

A yellow pencil and a pink eraser are positioned in the top right corner of the slide, as if they are on a piece of paper.

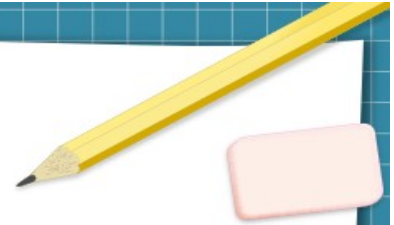
Introducción shaders

Nancy Hitschfeld Kahler
(nancy@dcc.uchile.cl)

Departamento de Ciencias de la Computación
Universidad de Chile

Contenido

- Historia y alcance
- GPUs como poder gráfico
- Pipeline gráfico
- Programando con shaders (GLSL)
 - Pipeline gráfico simple
 - Modelo iluminación de Phong
 - Pipeline gráfico actual



Introducción

- Early 2000s started for computer graphics (Video games)
 - GPU: Graphic parallel Unit
 - GLSL (OpenGL Shading Language)
 - Lighting, shading, geometry effects in real time
 - Massive paralelism designing per-vertex and per-fragment algorithms to work in a set of millions of primitives.



F. Cohen, P. Decaudin, and F. Neyret. GPU-based lighting and shadowing of complex natural scenes. In Siggraph'04 Conf. DVD-ROM (Poster), August 2004. Los Angeles, USA

Phong model

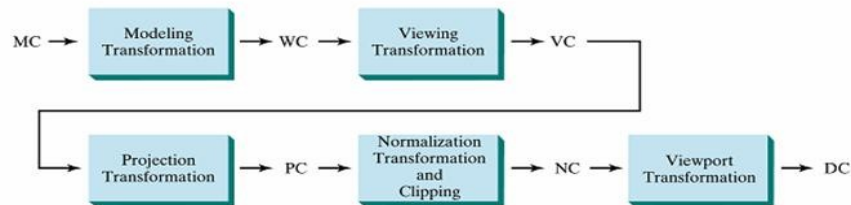
Introducción



- Por qué la comunidad científica se interesó en las Gpus?
 - Su bajo costo comparado con clusters, super-computers
- Evolución de los lenguajes
 - In 2002, McCool et al. published a paper detailing a meta-programming GPGPU language, named Sh
 - In 2004, Buck et al. proposed Brook for GPUs
 - In 2006, Nvidia proposed CUDA (Compute Unified Device Architecture)
 - In 2008, an open standard was released with the name of OpenCL (Open Computing Language), allowing the creation of multi-platform, massively parallel code

3D rendering pipeline

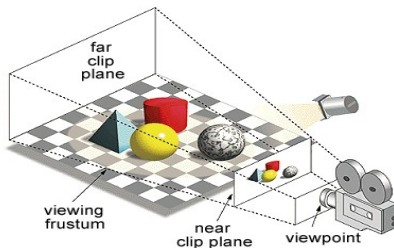
- Recordemos que, el proceso completo de rendering incluye las siguientes transformaciones:

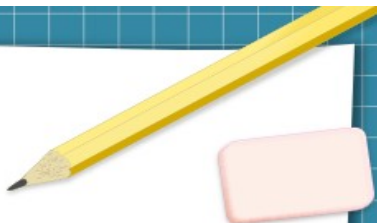


General three-dimensional transformation pipeline, from modeling coordinates to world coordinates to viewing coordinates to projection coordinates to normalized coordinates and, ultimately, to device coordinates.

Transformación de *Viewing* y proyección en 3D

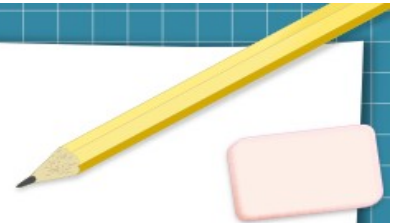
- ¿Por qué es más complejo que en 2D?
 - Especificar desde donde se mira la escena (dónde posicionamos y qué orientación le damos a la “cámara”)
 - Especificar proyección (paralela o perspectiva) a utilizar: Objetos en 3D deben ser proyectados en un plano 2D
 - Especificar el *view volume*, es decir qué parte de la escena se desea proyectar: sólo los objetos que están dentro de este volumen de visión son proyectados a dos dimensiones
 - El recorte de los objetos (algoritmos de clipping) se aplican sobre el volumen de visión



A yellow pencil and a pink eraser are positioned in the top right corner of the white paper, as if they are tools for writing or editing.

