad Trees -RQT-) suelen regenerar en cada paso de tiempo la malla, sin explotar la coherencia temporal. Ambos algoritmos tienen sus ventajas y desventajas y han sido utilizados con éxito en diferentes escenarios. El mayor problema con que se han encontrado dichos algoritmos es que han sido diseñados para ejecutarse en CPU y no explotan las capacidades de paralelismo de los GPU de última generación. En respuesta a este inconveniente han surgido nuevos algoritmos basados en GPU tratando de explotar su potencia de cálculo. Uno de los más populares es Geometry Clipmaps (Hoppe, 2004). Siguiendo esta linea de investigación en el presente trabajo se presenta un algoritmo de visualización de terrenos que se ejecuta íntegramente en GPU. El mismo explota las características introducidas en la última generación de placas de video y Direct3D 11 (Microsoft) conocida como teselación por hardware (Hardware Tessellation).

Figura 1: Nuevas etapas del pipeline

Por otro lado, como se dijo anteriormente, las mallas deben ser sombreadas para lograr una representación realista de los terrenos. La forma más realista de sombrearlas seguramente sería utilizar fotografías satelitales del área en cuestión. Pero esto presenta varios inconvenientes. En primer lugar las fotografías satelitales ocupan cuantiosas cantidades de memoria debido a su tamaño -siempre y cuando se desee una buena resolución-. Por otro lado, es difícil conseguir de fuentes públicas fotografías satelitales con buena resolución. Además muchas veces estas fotografías de dominio público exigen un alto nivel de preprocesado antes de ser incluidas en el simulador (corrección de bordes, relleno de información faltante, etc). En este trabajo se usan las ideas introducidas en (Szofran, 2006) para sombrear los triángulos de manera realista basándose en información satelital y utilizando un diccionario de imágenes que son combinadas mediante máscaras.

2. PROBLEMA GEOMÉTRICO

El objetivo del presente trabajo fue desarrollar un algoritmo para visualización de terrenos que se ejecute exclusivamente en GPU . Para ello se decidió utilizar las características de última generación del hardware de video. En particular se desarrolló sobre las capacidades de teselación de Direct3D 11. En el algoritmo desarrollado el CPU no genera una malla sino que genera una serie de patches o superficies las cuales serán teseladas en el GPU creando la malla íntegramente en el hardware de video. Al ser teselada en la placa de video se logra descargar la carga del CPU liberándolo para tareas de simulación. Además es posible controlar la teselación en tiempo de ejecución lo que da como resultado un CLOD por construcción.

2.1. Teselación en Direct3D 11

En Direct3D11 se introdujo la posibilidad de controlar cómo se tesela una superficie en el hardware de vídeo. Para ello se introdujeron tres estapas más al pipeline, dos de ellas programables. Las mismas se ubican entre la etapa de Vertex Shader y la de Geometry Shader. Las etapas se denominan Hull Shader, Domain Shader y Tessellator. En la Fig.1 se muestra un diagrama parcial del nuevo pipeline. En negrita se pueden observar las tres etapas nuevas, de las cuales Tessellator no es programable.

2.1.1. Hull Shader

El objetivo de esta etapa es calcular cuál será el factor de teselación para cada patch. Es decir en cuántos triángulos se debe dividir cada patch. Este cálculo es de suma importancia si se desea obtener una teselación buena. Se recibe como argumento un patch de terreno ya procesado en el Vertex Shader y el shader debe devolver el factor de teselación para el patch y los puntos de control del patch. Los factores de teselación son enviados al teselador mientras que las coordenadas de los puntos de control se envian directametne al Domain Shader. La Fig.2 muestra este comportamiento.

