Dani Barca 146727
Lucas Enric De Haan 146736
Álvaro Ibáñez 124805

INFOGRAFÍA
Práctica 2: Pipeline Gráfica

Introducción y Explicación del Algoritmo

La segunda y última práctica de la asignatura tiene por nombre Pipeline Gráfica. Esta práctica se puede subdividir en varias partes y algoritmos, a diferencia de la primera, en la que el propio algoritmo Seam-Carving te daba el proceso a seguir.

Para entender qué teníamos que hacer, tuvimos que complementar las notas de clase de teoría, en la que nos introducen los algoritmos a utilizar, con la lectura del libro “3D Computer Graphics”, en la que se explican con más detalle los algoritmos y procesos para llevar a cabo la práctica. Esos capítulos del libro nos hablan de transformaciones 3D-2D, culling, rasterización, renderización, z-buffer, etc. Para desarrollar la práctica, hay que hacer uso de todos estos algoritmos paso por paso, en la que a medida que vayamos usando más algoritmos para la visualización de la imagen, la calidad será mayor. A continuación, explicamos en qué consiste esta práctica y cada uno de los algoritmos.

En esta práctica se lee un fichero de texto en el que se especifica un número de vértices y un número de polígonos (que representan un objeto), las coordenadas de los vértices, y, por último, por cada polígono, te dice cuántos vértices tiene y cuáles son. Estos datos nos darán la manera en la que el objeto se construye en el “mundo”.

Lo siguiente que se hace es añadir una cámara a nuestro mundo, que será la que nos dará la visión respecto al plano. Una vez tenemos el objeto y la cámara, primero hemos de transformar las coordenadas del objeto a punto de vista, y después, tenemos que pasar nuestro objeto en coordenadas mundo a pantalla.

Para ello, se usa un algoritmo de proyección 3D-2D que renderiza la imagen, obviando aquellos puntos que estan fuera del alcance de visión de la cámara, y proyecta los puntos del mundo a nuestra pantalla. Rasterizando los puntos, obtenemos la unión de los vértices y nuestro objeto ya toma forma.

Una vez tenemos el objeto en nuestra pantalla, tenemos que continuar añadiendo optimizaciones. Usando el culling, haremos que las caras que no sean visibles por la cámara, tampoco sean visibles en nuestra pantalla. Si añadimos otro objeto a nuestro mundo, y uno tuviera partes por delante del otro, con el algoritmo del z-buffer haríamos que sólo mostrara las partes de los polígonos que fueran visibles. Esto es debido a que el algoritmo z-buffer tiene en cuenta la componente z, es decir, la profundidad de los puntos. Si la z de un punto A es mayor que la de un punto B, significa que B está por delante de A, entonces las partes que A tenga por detrás de B no serán visibles.

En la práctica también se le puede añadir un punto de luz al mundo, y usando shading, se hace que las zonas sobre las que influya la luz directamente tendrán una luminosidad mayor a las que estan menos accesibles.

Explicación del código

Hemos ido estructurando el código siguiendo las indicaciones detalladas del libro y de las notas de teoría.

Tenemos por un lado las clases *image* y *text*, con el código proporcionado por el profesor para el tratamiento de imágenes y para parsear documentos de txt. Después, hemos creado una serie de clases para estructurar el código:

- *Framework*, en la que realizamos todas las operaciones correspondientes con las matrices y vectores que tenemos que usar en nuestros algoritmos, así como algunas funciones matemáticas y definiciones que pueden ser útiles. Tanto los vectores como las matrices incluyen todo tipo de operaciones que puedan ser útiles, como por ejemplo la multiplicación de matrices, producto escalar y vectorial de vectores, rotaciones y traslaciones con matrices, etc.
- *Polygon*, en la que guardamos el número de vértices que tiene el polígono en cuestión y qué vértices son.
- *Mesh*, en la que parseamos el fichero de texto y guardamos la información sobre vértices y los polígonos.
- *Object*, en la que guardamos toda la información sobre posición y rotación del objeto, así como su mesh.
- *Camera*, en la que introducimos la cámara al mundo y realizamos todas las operaciones que tienen que ver con ella: visualizar el objeto y transformarlo a punto de vista, rasterizar sus bordes, etc.

Para parsear el texto hemos hecho uso de la clase *text*, en la que se facilita la tarea de leer el archivo de texto y se nos permite obtener un código más limpio.

Empezamos la explicación de nuestra práctica en el *main*, con dos variables de entrada en las que escribiremos la dirección de nuestra imagen: una del input (fichero de texto con las posiciones de los vértices y polígonos), y la ruta output de salida de nuestra imagen final.

