Advanced Computer Graphics

Lab 3: Volume Rendering

03/12/2020

Abel Pérez
abel.perez01@estudiant.upd.edu
NIA: 205254

Sixto Pineda
sixtosarwed.pineda01@estudiant.upf.edu
NIA: 205614

I. Información general sobre el Framework e ImGui

Para poder llevar a cabo Volume Rendering debimos modificar los ficheros ”application.cpp”, ”material.h” y ”material.cpp” del framework:

- **application.cpp**: aquí modificamos el constructor de Application, el update y renderInMenu (ImGui). Dado que usaremos variables que modificarán las propiedades del Volume Rendering desde el menú ImGui, creamos como variables globales el volumen (vrVolume), su material (vrMaterial), su respectivo shader (vrShader), la mesh (vrMesh), la textura del volumen (vrTexture), el Step Length inicializado (modificable desde ImGui) y la textura con ruido para realizar el Jittering (jittering). Éstas hacen referencia a las usadas para la primera parte de la práctica.
  Dado que nosotros escogimos como segunda parte la implementación con enfoque Médico, creamos de igual modo el mapa LUT para realizar la Transfer Function en el shader (LUT), dos flotantes para dar la opción de modificar los límites de color verde, rojo y blanco de manera manual (sl1, sl2), el threshold, el valor h y una luz para poder aplicar la Isosurface y el Gradiente (th, h, light), y por último un vector de cuatro componentes que usaremos como A, B, C y D para crear el plano que nos permita realizar el Clipping (clipping).
  Además de las distintas variables para cada una de las partes de la práctica, con el fin de poder ver cada una por separado, aplicar la textura LUT en la Transfer Function o usar sliders para modificar los límites de color, y poder activar o no el gradiente, creamos cinco variables booleanas que mediante su modificación desde ImGui y condicionales en el shader, podremos activar o desactivar.
  Una vez creadas todas las variables globales pasamos a modificar los métodos mencionados:
  - **application**: creamos el nodo de la escena e inicializamos la mesh cúbica que le asignaremos a dicho nodo. De igual modo, creamos y cargamos el shader, el volumen, la luz y las texturas, tanto la del volumen como la de Jittering y LUT. Dado que cada volumen tiene unas dimensiones en concreto, modificamos la escala del modelo pasado al nodo según las dimensiones y el espaciado del volumen (i.e. $dimension * espaciado$).
  - **update**: en este método inicializaremos el material (vrMaterial) con cada una de las variables que se necesite, y de igual modo que en las anteriores prácticas, asignamos dicho material al nodo.
  - **renderInMenu**: creamos un desplegable de propiedades a modificar sobre el Volume Rendering y dentro de éste podremos variar el valor de Step Length que le pasamos al material del volumen, activar o desactivar cada una de las partes de la práctica, y modificar el plano para realizar el Clipping.
    Cabe decir que en activar la segunda parte podremos usar la textura LUT o los sliders para la Transfer Function, y activar o no el gradiente. Si decidimos activarlo nos aparecerá un desplegable para poder modificar el threshold, el valor h y la luz (posición, color e intensidad). Otro dato importante es que el valor por defecto del plano de Clipping es el óptimo para eliminar la camilla pero si tenemos activada la Transfer Function parte de la espalda del cuerpo se ve cortada.
- **material.h**: para crear el material del volumen creamos una subclase que hereda de “StandardMaterial”. Ésta contendrá como variables locales: todas las necesarias, como hemos mencionado anteriormente, para realizar cada una de las partes de la práctica en el shader, con éste último incluido como variable local; y sus respectivos métodos constructor, destructor, y setUniforms para pasar toda la información necesaria al shader.
- **material.cpp**: mediante el método constructor de la clase creada para el material del volumen, inicializamos todas las variables pasadas a éste, y mediante el método “setUniforms” le pasamos al shader todos los parámetros necesarios para computar “Ray marching algorithm” y ”Medical Path”.

II. Ray Marching Setup

En esta sección, presentaremos algunos pasos que realizamos antes de usar el bucle para computar el algoritmo de Ray-marching. Desde este apartado en adelante toda la implementación está realizada en el shader:

Inicializamos finalColor (color final) y sampleColor (sample de color de la textura del volumen, la densidad) como vectores cero de cuatro componentes, y step length con la variable pasada al shader [1].

Dado que no todos los vectores con los que tenemos que trabajar se encuentran en el mismo sistema de coordenadas, nosotros decidimos, pasar todo a coordenadas de textura. Con el fin de poder realizar dicha conversión, usamos la inversa de la matriz "u_model" (u_model_inv) que nos permite pasar de coordenadas de mundo a locales.
Cuando nos encontramos en coordenadas locales (i.e. coordenadas_mundo*u_model_inv), los valores van de -1 a 1, por lo tanto, si queremos pasar a coordenadas de textura (i.e. valores que van de 0 a 1), tan solo debemos sumar 1 a cada componente del vector y dividir entre 2.

