Informática Gráfica

Tema 5. Modelado y visualización avanzados.

Domingo Martín

Dpto. Lenguajes y Sistemas Informáticos
ETSI Informática y de Telecomunicación
Universidad de Granada

Curso 2013-14

Índice

Informática Gráfica
Tema 5. Modelado y visualización avanzados.

Introducción

- Hemos trabajado con el modo inmediato de OpenGL
- Este modo ha quedado obsoleto porque es lento (aunque es fácil para enseñar)

```
glBegin(GL_POINTS);
for( i= 0 ; i < Vertices.size() ; i++ ){
   glVertex3f(Vertices[i].x, Vertices[i].y, Vertices[i].z ) ;
}
```

- ¡Una llamada para cada vértice!

Solución $\rightarrow$ Pasar la información como bloques

Modo inmediato

```
void draw_line(_vertex4f &Color,float Line_width)
{
  glColor4fv((GLfloat *) &Color);
  glLineWidth(Line_width);
  glBegin(GL_TRIANGLES);
  glPolygonMode(GL_FRONT_AND_BACK,GL_LINE);
  for (unsigned int i=0;i<Triangles.size();i++){
    glVertex3fv((GLfloat *) &Vertices[Triangles[i]._0]);
    glVertex3fv((GLfloat *) &Vertices[Triangles[i]._1]);
    glVertex3fv((GLfloat *) &Vertices[Triangles[i]._2]);
  }
}
```

DrawArrays

- Los datos están en la memoría del cliente
- Los datos se copian a la memoria del servidor
- Los datos deben estar organizados para de tal manera que sean coherentes con el modo de dibujo seleccionado: por ejemplo, para cada triángulo sus tres vértices, aunque se repitan
- Hay que seguir los siguientes pasos:
  1. Habilitar el array
     glEnableClientState(...)
  2. Especificar el formato y la dirección de los datos
     glVertexPointer(...)
  3. Referenciar y dibujar los datos dependiendo del modo
     glDrawArrays(...)

DrawArrays

glEnableClientState(ARRAY)

- ARRAY
  - GL_VERTEX_ARRAY
  - GL_COLOR_ARRAY
  - GL_NORMAL_ARRAY
  - GL_TEXT_COORD_ARRAY
  - GL_SECONDARY_COLOR_ARRAY
  - GL_INDEX_ARRAY
  - GL_FOG_COORD_ARRAY
  - GL_EDGE_FLAG_ARRAY

DrawArrays

glVertexPointer(NUMERO,TIPO,DESPLAZAMIENTO,PUNTERO)

- NUMERO: número de coordenadas por vértice
- TIPO
  - GL_SHORT
  - GL_INT
  - GL_FLOAT
  - GL_DOUBLE
- DESPLAZAMIENTO: espacio entre vértices consecutivos
- PUNTERO: puntero a la posición donde están los datos

Hay instrucciones para cada tipo de array

- glColorPointer(...)
- glNormalPointer(...)
- glTextCoordPointer(...)
- ...

DrawArrays

glDrawArrays(MODO,PRIMERO,NUM_ELEMENTOS)

- MODO
  - GL_POINTS
  - GL_TRIANGLES
  - GL_LINES
  - ...
- PRIMERO: posición del primer elemento
- NUM_ELEMENTOS: número de elementos que se van a dibujar

DrawArrays

```
void draw_line_drawarrays(_vertex4f &Color,float Line_width)
{
  glColor4fv((GLfloat *) &Color);
  glLineWidth(Line_width);
  glPolygonMode(GL_FRONT_AND_BACK,GL_LINE);

  glEnableClientState(GL_VERTEX_ARRAY);

  glVertexPointer(3,GL_FLOAT,0,&Vertices_triangles[0]);

  glDrawArrays(GL_TRIANGLES,0,Vertices_triangles.size());
}
```

Vertices_triangles contiene cada uno de los vértices de cada uno de los triángulos, aunque aparezcan repetidos

DrawElements

Para evitar la repetición de vértices se recurre a los índices.

glDrawElements(MODO,NUM_ELEMENTS,TIPO,PUNTERO)

- MODO
  - GL_POINTS
  - GL_TRIANGLES
  - GL_LINES
  - ...
- NUM_ELEMENTS: número de vértices a usar
- TIPO
  - GL_UNSIGNED_BYTE
  - GL_UNSIGNED_SHORT
  - GL_UNSIGNED_INT
- PUNTERO: puntero a la posición donde están los datos de los índices

