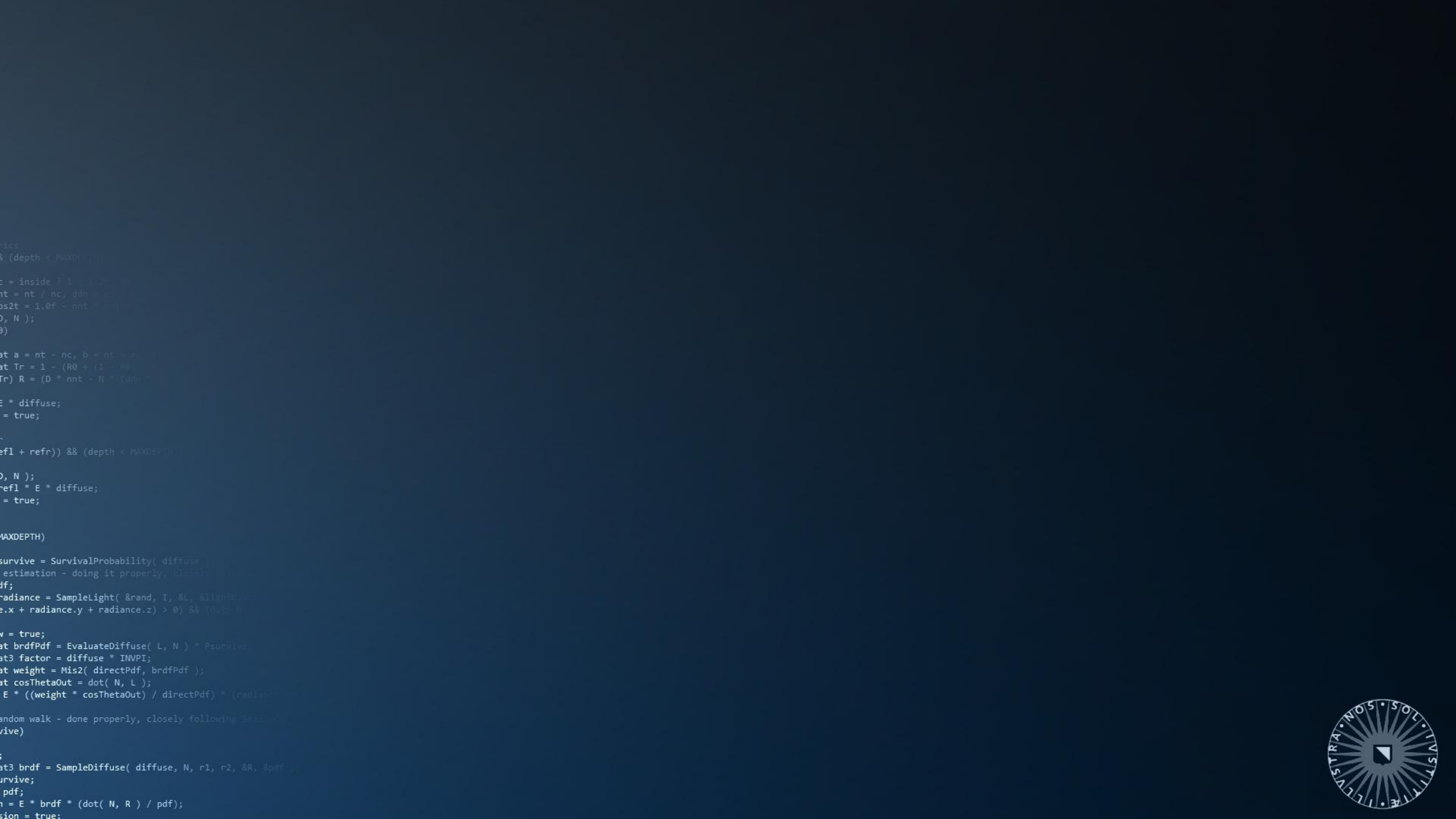


INFOGR – Computer Graphics

Jacco Bikker & Debabrata Panja - April-July 2019

Lecture 9: “OpenGL”

Welcome!