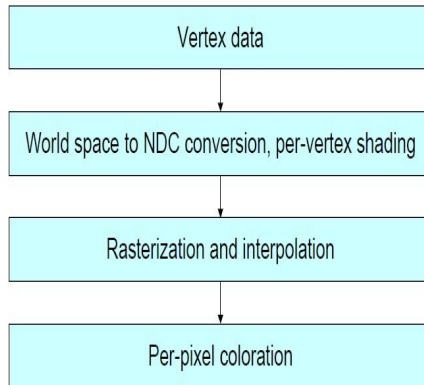
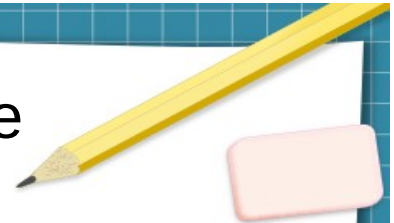
GPUS como poder gráfico
¿En qué consiste el lenguaje de shaders?

GLSL: alcance

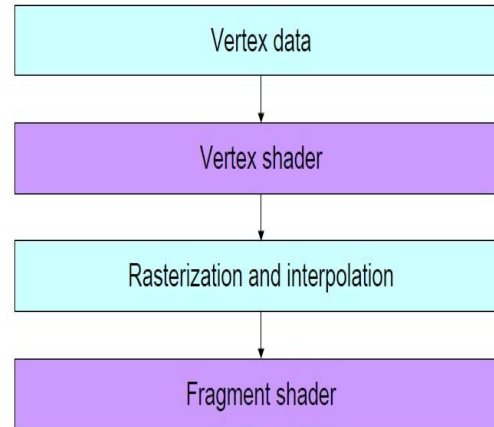


- Lenguaje para escribir programas sobre la GPU
- Permite programar varios tipos de shaders: **vertex shaders** y **fragment shaders** (los primeros)
 - También hay **geometry shaders** y **tessellation shaders**
- Estos shaders permiten **sobre-escribir partes específicas del pipeline gráfico**

Pipeline gráfico simple

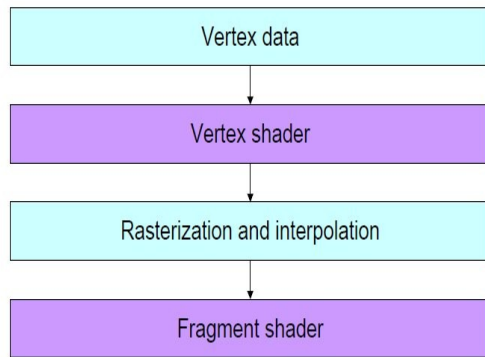
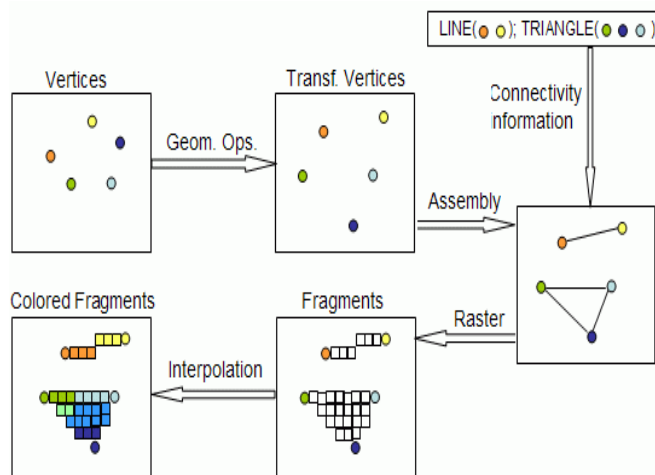


OpenGL (default)



OpenGL con shaders

Simple shader pipeline

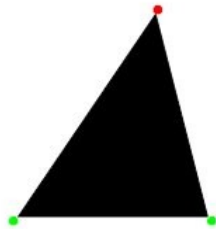


Shader pipeline

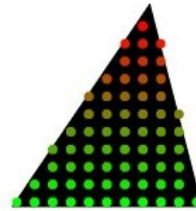


- **Vertex shader**: responsable de todos los cálculos sobre los vértices. Pipeline default:
 - Convertir vértices desde wc a ndc
 - Ejecutar cálculos de textura e iluminación sobre vértices
- **Fragment shaders**: responsables de todos los cálculos sobre pixeles
 - Antes de aplicar este shader, toda la información dada para los vértices es interpolada por el rasterizador
 - En el pipeline default:
 - Define (asigna) el color de un pixel como el valor de color interpolado desde los vértices

