GPU Programming and graphics pipeline
The graphics pipeline

**Geometry:** objects, surfaces, light sources...

**Lighting/shading model**

**Camera:** viewpoint, direction, field-of-view (frustum)

**Window (viewport):** pixel grid where the picture is displayed

**Modeling Transformations**

**Illumination** (Shading)

**Viewing Transformation** (Perspective / Orthographic)

**Clipping**

**Projection** (to Screen Space)

**Scan Conversion** (Rasterization)

**Visibility / Display**

**Colors, intensities tailored for display** (ex: 24 bits, RVB)
Object transformations

- From each object’s **local coordinate system** (object space) to a **global coordinate system** (world space)

```
Object space          World space
```

<table>
<thead>
<tr>
<th>Modeling Transformations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Illumination (Shading)</td>
</tr>
<tr>
<td>Viewing Transformation (Perspective / Orthographic)</td>
</tr>
<tr>
<td>Clipping</td>
</tr>
<tr>
<td>Projection (to Screen Space)</td>
</tr>
<tr>
<td>Scan Conversion (Rasterization)</td>
</tr>
<tr>
<td>Visibility / Display</td>
</tr>
</tbody>
</table>
Lighting/shading

- Each primitive is shaded depending on its material and the light sources.
- Local illumination only (no shadows), independent computation for each primitive
Camera transformation

- From world coordinate system to viewpoint (eye space).

`Modeling Transformations`

`Illumination (Shading)`

`Viewing Transformation (Perspective / Orthographic)`

`Clipping`

`Projection (to Screen Space)`

`Scan Conversion (Rasterization)`

`Visibility / Display`

Eye space

World space
Clipping

- Normalized coordinates:

  ![Normalized coordinates diagram]

  - Anything outside the viewing frustum is clipped:

  ![Clipping example]

- Modeling Transformations
- Illumination (Shading)
- Viewing Transformation (Perspective / Orthographic)
- Clipping
- Projection (to Screen Space)
- Scan Conversion (Rasterization)
- Visibility / Display
Projection

- 3D primitives are projected onto a 2D picture (screen space)

Modeling Transformations

Illumination (Shading)

Viewing Transformation (Perspective / Orthographic)

Clipping

Projection (to Screen Space)

Scan Conversion (Rasterization)

Visibility / Display
Rasterization

- Convert the 2D primitive in pixels
- Interpolate values known at the vertices (color, depth...)

![Diagram of rasterization process]

- Modeling Transformations
- Illumination (Shading)
- Viewing Transformation (Perspective / Orthographic)
- Clipping
- Projection (to Screen Space)
- Scan Conversion (Rasterization)
- Visibility / Display
Visibility and display

- Hidden surface removal
- Filling the frame buffer with the right color format
What is a GPU?

- “Graphics Processing Unit”
- Specialized processor for graphics rendering
- Spécificities:
  - Highly parallel (SIMD)
  - Fast local memory
  - Large throughput
What is a GPU?

- Highly efficient parallel processor:
  - GPGPU: “General-Purpose computation on GPU”
Before graphics hardware (1970s)
### Graphics pipeline

1st generation graphics hardware (1980s)

<table>
<thead>
<tr>
<th>Software configurable</th>
<th>Hardware (GPU)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modeling Transformations</td>
<td>Scan Conversion (Rasterization)</td>
</tr>
<tr>
<td>Illumination (Shading)</td>
<td>Visibility / Display</td>
</tr>
<tr>
<td>Viewing Transformation (Perspective / Orthographic)</td>
<td></td>
</tr>
<tr>
<td>Clipping</td>
<td></td>
</tr>
<tr>
<td>Projection (to Screen Space)</td>
<td></td>
</tr>
</tbody>
</table>
Graphics pipeline

2nd generation graphics hardware (1990s)

Hardware configurable

- Modeling Transformations
- Illumination (Shading)
- Viewing Transformation (Perspective / Orthographic)
- Clipping
- Projection (to Screen Space)
- Scan Conversion (Rasterization)
- Visibility / Display
Configurable?

- **API** (Application Programming Interface) for graphics hardware

- Mostly 2 different graphics APIs:
  - Direct3D (Microsoft)
  - OpenGL (Khronos Group)
Graphics pipeline

3rd generation graphics hardware (2000s)

Hardware programmable (shaders)

Modeling Transformations
Illumination (Shading)
Viewing Transformation (Perspective / Orthographic)
Clipping
Projection (to Screen Space)
Scan Conversion (Rasterization)
Visibility / Display
Programmable?