DrawElements

```
void draw_line_drawelements(_vertex4f &Color,float Line_width)
{
  glColor4fv((GLfloat *) &Color);
  glLineWidth(Line_width);
  glPolygonMode(GL_FRONT_AND_BACK,GL_LINE);

  glEnableClientState(GL_VERTEX_ARRAY);

  glVertexPointer(3,GL_FLOAT,0,&Vertices[0]);

  glDrawElements(GL_TRIANGLES,Triangles.size()*3,GL_UNSIGNED_INT,&
       Triangles[0]);
}
```

Buffer objects (DrawArrays)

- Con DrawArrays y DrawElements se copian los datos desde la memoria del cliente a la del servidor cuando se va a dibujar (memoria general a la memoria de la GPU)
- El copiar bloques de memoria evita la sobrecarga de las múltiples llamadas, pero también es costosa
- El ancho de banda de la memoria de la GPU es mucho mayor que el de la CPU:
  ¡288GB/s Nvidia GTX Titan vs 21.33 GB/s Intel Core i7 4770k!

Solución $\rightarrow$ Copiar una sóla vez los datos a la memoria del servidor (GPU) y usarlos repetidas veces

Buffer objects (DrawArrays)

- Para poder hacerlo se reservan buffers de memoria
- hay que seguir los siguientes pasos:
  1. Generar identificadores: glGenBuffers
  2. Activar el buffer: glBindBuffer
  3. Reservar el espacio y, opcionalmente, copiar la información: glBufferData
  4. Usar el procedimiento de DrawArrays o DrawElements para dibujar

Buffer objects (DrawArrays)

glGenBuffers(NUM_BUFFERS,PUNTERO)

- NUM_BUFFERS: número de identificadores de bufferes que se quieren reservar
- PUNTERO: posición donde se devuelven los valores

Los identificadores son números enteros

Buffer objects (DrawArrays)

glBindBuffer(OBJETIVO,IDENTIFICADOR)

- OBJETIVO
  1. GL_ARRAY_BUFFER: para los datos: vértices, colores, coordenadas de textura, etc.
  2. GL_ELEMENT_ARRAY_BUFFER: para los índices
  3. ...
- IDENTIFICADOR: indentificador del buffer

Buffer objects (DrawArrays)

glBufferData(OBJETIVO,TAMAÑO,PUNTERO,USO)

- OBJETIVO
  1. GL_ARRAY_BUFFER
  2. GL_ELEMENT_ARRAY_BUFFER
  3. ...
- TAMAÑO: cantida de datos que se van a copiar ¡en bytes!
- PUNTERO: puntero al vector de datos que se van a copiar
- USO: pista para el controlador para que pueda optimizar el uso de la memoria: por ejemplo, si los datos no van a cambiar, si se van a usar muchas o pocas veces, etc.

Buffer objects (DrawArrays)

```
void draw_line_bufferobjects_drawarrays(_vertex4f &Color,float Line_width
     )
{
  GLuint Buffer;

  glColor4fv((GLfloat *) &Color);
  glLineWidth(Line_width);
  glPolygonMode(GL_FRONT_AND_BACK,GL_LINE);

  glGenBuffers(1,Buffer);

  glBindBuffer(GL_ARRAY_BUFFER,Buffer);

  glBufferData(GL_ARRAY_BUFFER,Vertices_triangles.size()*3*sizeof(float)
       ,&Vertices_triangles[0],GL_STATIC_DRAW);

  glVertexPointer(3,GL_FLOAT,0,0);

  glEnableClientState(GL_VERTEX_ARRAY);

  glDrawArrays(GL_TRIANGLES,0,Vertices_triangles.size()*3);

  glDisableClientState(GL_VERTEX_ARRAY);
}
```

Buffer objects (DrawArrays)

- El anterior código contiene varios errores/fallos:
  1. Se genera un identificador en cada llamada
  2. Se copian los datos al buffer en cada llamada
  3. No se comprueba si había memoria para el buffer

En glVertexPointer se cambia el PUNTERO, ya no es necesario pues los datos están en el buffer