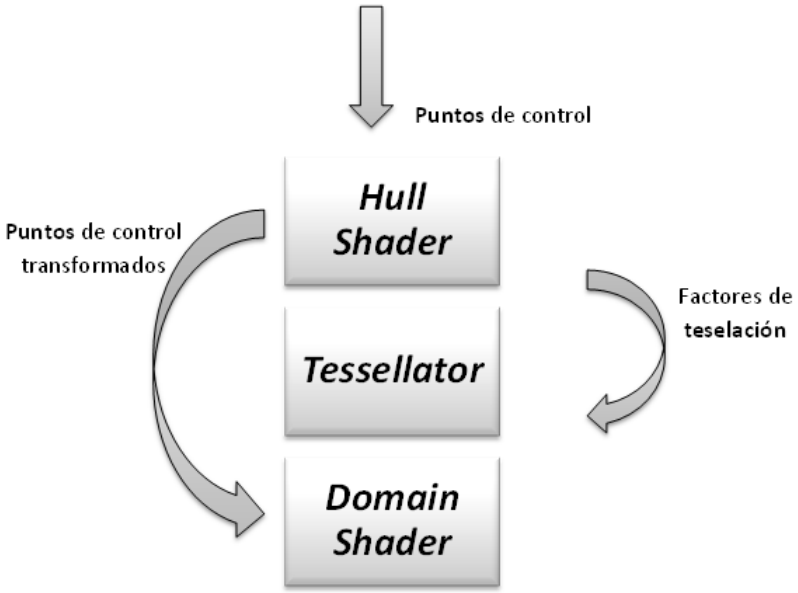


Figura 2: Entradas y salidas del Hull Shader

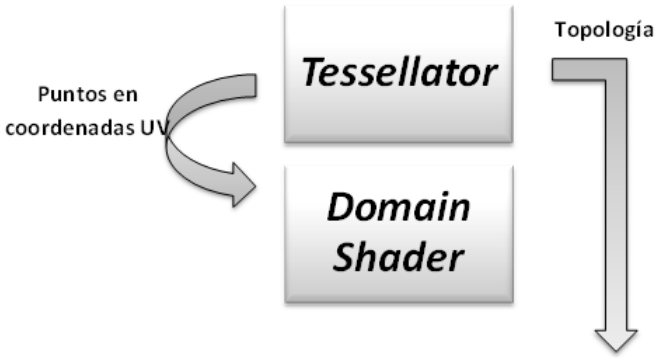


Figura 3: Salidas del Tessellator

2.1.2. Tessellation

Esta etapa no es programable y se encarga de llevar adelante la teselación de los patches utilizando el factor calculado en el Hull Shader. Esta etapa produce como salida la topología y los puntos en coordenadas baricéntricas sobre el patch. La Fig. 3

2.1.3. Domain Shader

Una vez terminada la etapa de teselación debemos calcular las posiciones y propiedades de los nuevos vértices generados. En el caso más simple se debe interpolar la posición del vértice de acuerdo a las coordenadas de los puntos de control del patch que le dio origen.

2.2. Algoritmo

El método propuesto utiliza mapas de altura para almacenar la información de elevación de la geometría. El mapa de altura se almacena como una figura en escala de grises normalizada junto a información de su altura máxima, mínima, y coordenadas WGS84(Szofran, 2006).Esta imagen será utilizada para desplazar los vértices y determinar uno de los multiplicadores del

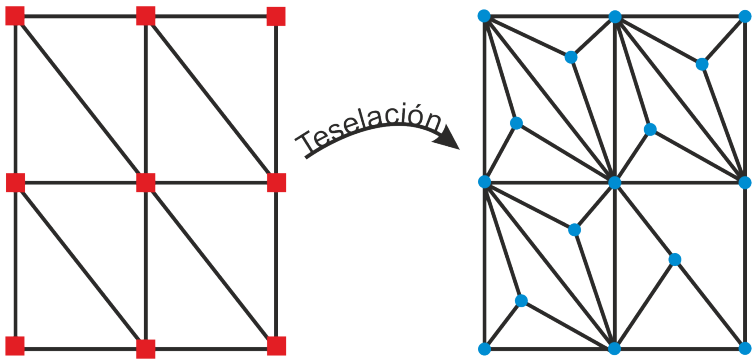


Figura 4: Patches y su posterior teselación

factor de teselación. En líneas generales el algoritmo parte de una malla de patches base, un mapa de altura y un mapa de densidad. En primer lugar se teselan los patches usando información de la densidad y de la cámara. Luego se desplazan en altura los vértices usando el mapa de altura y finalmente se transforma el bloque de terreno usando el sistema WGS84 para darle la curvatura y posición correcta en el espacio.

2.2.1. Malla base

El algoritmo en una etapa de preprocesamiento genera una serie de patches de base con sus puntos de control. Estos patches son triangulares y conforman entre todos un rectángulo proporcional al área de terreno a representar. Estos patches serán luego teselados en GPU para generar la malla del terreno. La cantidad de patches pondrán luego un límite al refinamiento.

La Fig.4 muestra un ejemplo de malla de base y su teselación. A la izquierda se puede observar el patch de base con los puntos de control mientras que a la derecha se dibuja la malla originada por dicho patch.

2.2.2. Mapa de densidad

Para teselar se aplican dos criterios. En primer criterio es estático y depende de la complejidad del terreno. La idea es refinar dónde existan cambios bruscos de pendiente, como en zona de montañas. Por otro lado la zona con pocas variaciones debería ser representada con la menor cantidad de triángulos posibles. Para ello en una etapa de preprocesamiento es necesario extraer esta información del mapa de altura. Para ello se genera un mapa de densidad. El mismo es una textura que contendrá valores altos en las zonas con pendiente alta. Para ello se calcula el valor de densidad para el píxel (i, j) de la siguiente manera:

$$D_{ij} = \alpha_0 P_{\Delta 0} + \alpha_1 P_{\Delta 1} + \alpha_2 P_{\Delta 2} + \alpha_3 P_{\Delta 3} \tag{1}$$

con

$$\begin{aligned} P_{\Delta 0} &= |(P_{i,j+1} - P_{i,j}) - (P_{i,j} - P_{i,j-1})| \\ P_{\Delta 1} &= |(P_{i+1,j} - P_{i,j}) - (P_{i,j} - P_{i-1,j})| \\ P_{\Delta 2} &= |(P_{i+1,j+1} - P_{i,j}) - (P_{i,j} - P_{i-1,j-1})| \\ P_{\Delta 3} &= |(P_{i+1,j-1} - P_{i,j}) - (P_{i,j} - P_{i-1,j+1})| \end{aligned} \tag{2}$$