- **texturePos**: dado que v_position se encuentra en coordenadas locales, tan solo hacemos el proceso mencionado anteriormente para obtener la posición del pixel en coordenadas de textura.
- **Cámara**: al shader se le pasa la posición de la cámara con coordenadas de mundo. Por lo tanto, multiplicamos dicho vector por la matriz inversa de la "u_model", pasando a coordenadas locales y nuevamente pasamos éstas a coordenadas de textura.
- **Rayo**: el rayo lo obtenemos como el vector que va desde la posición de la cámara hacia el pixel. Al tener ya calculados ambos vectores en coordenadas de textura, tan solo hacemos la diferencia entre éstos y normalizamos el vector. Además, al multiplicar por el step length obtenemos el vector que usaremos para ir avanzando al realizar el algoritmo dentro del bucle (stepVector) [1].
- **Jittering**: cargamos la textura de ruido con los valores x e y del vector texturePos (posicion actual dentro del volumen) y lo multiplicamos por el step length para obtener el valor de jittering que deberemos añadir en la primera posición de cada rayo (stepVector) con el fin de eliminar el "Wood pattern" causado por el hecho de que todas las muestras empiezan en paralelo y tienen el mismo step [1].
- **Light**: de igual forma que con la cámara, la posición de la luz pasada al shader se encuentra en coordenadas de mundo. Por lo tanto, realizamos el mismo proceso para obtener las coordenadas de textura de la luz. A continuación, puesto que tenemos la luz y la posición del pixel en coordenadas de textura, tan solo debemos hacer la diferencia entre éstos para obtener el vector de la luz manteniendo la consistencia de las coordenadas.

III. Parte 1: Ray Marching algorithm & Jitrreing (Fig. 1)

Para realizar el algoritmo crearemos un bucle de tipo "for" de mil iteraciones como caso límite y cargaremos la densidad del punto actual del volumen con el vector texturePos. Dado que la textura es en escala de grises (i.e. x = y = z), para obtener el valor de densidad, tan solo con coger una de las tres componentes resultantes de cargar la textura ya nos sirve.
Cabe recordar que en la variable texturePos ya hemos añadido el valor de jittering antes de empezar el bucle. Por lo tanto, la corrección ya se está aplicando [2].

1) **Ray Marching algorithm [1]**: una vez tenemos la densidad del sample en el que nos encontramos necesitamos obtener el color. Para ello, igualamos el sampleColor a un vector de cuatro componentes, donde cada una será la densidad:

```
sampleColor = vec4(d,d,d,d);
```

Una vez tenemos el sampleColor podemos añadirlo al color final. Esto lo realizamos mediante el esquema propuesto para la primera parte de la práctica, que suma la densidad de una manera ponderada:

```
finalColor += stepLength*(1 - finalColor.a) * sampleColor;
```

Y por último avanzamos a la siguiente posición a samplear sumando el stepVector a la posición del pixel (texturPos).

```
texturePos += stepVector;
```

Una vez alcanzamos la densidad máxima visible ($finalColor.a = 1$), nos salimos del volumen o, en el peor de los casos, llegamos al máximo de iteraciónes (lo vemos en el siguiente apartado), salimos del bucle y mostramos el color total calculado.

IV. Parte 2: Medical Visualization (Fig. 2 y 3)

1) **Volume clipping**: primero de todo, antes de empezar a computar el algoritmo, aplicaremos un condicional usando el plano de Clipping, definido como $ax + by + cz + d < 0$ [2]:

```
dot(clipping,vec4(texturePos,1.0)) < 0
```

Esta condición nos permite eliminar la camilla que se encuentra debajo del volumen de la persona (Fig. 3a). En caso de que en la posición del pixel con la que trabajar se encuentre por debajo de dicho plano, procederemos a hacer el algoritmo. En caso contrario únicamente añadiremos el stepVector a la posición actual y pasaremos a la siguiente iteración (Fig. 3b).

2) **Transfer function**: nos permite mapear las propiedades visuales y las propiedades del volumen. Por lo tanto, según la densidad del volumen podemos dar un color u otro de manera que podamos diferenciar entre órganos, musculos y huesos. Esto se puede realizar de dos formas: utilizando condicionales, donde dependiendo de la densidad obtenida, asignaremos un color u otro a sampleColor, o mediante un mapa LUT de una dimensión, donde según la densidad añadiremos un color u otro.