Llamadas a los vertex y fragment shaders



Vertex shader runs
once for every vertex



Fragment shader runs
once for every pixel
rendered in the scene,
with vertex data
interpolated

Sintaxis y programación en GLSL



- Sintaxis similar a C
 - `main()` es el punto de entrada a los vertex y fragment shader
 - Algún soporte extra para manejo de vectores y matrices
 - Muchas funciones y variables built-in:
 - <http://www.khronos.org/files/opengl-quick-reference-card.pdf>
- Tipos:
 - Floating points: `float`, `vec2`, `vec3`, `vec4`
 - Enteros: `int`, `ivec2`, `ivec3`, `ivec4`
 - Booleans: `bool`, `bvec2`, `bvec3`, `bvec4`
 - Matrices: `mat2`, `mat3`, `mat4`

Vertex shaders: variables built-in



- Inputs:
 - `gl_Vertex`: posición espacial del objeto en WC
 - `gl_Normal`: vector normal del objeto en ese punto
- Outputs:
 - `gl_FrontColor`: escribir el color del vértice en esta variable (o este será calculado por un fragment shader)
 - `gl_Position`: escribir posición en NDC

Fragment shaders: variables built-in



- Input:
 - `gl_Color`: color interpolado por vértice
- Output:
 - `gl_FragColor`: escribir el color del pixel aquí.

Funciones Built-in

- Algunas son:
 - `dot()`: producto punto entre vectores
 - `sin`, `cos`, `pow`, etc como en `math.h`
 - `ftransform()`: aplica transformación a NDC (viewing, proyección, etc)
 - `reflect()`: refleja un vector alrededor de otro
- Ver más en la quick-reference card

Comunicación entre shaders

- Qué pasa si un shader necesita más información que la que está disponible?
 - Usar palabras claves (keywords) especiales
- GPU ---> **in y out** ;
- CPU ---> a ambos (para render): **uniform**
- CPU ---> a ambos (por vertex): **attribute**
- CPU ---> a ambos (gran cantidad de info): **usar texturas**

A yellow pencil and a pink eraser are positioned in the top right corner of the slide, appearing to be on a piece of paper.

Ejemplo en c:
Modelo de iluminación considera
luz ambiental y difusa

Ejemplo de vertex shader



El mismo código se aplica a todos los vértices y para cada vértice se corre un thread (hilo) distinto (uso de la GPU)

Inputs: azul y output: rojo

- out int ambientEnabled;
 - out vec3 normal, lightDir;
 - void main() {
 - normal = normalize(gl_NormalMatrix * gl_Normal);
 - vec3 worldPos = vec3(gl_ModelViewMatrix * gl_Vertex);
 - lightDir = normalize(vec3(gl_LightSource[0].position) - worldPos);
 - gl_Position = ftransform();
- }

Ejemplo de fragment shader

El mismo código se aplica a todos los pixeles y para cada pixel se corre un thread distinto (uso de la GPU)

Inputs: azul y output: rojo

- **in** int **ambientEnabled**; // mismo del vertex shader
- **in** vec3 **normal**, **lightDir**; // mismo del vertex shader
- void main() {
 - vec4 color = vec4(0, 0, 0, 0);
 - if (ambientEnabled)
 color = gl_LightModel.ambient * gl_FrontMaterial.ambient;
 - float NdotL = max(dot(normalize(normal), normalize(lightDir)), 0.0);
 - color += (gl_FrontMaterial.diffuse * gl_LightSource[0].diffuse * NdotL);
 - **gl_FragColor** = color;}

A yellow pencil and a pink eraser are positioned in the top right corner of the slide, resting on a blue grid background.

