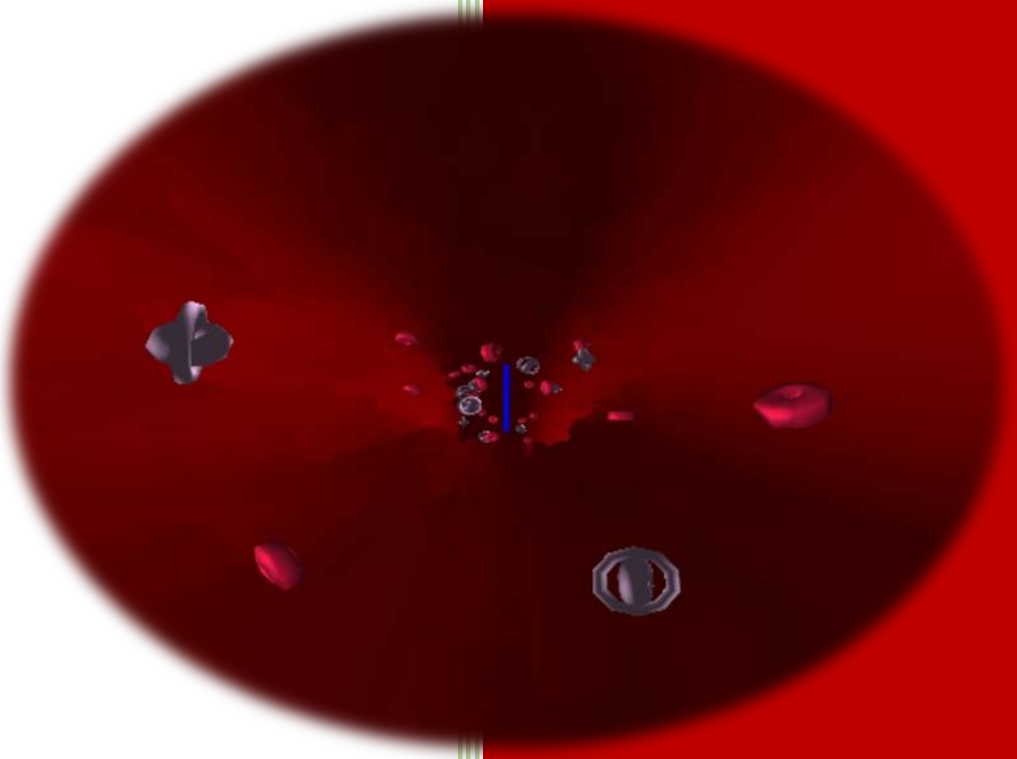
Práctica 1

El viaje alucinante



Julio Martín Saez y Raquel Peces Muñoz.

Contenido

El viaje alucinante

Introducción

En la siguiente memoria describimos la realización, el planteamiento del código que hemos escrito y se incluyen una serie de capturas de pantalla que muestran los resultados.