Today’s Agenda:

- Introduction
- OpenGL
- GPU Model
- Upcoming
- Assignment P3

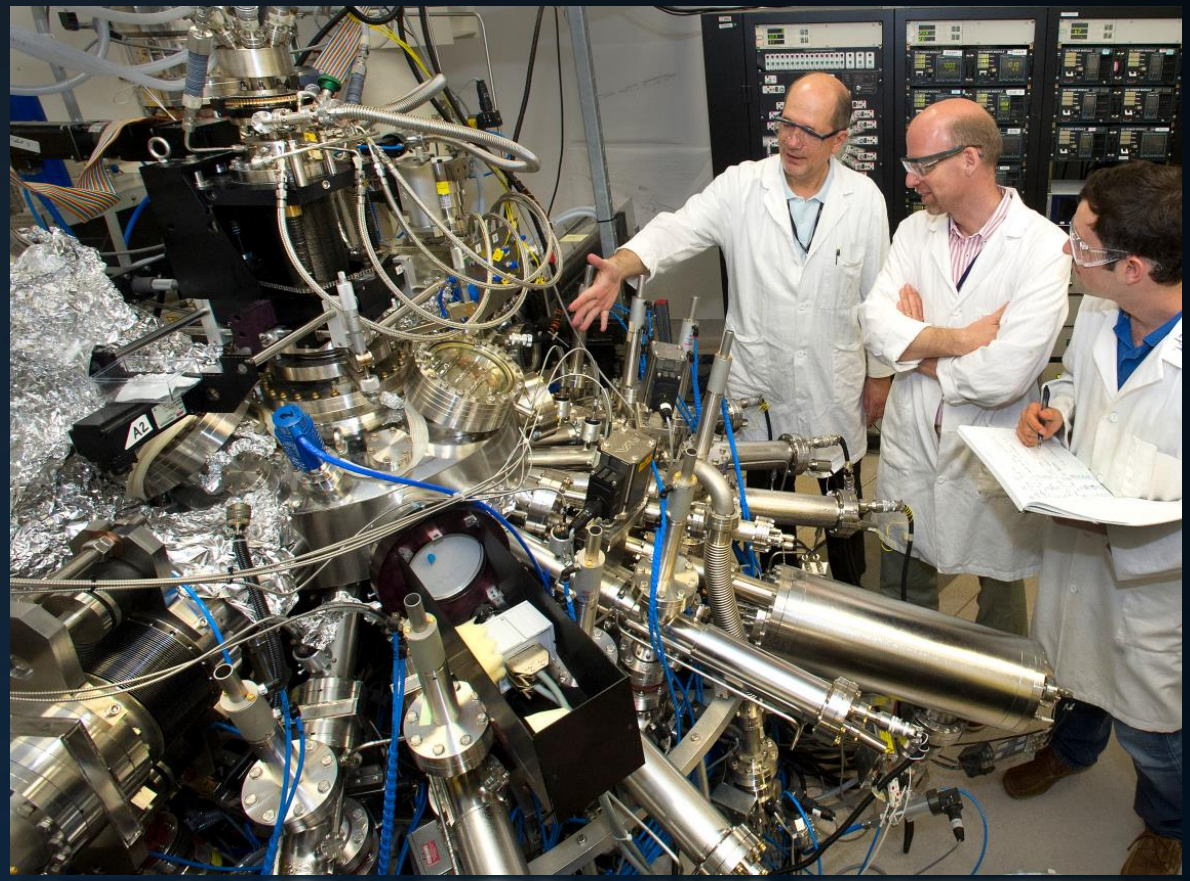
Introduction

Topics covered so far:

