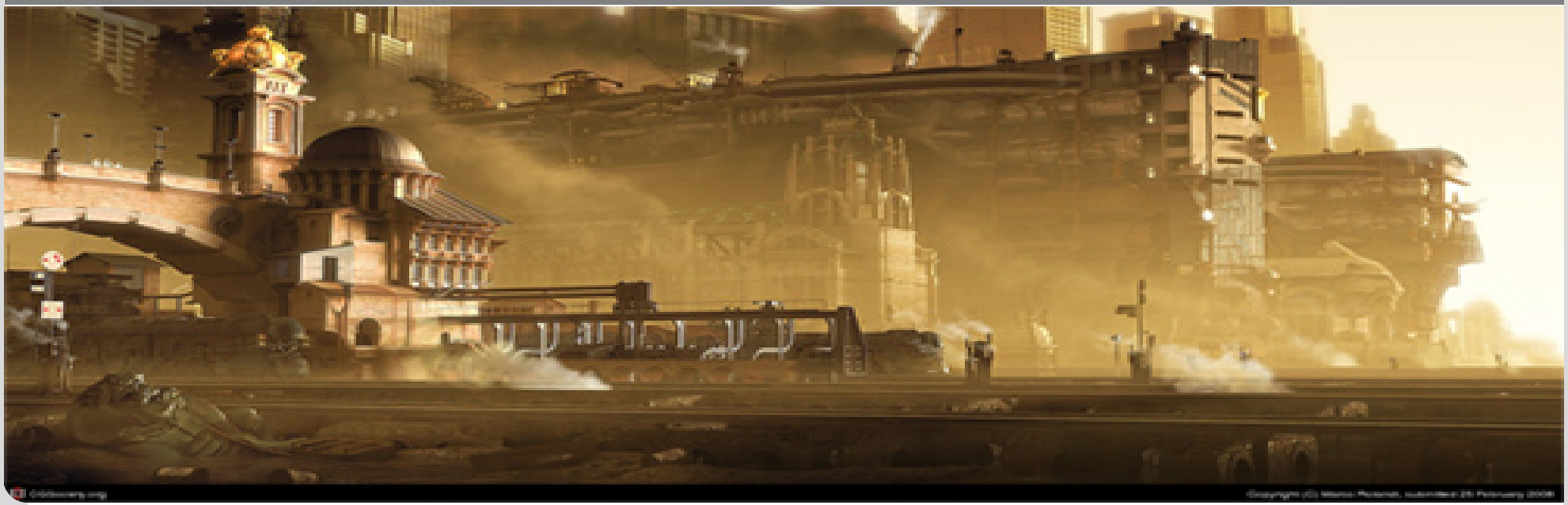
Shaders

Computer Graphics - SET08116

EDINBURGH NAPIER UNIVERSITY



Outline

What is a Pipeline?

A pipeline is just a series of processes that occur in order:

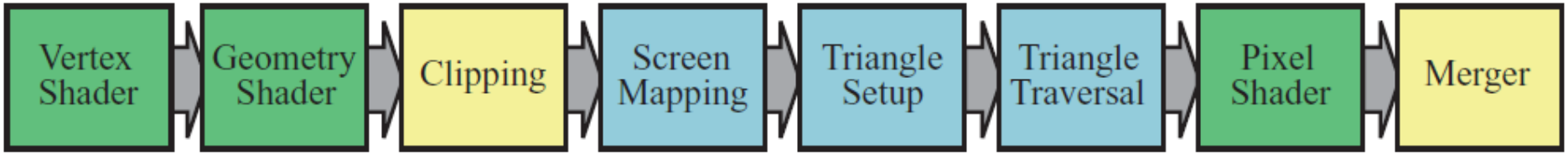
$$\text{Read File} \Rightarrow \text{Process Data} \Rightarrow \text{Print Results}$$

A pipeline can have any number of processes within it.

The point is that data from one process is fed into the next process.

- Think of this as an assembly line.

Graphics Pipeline



- Vertex Shader - manipulates individual vertexes.
- Geometry Shader - manipulates primitives.
- Clipping - removes unseen geometry, and fixes partially seen.
- Screen Mapping - mapping triangles to the screen
- Triangle Setup and Traversal - discover which triangles affect which pixels
- Pixel Shader - manipulates pixel colour
- Merger - determines final colour of a pixel

Vertex Data

Initially we are going to take a step back and think about how data is presented to the graphics card. This is important to understand as we will shortly be working with buffers and shaders.