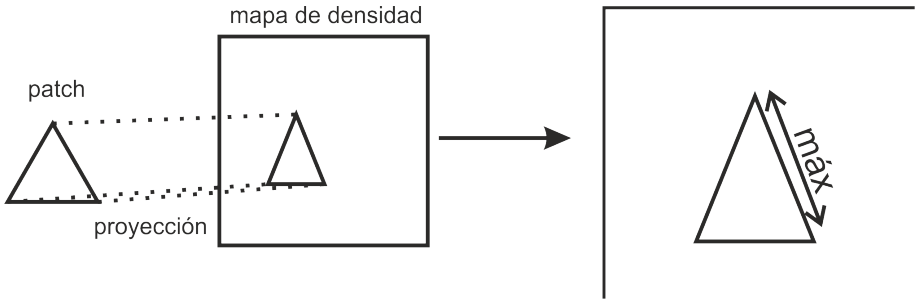


Figura 5: Obtención del factor de teselación

y

$$\alpha_0 = \begin{cases} 0 & \text{si } P_{\Delta 0} \leq tol \\ 0{,}293 & \text{si } P_{\Delta 0} > tol \end{cases} \tag{3}$$

$$\alpha_1 = \begin{cases} 0 & \text{si } P_{\Delta 1} \leq tol \\ 0{,}293 & \text{si } P_{\Delta 1} > tol \end{cases} \tag{4}$$

$$\alpha_2 = \begin{cases} 0 & \text{si } P_{\Delta 2} \leq tol \\ 0{,}207 & \text{si } P_{\Delta 2} > tol \end{cases} \tag{5}$$

$$\alpha_3 = \begin{cases} 0 & \text{si } P_{\Delta 3} \leq tol \\ 0{,}207 & \text{si } P_{\Delta 3} > tol \end{cases} \tag{6}$$

donde tol es un valor de tolerancia ajustable que mide la velocidad del cambio de altura. Cabe aclarar que la resolución del mapa de densidad es la misma que la del mapa de altura.

Luego es necesario determinar cómo será aplicada la información de densidad sobre los patches. La misma puede aplicarse sobre los patches triangulares o sobre las aristas (en el primer caso la densidad será constante en toda la cara). Si la densidad es constante sobre la cara, puede suceder (y de hechó lo hará) que dos patches que comparten una arista posean distintos factores de teselación, lo cual llevará a que de un lado existan más vértices que del otro. Esto provocará discontinuidades en la malla, las cuales son inaceptables. Para evitar esto la información de densidad se aplica sobre las aristas, de manera que dos patches que comparten una arista sean teselados de manera congruente.

Para determinar el valor de densidad de una arista se utiliza el criterio del máximo valor. Es decir se examinan los valores de densidad que atraviesa la arista proyectándola sobre el mapa de densidad y se escoge el mayor. En la Fig.5 se ilustra el proceso. Esta información se almacena en un buffer de densidad el cuál posee un vector de 4 elementos por cada patch de la malla. El cuarto valor corresponde a la densidad en el centro del patch y se calcula como el máximo de los valores de las aristas. Este buffer será el que finalmente se introduzca en el pipeline de Direct3D 11. De ahora en adelante lo llamaremos *buffer de densidad*.

2.2.3. Teselación Adaptativa

Además de la densidad como criterio de teselación también se utiliza la distancia a la cámara. Por cada vértice se realiza el cálculo del factor de teselación adaptativo.

$$F_d = 1 - clamp(\frac{d - d_{min}}{d_{max} - d_{min}}, 0, 1) \tag{7}$$

donde d es la distancia del vértice a la cámara, d_{max} y d_{min} son los límites de teselación mínima y máxima relativa a la cámara y $clamp$ es una función que acota el valor de su primer argumento entre los valores provistos como tercer y cuarto argumento.

2.2.4. Pegando todo junto

Una vez calculados los factores que afectarán la teselación se procede a calcular el valor final. Dado un patch p cuyos vértices poseen las coordenadas p_0, p_1 y p_2, y el factor de teselación adaptativa de cada vértice es F_{d0},F_{d1} y F_{d2} respectivamente:

$$T = T_d \odot T_e \odot T_g; \tag{8}$$

donde $\odot$ indica el producto componente a componente. T_d es el factor de teselación adaptativo:

$$\begin{aligned} T_{d1} &= \frac{1}{2}(F_{d1} + F_{d2}) \\ T_{d2} &= \frac{1}{2}(F_{d2} + F_{d0}) \\ T_{d3} &= \frac{1}{2}(F_{d0} + F_{d1}) \\ T_{d4} &= T_{d1} \end{aligned} \tag{9}$$

T_e es el factor de teselación estático dependiente del mapa de altura, es decir es el valor leído del *buffer de densidad* y T_g es un factor de teselación global controlado por el usuario. Variando este último parametro es posible controlar el refinamiento general del terreno.

Finalmente en T es el factor de teselación a introducir en el pipeline de teselación (los tres primeros valores corresponden a las aristas del patch y el último al centro del mismo).

2.2.5. Culling

Durante la etapa de teselación es posible que se gaste potencia de cómputo teselando patches que no se encuentran dentro del frustum de la cámara. Para evitar que esto suceda, se lleva a cabo una etapa de culling para determinar si el patch debe ser teselado o no. Para ello se utilizan los planos que conforman el frustum de la cámara. Luego por cada patch antes de devolver el valor de teselación se calcula si el mismo no se encuentra fuera de los planos, en este caso el valor de teselación será 0, indicando que no se debe generar geometría para el mismo.

2.3. Implementación en GPU

La implementación del algoritmo descripto se distribuye en las distintas etapas del pipeline programable. A continuación se explica qué tarea se realiza en cada etapa.

2.3.1. Preprocesamiento

En esta etapa se debe calcular el buffer de densidad descripto anteriormente y generar el patch de base. Esto se realiza al arrancar el programa.