Basics:
- Rasters
- Vectors
- Color representation

Ray tracing:
- Light transport
- Camera setup
- Textures

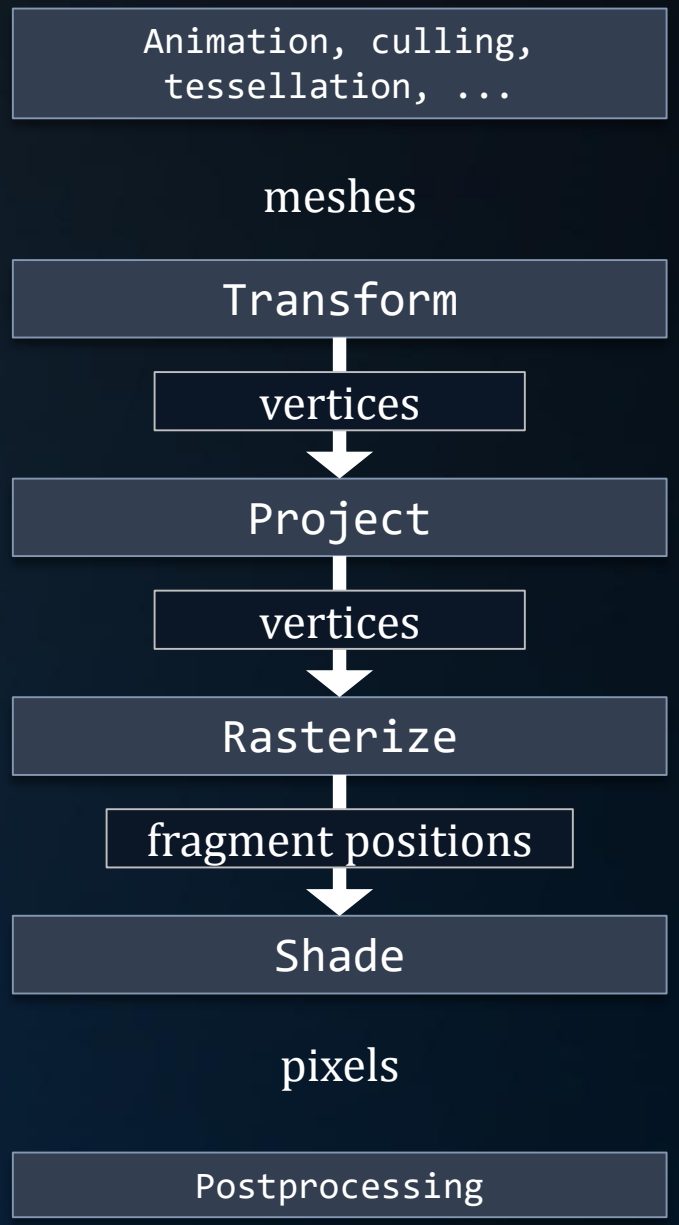
Shading:
- N dot L
- Distance attenuation
- Pure specular