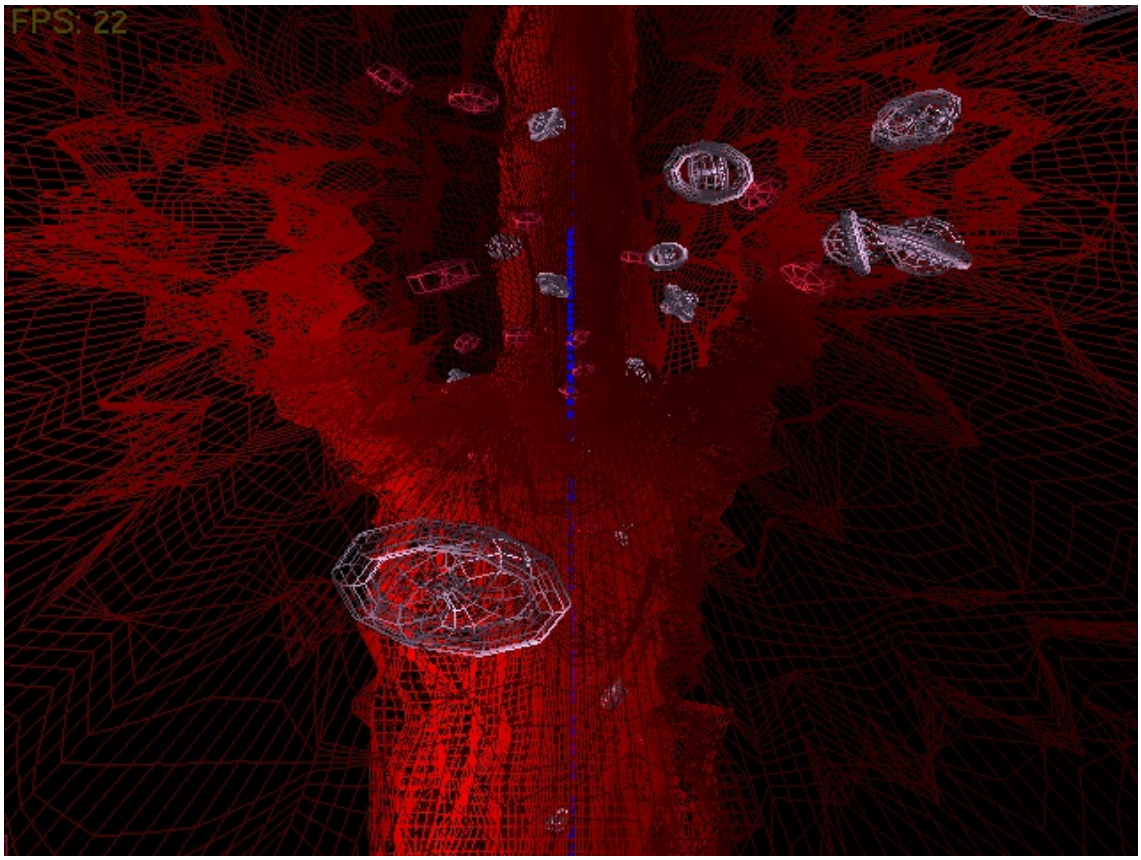


Figura 1 Vena en modo alámbrico

Generación de la curva

Generación de la spline

Para la generación de la curva nos hemos basado en el documento facilitado en el que se explica la interpolación por Catmull-Rom, en el que el conjunto original de puntos también es parte de los puntos de control de la curva spline que se genera.

Por un lado la clase "BezierCurve" se encarga de a partir de unos puntos de control especificados, los cuales pueden variar al aplicarle un random a esos números de suma o resta entre unos límites, para evitar que la vena se cruce entre sí, generar la curva a través de la interpolación Catmull-Rom.

Por otro lado, para poder ver la línea que sigue la curva, con los puntos generados, los hemos pintado. La clase "DrawCurve" es la encargada de coger dichos puntos y mandárselos al shader de vértices y fragmentos correspondientes, en este caso "curveVshader" y "curveFshader" los cuales toman dicha curva en forma de puntos y la pintan en modo línea continua, discontinua o puntos, dependiendo del modo en el que nos encontremos.

Cálculos sobre la spline

Para que la cámara pueda seguir correctamente la curva, necesitamos situarla correctamente en ella. Para ello necesitamos el vector tangente a la curva en ese punto, y el vector binormal en ese punto. Por otro lado, para el cálculo de la estructura de la vena a partir del marco de Frenet también necesitamos la normal en dicho punto.

Es por esto que para cada punto de la curva calculamos estos valores. Una de las decisiones que tuvimos que tomar era generar una curva en el plano XY, para que la normal fuese el vector Z, porque en otro caso la generación de la vena nos hacía pliegues a la hora de construir las caras que la forman.

Por tanto, el cálculo de estos vectores para cada punto queda del siguiente modo:

$$Tangente_i = \tau\,(p_{i+1} - p_{i-1})$$

$$Normal_i = (0, 0, 1)$$

$$Binormal_i = (Normal_i\ x\ Tangente_i)$$

Figura 2 Vista de la curva

Generación de la vena

Geometría de la vena

Para el cálculo de la geometría de la vena,en primer lugar hacemos uso de la clase “Poligon”.Esta clase es la encargada de generar un polígono regular de N lados centrado en el origen, el cual lo utilizamos para aproximar la circunferencia.

