



Ejercicios Tema 4 – parte 2 (Shaders de geometría)

Materia de consulta

Se recomienda repasar los siguientes recursos online antes de comenzar a resolver los ejercicios.

- Diapositivas del tema 4 de la asignatura, diapos: 38-63.
- Video explicativo de las diapositivas 38-63 del tema 4.
(<https://aulavirtual.uv.es/mod/kalvidres/view.php?id=583929>)
- Red book OpenGL Programming guide, 8th Edition, pag: 623-649.
(<https://www.cs.utexas.edu/users/fussell/courses/cs354/handouts/Addison.Wesley.OpenGL.Programming.Guide.8th.Edition.Mar.2013.ISBN.0321773039.pdf>)
- Graphics Shaders : Theory and Practice, 2nd Edition, pag: 291-313.
Disponible online a través de VPN en trobes +, Servei de biblioteques de la UV
(<https://ebookcentral.proquest.com/lib/univalencia/detail.action?docID=830238>)
- OpenGL 4 Shading Language Cookbook, 2nd Edition, pag: 215-242
Disponible online a través de VPN en trobes +, Servei de biblioteques de la UV
(<https://ebookcentral.proquest.com/lib/univalencia/detail.action?docID=1441782>)

Ejercicios Tema 4 – parte 2

Crea un proyecto en Microsoft Visual Studio e incluye en él los ficheros de la carpeta parte 2; compila y ejecuta el programa. En este ejemplo, se dibuja la tetera, asignándole como colores a los vértices la normal en valor absoluto.

2.1 - Shader de geometría: cambio del tipo de primitiva.

En este ejercicio hay que conseguir convertir los triángulos de la tetera en puntos usando un shader de geometría. Para conseguirlo, dentro del shader de geometría obtenemos el baricentro del triángulo y emitimos como primitiva de salida ese punto.

Para resolver el ejercicio hay que modificar el código de los siguientes ficheros:

- *Código de la aplicación* (función “init”): para incluir dentro del objeto programa, además de los shaders de vértice y fragmento, el shader de geometría.
- *Código del shader de vértice*: Guardamos en la variable `gl_Position` la posición del vértice en el S.R. de la vista (en lugar del espacio de la ventana, como está ahora). Y le pasamos al shader de geometría el color del vértice y su normal en el S.R. vista (en este sistema de referencia, el eje z es siempre perpendicular a la pantalla).



- *Código del shader de geometría:* al principio del código hay que indicar el tipo de primitiva de entrada (en este caso, triángulos) y el de salida (puntos). Dentro del `main`, obtenemos la posición media de los tres vértices del triángulo (el baricentro), y esta posición transformada al espacio de la ventana (multiplicada por la matriz de proyección) será la posición del punto que emitiremos como primitiva de salida. Antes de emitir dicho vértice, fijaremos, además, como color para ese vértice el color medio de los tres vértices del triángulo de entrada.
- *Código del shader de fragmento:* Recogemos como dato de entrada (`in`) el dato proporcionado por el shader de geometría como dato de salida (el color del vértice) y le asignamos como color al fragmento ese valor.

2.2 - Shader de geometría: simulación de explosión del objeto.

Vamos a modificar la aplicación para que cuando el usuario pulse la tecla 'E' los puntos del objeto se desplacen, de forma gradual, de su posición, en la dirección de su normal (y cuando se vuelva a pulsar la tecla 'E' vuelvan a su posición inicial).

Para ello, hay que modificar el código de los siguientes ficheros:

- *Código de la aplicación:* Para activar esta opción por teclado y para que dentro de la función `display` se le pase a la variable uniform `uExplosionFactor` (declarada en el shader de geometría) un valor que se vaya incrementado en cada frame una cierta cantidad (la cantidad `inc_explosion`, declarada como constante al inicio de `display`).
- *Código del shader de geometría:* obtenemos la media de las normales de los tres vértices del triángulo y desplazamos la posición del punto en esa dirección la cantidad fijada en la variable uniform `uExplosionFactor`.

2.3 - Shader de geometría: impostores.

En este ejercicio hay que conseguir que cuando el usuario pulse la tecla 'I' se representen los puntos como esferas. Para aligerar la carga gráfica, se utilizará una técnica conocida como "impostores" que, en este caso, consiste en dibujar sobre un plano orientado siempre perpendicularmente a la dirección de la vista (es decir, paralelo al plano de la pantalla) una textura con la imagen de una esfera.

Para conseguir el anterior objetivo vamos a utilizar otro shader de geometría, con el cual convertiremos los puntos en strips de triángulos (2 triángulos) formando un plano, que se orientará perpendicularmente a la vista, y en el shader de fragmento pegaremos sobre el plano la textura con la imagen de la esfera.

Nota: Debemos emplear un shader de geometría distinto del anterior porque la primitiva de salida es distinta de la del shader anterior. Ahora debemos generar como primitiva de salida tiras de triángulo, mientras que antes la primitiva era puntos.

