

Procesadores Gráficos y Aplicaciones en Tiempo Real

Proyecto de investigación: Calculo de la distancia de
Hausdorff en CUDA

Iván Velasco González

Índice

1. Introducción	2
2. Distancia de <i>Hausdorff</i>	2
3. Distancia punto a triangulo	2
4. Implementación	2
5. Resultados	3
5.1. Rendimiento	3
6. Mejoras	4

1. Introducción

El objetivo de esta práctica consiste en la implementación de un algoritmo que calcule la distancia de *Hausdorff* entre dos mallas de triángulos de forma masivamente paralela utilizando CUDA.

2. Distancia de *Hausdorff*

La distancia de *Hausdorff* es una forma genérica de calcular la distancia entre dos conjuntos de puntos. Concretamente se define como la máxima distancia de cada uno de los puntos del conjunto, calculándose esta distancia de cada punto como la distancia mínima desde ese punto dado al resto de puntos del otro conjunto. Sin embargo, en este trabajo no se va a trabajar con conjuntos sino que se utilizarán mallas de triángulos. Por lo tanto, es necesario definir tanto la distancia de una malla a otra como la distancia de un punto a una malla. Respecto a la distancia de punto a malla esta puede calcularse como la distancia mínima encontrada entre el punto y todos los triángulos que conforman la malla contra la que se compara, mientras que, la distancia de malla a malla será la distancia máxima encontrada en un punto utilizando el método anteriormente descrito.

3. Distancia punto a triángulo

Como puede deducirse del funcionamiento del algoritmo el cálculo de la distancia entre mallas va a requerir un gran número de cálculos de distancia de punto a triángulo. Por lo tanto, se va a proponer un algoritmo que resuelva este problema de manera eficiente. El algoritmo consiste en transformar el problema de distancia de 3D a 2D calculando la matriz de transformación que coloca el triángulo de tal manera que uno de sus vértices se encuentre en el origen, otro en el eje z , y el último, en el plano zy . Una vez calculada esta transformación se transforma el punto desde el que se quiere calcular la distancia y se proyecta al plano del triángulo, simplemente poniendo la coordenada x del punto a 0. Una vez proyectado el punto pueden darse dos casos, o bien que el punto proyectado caiga dentro del triángulo y, por lo tanto, la distancia sea el valor absoluto de la coordenada x , o bien, que el punto caiga fuera del triángulo, en este caso se calcula la distancia del punto sin proyectar a cada una de las aristas del triángulo y se escoge la menor distancia encontrada.

4. Implementación

En esta sección se detallará la implementación realizada de los algoritmos descritos anteriormente.

En primer lugar, se ha desarrollado una pequeña función la cual es capaz de leer un archivo en formato `obj` y almacena la malla leída de una manera muy común, como una lista de vértices y una lista de triángulos que referencian a los vértices de la lista de vértices.

Una vez se ha leído la primera malla, se procede a cambiar la forma de almacenarla a una que sea más adecuada para el proceso que se hará a continuación. Esta nueva representación escogida consiste en una lista de triángulos donde cada triángulo contiene la posición de sus tres vértices, prescindiendo de la lista de triángulos. Para realizar el cambio en la representación, en primer lugar, se reserva la memoria necesaria en la GPU para almacenar la malla, tanto en la representación usada en CPU como a la que se va a transformar para usarla en GPU. Seguidamente, se ejecuta el *kernel* el cual por cada triángulo accede a sus vértices correspondientes y los copia en la memoria de la nueva

representación. Una vez finalizado este proceso se libera la memoria reservada en GPU para la representación usada en CPU, ya que no se volverá a utilizar.

El siguiente caso consiste en un preproceso el cual calcula la transformación asociada a cada triángulo para posicionarlo de tal manera que su primer vértice se encuentre en el origen, el segundo en el eje z y el tercero en el eje zy . Para ello se ejecuta un *kernel* que, en primer lugar, coloca el primer vértice en el origen y seguidamente calcula 3 matrices de rotación por medio de trigonometría para colocar los otros dos vértices en los lugares indicados, por último, compone una única matriz de transformación y la almacena en un vector reservado a tal efecto en la memoria de la GPU.

Una vez que se ha realizado todo el preproceso de la primera malla se carga la segunda malla y se reserva memoria para almacenar sus vértices en la GPU. Seguidamente se procede a llamar al *kernel* que calculará las distancias de cada uno de estos vértices a la primera malla ya preprocesada. Para hacer esto el *kernel* por cada triángulo transforma el punto que está tratando con la correspondiente transformación del triángulo y calcula la distancia del punto a cada triángulo por medio de una llamada a la función *pointToTriangleDistance* la cual se verá más adelante, por último, se almacena en un vector de distancias la distancia mínima encontrada para cada vértice de la malla.

Por último, para calcular la distancia de *Hausdorff* entre las dos mallas se calcula la máxima distancia encontrada en los vértices. Para ello se ha utilizado un algoritmo de reducción muy similar al utilizado en la práctica. Este algoritmo consiste en que cada bloque de *Threads* realiza una reducción parcial del vector en memoria compartida y en última instancia almacena el resultado de la reducción parcial en la posición correspondiente en el vector global, siendo esta posición el número de bloque.

Respecto a la implementación de la función *pointToTriangleDistance* esta recibe como parámetros los vértices de un triángulo (solo 2 ya que el otro es siempre el origen de coordenadas) y el punto ya transformado. Lo primero que realiza la función es determinar si el punto se encuentra dentro del triángulo, para ello utiliza funciones de arista (*Edge Equation*), similares a las utilizadas en el proceso de rasterizado, que devuelven la distancia del punto a la arista con signo, indicando este signo a qué lado de la arista se encuentra. Por tanto, a partir de dos de estas ecuaciones (una arista es siempre el eje z) se determina si el punto se encuentra fuera o dentro del triángulo dependiendo de a qué lado de cada una de las aristas se encuentre. En el caso en el que el punto se encuentre dentro del triángulo se devuelve el valor absoluto de la coordenada x del punto, si no, por el contrario, el punto se encuentra fuera del triángulo se calcula la distancia del punto a los 3 segmentos que conforman el triángulo y se devuelve la distancia mínima encontrada.

5. Resultados

Para comprobar la validez del algoritmo implementado se han comparado los resultados obtenidos con el cálculo de distancias de *Hausdorff* proporcionado por MeshLab obteniendo resultados muy similares con errores que pueden deberse a problemas de precisión.

5.1. Rendimiento

El rendimiento obtenido con esta implementación ha sido bastante bueno tardando solamente 1351 milisegundos en calcular la distancia desde un modelo con 17.000 vértices a otro con 573.440 caras TODO COSAS DE CUDA

Sin embargo, como era de esperar la implementación en CPU proporcionada por MeshLab es mucho mas rapida realizando la operación en tan solo 271 milisegundos. Esto se debe principalmente a que MeshLab utiliza una tecnica de *Spatial Hashing* para minimizar los calculos de distancias de punto a triangulo, mientras que el algoritmo propuesto realiza esto por fuerza bruta testeando todos los triangulos, pero esto tiene la ventaja de que el algoritmo propuesto es mas robusto, ya que , en ocasiones durante las pruebas MeshLab no era capaz de calcular la distancia (devolviendo 0) posiblemente debido a esta optimización.

6. Mejoras

En esta sección se detallaran algunas mejoras que pueden realizarse al algoritmo para mejorar su eficiencia.