2.3.2. VertexShader

En el vertex shader se recibe por cada vértice su posición y sus coordenadas de textura:

```
struct VS_INPUT
{
    float3 inPositionOS   : POSITION;
    float2 inTexCoord     : TEXCOORD0;
    uint   uVertexID      : SV_VERTEXID;
};
```

Y genera como salida la posición en el mundo del vértice, con su coordenada de textura y la distancia a la cámara.

```
struct VS_OUTPUT_HS_INPUT
{
  float3 vWorldPos : WORLDPOS;
  float2 texCoord  : TEXCOORD0;
  float fVertexDistanceFactor : VERTEXDISTANCEFACTOR;
};
```

Las coordenadas de textura se pasan sin modificarlas a la siguiente etapa del pipeline. Para calcular la distancia la cámara se tiene en cuenta la proyección geodésica del vértice.

```
VS_OUTPUT_HS_INPUT VS( VS_INPUT i )
{
  VS_OUTPUT_HS_INPUT Out;

  // Se calcula la posicion en el mundo
  float4 vPositionWS = mul( i.inPositionOS.xyz, g_mWorld );

  // Propagamos coordenadas de textura
  Out.texCoord = i.inTexCoord ;

  //Desplazamos el patch usando el mapa de normales
  float4 vNormalHeight = g_nmhTexture.SampleLevel( g_samLinear, i.inTexCoord, 0);
  vNormalHeight=(mMaxH-mMinH)*vNormalHeight+mMinH;
  vPositionWS.y=vNormalHeight;

  //Para la teselacion adaptativa
  //Valores hardcodeados de ejemplo
  const float fMinDistance = 1.0f;
  const float fMaxDistance = 6000.0f;

  //Se calcula la distancia del vertice a la camara y un factor de distancia a partir de la misma
  float3 w=abs(latlon2xyz(vPositionWS.xzy).xyz  - float3(mMinX,0.0f,mMinZ));
  w.x*=-1;
  w.xyz=w.xyz/1;
  float fDistance = distance( w , g_vEye.xyz )-1000;
  Out.fVertexDistanceFactor = 1.0 - clamp( ( ( fDistance - fMinDistance ) / ( fMaxDistance - fMinDistance ) ),
                                           0.0, 0.99);
  // Escribimos la posicion en coordenadas del mundo
  Out.vWorldPos = float3( vPositionWS.xyz );

  return Out;
}
```

La función *latlon2xyz* convierte de latitud y longitud a coordenadas cartesianas usando la proyección geodésica descripta anteriormente. Aquí se puede observar cómo se implementó la ecuación (7).

2.3.3. Hull Shader

El hull shader se compone de dos rutinas. La primera simplemente copia los datos del patch de entrada al patch de salida sin modificarlo.

```
[domain("tri")]
[partitioning("fractional_odd")]
[outputtopology("triangle_cw")]
[outputcontrolpoints(3)]
[patchconstantfunc("ConstantsHS")]
[maxtessfactor(64.0)]
HS_CONTROL_POINT_OUTPUT HS( InputPatch<VS_OUTPUT_HS_INPUT, 3> inputPatch, uint uCPID : ↩
    SV_OutputControlPointID )
{
    HS_CONTROL_POINT_OUTPUT    output = (HS_CONTROL_POINT_OUTPUT)0;

    //Copiamos entradas a salidas.
    output.vWorldPos = inputPatch[uCPID].vWorldPos.xyz;
    output.texCoord =  inputPatch[uCPID].texCoord;

    return output;
}
```

La estructura de entrada del mismo utiliza la salida del VerteShader descripta anteriormente. Y genera como salida la estructura

```
struct HS_CONTROL_POINT_OUTPUT
{
  float3 vWorldPos : WORLDPOS;
  float2 texCoord  : TEXCOORD0;
};
```

Mientras que la segunda, es la que efectivamente dado un patch determina el valor de teselación del mismo. Esta rutina implementa las ecuaciones (8) y (9).

```
HS_CONSTANT_DATA_OUTPUT ConstantsHS( InputPatch<VS_OUTPUT_HS_INPUT, 3> p, uint PatchID : ↩
    SV_PrimitiveID )
{
  HS_CONSTANT_DATA_OUTPUT output  = (HS_CONSTANT_DATA_OUTPUT)0;

  float4 vEdgeTessellationFactors;

  //Teselación según distancia
  // Sólo se computa el promedio entre los dos vértices que unen el edge
  vEdgeTessellationFactors.x = 0.5 * ( p[1].fVertexDistanceFactor + p[2].↩
      fVertexDistanceFactor );
  vEdgeTessellationFactors.y = 0.5 * ( p[2].fVertexDistanceFactor + p[0].↩
      fVertexDistanceFactor );
  vEdgeTessellationFactors.z = 0.5 * ( p[0].fVertexDistanceFactor + p[1].↩
      fVertexDistanceFactor );
  vEdgeTessellationFactors.w = vEdgeTessellationFactors.x;

  //Lo multiplicamos por el factor global
  vEdgeTessellationFactors *= g_vTessellationFactor.xxxy;


  //Teselación según densidad
  //Leemos el buffer de densidad
  vEdgeTessellationFactors *= g_DensityBuffer.Load( PatchID ).yzxw;

  // Asignamos los valores finales de teselación
  output.Edges[0] = vEdgeTessellationFactors.x;
  output.Edges[1] = vEdgeTessellationFactors.y;
  output.Edges[2] = vEdgeTessellationFactors.z;
```

```
  output.Inside   = vEdgeTessellationFactors.w;

  //Realizar culling
  //se omite para claridad del código
  return output;
}
```

La salida es el factor de teselación de cada arista y del centro del patch:

```
struct HS_CONSTANT_DATA_OUTPUT
{
  float    Edges[3]         : SV_TessFactor;
  float    Inside           : SV_InsideTessFactor;
};
```