Introduction

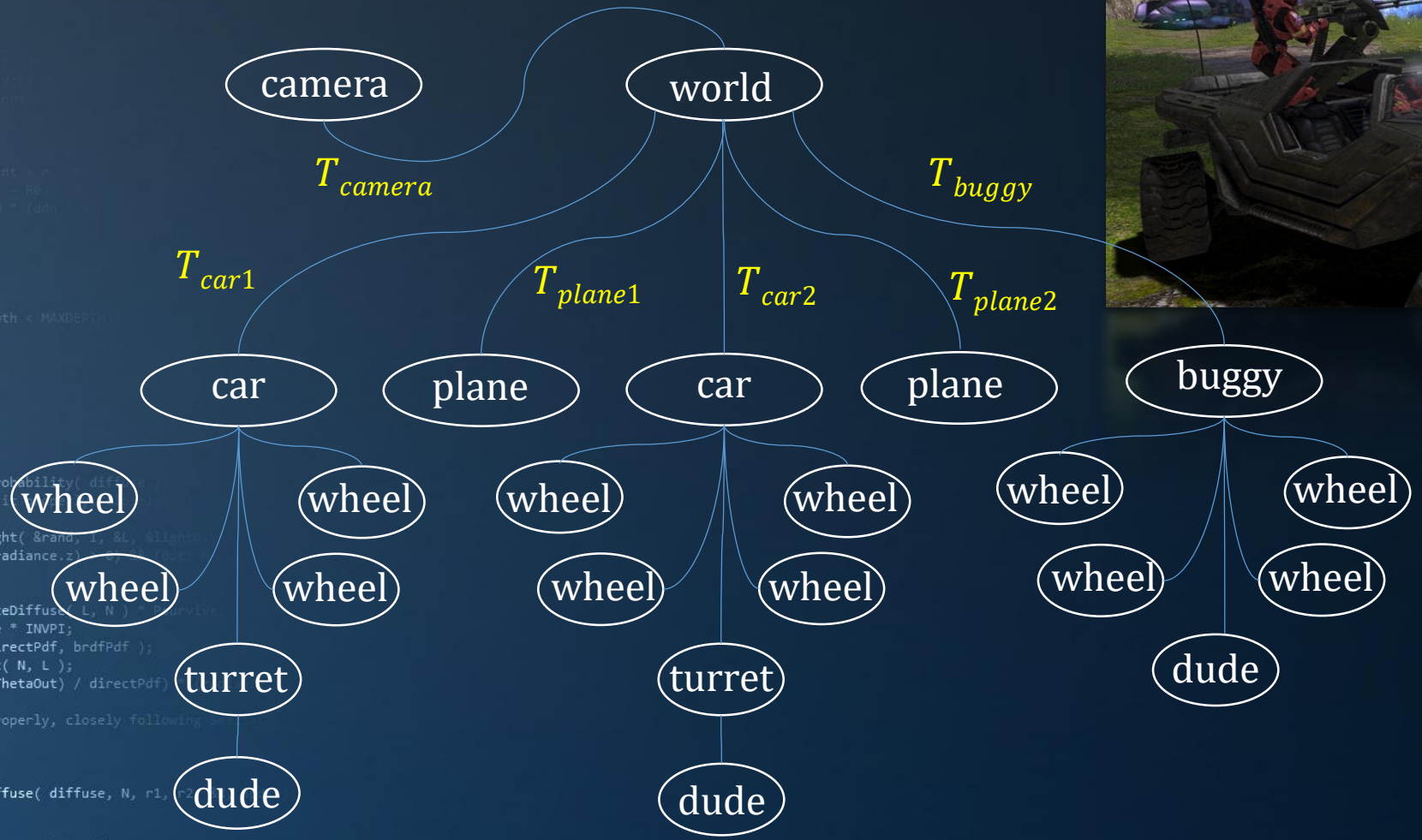
Rendering – Functional overview

1. Transform:
   translating / rotating / scaling meshes
2. Project:
   calculating 2D screen positions
3. Rasterize:
   determining affected pixels
4. Shade:
   calculate color per affected pixel



Introduction

Rendering – Data Overview



Introduction

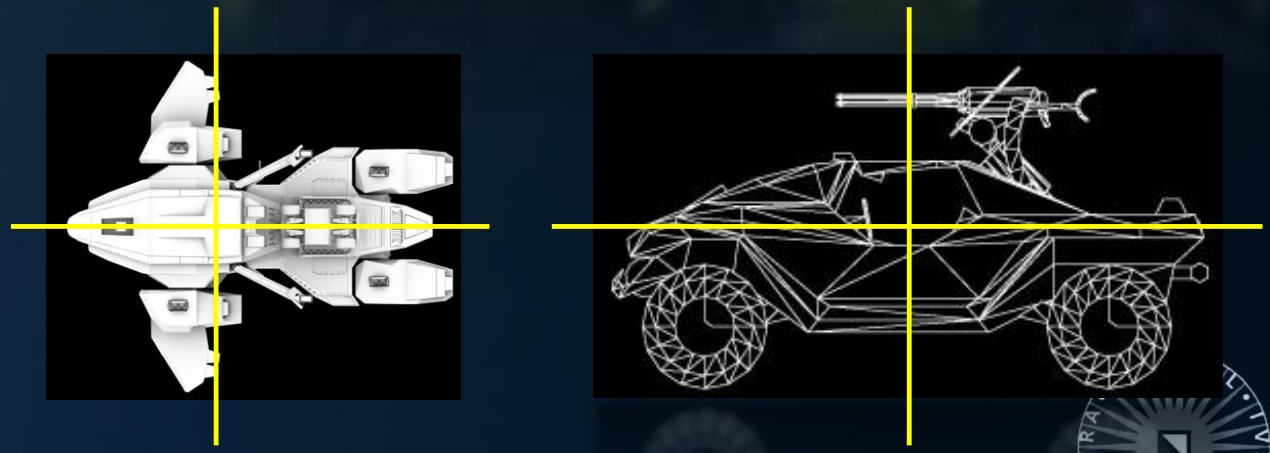
Rendering – Data Overview

Objects are organized in a hierarchy: the *scenegraph*.