Data sent to the graphics card is called vertex data:

- Position of the vertex
- Colour of the vertex
- Texture coordinates
- Normals

These items are used to define a geometric object.

```
// Vertex 1
glVertex3f(1.0f,0.0f,0.0f);
glColor3f(1.0f,0.0f,0.0f);

// Vertex 2
glVertex3f(1.0f,1.0f,0.0f);
glColor3f(1.0f,0.0f,1.0f);

// etc.
```

Programmable Pipeline

To add flexibility, GPUs were given programmable stages.

- The programs that run on the GPU are referred to as shaders.

First shader programs were used in the 1980s.

- Pixar's RenderMan

2001 Nvidia introduced the first GPU with programmable shader support.

- Originally only vertex shading was supported.

Today, nearly all GPUs support some form of programmable shading.

- Vertex shader
- Geometry shader
- Tessellation shader (new - DirectX 11 and OpenGL 4)
- Pixel / fragment shader

Programmable Shaders

As already mentioned, there are three stages in the graphics pipeline that are programmable:

- Vertex Shader - manipulates individual vertexes
- Geometry Shader - works with primitives (triangles, lines, etc.)
- Pixel Shader - determines colour of individual pixels

There is also the tessellation shader and compute shaders, but these are beyond the scope of the module.

Having the ability to write programs for individual stages of the pipeline is advantageous, and allows us to both optimise, and produce more elaborate effects than the fixed function pipeline was capable of.

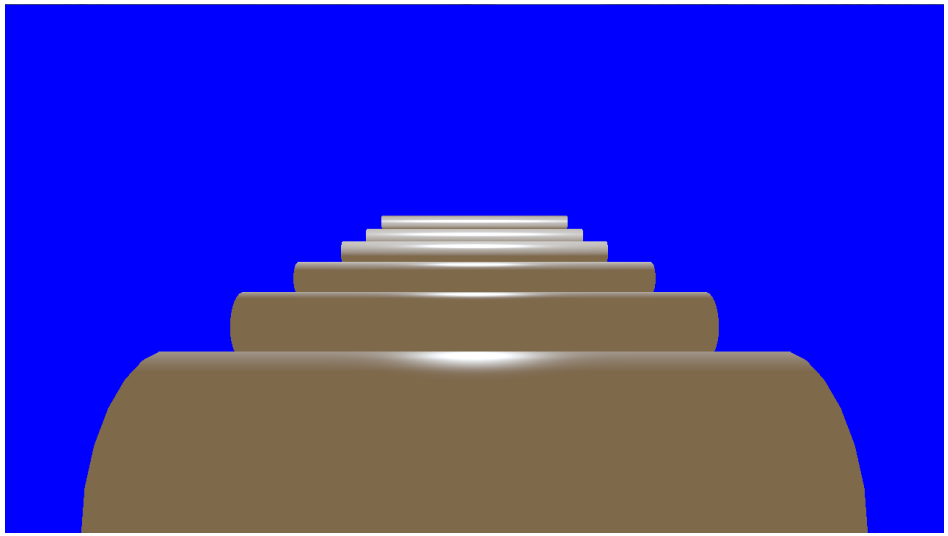
Vertex Shader

The vertex shader lets us work with the incoming vertex data.

- Position
- Colour
- Normals

We can use these values to perform calculations for the rest of the render:

- Light Colour
- World Position



Geometry Shader

The geometry shader lets us work with individual geometric primitives:

- Lines
- Triangles
- Quads

The geometry shader can also introduce new geometry, allowing effects such as particle systems, grass, etc.

Stream Output

It is possible to take the output from the geometry shader (or just the vertex shader if no geometry shader is present) and use the data for other information.

- Information for final render
- Physics calculations

The point is that no rendering is attempted to an image (e.g. the screen). The rasterization stage does not happen.

Essentially, we want to work with geometric data that we can utilise later in another rendering pass, and we only wish to manipulate the rendering data, which the GPU is very good at.