   Para realizar la Transfer Function implementamos ambas opciones seleccionables desde ImGui:

   - **LUT map** (Fig. 2a): cargamos el color del mapa según la densidad y creamos un vector que contendrá el valor del color del mapa y como cuarta componente la densidad.
     Cabe decir que en el momento de crear la textura LUT probamos de manera empírica qué rango de valores para la densidad nos daba un mejor resultado visual.

```
    sampleColor = vec4(texture2D(lut, vec2(d,1)));
```

   - **Sliders**: para ello usaremos dos flotantes modificables desde ImGui. Según la densidad del volumen se cumplirá una condición u otra y en función de esto se le asignará un color a sampleColor.

3) **Isosurfaces & Gradient**: con el fin de poder conseguir sombras o una sensación de profundidad aplicaremos una iluminación básica (NdotL, donde N es el vector normal y L el vector que va de la luz al punto). Dado que no disponemos de superficies para calcular N, utilizamos el concepto de isosurfaces (conjunto de puntos en el volumen con la misma densidad) para poder simularlas, y el gradiente (dirección en la que aumenta la densidad). Dado que el gradiente tiene exactamente la misma dirección que el vector normal de las isosurfaces, es lo que que utilizaremos para sustituir N [2]. Partiendo de esto podemos sombrear el volumen utilizando el NdotL.

   - **Gradiente**: para obtener el pendiente cogemos dos puntos cercanos (una muestra antes y una muestra después) y calculamos la dirección en la que aumenta la densidad. Este concepto lo extrapolamos a 3D y simplemente cogemos el valor que hay en esas posiciones.

```
    float x = texture3D(texture, texturePos + vec3(h, 0.0, 0.0)).x - texture3D(texture, ...
        texturePos - vec3(h, 0.0, 0.0)).x;
    float y = ...;
    float z = ...;
    vec3 gradient = vec3(x,y,z)*(2.0*h);
```

Calculada la normal (N) mediante el gradiente, realizamos el producto escalar entre N y L. A partir de aquí podemos añadir dicho sombreado al color final, pero dado que también hemos decidido otras propiedades de la luz como el color y la intensidad, sumaremos la multiplicación de las tres variable.

```
sampleColor = vec4(NdotL)*vec4(light_intensity)*vec4(light_color,1);
```

Todo este proceso se lleva a cabo de forma selectiva ya que calcular el gradiente es muy costoso. Para ello, establecemos un "threshold" (th) para discriminar según la densidad del volumen. De esta manera realizaremos este proceso únicamente cuando la densidad del punto supere el thereshold establecido. Por esta razón después de computar y sumar el sombreado, modificaremos la densidad del volumen a 1, y de este modo saldremos del bucle.

También es importante mencionar que para ambos apartados, tras cada iteración tenemos dos condiciones por los cuales saldremos del bucle: que nos salgamos del volumen, donde tendremos en cuenta el stepLength para evitar artefactos al utilizar el Jittering, o que alcancemos la densidad máxima. Como hemos mencioando antetiormente, solamente en los peores casos deberemos realizar todas las iteraciones.

```
if (finalColor.a≥1.0) break; //alcanzamos la maxima densidad
if (texturePos.x≤-0.0 || texturePos.y≤-0.0 || texturePos.z≤-0.0) break; //nos salimos del volumen
if (jitBool==true){if (texturePos.x≥1.0 || texturePos.y≥1.0 || texturePos.z-stepLength≥1.0) break;}
else{if (texturePos.x≥1.0 || texturePos.y≥1.0 || texturePos.z≥1.0) break; //nos salimos del volumen}
```

