Procesadores Gráficos y Aplicaciones en Tiempo Real

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

Iván Velasco González

Índice

1. Introducción

El objetivo de esta práctica consiste en la implementación de un algoritmo que calcuque la distancia de *Hausdorff* entre dos mallas de triangulos de forma masimavemente paralela utilizando CUDA.

2. Distancia de *Hausdorff*

La distancia de *Hausdorff* es una forma generia de calcular la distancia entre dos conjuntos de puntos. Concretamente se define como la maxima distancia de cada uno de los puntos del conjunto, calculandose esta distancia de cada punto como la distancia minima 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 utilizaran mallas de triangulos. 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 minima encontrada entre el punto y todos los triangulos que conforman la malla contra la que se compara, mientras que, la distancia de malla a malla sera la distancia maxima encontrada en un punto utilizando el metodo anteriormente descrito.

3. Distancia punto a triangulo

Como puede deducirse del funcionamiento del algoritmo el calculo de la distancia entre mallas va a requerir un gran numero de calculos de distancia de punto a triangulo. 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 transformacion que coloca el triangulo de tal manera que uno de sus vertices se encuentre en el origen, otro en el eje z , y el ultimo, 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 triangulo , 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 triangulo y, por lo tanto, la distancia sea el valor absoluto de la coordenada x, o bien, que el punto caiga fuera del triangulo, en este caso se calcula la distancia del punto sin proyectar a cada una de las aristas del triangulo y se escoge la menor distancia encontrada.

4. Implementación

En esta sección se detallara 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 leida de una manera muy comun, como una lista de vertices y una lista de triangulos que referencian a los vertices de la lista de vertices.

Una vez se ha leido la primera malla, se procede a cambiar la forma de almacenarla a una que sea mas adecuada para el proceso que se hara a continuacion. Esta nueva representacion escogida consiste en una lsita de triangulos donde cada triangulo contiene la posición de sus tres vertices, prescindiendo de la lista de triangulos. 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 triangulo accede a sus vertices correspondientes y los copia en la memoria de la nueva

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

El siguiente caso consiste en un preproceso el cual calcula la transformación asociada a cada triangulo para posicionarlo de tal manera que su primer vertice 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 vertice en el origen y seguidamente calcula 3 matrices de rotación por medio de trigonometria para colocar los otros dos vertices en los lugares indicados, por ultimo, compone una unica matriz de transformación y la almacena en un vector reservado a tal efecto en la memrio 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 vertices en la GPU. Seguidamente se procede a llamar al *kernel* que calculara las distancias de cada uno de estos vertices a la primera malla ya preprocesada. Para hacer esto el kernel por cada triangulo transforma el punto que esta tratando con la correspodiente transformacion del triangulo y calcula la distancia del punto a cada triangulo por medio de una llamada a la funcion *pointToTriangleDistance* la cual se vera mas adelante, por ultimo, se almacena en un vector de distancias la distancia minima encotrada para cada vertice de la malla.

Por ultimo, para calcular la distancia de *Hausdorff* entre las dos mallas se calcula la maxima distancia encontrada en los vertices. Para ello se ha utilziado un algoritmo de reducción muy similar al utilizado en la practica. Este algoritmo consiste en que cada bloque de *Threads* realiza una reduccion parcial del vector en memoria compartida y en ultima instancia alamacena el resultado de la reduccion parcial en la posicion correspoendiente en el vector global, siendo esta posicion el numero de bloque.

Respecto a la implementacion de la funcion *pointToTriangleDistance* esta recibe como parametros los vertices de un triangulo (solo 2 ya que el otro es siempre el origen de coordenadass) y el punto ya tranformado. Lo primero que realiza la funcion es determinar si el punto se encuentra dentro del triangulo, 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 que 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 triangulo dependiendo de a que lado de cada una de las aristas se encuentre. En el caso en el que el punto se encuentre dentro del triangulo se devuelve el valor absoluto de la coordenada x del punto, si , por el contrario, el punto se encuentra fuera del punto se calcula la distancia del punto a los 3 segmentos que conforman el triangulo y se devuelve la distancia minima encontrada.

5. Resultados

Para comprobar la validez del algoritmo implementado se han comparado los resultados obtenidos con el calculo 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 vertices 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 minimar 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 le algormitmo propuesto es mas robusto, ya que , en ocasiones durante las pruebas MeshLab no era capaz de calcular la distancia (devolveindo 0) posiblemente debido a esta optmimiacion.

6. Mejoras

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

La mejora mas obvia seria implementar la tecnica de *Spatial Hashing* tambien en esta implementacion en GPU. Sin emargo realizar esto supone una serie de problemas, el primero de ellos consiste en lidiar con las colisiones propias de estas tablas en su construccion de forma masivamente paralela, donde habra que tener algun metodo para lidiar con estas condiciones que no requiera de muchos accesos a memoria y que no provoque demasiados intentos fallidos de colocación, ya que esto puede provocar mucha divergencia. Por otro lado, tambien hay que tener en cuenta que estas tabals son por definicion dispersas y ademas fuerzan a accesos no coalescentes en la GPU lo que provoca que el acceso a la tabla no sea tan rapido como cabria esperar.