Introduction to Shader (GPU) Programming

Background

- We expect visual effect like reality

Background

- 1998 — The theory of GPU is advanced
- 2001 — video card includes programmable vertex shader
- 2002 — fragment shader
- 2006 — geometry shader
- Later — GPGPU technology: CUDA, ATI Stream, DirectX11 Computer Shader...
Hardware and software

- OpenGL Shading Language (GLSL)
  - Developed by OpenGL ARB
- High Level Shading Language (HLSL)
  - Developed by Microsoft
- C for Graphics (Cg)
  - Developed by Nvidia

API for Shader Programming

Programming Structure

- glViewport
- glutPerspective/gluOrtho2D
- gluLookAt
- glTranslate/glRotatef
- glVertex/glNormal/glTexCoord
Programming Structure

- Context
- Profile
  - Vertex & Fragment
- Program
  - Vertex & Fragment
- Parameters

Programming Structure

1. Get ModelViewProj Matrix
   - Convert from Model Coordinate to Projection Coordinate
2. Set parameters for Cg program
   - The matrix, and texture ID
3. Enable GPU
4. Run the normal core drawing procedure
   - Plot the model and assign its texture coordinates
5. Disable GPU

Programming Structure

```
struct Output {
    float4 position : POSITION;
    float3 color    : COLOR;
    float2 texCoord : TEXCOORD0;
};

Output vertex_passthrou(float4 position : POSITION,
    float3 color    : COLOR,
    float2 texCoord : TEXCOORD0, //multiple textures is OK
    uniform float4x4 modelViewProj)
{
    Output OUT;
    OUT.position = mul(modelViewProj, position);
    OUT.color    = color;
    OUT.texCoord = texCoord;
    return OUT;
}
```
Programming Structure

• TRIANGLE void geometry_passthru(AttribArray<float4> position : POSITION,
  AttribArray<float4> color    : COLOR)
  {
    for (int i=0; i<position.length; i++) {
      emitVertex(position[i], color[i]);
    }
  }

• float4 C3E3f_texture(float2 texCoord : TEXCOORD0,
  uniform sampler2D decal : TEX0)
  {
    float4 OUT;
    OUT = tex2D(decal,texCoord);
    return OUT;
  }

Conclusion

How much do we rely on OpenGL now?
  Texture data and its coordinate
  +
  Vertices coordinates
  +
  Transformation matrices (optional)

How to make profit from GPU?
  Enrich the rendering process in GPU
VA, VBO & VAO

- **VA: Vertex Array**
  - To prepare the data in an array and call 1 function to draw it.
  - Reduces the # of function calls and redundant usage of shared vertices.

- **VBO: Vertex Buffer Object**
  - Similar to VA, but now the data transfers to GPU video memory.
  - Have to create separate VBOs for vertex and texture coordinates.

- **VAO: Vertex Array Object**
  - Solution to store multiple VBOs in one VAO.
  - For instance, vertex and texture VBO can now be set into one VAO.

---

**Example**

```c
GenBuffers(sizei n, uint *buffers)
Generates a new VBO and returns its ID number as an unsigned integer. Id 0 i
s reserved.

BindBuffer(enum target, uint buffer)
Use a previously created buffer as the active VBO.

BufferData(enum target, sizeiptrARB size, const void *data, enum usage)
Upload data to the active VBO.

DeleteBuffers(sizei n, const uint *buffers)
Deletes the specified number of VBOs from the supplied array or VBO id.
```

---

```c
//Initialise VBO - do only once, at start of program
//Create a variable to hold the VBO identifier
GLuint triangleVBO;

//Vertices of a triangle (counter-clockwise winding)
float data[] = {1.0, 0.0, 1.0, 0.0, 0.0, -1.0, -1.0, 0.0, 1.0};

//Create a new VBO and use the variable id to store the VBO id
glGenBuffers(1, &triangleVBO);

//Make the new VBO active
//Upload vertex data to the video device
//Actually draw the triangle using the VBO. Repeat here incase changed since initialisation
//Force display to be drawn now
```