Una vez calculado el polígono, se obtiene la transformación del sistema de coordenadas global en un sistema de coordenadas local cuyo origen se encuentra en los puntos de la curva. Para ello haremos uso del Marco de Frenet, en el cual se parte de un perfil dado, en nuestro caso el polígono mencionado, y se producen sucesivos perfiles, cuyos puntos se unen, igual que se hace en una malla por extrusión.

El cálculo de la transformación por el Marco de Frenet es la matriz:

$$M(t) = (N(t), B(t), T(t), C(t)))$$

Donde N, B y T son los vectores normal, binormal y tangente respectivamente correspondientes a las coordenadas C en el punto t.

Por otro lado, para el cálculo de los vectores normales por cada cara, hacemos uso del Método de Newell, en el cual por cada cara se calcula su vector normal a partir de las coordenadas de los puntos que lo forman.

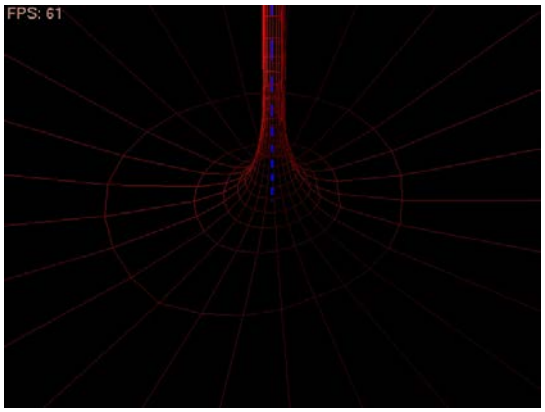


Figura 3 Vista de la curva en modo alámbrico

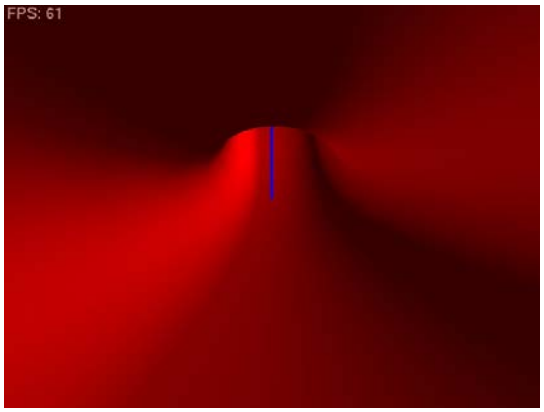


Figura 4 Vista de la curva rellena

Ruido de Perlin

La deformación de la vena se realiza a través de un mapa de Perlin. Este cálculo lo realizamos en la CPU después de tener calculada la geometría. Y se aplica una deformación de los vértices hacia el centro de la vena.

Para el cálculo del mapa de ruido de Perlin se usa la clase “PerlinGenerator” el cual tiene dos métodos relevantes “generate” y “generateNextLevel”, los cuales se encargan de generar la imagen final del ruido de Perlin y de generar la imagen de un nivel específico respectivamente.

Para ello el método “generateNextLevel” recibe el número de cuadrados que forman dicho nivel y obtiene los valores aleatoreos que corresponderán a las aristas exteriores, en este caso, están asignados para que pueda pegarse de manera circular, y después se itera por los puntos interiores de cada cuadrado. Por último se interpolan los valores de manera bilineal a lo largo de cada cuadrado.

Para depurar la generación de los mapas y comprobar su correcto funcionamiento, se ha utilizado una clase *BitmapSaver* que se encarga de escribir una imagen. Al ser una funcionalidad totalmente irrelevante y con carácter únicamente depurativo, se ha obtenido el código de internet y se ha adaptado para los fines específicos.

A continuación se muestran las imágenes para una ejecución concreta:

Figura 5 Nivel 0

Figura 6 Nivel 1

Figura 7 Nivel 2

Figura 8 Nivel 3

Figura 9 Nivel 4

Figura 10 Nivel 5

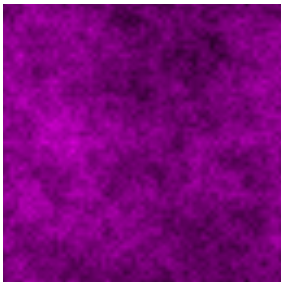


Figura 11 Combinación de los niveles

Color

Para colorear los vértices de la vena, hemos utilizado la deformación generada por Perlin, y dependiendo de dicha deformación aplicamos un color u otro a dicho vértice.

