Tema-5-Resumen.pdf

LosCocos

Informática Gráfica

3º Grado en Ingeniería Informática

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación
Universidad de Granada

WUOLAH

El más PRO del lugar puedes ser Tú.

¿Quieres eliminar toda la publi de tus apuntes?

¡Fuera Publi!
Concéntrate al máximo

Apuntes a full.
Sin publi y sin gastar coins

Para los amantes de la inmediatez, para los que no desperdician ni un solo segundo de su tiempo o para los que dejan todo para el último día.

Quiero ser PRO

4,95 / mes

TEMA 5: INTRODUCCIÓN A LA PROGRAMACIÓN EN GPU

Las primeras GPUs estaban optimizadas para coger un conjunto de coordenadas, colores, texturas... y generar una imagen de manera eficiente.
Con el tiempo, las GPUs se han hecho cada vez más programables, y esos pequeños programas, los shaders, se pueden aplicar a todos los triángulos, vértices o píxeles, consiguiendo resultados más complejos.

OPTIMIZACIÓN DEL RENDERING MEDIANTE LA PROGRAMACIÓN DE SHADERS EN GPU

Los shaders en OPENGL están escritos en lenguaje GLSL. Un lenguaje basado en C pero adaptado para que encaje con los procesadores gráficos. El mismo OpenGL lo compila. La idea es:

- El código fuente del shader se trata de una cadena de texto que se guarda en una variable.
- Esta variable y todos los shaders en su variable se compilan
- Se crea un programa con todos los shader juntos.

Podemos decir que el vertex shader se encarga de la geometría, y el fragment shader, de generar la imagen final.

VERTEX SHADER

El vertex shader se encarga de recoger la info de los vértices del sistema OpenGL y pasarlos a variables uniform o atributo para poder realizar cálculos con ellas.

- Transformaciones de vértices
- transformaciones de normales
- Iluminación
- Gestión de coordenadas de textura

Podemos decir que el vertex shader se encarga de tomar todas las variables atributo y pasadas o copiarlas en variables de salida para ser usadas por otros shaders
Tipos de variables:

- Variables atributo: son las propiedades del vértice que se está procesando y son solo de lectura para el vertex shader.

- Variables uniformes: Son las que persisten durante todo el procesamiento y son de solo lectura.
  - Matrices originales y derivadas de OpenGL
  - Propiedades de la luz y materiales
  - etc.
- Variables de salida: son el resultado del procesamiento y son de sólo escritura. Algunas como gl_Position o gl_PointSize, son obligatorias.

FRAGMENT SHADER

Son las que actúan sobre cada píxel para determinar su color:
- Cálculo de color
- Texturización
- ~~Ilu~~ Iluminación
- etc.

Tipos de variables:
- Uniformes
- De entrada: deben ser las mismas que las de salida del vertex shader
  - El 'color' se calcula por medio de interpolación de los colores de cada vértice, o interpolando las coordenadas de textura, usando la textura para calcular el color. Una vez calculado, se almacena en gFragColor
  - Si se utilizan texturas, podemos calcular las coordenadas de textura de cada vértice y almacenarlas en vST.
- De salida: debe contener al menos la variable de salida del color del pixel, almacenada como: out vec4 gFragColor

IMPORTANTE: los nombres y tipos de los parámetros de las variables de in del fragment shader deben COINCIDIR con las variables del out del vertex shader

ESQUEMA GENERAL DE UNA APLICACIÓN OPENGL 4.x

Cuando se usan shaders, no solo hay que proporcionar el código de estos.

Hay que hacer unos pasos:

1. Crear el código fuente de los shaders y almacenarlo en ficheros.
2. Leer los archivos en una cadena de texto. → `textFileRead(); char *vs = vs;`
3. Crear un objeto shader para cada cadena. → `glCreateShader(GL_VERTEX_SHADER);`
4. Asignar a cada objeto su código fuente asociado → `glShaderSource(v, 1, &vv, NULL);`
5. Compilar cada objeto → `glCompileShader(v)`
6. Crear programa shader global → `glCreateProgram(); p;`
7. Conectar todos los shaders con ese programa `glAttachShader(p, v);`
8. Enlazar los shaders `glLinkProgram(p)` `glUseProgram(p)`
9. Avisar de que vamos a usar shaders y no el pipeline fijo