In this hierarchy, objects have translations and orientations relative to their parent node.

Relative translations and orientations are specified using matrices.

Mesh vertices are defined in a coordinate system known as *object space*.



Introduction

Writing a 3D Engine
(before the summer holiday)

3D engine raison d'être: *it's a visualizer for a scene graph.*

We typically build 3D engines for GPUs. A GPU is not a CPU.

For real-time graphics, we use rasterization, rather than ray tracing.

We still want realistic images.

- Math: matrices
- OpenGL
- GPU architecture
- Shading
- Post processing
- Visibility

Today’s Agenda:

- Introduction
- OpenGL
- GPU Model
- Upcoming
- Assignment P3

Introduction

A Brief History of OpenGL

OpenGL: based on Silicon Graphics IRIS GL (~1985).

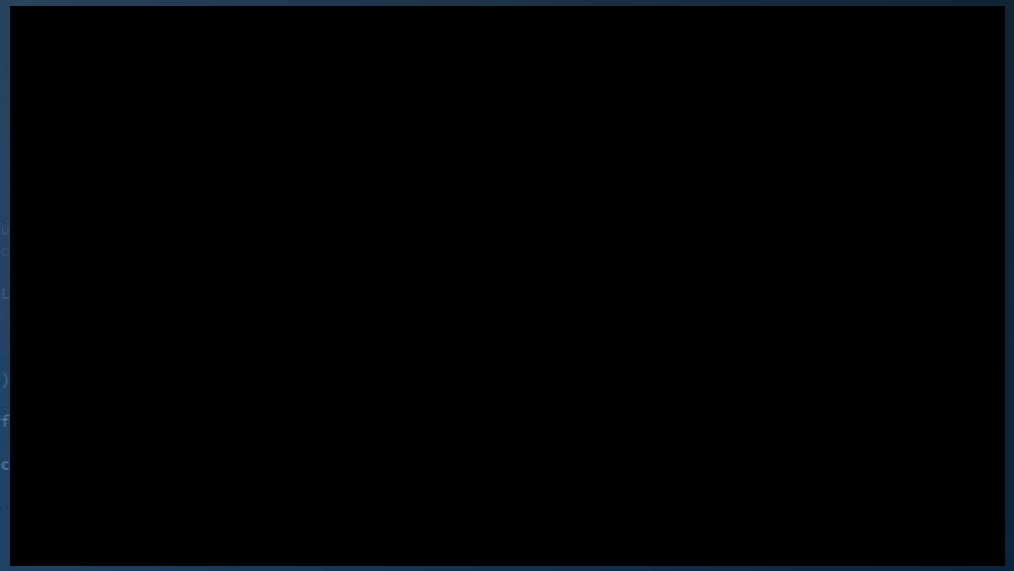
1992: OpenGL Architecture Review Board (ARB)
1995: Direct3D
1997: Fahrenheit (☠1999)

Purpose: generic API for 2D and 3D graphics.

- Platform-independent
- Language-agnostic
- Designed for hardware acceleration


3D
REALMS
REALITY IS OUR GAME


Select Pilot:
52 Paul Lucas
13 bunnyhunter
1 Samuel Stinger
0 Virginia Wolfblitz
0 Pow
0
0
0
(Press DEL to enter a new player)




PIECE OF CAKE
LET'S ROCK
COME GET SOME
DAMN I'M GOOD

Introduction

A Brief History of OpenGL

OpenGL: based on Silicon Graphics IRIS GL (~1985).