Buffer objects (DrawArrays)

```
glGenBuffers(1,Buffer);
Data_loaded=false;

void draw_line_bufferobjects_drawarrays(_vertex4f &Color,float Line_width
     )
{
  glColor4fv((GLfloat *) &Color);
  glLineWidth(Line_width);
  glPolygonMode(GL_FRONT_AND_BACK,GL_LINE);

  glBindBuffer(GL_ARRAY_BUFFER,Buffer);
  if (Data_loaded==false){
    glBufferData(GL_ARRAY_BUFFER,Vertices_triangles.size()*3*sizeof(float
         ),&Vertices_triangles[0],GL_STATIC_DRAW);
    if (glGetError()==GL_OUT_OF_MEMORY){
      cout << "Error: not enough memory" << endl;
      exit(-1);
    }
    Data_loaded=true;
  }
  glVertexPointer(3,GL_FLOAT,0,0);
  glEnableClientState(GL_VERTEX_ARRAY);
  glDrawArrays(GL_TRIANGLES,0,Vertices_triangles.size()*3);
  glDisableClientState(GL_VERTEX_ARRAY);
}
```

Buffer objects (DrawArrays)

- Resultados comparación (happy.ply:543652 vértices, 1087716 triángulos; se dibuja 100 veces seguidas para obtener el tiempo):
  1. Normal: 13.5 segundos
  2. DrawArrays: 3.6 segundos
  3. DrawElements: 4.3 segundos
  4. BufferObjects (DrawArrays): 0.5 segundos

Cauce gráfico

- OpenGL es una máquina de estados
- La información de entrada pasa a través de unos procesos-estados que van transformandola hasta convertirla en una imagen
- Estos procesos eran fijos hasta que apareció la programación con la verión 2.0
- Se cambia la funcionalidad fija por la flexibilidad de la programación
- En la versión 2.0 aparece los programas (shaders) que operan sobre los vértices y sobre los fragmentos

Cauce gráfico

- Cauce gráfico original (OpenGL 1.0 hasta 1.5)