Los colores los hemos elegido de acuerdo a una gama de colores monocromáticos los cuales los hemos seleccionado siguiendo la gama que se muestra en la Figura 12, y la vena queda como muestra la Figura 11 tras aplicar las transformaciones del ruido de Perlin y la gama de colores conforme a la altura.

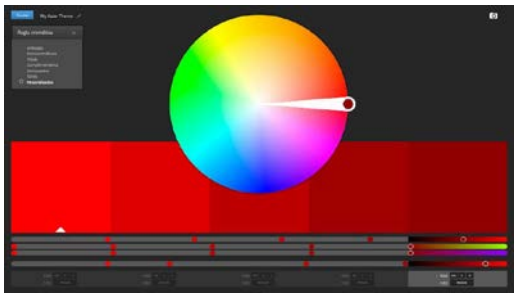


Figura 12 Rueda de colores

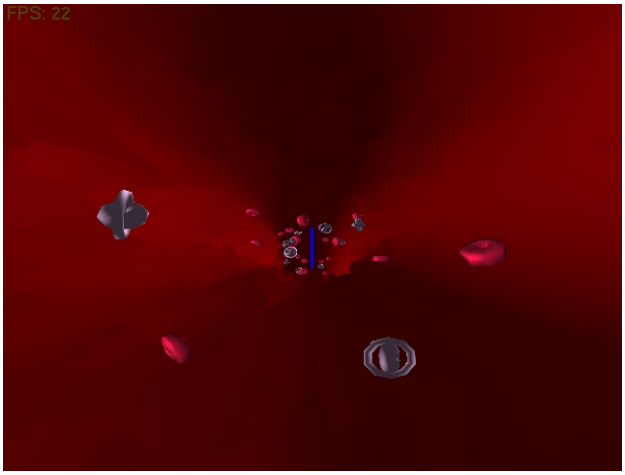


Figura 13 Vena con el gradiente de color

Iluminación

Para la iluminación de la vena, hacemos uso de la fórmula de Phong utilizando una única luz direccional. De este modo eliminamos el cálculo del resto de posibles luces de la escena.

Mutación

Para realizar la parte de la mutación de la vena, pasamos una variable “mutation” la cual realizará un coloreado similar al efecto plasma que aprendimos con los shaders de frágmentos en la asignatura de Procesadores Gráficos, en este modo de simulación hemos definido que sea en modo alámbrico.

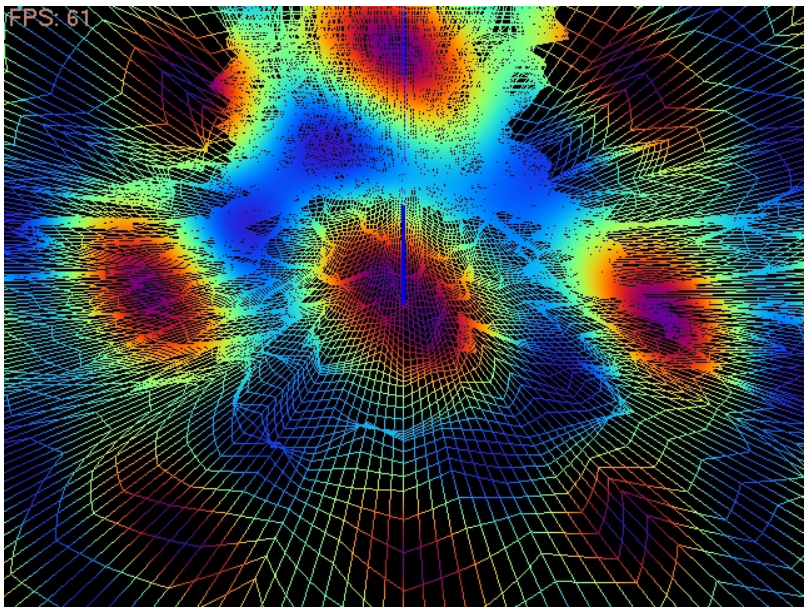


Figura 13 Mutación de la vena

Generación de los glóbulos

Cálculo de la geometría

Para la generación de los glóbulos hemos hecho uso de la clase “Blood” la cual contiene un array de “BloodElement”. Esta clase recibe el número de glóbulos rojos y blancos que queremos y procede a crearlos.