1992: OpenGL Architecture Review Board (ARB)
1995: Direct3D
1997: Fahrenheit (☠1999)
1997: Glide / 3Dfx
2006: ARB ➔ Khronos Group

KHRONOS GROUP

Introduction

A Brief History of OpenGL

OpenGL 1.0 – 1992 (initial version)

1995: Windows 95

1996: Direct3D 2.0 & 3.0

OpenGL 1.1 – 1997 - Textures

1997: GLQuake

1998: Direct3D 6.0

OpenGL 1.2 – 1998 - 3D textures

1999: Direct3D 7.0: HW T&L, vertex buffers in device mem

OpenGL 1.3 – 2001- Environment maps, texture compression

OpenGL 1.4 – 2002 - Blending, stencils, fog

OpenGL 1.5 – 2003 - Vertex buffers

Introduction

A Brief History of OpenGL

2001: GeForce 3, vertex/pixel shaders

OpenGL 2.0 – 2004 - Shaders

OpenGL 3.0 – 2008 – Updated shaders, framebuffers, floating point textures

OpenGL 3.1 – 2009 – Instanced rendering

2009: GeForce 8, geometry shaders

OpenGL 3.2 – 2009 – Geometry shaders

OpenGL 3.3 – 2010 – Support Direct3D 10 hardware

OpenGL 4.0 – 2010 – Direct3D 11 hardware support

Introduction

A Brief History of OpenGL

OpenGL 4.1 – 2010

OpenGL 4.2 – 2011 – Support for atomic counters in shaders

OpenGL 4.3 – 2012 – Compute shaders

OpenGL 4.4 – 2013

OpenGL 4.5 – 2014

Vulkan - 2016

Apple Metal - 2014
AMD Mantle - 2015
MS DirectX 12 - 2016

Vulkan:

- “OpenGL next”
- Support for multi-core CPUs
- Derived from AMDs Mantle
- Low-level GPU control
- Cross-platform

Introduction

OpenGL

Vulkan

OpenGL|ES

PSGL
PlayStation Graphics Library

Microsoft DirectX 11

android

NINTENDO SWITCH

Wii U

Introduction

A Brief History of OpenGL

Digest:

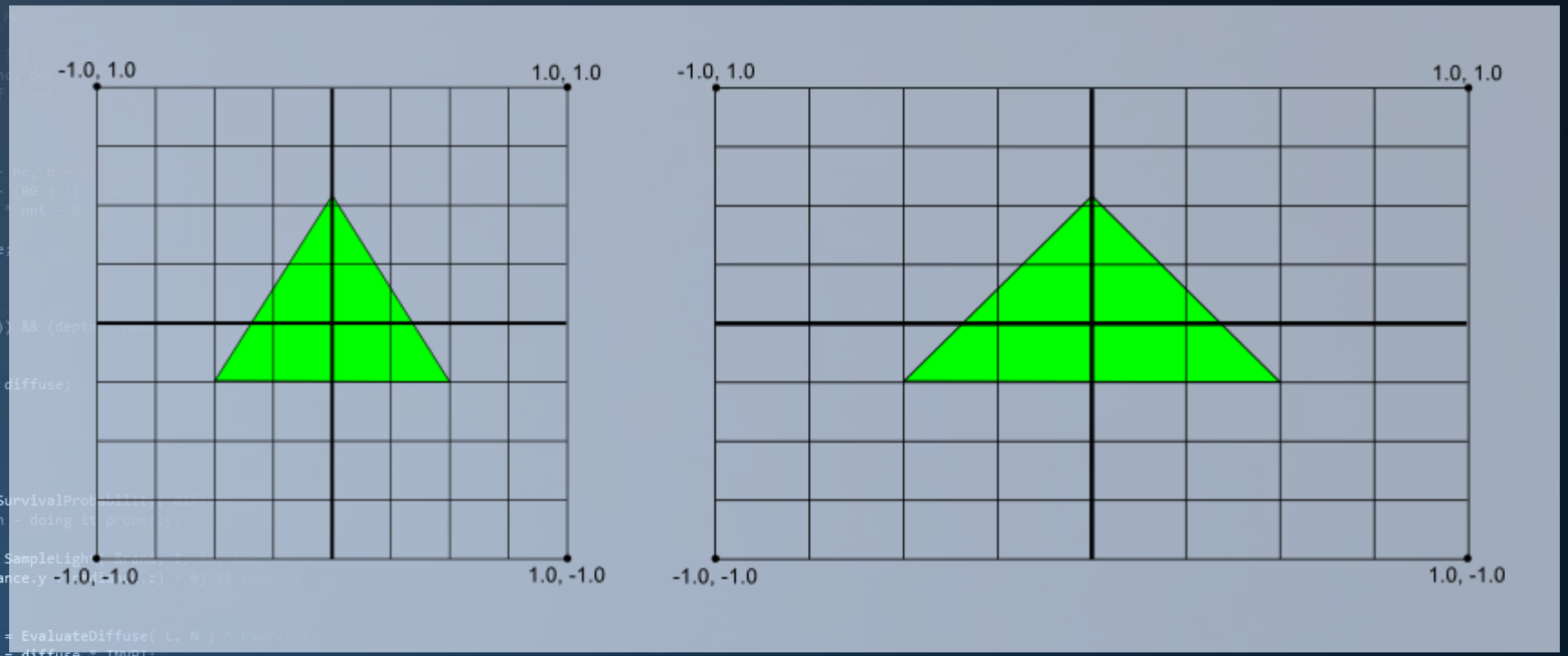
- Open standard graphics API, governed by large body of companies
- Initially slow to follow hardware advances
- After transfer to Khronos group: closely following hardware
- Currently more or less ‘the standard’, despite DirectX / Metal
- Moving towards ‘close to the metal’ ➔ Vulkan.

