Análisis de modelo de visualización 3D BRAINIAC

Este proyecto surge como un viaje de exploración dentro del vasto mundo de la computación gráfica. Se inició por conocer las bases fundamentales intrínsecas a la visualización de modelos 3D en ordenadores, por ejemplo: vértices y aristas, proceso de renderizado, API de gráficos para comunicarse con la GPU... Esto llevó a la investigación e implementación de algoritmos para crear formas a partir de puntos en una matriz. En el caso de BRAINIAC y, en la mayoría de renderizados populares, el modelo de polígonos triangulares agrupados en objetos de tipo Mesh quedó para la posteridad del proyecto.

BRAINIAC inició siendo una idea con algunas diferencias significativas en el propósito y dirección del proyecto. En una primera iteración se observaron los primeros signos de vida de BRAINIAC en un framework de C++ llamado Cinder. Este proporcionaba un conjunto de herramientas para trabajar con gráficos a bajo y medio nivel, incluyendo implementación de una rendering loop eficiente y una forma bastante simple de manejar eventos dentro del rendering pipeline. De cajón todo esto dentro de un contexto de OpenGL. Las pruebas iniciales fueron buenas, sin embargo, la visión de bajo nivel del proyecto contemplaba un intervalo de tiempo mayor para llegar a resultados satisfactorios. La respuesta a esta problemática era clara, pues, dentro del mismo planteamiento se señala indiscretamente el impedimento principal: la implementación a bajo nivel, si bien, eficiente, tomaba más tiempo de desarrollo e investigación. Fue así como surgió BRAINIAC-WEB.

BRAINIAC-WEB es creado como un sucesor de BRAINIAC pero orientado a navegadores. Cuando se trata de implementar gráficos en navegadores web la opción predilecta es WebGL, no obstante, esta

cuenta con una curva de aprendizaje bastante elevada. La solución fue utilizar Three.js, un framework para el desarrollo de gráficas en navegadores basado en WebGL. Optimizado por el eficiente renderizado de React y el tipado fuerte de TypeScript.

BRAINIAC y su sucesor comparten los mismos fundamentos en el desarrollo del proyecto: crear una visualización del cerebro y las fibras neuronales del mismo. Para generar los modelos 3D de ambos elementos se realizó un tipo de IRM especial llamado Imagen por Resonancia Magnética de Tensor de Difusión (DTI). Esta técnica consiste en medir cómo se difunden las moléculas de agua a lo largo de los diferentes caminos del cerebro. Esta metodología aprovecha que las fibras neuronales, como los axones y dendritas, están rodeadas de una capa llamada mielina que restringe el movimiento del agua y permite a la DTI detectar la dirección de estas fibras. En otras palabras, el agua se mueve más fácilmente a lo largo de las fibras que a través de ellas. La información recolectada atreves de este análisis se traduce digitalmente como Voxeles, el equivalente 3D de los pixeles. Cada voxel representa un tramo de una fibra y la unión final de todos los voxeles forma el circuito completo de las fibras neuronales.

El siguiente paso en el desarrollo del proyecto fue la creación de una arquitectura de software modular y escalable. Cada elemento del setup fue implementado en componentes independientes para facilitar el debug y la futura integración de detalles y funcionalidades adicionales. Tras haber definido la arquitectura y los componentes del programa tocaba ensamblarlos en un escenario digital para la demostración del proyecto. Los resultados de la tractografía fueron serializados para una carga de los datos sumamente rápida y eficaz debido a la gran cantidad de datos que eran manejados. El modelo del cerebro y el suelo se manejaron como archivos de tipo obj y, como se había aclarado anteriormente, como estructuras de Mesh dentro de la simulación. Se agregaron también iluminación y controles de cámara a la escena para crear un ambiente más agradable dentro de la visualización.

Para este punto del proyecto se decidió cambiar la finalidad de la visualización. De aquí en adelante la exposición va enfocada a mostrar de manera visual la cardinalidad o nivel de conexión de grafos resultantes de la investigación del proyecto: “Clasificación y visualización entre monólogo interno y lectura silenciosa analizando sincronía funcional de datos electroencefalográficos”, que, de ahora en adelante nos referiremos como “Clasificación monólogo interno y lectura silenciosa”.