Para ello, ambas funciones realizan un algoritmo similar, eligen aleatoriamente un punto de la curva, y a él le suman un valor acotado, para que la posición no se salga de la vena. Una vez calculada la posición del glóbulo se calcula su rotación, para ello también obtenemos un número aleatorio para cada componente de la rotación con respecto a cada uno de los ejes. Con estos valores, realizamos el cálculo de los glóbulos.

La geometría de los glóbulos se ha obtenido a través de la geometría de un toroide. Esta función la hemos obtenido de internet, para poder obtener las coordenadas de los vértices, así como sus normales y sus coordenadas de textura, puesto que usando funciones como “glutSolidTorus” no podríamos obtener a ellas. El cálculo de esta geometría se realiza en el origen y luego se transforma a través de las matrices de rotación y translación a las coordenadas que hemos generado aleatoriamente.

Los “BloodElements” pueden ser de dos tipos, rojos y blancos. Los glóbulos rojos se pasan los parámetros específicos para generar una toroide de tipo “horn torus” para que tenga una forma más similar al glóbulo rojo. Por otro lado los glóbulos blancos están formados de dos toroides uno rodeando a otro.

Una vez calculada toda la geometría se pasan a sus shaders correspondientes, en este caso el “bloodVShader” y “bloodFShader” los cuales se encargan de realizar el pintado de los glóbulos. Estos shaders son muy similares a los de la vena, pero en este caso se pasa un color como uniform encargado de pintar dicho color en todos los vértices de la geometría.

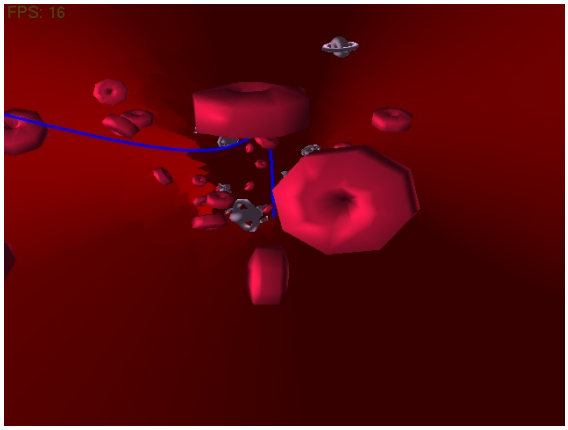


Figura 14 Vista en detalle de los glóbulos rojo

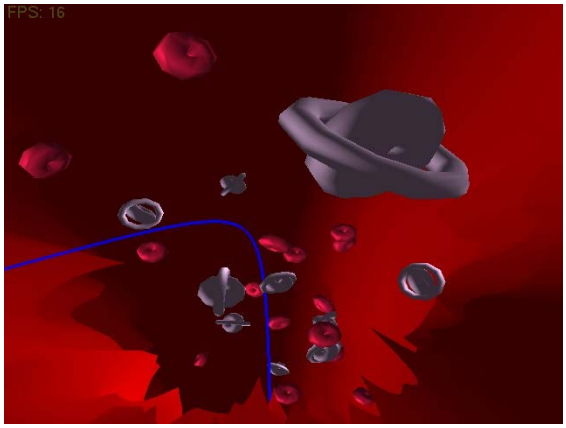


Figura 15 Vista en detalle de glóbulos blancos

Mutación

Para realizar la parte de la mutación de los glóbulos, pasamos una variable “mutation” la cual realizará un coloreado similar al efecto plasma que aprendimos con los shaders de fragmentos en la asignatura de Procesadores Gráficos.