Nota: También tendremos que preparar la aplicación para que, cuando se active la opción con la tecla 'E', se desplacen las posiciones de las esferas igual que antes con los puntos.


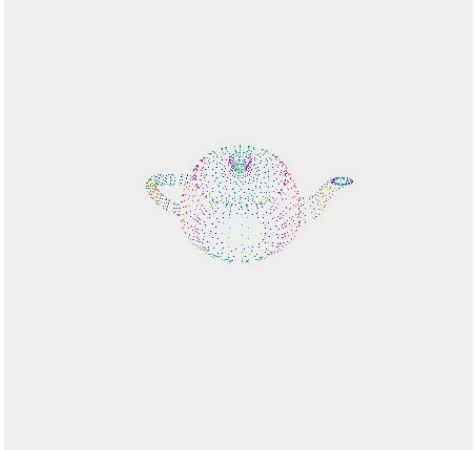
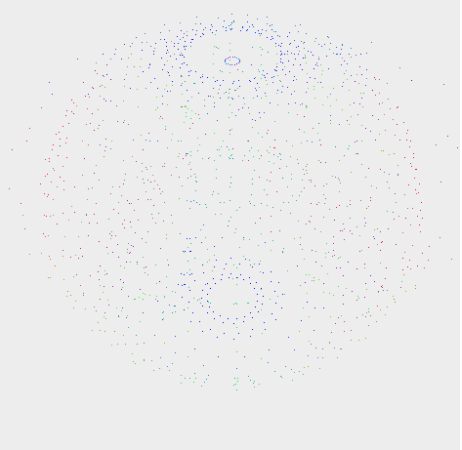

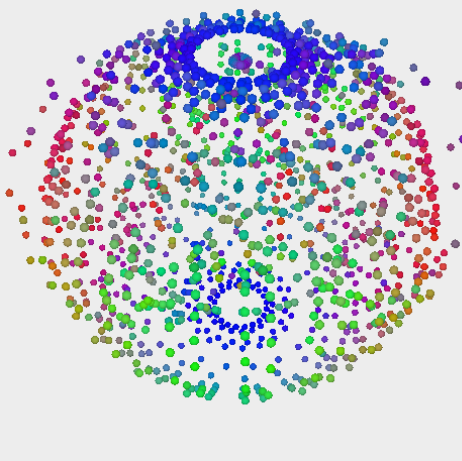


Para ello, hay que modificar el código de los siguientes ficheros:

- *Código de la aplicación:* En la función `init`, debemos crear otro objeto programa, aparte del anterior, para incluir en él el nuevo shader de geometría (además de un nuevo shader de vértice y otro de fragmento). También, dentro de la función `init`, debemos cargar la imagen de la esfera. Ya, dentro de la función `display`, activamos uno u otro objeto programa en función de si está o no activa la opción de los impostores y le pasamos al shader de fragmento la unidad de textura donde está la textura con la imagen de esfera.
- *Código del nuevo shader de vértice:* Será idéntico al anterior shader de vértice.
- *Código del nuevo shader de geometría:* Ahora hay que indicar que el tipo de primitiva salida es "strip de triángulos". Dentro del `main`, generaremos los vértices de la strip. Para ello Partimos de la posición del baricentro en el S.R. vista. En este sistema de referencia, los vértices de la strip se obtienen desplazando las coordenadas `x` e `y` del punto, la cantidad `uSize2` declarada en el código como `constate`, manteniendo fija la coordenada `z` (para que el plano resultante sea perpendicular a la dirección de la cámara). Para cada uno de los vértices de la strip, además de su posición, fijamos el color como antes (le asignamos a los cuatro vértices el mismo color) y, además, fijamos las coordenadas de textura de cada uno de los cuatro vértices para que la imagen completa de la esfera se pegue en la tira.
- *Código del nuevo shader de fragmento:* Recogemos como datos de entrada los datos proporcionado por el shader de geometría. Accedemos al texel correspondiente de la textura con la imagen de la esfera y lo multiplicamos por el color que se le pasa desde el shader de geometría. El resultado del producto será el color final del fragmento.
Nota: Para conseguir la forma de la esfera, dentro del shader de fragmento, debemos descartar el fragmento cuando estemos seguro que éste cae fuera del dominio de la esfera; cosa que podemos averiguar comprobando si sus coordenadas de textura se desplazan del centro de la textura (de las coordenadas (0.5, 0.5)) una cantidad superior al radio de la esfera (que es igual 0.38, en el espacio de la textura).



Resultados

	
Tetera coloreada	
	
Ejercicio 2.1 (nube de puntos)	Ejercicio 2.2 (explosión de la nube)
	
Ejercicio 2.3 (sustituyendo los puntos por esferas)	Ejercicio 2.3 (explosión con esferas)