- **Shaders:**
  - Short programs, that the GPU runs at specific steps in the pipeline
  - Different languages (C-like), depending on the API:
    - NVIDIA ➔ Cg (2002)
    - Direct3D ➔ HLSL (2003)

- For GPGPU:
  - CUDA (NVIDIA)
  - ATI Stream
  - OpenCL (Khronos Group)
Shaders

- 3 types of shaders
  1. **Vertex** shader
  2. **Geometry** shader
  3. **Pixel** shader

- Local effect
  1. one vertex
  2. one primitive (& neighbors)
  3. one pixel
Shaders

Vertex shader

Geometry shader

Fragment shader

(x, y, z, w)
(nx, ny, nz)
(s, t, r, q)
(r, g, b, a)

(x', y', z', w')
(nx', ny', nz')
(s', t', r', q')
(r', g', b', a')

(x, y)
(r', g', b', a')
(depth')

(x, y)
(r, g, b, a)
(depth)
What you can do:
- Geometric transformations, changing position
- Lighting, shading, computing a color per vertex
- Computing texture coordinates

$$(x, y, z, w)$$  
$$(nx, ny, nz)$$  
$$(s, t, r, q)$$  
$$(r, g, b, a)$$  

$$(x', y', z', w')$$  
$$(nx', ny', nz')$$  
$$(s', t', r', q')$$  
$$(r', g', b', a')$$
Geometry shader

- What you can do:
  - Add/remove vertices
  - Change the primitives
  - Get the actual geometry, before rasterization
Fragment shader

- What you can do:
  - Lighting, shading, computing a color... per pixel
  - Use the textures as input for computations
  - Change pixel depth
uniform mat4 modelViewProjectionMatrix;

in vec4 vertex;

out vec3 color;

vec4 UneFonction(vec4 Entree) {
    return Entree.zxyw;
}

void main() {
    vec4 pos = modelViewProjectionMatrix * vertex;
    gl_Position = pos + UneFonction(vertex);
    color = vec3(1.0, 0.0, 0.0);
}

Output
Function
Input
Local variable
Matrix-vector multiplication
OpenGL Output
Swizzle

My first Vertex Shader
A piece of advice

- Code slowly, step by step, and **test often**!
  - Debugging is really difficult

**Optimization**

- Best place for each computation:
  - Vertex shader: 1x per vertex
  - Fragment shader: 1x per fragment: much more frequent!

- Use textures to tabulate complicated functions
- Use the functions in the language, rather than coding them yourself (sin, sqrt,...)
Graphics pipeline

4th generation graphics hardware (2010s)

- Model generation & transformation (tessellation / transformations)
- Illumination (Shading)
- Viewing Transformation (Perspective / Orthographic)
- Clipping
- Projection (to Screen Space)
- Scan Conversion (Rasterization)
- Visibility / Display

Hardware programmable (shaders)
Another step, between vertex and geometry shaders

Tessellation shaders

Before
Tessellation shaders

- Between vertex and geometry shaders
Before “primitive assembly”

Input: a patch
- Control points, coordinates inside the patch
- Patch type (triangles, quads, iso-lines)

Output:
- a vertex
- Called several times, once for each vertex generated
- Local effect, no global view of the patch

What for?
- Subdivision surfaces, splines...
Tessellation Control Shader

- Called before the Tessellation Eval Shader
- Facultative
- Controls tessellation level for each patch
- Once for each control point
  \[ \text{gl}\_\text{InvocationID} \]
- *Can* modify the control points

- Tessellation Primitive Generator
  - Generates the coordinates where we call TessEval
  - Fixed functionality
Tessellation: what for?

- Apparently, useless:
  - Anything it does, you can do with geometry shader

- In practice:
  - GPU needs predictability
  - Parallel processor / resource allocation
  - Useful for subdivision surfaces, splines
GPGPU

- General-Purpose Computation Using Graphics Hardware
- GPU = a SIMD processor (Single Instruction Multiple Data)
- One texture = array of input data
- One picture = array of output data
GPGPU – Applications

- Advanced rendering
  - Global illumination
  - Image-based rendering
  - ...
- Signal processing
- Algorithmic geometry
- Genetic algorithms
- Anything you can massively parallelize
GPGPU

- Get back the data (from GPU to CPU) = slower
  - PCI Express
- Limited operators, functions, types
- A parallel algorithm is not necessarily faster than the sequential version
  - Synchronization between multiple cores
References

- OpenCL: [http://www.kronos.org/opencl/](http://www.kronos.org/opencl/)

- Debugging OpenGL/GLSL:
  - glslDevil: [http://www.vis.uni-stuttgart.de/glsldevil/](http://www.vis.uni-stuttgart.de/glsldevil/)

- Many examples (to use as a starting point):