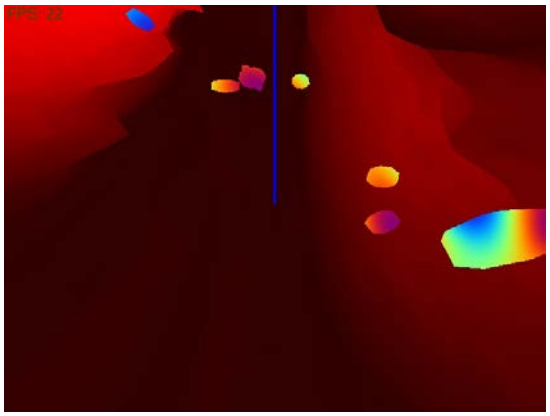


Figura 16 Mutación de los glóbulos (I)

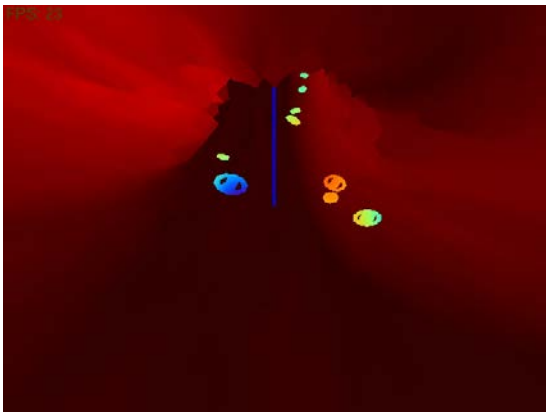


Figura 17 Mutación de los glóbulos (II)

Cámara

Implementación

Se ha implementado una clase “Camera” la cual contiene todos los controles de la cámara. Es la encargada de generar las matrices de proyección perspectiva y de transformación para las coordenadas de la cámara. Para ellos dichos valores se calculan en la creación de la cámara por medio de los parámetros que se le pasan, en este caso el ángulo de apertura, el aspect ratio y los planos near y far.

Controles

Para permitir al usuario el control de la cámara se han implementado una serie de métodos. Estas funciones son llamadas a partir de los eventos de teclado los siguientes métodos:

- “Move”: Este método translada la cámara en las cuatro direcciones, sin permitir salirse de la curva.
- “Rotate”: Este método rota la cámara con respecto a los vectores up, look y el vector normal.
- “addZoom” y “deductZoom”: Estos métodos permiten hacer zoom y disminuirlo entre un máximo y mínimo para evitar que no se abra demasiado el ángulo de la cámara dando lugar a que la cámara se voltee.
- Moverse por la curva adelante o atrás punto a punto.

Modos

Para permitir al usuario poder ver y moverse por la curva de distintos modos se han implementado una serie de métodos:

- “followCurve”: Este método se encarga de ir avanzando o retrocediendo por la cámara, dependiendo del valor del booleano que le pasemos. Si estamos en el modo de avance automático irá recorriendo los puntos uno a uno.
- “followOutCurve”: Este método situa la cámara una distancia por fuera de la curva, para poder seguir los puntos uno a uno y tener una vista global de toda la vena.
- “simulateHeartBeat”: Este método realiza un avance simulando los latidos del corazón, el cual se basa en un ciclo de avance lineal de los puntos, un frame de parada y unos frames en los que avanza más puntos de seguido.

Mutación

Cuando se llega al momento de la mutación, también realizamos un cambio en la matriz de proyección para ver una vista más deformada tanto en los métodos de avance lineal como en el de avance en forma de latidos, cada uno de ellos siendo distinto. La figura 19 muestra una captura de la mutación en el avance en modo de latidos.