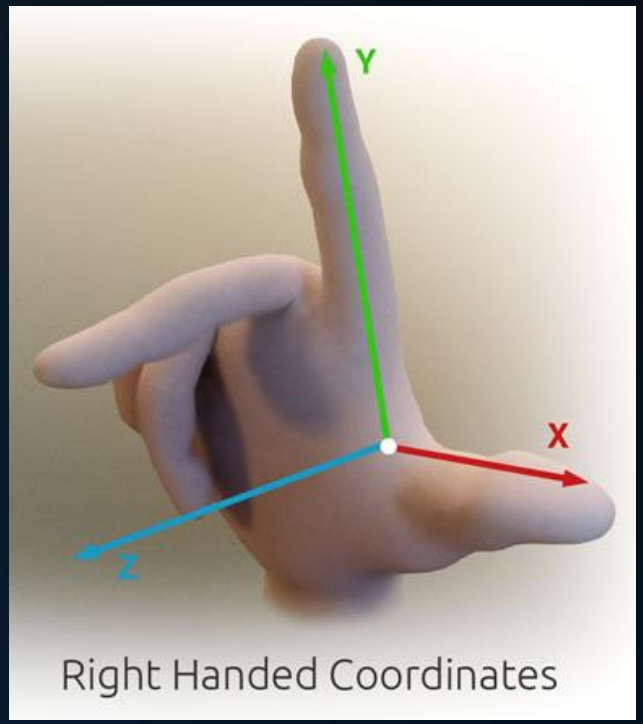
Today’s Agenda:

- Introduction
- OpenGL
- GPU Model
- Upcoming
- Assignment P3

OpenGL

OpenGL Coordinates





OpenGL

OpenGL Basics

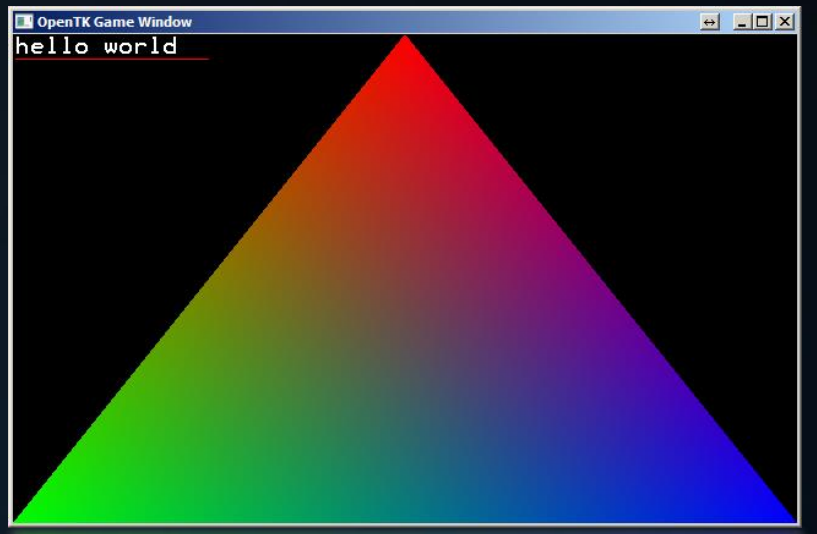
C# / OpenTK:

```
public void TickGL()
{
    GL.Begin( PrimitiveType.Triangles );
    GL.Color3( 1.0f, 0, 0 ); GL.Vertex2(  0.0f,  1.0f );
    GL.Color3( 0, 1.0f, 0 ); GL.Vertex2( -1.0f, -1.0f );
    GL.Color3( 0, 0, 1.0f ); GL.Vertex2(  1.0f, -1.0f );
    GL.End();
}
```