- Again, particle effects (explosions) are commonly performed here.

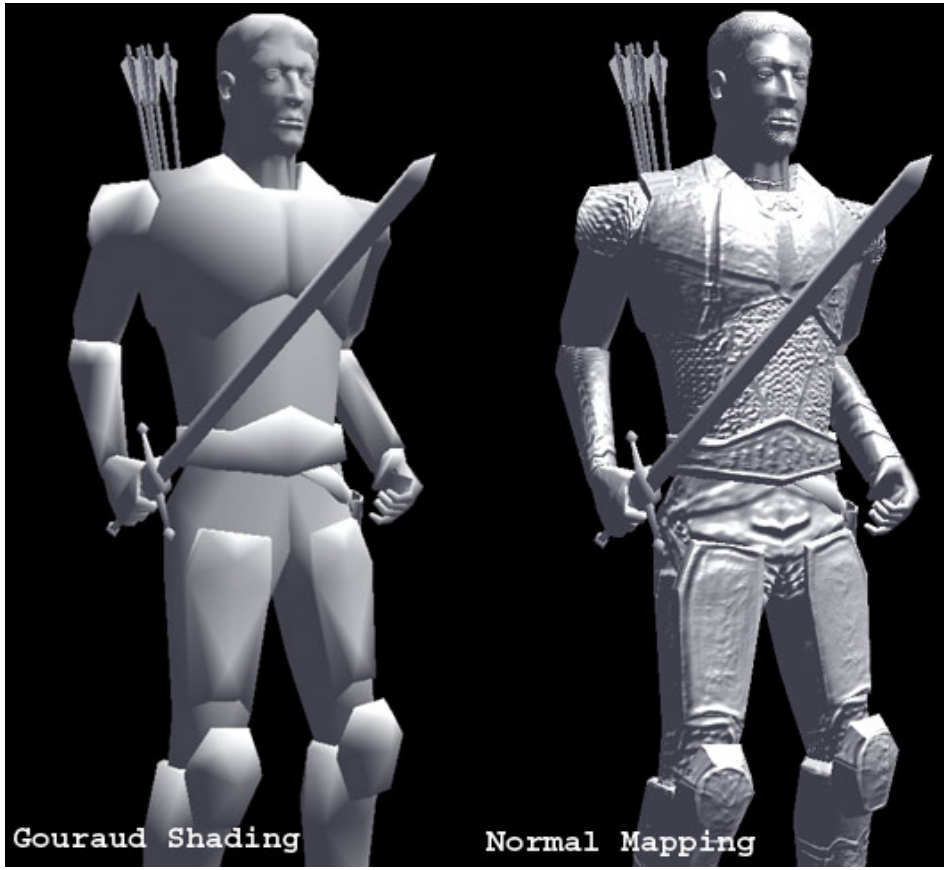
Pixel Shader

The pixel shader is used to determine the individual pixel colours in the final rendered image.

There are a number of common techniques to achieve this:

- Per-pixel lighting
- Texturing
- Normal (bump) mapping
- etc.

Several techniques are combined to determine the final render pixel colour.



Render to Texture

As mentioned, it is not necessarily the case that we render to the screen. We can also render to another texture.

The GPU has a render target that it is assigned prior to a render pass being undertaken. By default, the render target is the screen, but a texture can be assigned as the target also. The end output is stored in the texture and can be used later.

Typically, a render stored in a texture is used in a render pass called a post process. A post process applies some form of filter to the rendered image to provide a different visual effect (for example a red filter).

There are numerous useful parts of the render, and techniques that can be applied:

- Motion blur
- Depth of field
- Shadowing

Example Vertex Shader

```
#version 330

uniform mat4 modelViewProjection;

layout (location = 0) in vec3 position;
layout (location = 1) in vec4 in_colour;

out vec4 out_colour;

void main()
{
    gl_Position = modelViewProjection * position;
    out_colour = vec4(1.0, 0.0, 0.0, 1.0);
}
```

Breakdown of Vertex Shader

```
#version 330
```

The first line is a pre-processor, informing the GLSL compiler what version of OpenGL we are using.

```
uniform mat4 modelViewProjection;
```

Next we declare our uniforms. Uniforms are values which are the same for all vertexes passed through the shader. For example, here we have the world-view-projection matrix.

```
layout (location = 0) in vec3 position;
layout (location = 1) in vec4 in_colour;
```

Next, we indicate the incoming values to the shader. Here we have two - the position of the vertex, and the colour of the vertex. Also note that we indicate the location that these values come into the shader.