SHADERS

VERTEX SHADER

Variables uniformes

```
uniform mat4 uModelViewMatrix; // Matriz 4x4 ModelViews
uniform mat4 uModelViewProjectionMatrix; // Matriz 4x4 Modelview * Projection
uniform mat3 uNormalMatrix; // Matriz 3x3 de normales: inversa(ModelViews)
```

var. atributo (coord del vértice actual)

```
in vec4 aVertex; -> Vértices de cada punto (ENTRADA)
in vec4 aNormal; -> Normales (ENTRADA)
in vec4 aColor; -> Colores (ENTRADA)
```

var. salida (salida vertex shader)

```
out vec4 vColor; -> Colores (SALIDA)
out vec4 vMC position; -> Posición xyz del vértice (SALIDA)
out float vLightIntensity; -> Intensidad de la luz (SALIDA)
```

```
const vec3 LIGHTPOS = vec3(3., 5., 10.); -> Posición de la luz
void main(){
    vec3 transNorm = normalize(uNormalMatrix * aNormal); -> Normal en SCV
    vec3 ECposition = vec3(uModelViewMatrix * aVertex); -> Vertice en SCV
    vLightIntensity = dot(normalize(LIGHTPOS - ECposition), transNorm);
    • vLightIntensity = abs(vLightIntensity); -> Calcular intensidad de la luz
    • vColor = aColor; -> Color
    • vMCPosition = aVertex.xyz; -> Posición xyz del vértice
    • gl_Position = uModelviewProjectionMatrix * aVertex; -> posición 2D del vértice
}
```

gl_Position: Obligatoria (SALIDA)

FRAGMENT SHADER

```
in vec4 vColor; // Entrada color del vertice
in float vLightIntensity; // Entrada intensidad de la luz
out vec4 gFragColor; -> Salida color del pixel

void main(){
    gFragColor = vec4(vLightIntensity * vColor.rgb, 1.);
}
```

Cálculo del color: Intensidad x rgb

3 primeros elementos

SETSHADERS()

① Crear código fuente y guardarlo

```
void setShaders() {
    char *vs, *fs;

    v = glCreateShader(GL_VERTEX_SHADER);
    f = glCreateShader(GL_FRAGMENT_SHADER);          ] ③ Crear un objeto shader por cadena

    vs = textFileRead("verticesshader.vert");
    fs = textFileRead("fragmentshader.frag");        ] ② Leer archivos y guardarlos en una cadena de texto

    const char *vv = vs;
    const char *ff = fs;

    glShaderSource(v, 1, &vv, NULL);
    glShaderSource(f, 1, &ff, NULL);                 ] ④ Asociar los archivos fuente a cada objeto shader

    free(vs); free(fs);                              → Liberar espacio

    glCompileShader(v);
    glCompileShader(f);                              ] ⑤ Compilar los shaders

    p = glCreateProgram();                           ] ⑥ Crear el programa shader global
    glAttachShader(p, v); glAttachShader(p, f);      ] ⑦ Añadir shaders al programa

    glLinkProgram(p); glUseProgram(p);               → usar ese programa p
        ⑧ Enlazar shaders

    + (practicas.cc, main())
    :
    setShaders();                                    ] ⑨ Avisar de que usamos shaders en vez del pipeline fijo
}
```

=== VERTEX SHADER ===

```
layout (0) vec2 coord;
layout (1) in vec3 v_color;   → Entrada
layout (2) out vec3 f_color;  → salida color

void main (void) {
    gl_Position = vec4 (coord2d, 0.0, 1.0);  → Calcula posición OBLIGATORIO
} f_color = v_color;  → Calcula color
```

=== FIN VERTEX SHADER ===

```
/*****/
//Variables globales o atributos de la clase
/*****/
GLuint vbo_triangle, vbo_triangle_colors;
GLuint attribute_coord2d, attribute_v_color;  → declaro variable color
/*****/
//En la inicialización de recursos
/*****/

GLfloat triangle_colors[] = {...};
GlGenBuffers (1, &vbo_triangle_colors);
glBindBuffer (GL_ARRAY_BUFFER, vbo_triangle_colors);
glBufferData (GL_ARRAY_BUFFER, sizeof (triangle_color)*triangle_colors, GL_STATIC_DRAW);
/*****/
//En el método dibujar
/*****/
attribute_name = "v_color";
attribute_v_color = glGetAttribLocation (program, attribute_name);  → obtengo v_color de VERTEX SHADER del programa shader program.
if (attribute_v_color) == -1) {
    fprintf (stderr, "Could not bind attribute %s\n", attribute_name);
} return 0;

glEnableVertexAttribArray (attribute_v_color);  → lo activo
glBindBuffer (GL_ARRAY_BUFFER, vbo_triangle_colors);
glVertexAttribPointer (attribute_v_color, 3, GL_FLOAT, GL_FALSE, 0, 0);
```

↳ Especificamos variables atributo de los vértices como un array.