2.4. Domain Shader

En esta etapa debemos configurar la geometría generada usando información del patch que le dio origen y del mapa de altura. Para ello se utilizan las coordenadas baricéntricas de la geometría generada. La salida de esta etapa es la posición y las coordenadas de textura del triángulo.

```
struct DS_OUTPUT
{
  float2 texCoord          : TEXCOORD0;
  float4 vPosition         : SV_POSITION;
};
\end{lstlistings}

\begin{lstlisting}
[domain("tri")]
DS_OUTPUT DS( HS_CONSTANT_DATA_OUTPUT input, float3 BarycentricCoordinates : ↩
    SV_DomainLocation,
            const OutputPatch<HS_CONTROL_POINT_OUTPUT, 3> TrianglePatch )
{
  DS_OUTPUT output = (DS_OUTPUT)0;

  Interpolamos la posicion en el mundo usando coordenadas baricentricas
  float3 vWorldPos = BarycentricCoordinates.x * TrianglePatch[0].vWorldPos +
                     BarycentricCoordinates.y * TrianglePatch[1].vWorldPos +
                     BarycentricCoordinates.z * TrianglePatch[2].vWorldPos;

  //Interpolamos los otros campos usando coordenadas baricentricas
  output.texCoord = BarycentricCoordinates.x * TrianglePatch[0].texCoord +
                    BarycentricCoordinates.y * TrianglePatch[1].texCoord +
                    BarycentricCoordinates.z * TrianglePatch[2].texCoord;

  //Sampleamos el mapa de altura
  float4 vNormalHeight = g_nmhTexture.SampleLevel( g_samLinear, output.texCoord, 0);

  //Se calcula el desplazamiento horizontal del vertice
  vNormalHeight=(mMaxH-mMinH)*vNormalHeight+mMinH;
  vWorldPos.y = ( vNormalHeight ) ;

  //Se proyecta usando WGS84 y se corre al origen
  float3 w=abs(latlon2xyz(vWorldPos.xzy).xyz  - float3(mMinX,0.0f,mMinZ));
  w.x*=-1;
  w.xyz=w.xyz/1 ;
  vWorldPos.xyz = w;

  //Finalmente se transforma el vertice usando la worldviewprojection matrix
  output.vPosition= mul(float4(vWorldPos.xyz,1.0), g_mView);
  output.vPosition = mul( output.vPosition, g_mProjection );

  return output;
```

```
}
```

De esta forma se ha generado la nueva geometría y se ha posicionado correctamente en el mundo. Debe notarse que el desplazamiento según el mapa de altura se realiza en esta etapa recién. Si se hubiera hecho antes no se hubiera ganado resolución en la malla, sino que sólo más triángulos.

3. COBERTURA

Cómo se dijo anteriormente además del problema geométrico de generar la malla es necesario abordar el problema sobre cómo sombrearla. Utilizar fotos satelitales es problemático en varios sentidos:

- Poseen un gran tamaño, requiriendo mucho espacio de almacenamiento si se desea una resolución buena.
- Muchas bases de datos no son de acceso público y las que lo son suelen necesitar una gran cantidad de postprocesamiento para adecuarlas al simulador. Muchas veces incluso ni siquiera las fotografías de distintas áreas se unen correctamente.

Para tratar de sobrellevar estas dificultades se utilizó el mismo enfoque que en (Szofran, 2006) utilizando la técnica conocida como *texture splatting*.

3.1. Diccionario

En principio se define un diccionario con 8 posibles tipos de cobertura como por ejemplo desierto, bosque, jungla, etc. En la Fig.6 se muestran dos imágenes del diccionario. Estas texturas no deben ser necesariamente de alta resolucíón ya que se las suele repetir varias veces sobre el terreno.

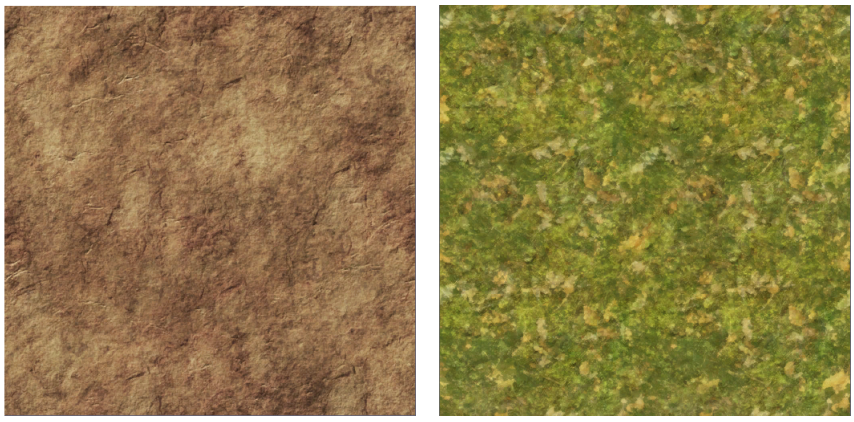


Figura 6: Diccionario

3.2. Máscaras

Luego es necesario definir por cada bloque de terreno cómo aplicar y cuáles. Para ello se utilizan máscaras, una máscara es una imagen que cubre todo el terreno y por cada pixel indica qué imagenes del diccionario deben ser blendeadas y en que porcentaje. Como existen 8 imágenes en el diccionario, entonces necesitaremos 8 valores de blend por cada pixel resultante. Estos

valores de blend o alfa pueden ser almacenados en texturas. Si usáramos una textura por cada canal de blend necesitaríamos 8 texturas por cada bloque de terreno lo cual atentaría contra el intento de optimizar el uso de memoria. Para evitar esto se recurre a comprimir los valores de alfa. Se utilizan dos texturas de 32 bits cada una, y se codifican 4 canales por textura, disponiendo de 256 valores de alfa para cada canal. Esto permite obtener transiciones suaves entre las diferentes texturas.