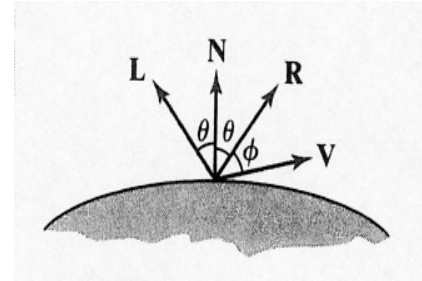
Ejemplo en c: Modelo de iluminación Phong con shaders

Phong illumination model with shaders

Vertex shader; *shading.vert*

```
uniform mat4 projection;
varying vec3 normal, light_dir, eye_dir;
void main(){
    normal = normalize(gl_NormalMatrix * gl_Normal);
    vec3 wp = vec3(gl_ModelViewMatrix * gl_Vertex);
    light_dir = normalize(vec3(gl_LightSource[0].position) -
wp);
    eye_dir = -wp;
    gl_Position = projection * gl_ModelViewMatrix * gl_Vertex;
}
```

$$I = I_a K_a + I_p K_d N \cdot L + I_p K_s (R \cdot V)^n$$



Fragment shader: *shading.frag*

```
varying vec3 normal, light_dir, eye_dir; // same as in vertex shader
void main(){
    vec4 color = gl_LightModel.ambient * gl_FrontMaterial.ambient;
    float lambert = max(dot(normalize(normal), normalize(light_dir)), 0.0);
    color += (gl_FrontMaterial.diffuse * gl_LightSource[0].diffuse * lambert);
    float reye = max(dot(reflect(-normalize(light_dir), normalize(normal)),
normalize(eye_dir)), 0.0);
    float i_s = pow( reye, gl_FrontMaterial.shininess );
    color += gl_LightSource[0].specular * gl_FrontMaterial.specular * i_s;
    gl_FragColor = color;
}
```

Phong illumination with shaders

- Mirar código: [shading.c](#), [shading.h](#), [tools.h](#), [shading.frag](#), [shading.vert](#)

- main (shading.c)
- init_all (Shading.h)
 - glutDisplayFuction(display_gl)
 - `dprog = make_shader_program(----)`
(tools.h)

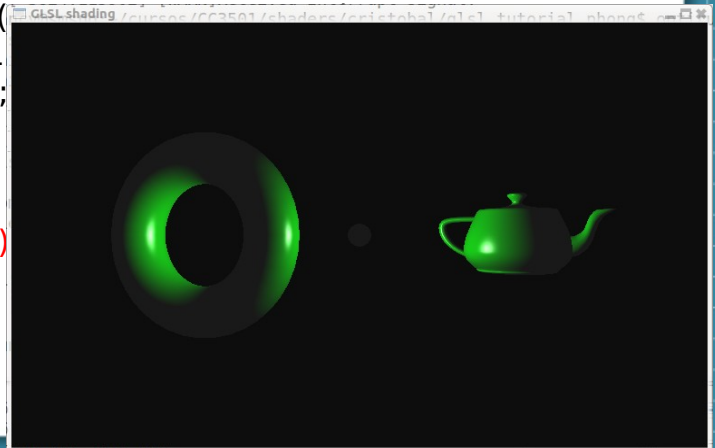
```
vs = glCreateShader(GL_VERTEX_SHADER);  
glShaderSource(vs, 1, &vs_src, &vs_len);  
glCompileShader(vs);
```

.....

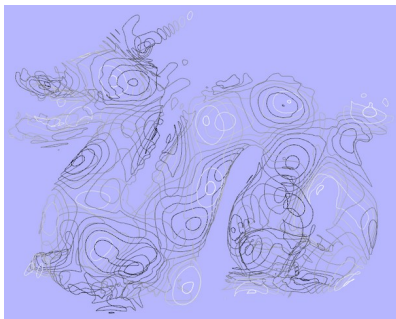
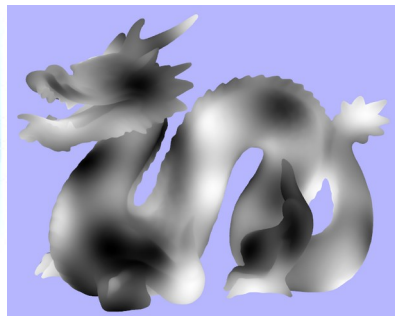
- display_gl() (shaing.h)
 - render_gsl(dprog)
 - `glUseProgram(dprog)`
 - `GlutSolidTorus ...`
 - `glUseProgram(0)`

What is very difficult?

- *Debugging*



Pipeline actual con shaders



Camarón

- Aldo Canepa et al. Camarón: An Open-source *Visualization Tool for the Quality Inspection of Polygonal and Polyhedral Meshes*. VISIGRAPP (1: GRAPP)2016: 130-137.