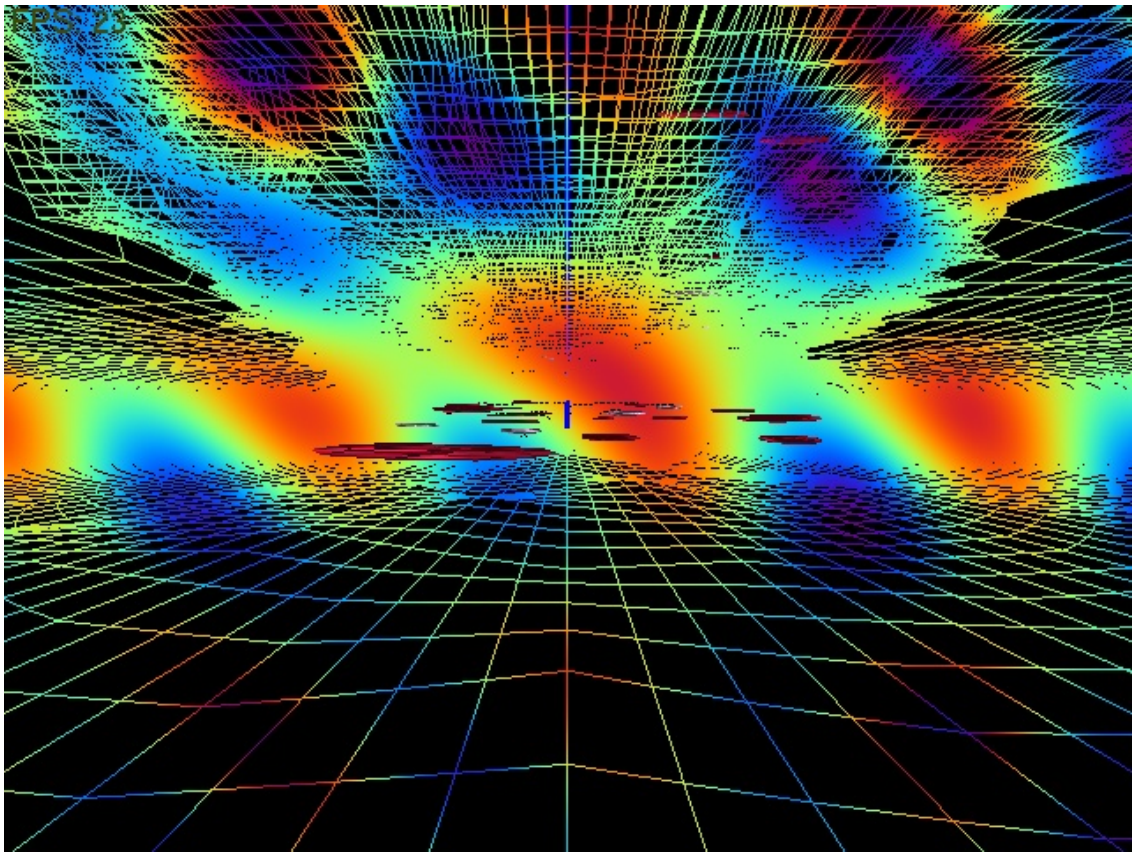


Figura 18 Mutación de la cámara

Material utilizado

Librerías

Para el desarrollo de la práctica hemos hecho uso de las siguientes librerías:

- GLM: Para el manejo y cálculo sobre matrices, vectores y las matrices de proyección y de vista de la cámara.
- FreeImage: Para la carga de imágenes.

Herramientas

Para el desarrollo de la práctica hemos hecho uso de las siguientes herramientas:

- Microsoft Visual Studio 2013.
- Git: Se ha utilizado un repositorio en github para el control de versiones y trabajo en grupo. El repositorio estq_alojado en el siguiente enlace: https://github.com/JulioUrjc/AdvancedRendering
- Youtube: Hemos generado un video de los resultados, el cual se encuentra en youtube en la siguiente dirección: http://youtube.com/watch?v=Dk0L6q_F80c

Problemas encontrados

Los problemas encontrados en el desarrollo de la práctica, los cuales han limitado las características que buscábamos implementar han sido los siguientes.

- Uso de texturas. Se ha intentado poner una textura, tanto en la vena como en los glóbulos pero nos hemos sentido incapaces de que se mostrase, lo cual nos ha hecho perder mucho tiempo de trabajo. Está preparada toda la estructura así como el cálculo de coordenadas u,v y la carga y lectura de ellas en el shader.
- Colocación de la cámara. En primer lugar realizamos una cámara que funcionaba en el cauce estático, pero para que funcionara en el cauce dinámico tuvimos que realizar una segunda implementación nueva, lo cual además nos dio bastantes problemas hasta que conseguimos obtener la matriz de vista y proyección correcta.