3.3. Generación de las máscaras

Para generar las máscaras se debe usar información satelital. Las mismas se pueden generar usando información de mapas de cobertura usando algún software GIS, o simplemente un poco de talento en algún software de edición de imágenes sino se desea representar una zona del mundo real. El desarrollo de una herramienta que permita pintar los terrenos de manera interactiva es una tarea que debería ser desarrollada a futuro para evitar el tedio del trabajo artesanal actual.

3.4. Implementación en GPU

El blending de las texturas se realiza también en GPU, especialmente en el PixelShader, ya que como se mostró anteriormente las otras etapas sólo transmiten las coordenadas de textura.

3.4.1. PixelShader

El pixel shader recibe como argumento la estructura:

```
struct PS_INPUT
{
   float2 texCoord          : TEXCOORD0;

};
```

y calcula el valor de salida de la siguiente manera:

```
float4 PS( PS_INPUT i ) : SV_TARGET
{
  float4 cResultColor0 = float4( 0, 0, 0, 1 );
  float4 cResultColor1 = float4( 0, 0, 0, 1 );
  float4 cResultColor = float4( 0, 0, 0, 1 );

  float4 cBase0 = float4( 0, 0, 0, 1 );
  float4 cBase1 = float4( 0, 0, 0, 1 );
  float4 cBase2 = float4( 0, 0, 0, 1 );
  float4 cBase3 = float4( 0, 0, 0, 1 );
  float4 cBase4 = float4( 0, 0, 0, 1 );
  float4 cBase5 = float4( 0, 0, 0, 1 );
  float4 cBase6 = float4( 0, 0, 0, 1 );
  float4 cBase7 = float4( 0, 0, 0, 1 );


  cBase0  = base0Texture.Sample(samLinear, i.texCoord* mTextureRepeatFactor);
  cBase1  = base1Texture.Sample(samLinear, i.texCoord* mTextureRepeatFactor);
  cBase2  = base2Texture.Sample(samLinear, i.texCoord* mTextureRepeatFactor);
  cBase3  = base3Texture.Sample(samLinear, i.texCoord* mTextureRepeatFactor);
  cBase4  = base4Texture.Sample(samLinear, i.texCoord* mTextureRepeatFactor);
  cBase5  = base5Texture.Sample(samLinear, i.texCoord* mTextureRepeatFactor);
  cBase6  = base6Texture.Sample(samLinear, i.texCoord* mTextureRepeatFactor);
  cBase7  = base7Texture.Sample(samLinear, i.texCoord* mTextureRepeatFactor);
```

```
  cResultColor0  = maskTexture0.Sample(samPoint, i.texCoord);
  cResultColor1  = maskTexture1.Sample(samPoint, i.texCoord);



  cResultColor= cResultColor0.x*cBase0; +
          cResultColor0.y*cBase1  +
          cResultColor0.z*cBase2  +
          cResultColor0.w*cBase3  +
          cResultColor1.x*cBase4; +
          cResultColor1.y*cBase5  +
          cResultColor1.z*cBase6  +
          cResultColor1.w*cBase7  ;


   return cResultColor;
}
```

donde base[i]Texture (i=0..8) y maskTexture[i] (i=1,2) son Texture2D, samLinear es un Sampler lineal y samPoint es un sampler puntual. Esta implementación dista de ser eficiente y se eligió por ser un poco más clara para la presentación.

4. EJEMPLO

En esta sección se mostrarán dos ejemplos para ilustrar el comportamiento del algoritmo desarrollado. En primer lugar se mostrará el funcionamiento del algoritmo adaptativo de teselación para un mapa de altura generado a partir de una función gaussiana. En la Fig.7 se puede observar el mapa de altura y su correspondiente mapa de densidades.

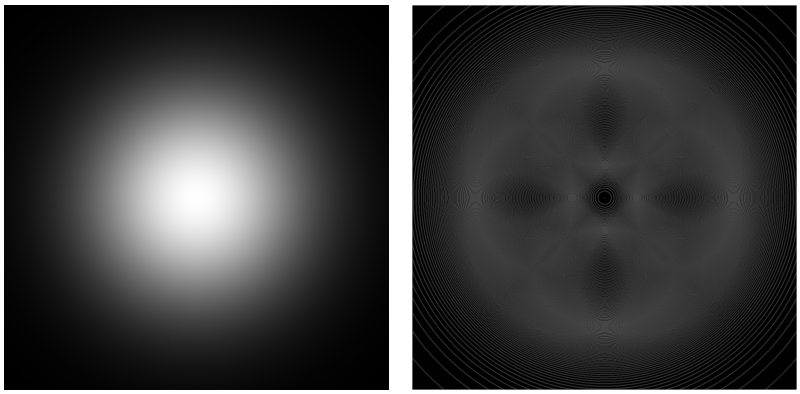
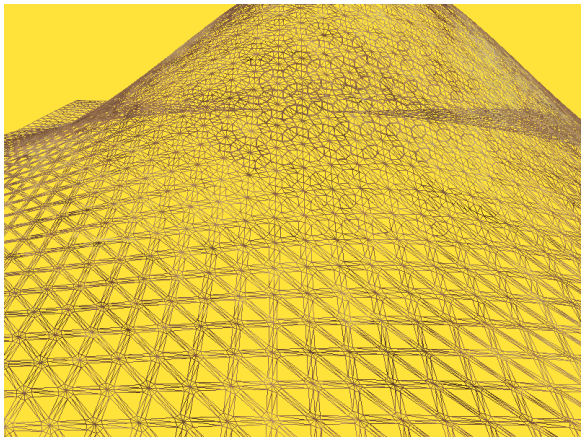