Creamos un objeto de tipo *Object*, y llamamos al constructor, pasándole la ruta de entrada del fichero y un vector en el que determinamos la posición del objeto dentro del mundo. Dentro del constructor de *Object*, le pasamos a *Mesh* el archivo para que lo trate.

En el constructor de *Mesh*, creamos unos vectores de *Vectors* y de *Polygons* en los que guardaremos la información del fichero que trataremos, y otro para guardar las posiciones en pantalla de los vértices. A continuación, llamamos a la función *parseFile, la cual* hace uso de la clase *text* y trata el fichero.

Una vez leido el fichero, generamos la matriz del objeto haciendo uso de los métodos setPosition y setRotation que hay la clase Matrix.

Una vez ya tenemos toda esta información, seguimos en el main y lo que hacemos es añadir la cámara, que como hemos comentado, nos permitirá ver el objeto y enfocarlo hacia el objeto. Para ello, creamos un objeto de tipo *Camera*, pasándole los vectores posición (con sus componentes x,y,z en la que estará la cámara), lookat (en el que le decimos a donde mirar), la dirección de salida de la imagen resultante, y el ancho y largo de la pantalla (imagen).

Lo primero que hacemos es definir la matriz de perspectiva, haciendo uso de los valores (arbitrarios) DISTPLANEC y DISTFARPLANE. Después, generamos la matriz de vista haciendo uso de los vectores C, N, U y V. El vector C es la posición x,y,z de la posición de la cámara, el vector N (normal del plano, o dicho de otro modo hacia dónde mira la cámara) lo calculamos mediante la resta normalizada del punto hacia donde mira la cámara y la posición de ésta. Para calcular el vector V, multiplicamos esta normal por el vector (0,1,0), y para calcular la U, lo mismo pero multiplicado esta vez por (0,0,1). Una vez tenemos los vectores calculados, hacemos uso de setCUVN para configurar la matriz de vista RT. De no existir estas matrices, no podríamos mover ni girar la cámara.

Con todo esto, ya tenemos suficiente información para renderizar. *Render* es una función dentro de la clase *Camera*, y con varios procedimientos lo que hacemos es renderizar, es decir, pasar el objeto de mundo a pantalla. Primero vemos en qué coordenadas de pantalla se encuentran los vértices: empezamos cogiendo los vértices del objeto, cuyas coordenadas son locales, y multiplicándolos por la matriz del objeto para obtener sus posiciones respecto al mundo, después lo multiplicamos por las matrices de perspectiva y de vista, obteniendo así las coordenadas en pantalla del vértice. Por último, comprobamos que esos puntos que hemos obtenido (tendrán componente x e y) no se salgan de pantalla. Si todo va bien, pintamos los vértices en pantalla.

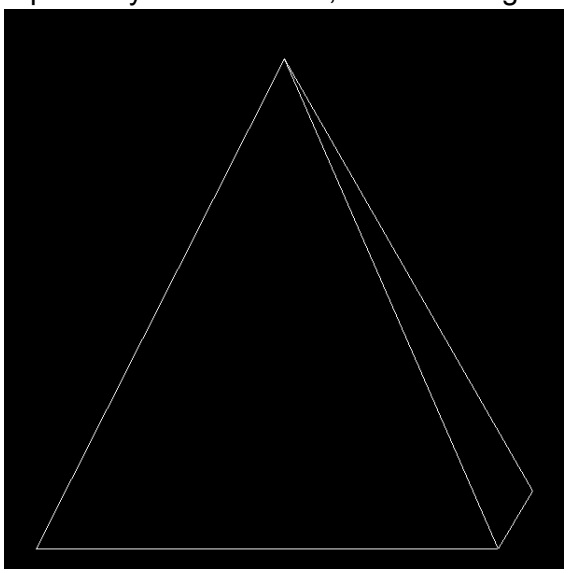
Una vez tenemos los vértices, continuamos con la función *rasterizePolygons*, empezamos recuperando la información que tenemos guardada, con las posiciones en pantalla de los vértices. Vamos llamando a la función *rasterize*, pasándole las posiciones de los vértices que queremos unir. Lo que hace la función *rasterize* son todos los cálculos correspondientes a pintar las líneas de unión entre los vértices. Como conocemos las normales de los polígonos, podemos hacer backface culling y renderizar sólo aquellos polígonos que son visibles desde la cámara, es tan sencillo como hacer el producto escalar entre la normal de los polígonos y la de la cámara; si ésta es superior a 0, las normales tienen un ángulo superior a 90º y por lo tanto el polígono no es visible.