Breakdown of Vertex Shader

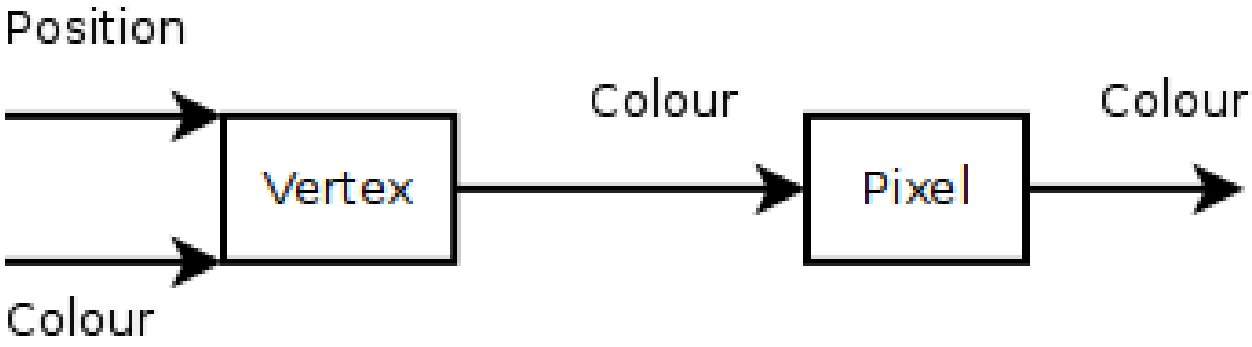
```
out vec4 out_colour;
```

We also have a collection of output values. Here we output the colour value for the vertex - which will be passed along to the pixel shader.

```
void main()
{
    gl_Position = modelViewProjection * position;
    out_colour = vec4(1.0, 0.0, 0.0, 1.0);
}
```

Finally, we have the main function of the shader. This is the actual code that runs on the GPU. It takes the incoming vertex data, the uniforms and performs some calculations. The vertex shader must set the position. We also set the outgoing colour value.

Data Flow



The data flows into the vertex shader program, where various values are set, and then flows out to the pixel shader.

Here we have the vertex shader inputting the position and colour, and outputting the colour to the pixel shader. The pixel shader will also output the colour.

Example Fragment Shader

```
#version 330

in vec4 in_colour;
out vec4 out_colour;

void main()
{
    out_colour = vec4(0.0f, 1.0f, 0.0f, 1.0f);
}
```

Breakdown of Fragment Shader

The fragment shader is simpler in this instance than the vertex shader, but this may not always be the case.

```
#version 330

in vec4 in_colour;
out vec4 out_colour;
```

The fragment shader uses the same approach to defining uniforms, incoming data, and outgoing data. Notice we have no location. The location syntax for internal communication came about in OpenGL 4.

Breakdown of Fragment Shader

```
void main()
{
    out_colour = vec4(0.0f, 1.0f, 0.0f, 1.0f);
}
```

The main function of the shader again uses uniforms and incoming data to calculate values. Our main goal is to output the colour, but this will likely involve more calculation than indicated.

Spot Quiz

Spot quiz - what is the final output colour for the shader? What other colours were used?

Effect Files

Shader programs are stored in effect files containing the necessary code and data requirements. These program files are compiled and linked as any other application. However, this is done by the host application, using the host GLSL compiler and linker (part of your graphics driver).

Values are set for the shader using application API calls (you will see this in the practical). Individual effects can be split across multiple shader files.