La forma en la que se logró la visualización fue mediante la creación de unas denominadas “Zonas de Influencia” dentro del modelo del cerebro. Cada una de estas Zonas de Influencia se ve plasmada como una esfera en el espacio 3D con centro en la posición en la que se colocaron los electrodos a la hora de realizar la experimentación. En el grafo resultante del experimento de Clasificación monólogo interno y lectura silenciosa cada electrodo representa un nodo, y cada nodo tiene un nivel de conectividad o cardinalidad definida por la cantidad de conexiones entre cada electrodo y el resto de los electrodos. De este nivel de conectividad de cada Zona de Influencia se define de manera proporcional el radio de cada esfera.

La visualización explica los resultados del experimento mediante el cálculo de cuáles fibras se encuentran dentro de las Zonas de influencia. Cada Zona de Influencia tiene asignado un color que se heredará a las respectivas fibras afectadas por la Zona generando, de este modo, una representación de la cardinalidad de cada nodo (Electrodo) utilizado en el experimento.

El proceso de identificar qué fibras se encuentran dentro de cada Zona de Influencia es computacionalmente costoso. La implementación de algoritmos eficientes para la ejecución de este algoritmo es vital para un análisis detallado de los resultados. Ante esta situación se llegó al planteamiento de una forma de más bajo nivel del algoritmo de reconocimiento. Este consiste de la implementación de dos estructuras de datos para manejar el espacio 3D.

La primera estructura de datos utilizada y, la cual es pieza fundamental del algoritmo, es un Octree. El Octree es una estructura de datos utilizada para particionar el espacio tridimensional semejantemente a como su versión para espacios bidimensionales, el Quadtree, lo hace. Es utilizado principalmente para la detección colisiones en motores gráficos. Este consiste en un grafo con topología de árbol, con la característica de que cada nodo puede tener cero u ocho hijos, pero no más. Cada nodo representa una partición en el espacio reflejada como un cubo con su respectivo volumen. Por cada nivel del árbol el cubo padre se particiona recursivamente en regiones o cubos más pequeños dando más precisión a la búsqueda dentro del árbol. De esta forma cuando se quiere saber si cierta geometría está contenida dentro de otra o bien, están colisionando, nos evitamos tener que comparar todos los vértices existentes en la escena con todas las demás. Al excluir vastas regiones de espacio y teniendo que comparar vértices ultimadamente en los nodos hoja del árbol.

La complejidad computacional del Octree básico está dada por el siguiente argumento:

E = {x | x ∈ {puntos en el espacio 3D}} // Todos los puntos en el espacio 3D. V = {v ∈ E | v ∈ {vértices de todas las fibras}} // Vértices pertenecientes a las fibras n = |V|

Particionar el espacio: Esta es una operación de tiempo constante, más específicamente, de 8. O(1)

Asignación de puntos: La asignación de puntos es de tiempo lineal pues, por cada nivel del Octree se van a tener que iterar todos los puntos para asignarlos a un nuevo sector. O(n)

División recursiva: Un cubo es dividido en ocho cubos más pequeños contenidos dentro del cubo padre. Posteriormente se ve van a dividir cada uno de estos octantes en más octantes de manera similar hasta que queden pocos vértices dentro de cada octante. La cantidad de

veces que se van a hacer las particiones es definida por la cantidad de niveles que tenga el Octree. Esta cantidad es dada por log_8 n.

Justificación de la complejidad total: Debido a que se tienen que hacer la asignación de puntos por cada nivel, tenemos que la complejidad promedio es de O(n log_8 n).

La segunda estructura de datos utilizada es un Spacial Map. Esta es la elección predilecta para acompañar al Octree debido a que los modelos de las fibras y los cerebros son estáticas en la escena. Los Spacial Maps son hash tables que mapean coordenadas a zonas del espacio. Estas toman como llave un vértice y regresan el octante en el que este se encuentra. Su creación de tiempo lineal con base en n pero su tiempo de búsqueda es constante, lo que les da una ventaja en el largo plazo.

En conclusión, el manejo adecuado de las estructuras de datos es crucial en el desarrollo de programas enfocados a gráficos computacionales debido a la cantidad de operaciones que se tienen que hacer por cada frame. De aquí en adelante la mejor forma posible de mejorar el rendimiento de la aplicación es mediante programación pluri-hilada.