Hemos usado 2 algoritmos para la rasterización: el *midpoint* y el de *Swanson y Thayer*. A la función se le pasan esas posiciones en forma de vectores, que definirán el punto start desde que se empieza el rasterizado y el end. Como comentamos en el código, el procedimiento para hacer el raster es el siguiente: primero de todo miramos si la línea a pintar es horizontal o vertical, para usar el algoritmo básico si es así. Como los algoritmos de rasterización funcionan de arriba a abajo, nos aseguramos de que start esté por encima de end en su componente ‘y’ y si no es así, los invertimos. Después, calculamos el ángulo que forma la línea con respecto a la horizontal. El algoritmo *midpoint* sólo funciona entre los ángulos 0/45º y 180/225º, y con algunas modificaciones nuestras lo hemos hecho funcionar de -45/45º y de 135/225º. De forma muy oportuna, el algoritmo de Swanson and Thayer funciona en todos los ángulos excepto en estos, así que sólo es cuestión de utilizar el algoritmo adecuado según el ángulo de la línea que queremos pintar.

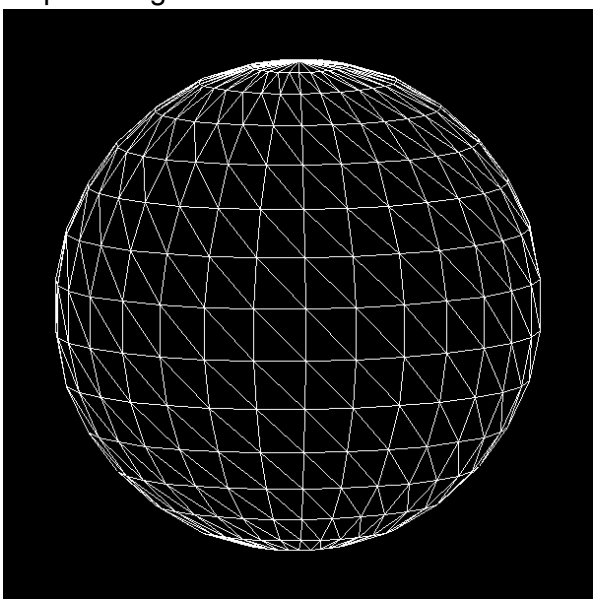
En el método render hay, comentadas, unas líneas que permiten testear el método de rasterizado, generando una imagen con líneas en todas las direcciones.

Experimentos

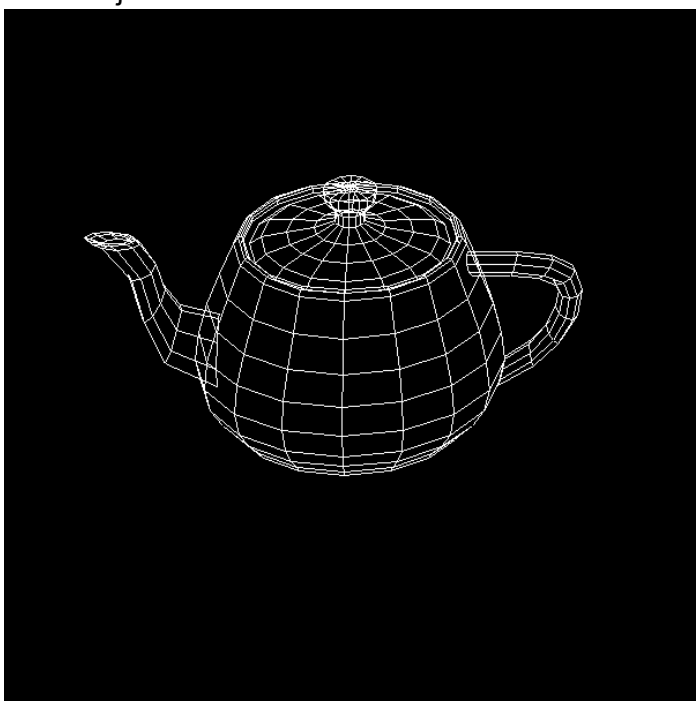
En esta imagen vemos una pirámide de base cuadrada. El algoritmo muestra las caras visibles de la pirámide y elimina los puntos y caras ocultas, con el culling.



En este otro ejemplo vemos como queda la entrada de la esfera (con 422 vértices y 840 polígonos) una vez pasado por el algoritmo.



Por último, vemos como quedaría la entrada que construye una tetera una vez pasado por el algoritmo y rotada 90º en el eje z.



Conclusiones

Aunque intensa, ha sido una práctica extremadamente interesante. Estamos muy satisfechos con el resultado, pero nos habría gustado tener tiempo de implementar las partes más complejas de la práctica como el shading y el z-buffer. Lo cierto es que esta práctica nos ha presentado mucho contenido que no habíamos visto nunca, pero hemos sabido solventar los fallos que nos hemos encontrado por el camino.