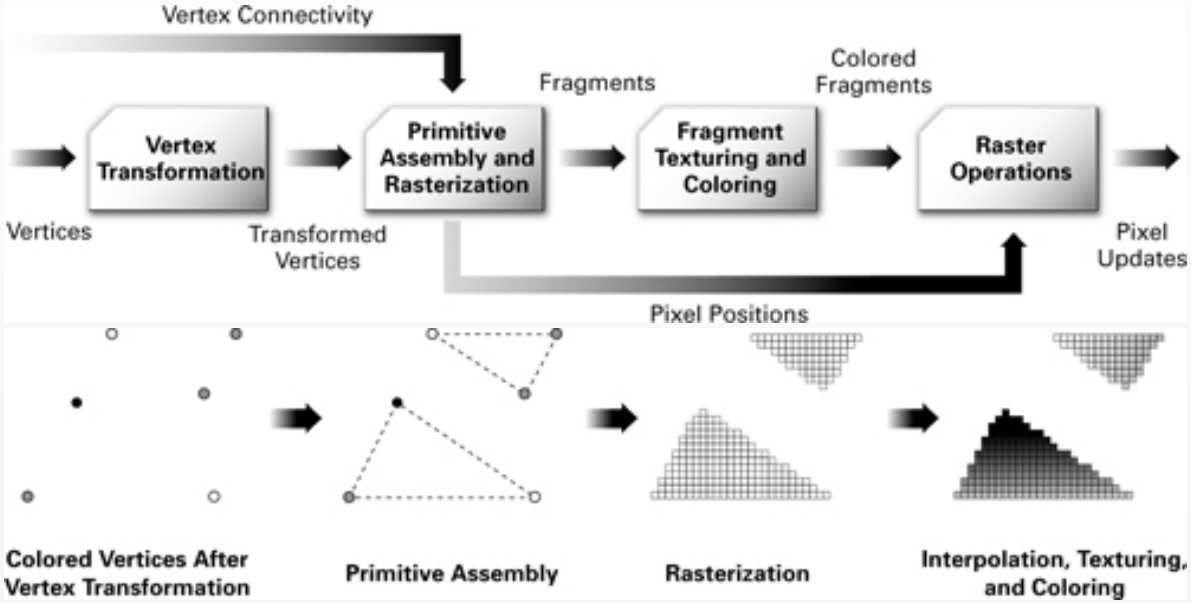

Figura: Nvidia©

Cauce gráfico

- Cauce gráfico OpenGL 2.0

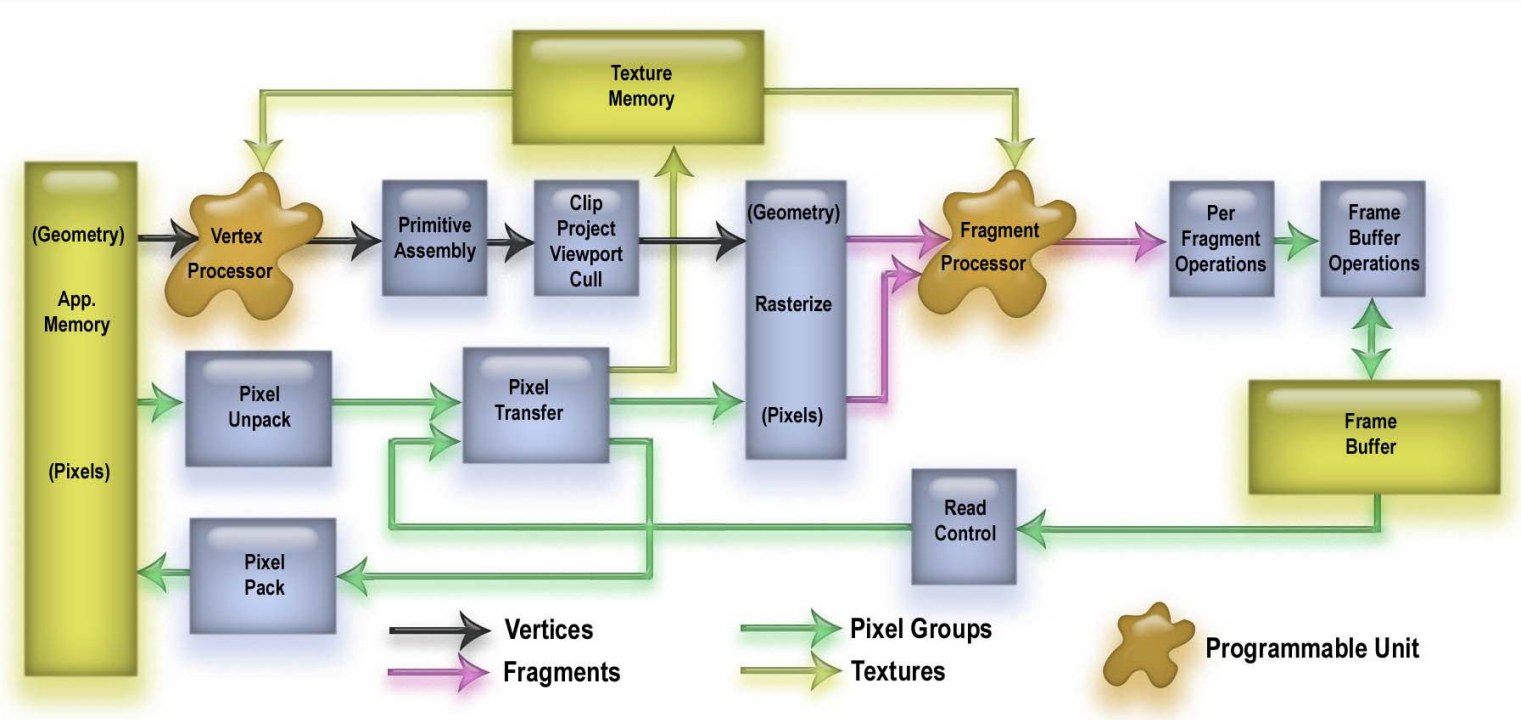

Figura: OpenGL Shading Language©

Cauce gráfico

- La funcionalidad fija perdida hay que obtenerla con programación
- Vertex shaders
  - Transformación de las coordenadas del vértice por la matriz Modelview
  - Transformación de las coordenadas del vértice por la matriz Projection
  - Transformación de las coordenadas del vértice por la matriz Texture
  - Transformación de las coordenadas de la normal a coordenadas de cámara
  - Reescalado y normalización de las normales
  - Generación de las coordenadas de textura
  - Cálculos de iluminación por vértice
  - Cálculos del material
  - Atenuación por distancia del tamaño del punto