V. Resultados obtenidos

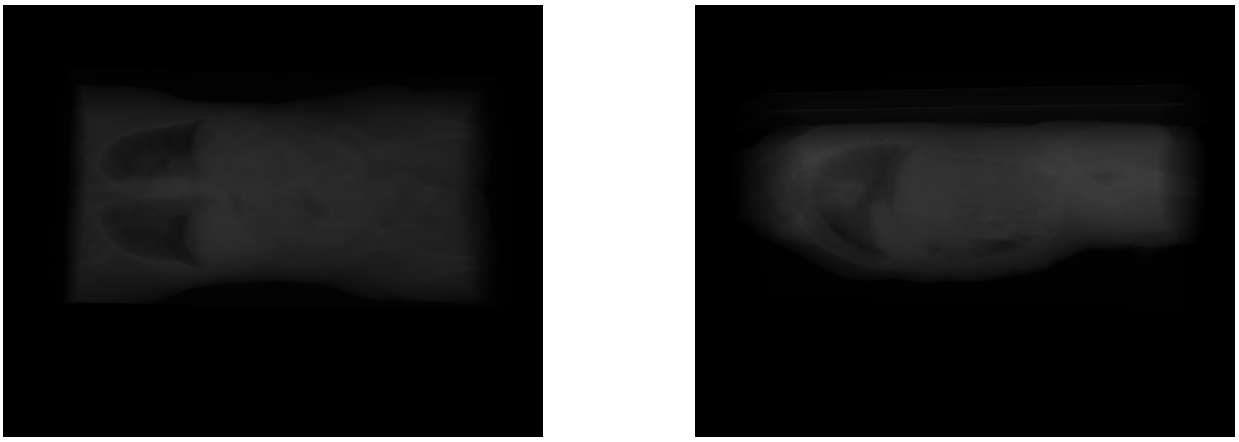
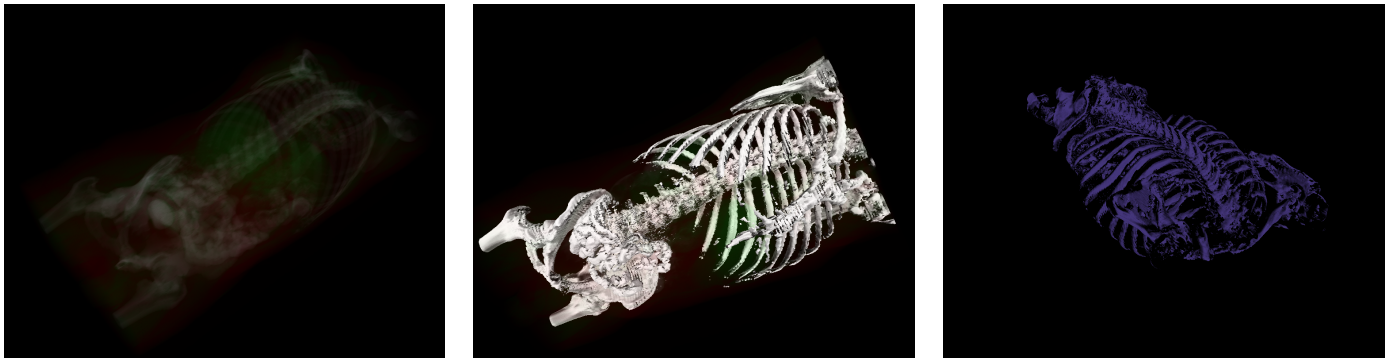
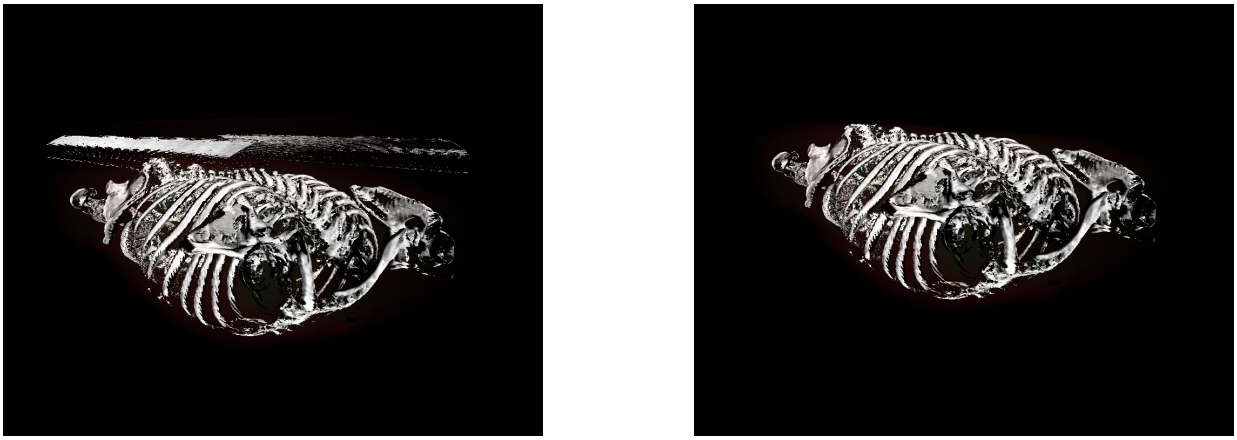


Fig. 1: Ray marching & Jittering



(a) Transfer function (b) Transfer function & Gradient (c) Gradient & Light color

Fig. 2: Transfer function, Gradient & Lighting



(a) No clipping (b) Clipping applied

Fig. 3: Clipping

References

[1] "Volume rendering - ray marching seminar," *slides de Advanced Computer Graphics*, 2020.

[2] "Volume rendering - lab3," *slides de Advanced Computer Graphics*, 2020.