- We will be doing this later in the module
- Enables re-usability

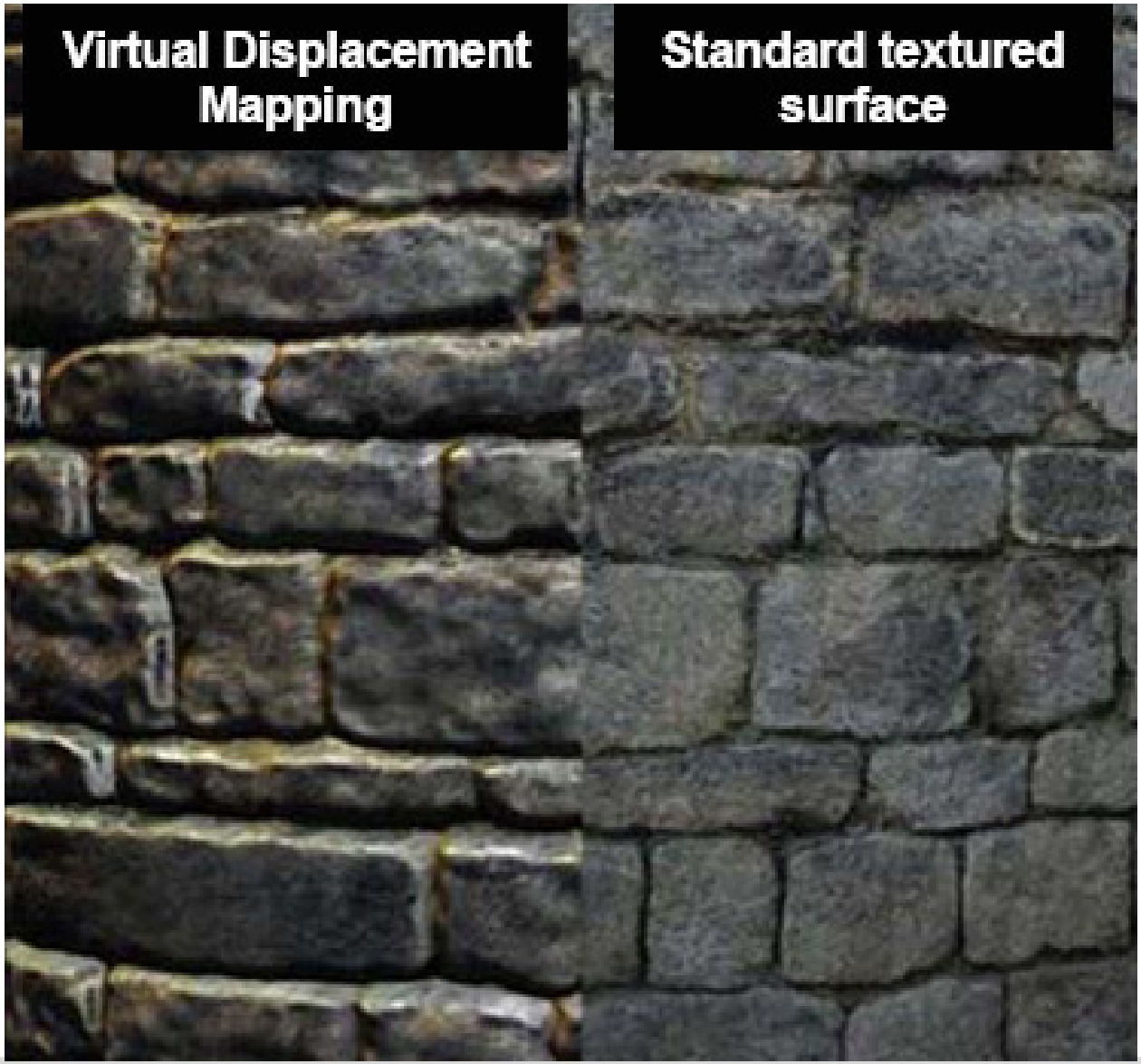
Languages

A number of shader languages exist. Essentially, the ideas are the same although the API and syntax may be slightly different.