Cauce gráfico

- La funcionalidad fija perdida hay que obtenerla con programación
- Vertex shaders
  - Funciones con texturas y entorno de texturas
  - Obtención del color
  - Niebla

Programación de los shaders

- Los shaders se programan en un lenguaje derivado de C y C++
- La secuencia de pasos para usar los shaders es la siguiente:
  1. Crear los programas (shaders)
  2. Crear los shaders con glCreateShader
  3. Proporcionar el código fuente de los shaders con glShaderSource
  4. Compilar cada shader con glCompileShader
  5. Crear un programa con glCreateProgram
  6. Adjuntar los shaders al programa con glAttachShader
  7. Enlazar el programa con glLinkProgram
  8. Activar el programa con glUseProgram
  9. Cargar los valores de las variables de los shaders
  10. Dibujar

Programación de los shaders

- Es necesario cargar la funcionalidad → GLEW

```
void initialize(void)
{
    const GLubyte* strm;
    strm = glGetString(GL_VENDOR);
    std::cerr << "Vendor: " << strm << "\n";
    strm = glGetString(GL_RENDERER);
    std::cerr << "Renderer: " << strm << "\n";
    strm = glGetString(GL_VERSION);
    std::cerr << "OpenGL Version: " << strm << "\n";
    if (strm[0] == '1'){
      std::cerr << "Only OpenGL 1.X supported!\n";
      exit(-1);
      }
    strm = glGetString(GL_SHADING_LANGUAGE_VERSION);
    std::cerr << "GLSL Version: " << strm << "\n";
    int err = glewInit();
    if (GLEW_OK != err){
      // Problem: glewInit failed, something is seriously wrong.
      std::cerr << "Error: " << glewGetErrorString(err) << "\n";
      exit (-1);
      }
    execute_shaders(File_vertex_shader,File_fragment_shader);
}
```

Programación de los shaders I

```
int execute_shaders(File_vertex_shader,File_fragment_shader)
{
  char *Vertex_shader_code=NULL;
  char *Fragment_shader_code=NULL;

// 1 Lectura de los programas
  read_file(File_vertex_shader,Vertex_shader_code);
  read_file(File_fragment_shader,Fragment_shader_code);

  // 2 Creación de los shaders
  Vertex_shader=glCreateShader(GL_VERTEX_SHADER);
  Fragment_shader=glCreateShader(GL_FRAGMENT_SHADER);

  // 3 Asignación del código fuente de los shaders a los shaders
  glShaderSource(Vertex_shader,1,(const GLchar **) &Vertex_shader_code,
       NULL);
  glShaderSource(Fragment_shader,1,(const GLchar **) &
       Fragment_shader_code,NULL);

  // 4 Compilación de los shaders
  glCompileShader(Vertex_shader);
  glCompileShader(Fragment_shader);
  ...
```

Programación de los shaders II

```
  ...
  // 5 Cración de un programa
  Program=glCreateProgram();

  // 6 Adjuntar los shaders al programa
  glAttachShader(Program,Vertex_shader);
  glAttachShader(Program,Fragment_shader);

  // 7 Enlazar
  glLinkProgram(Program);

  // 8 Activar el programa
  glUseProgram(Program);
}
```

Vertex shader

```
varying vec3 normal, lightDir;
varying vec4 material_d;

void main()
{
lightDir = normalize(vec3(gl_LightSource[0].position));
normal = normalize(gl_NormalMatrix * gl_Normal);
material_d = normalize (gl_FrontMaterial.diffuse);

gl_Position = ftransform();
}
```

Fragment shader

```
varying vec3 normal, lightDir;
varying vec4 material_d;

void main()
{
float intensity, r, g, b;
vec3 n;
vec4 color;
float sil;

n = normalize(normal);
intensity = max(dot(lightDir,n), 0.0);
sil = clamp(dot(vec3(0,0,1),n),0.0,1.0);
color = vec4(intensity);

// silueta
if (sil < 0.60) color = vec4(0,0,0,1.0);
else color = vec4(1,1,1,1);

// Líneas de forma
if (intensity > 0.90) color = vec4(2,2,2,1.0) * material_d;
if (intensity > 0.95) color = vec4(3,3,3,1.0) * material_d;

gl_FragColor = color * material_d;
}
```

Ejemplo