Figura 7: Mapa de altura y densidad gaussianos

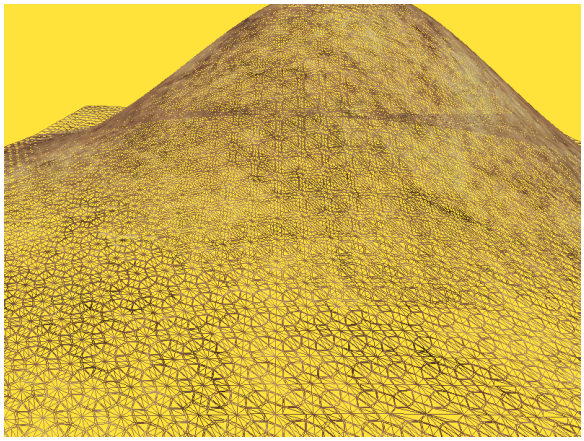
En las Figs.8(a) y 8(b) se puede observar el algoritmo en funcionamiento sobre dichos mapas. Allí no se muestran los efectos del factor de la distancia, sino que sólo se tesela en función del mapa de densidad. Se puede observar como el nivel de detalle varía continuamente a medida que se sale de la zona de curvatura grande. Lo mismo sucede con el centro de la función como se puede observar en la Fig.8(c) .

Por otro lado se probó el algoritmo con el mapa de altura de la Fig.9 obtenido de Reuter et al. (2007). También allí se puede observar el mapa de densidades generado por el algoritmo.

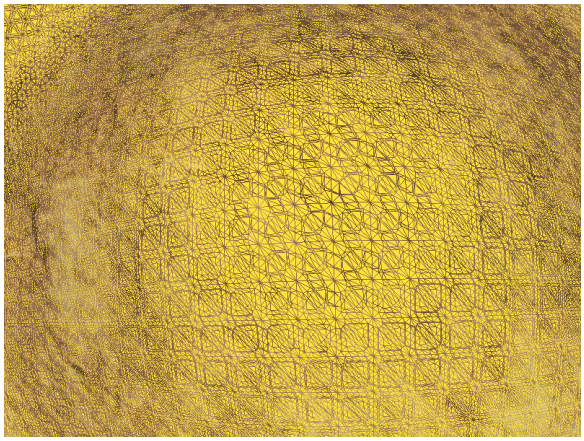
En la Fig.10(a) se puede observar una vista superior del terreno dónde se aprecia cómo varía continuamente el nivel de teselación para las zonas de mayor densidad. En este caso no interviene el factor adaptativo. En cambio en la Fig.10(b) se puede observar cómo varía el nivel de teselación dependiendo de la posición de la cámara. Al combinar ambos factores se pueden observar escenas como la de la Fig.10(c). Allí se puede apreciar cómo las zonas cercanas a la



(a) Vista lateral (factor de teselación bajo)



(b) Vista lateral (factor de teselación alto)



(c) Vista superior del centro de la gaussiana

Figura 8: Teselación de Gaussiana

cámara no se subdividen si el mapa de densidad no lo exige, también se puede apreciar cómo las zonas lejanas a la cámara tampoco se subdividen gracias al factor adaptativo de teselación.

5. TRABAJOS A FUTURO

Quedan aún mucho trabajo a realizar sobre el algoritmo presentado. Los puntos más importantes son:

- Optimizar el pipeline gráfico para un renderizado eficiente.
- Calcular métricas de error para comparar con otros métodos.
- Explorar alternativas para lograr terrenos continuos, pudiendo unir correctamente diferentes bloques de terreno.
- Mejorar la malla de base usando algún algoritmo en CPU que no sea costoso.
- Mejorar la compresión para el sombreado.

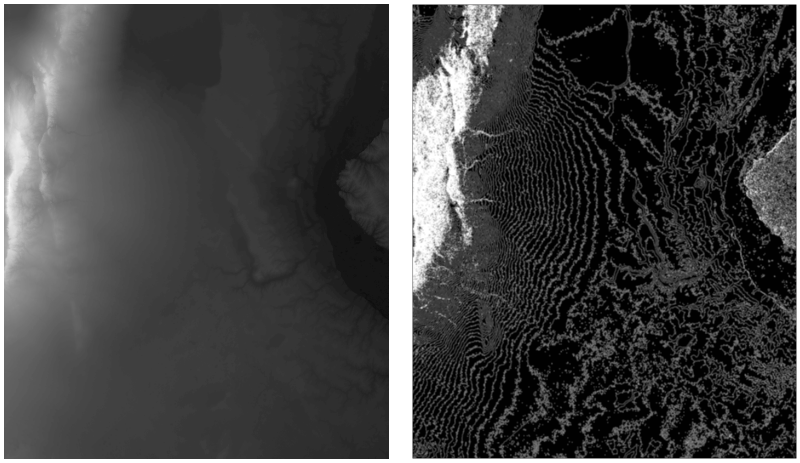


Figura 9: Mapa de altura y densidad de terreno

6. CONCLUSIONES

En el presente trabajo se introdujo un algoritmo de CLOD para visualización de terrenos que se ejecuta íntegramente en GPU. El mismo permite descargar del CPU el costo de generar y mantener la malla del terreno. Para ello se utilizaron las características del estado del arte de las placas gráficas mostrando flexibilidad y buenos resultados. Además se mostró el correcto funcionamiento utilizando una técnica de *texture splatting* para sombrearlo.