Some of the well known languages are:

- High Level Shader Langauage (HLSL)
  - DirectX
- GL Shader Langauge (GLSL)
  - OpenGL
- C for graphics (Cg)
  - nVidia language
  - Supported via libraries in DirectX and OpenGL
  - Slower due to higher level integration
- RenderMan
  - Pixar's shading language

Displacement Maps

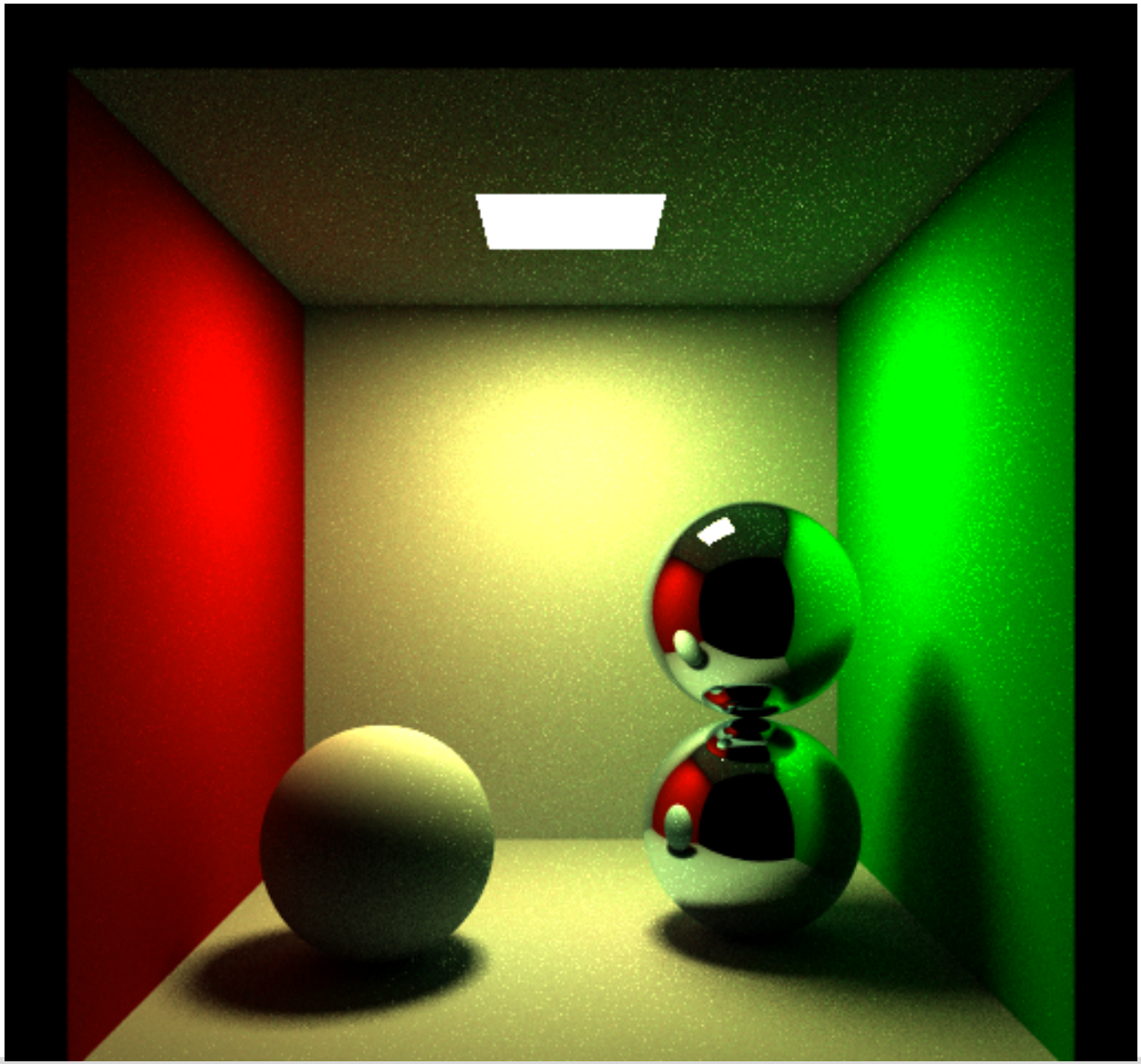


Motion Blur


4ème
/4
42"50
252 km/h
Points de Style
0
5
NOS
0%
290
220
100
90
Turbo
UNDRGND

Depth of Field

Environment Maps



Summary

From this lecture, you should now know about:

- The different programmable stages of the graphics pipeline, and their responsibilities
- The structure of a shader program
- How data is communicated into and out of a shader program
- The types of effects possible via the different programmable shader stages.

Recommended Reading

Interactive Computer Graphics, chapters 5 and 6.
Real-Time Rendering, chapter 3 review.