REFERENCIAS

Duchaineau M., Wolinsky M., Sigeti D.E., Miller M.C., Aldrich C., y Mineev-Weinstein M.B. Roaming terrain: real-time optimally adapting meshes. En *Proceedings of the 8th conference on Visualization '97*, VIS '97, páginas 81–88. IEEE Computer Society Press, Los Alamitos, CA, USA, 1997. ISBN 1-58113-011-2.

Hoppe F.L.H. Geometry clipmaps: Terrain rendering using nested regular grids. En *ACM Transactions on Graphics*. 2004.

Hwa L.M., Duchaineau M.A., y Joy K.I. Real-time optimal adaptation for planetary geometry and texture: 4-8 tile hierarchies. *IEEE Transactions on Visualization and Computer Graphics*, 11(4):355–368, 2005. ISSN 1077-2626. doi:10.1109/TVCG.2005.65.

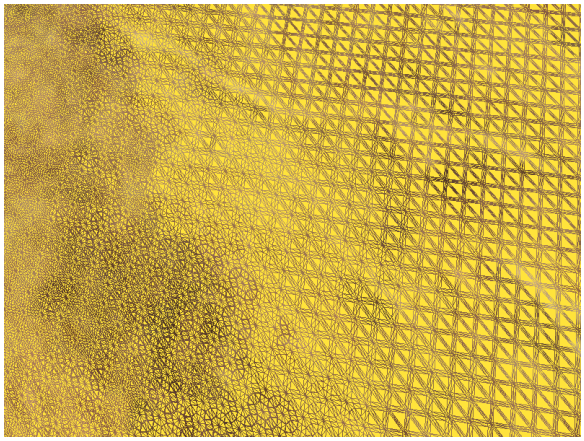
Lindstrom P., Koller D., Ribarsky W., Hodges L.F., Faust N., y Turner G.A. Real-time, continuous level of detail rendering of height fields. En *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, SIGGRAPH '96, páginas 109–118. ACM, New York, NY, USA, 1996. ISBN 0-89791-746-4. doi:10.1145/237170.237217.

Lindstrom P. y Pascucci V. Terrain simplification simplified: A general framework for view-dependent out-of-core visualization. *IEEE Transactions on Visualization and Computer Graphics*, 8(3):239–254, 2002. ISSN 1077-2626. doi:10.1109/TVCG.2002.1021577.

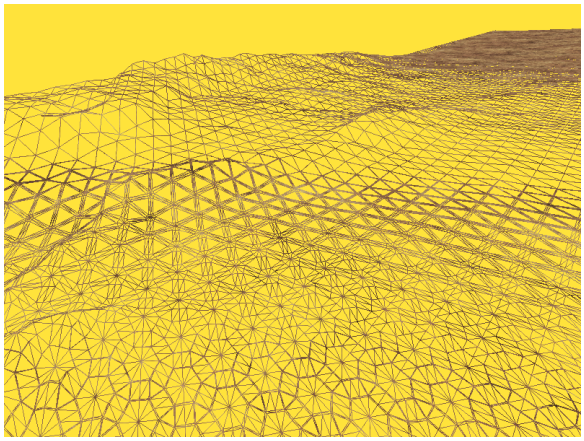
Microsoft. Directx 11. http://msdn.microsoft.com/en-us/directx/, ????

Pajarola R. Large scale terrain visualization using the restricted quadtree triangulation. En *Proceedings of the conference on Visualization '98*, VIS '98, páginas 19–26. IEEE Computer Society Press, Los Alamitos, CA, USA, 1998. ISBN 1-58113-106-2.

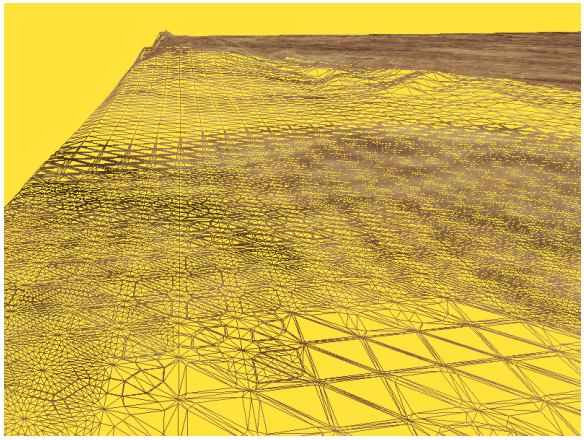
Pajarola R. y Gobbetti E. Survey of semi-regular multiresolution models for interactive terrain rendering. *Vis. Comput.*, 23(8):583–605, 2007. ISSN 0178-2789. doi:10.1007/s00371-007-0163-2.



(a) Teselación basada en densidad



(b) Teselación basada en distancia



(c) Teselación combinada

Figura 10: Teselación adaptativa

Reuter H., Nelson A., y A. J. An evaluation of void filling interpolation methods for srtm data. *International Journal of Geographic Information Science*, 21(9):983–1008, 2007.

Szofran A. Global terrain technology for flight simulation. Informe Técnico, Microsoft ACES Game Studio, 2006.