C++:

```
glBegin( GL_TRIANGLES );
glColor3f( 1.0f, 0, 0 ); glVertex2f(  0.0f,  1.0f );
glColor3f( 0, 1.0f, 0 ); glVertex2f( -1.0f, -1.0f );
glColor3f( 0, 0, 1.0f ); glVertex2f(  1.0f, -1.0f );
glEnd();
```

```
Points
Lines
LineLoop
LineStrip
Triangles
TriangleStrip
TriangleFan
Quads
QuadsExt
...
```

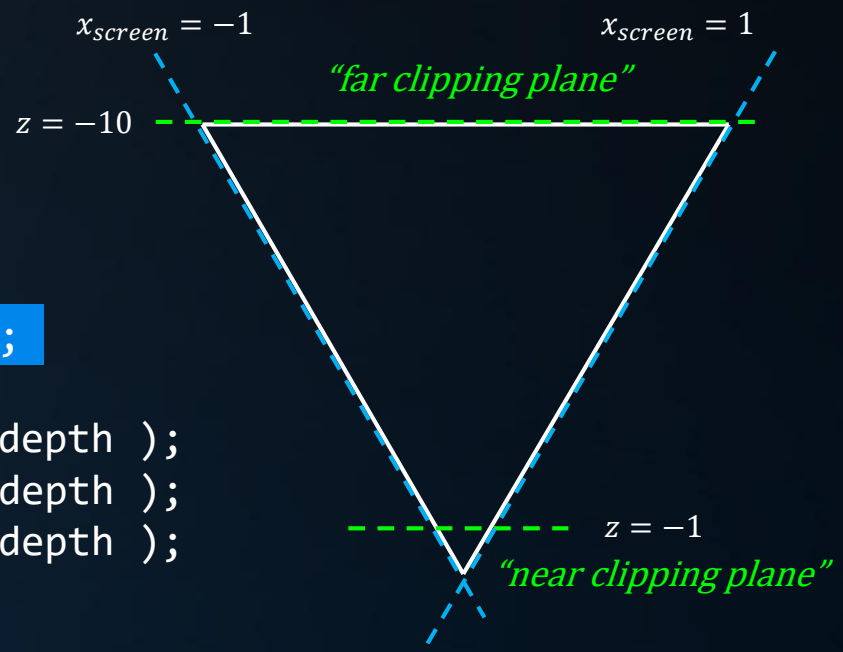


OpenGL

OpenGL Basics

```
static float depth = -1.0f;

public void TickGL()
{
    GL.Frustum( -1.0f, 1.0f, -1.0f, 1.0f, 1.0f, 10.0f );
    GL.Begin( PrimitiveType.Triangles );
    GL.Color3( 1.0f, 0, 0 ); GL.Vertex3(  0.0f,  1.0f, depth );
    GL.Color3( 0, 1.0f, 0 ); GL.Vertex3( -1.0f, -1.0f, depth );
    GL.Color3( 0, 0, 1.0f ); GL.Vertex3(  1.0f, -1.0f, depth );
    GL.End();
    depth -= 0.01f;
}
```



OpenGL

OpenGL Basics

```
static float r = 0.0f;

public void TickGL()
{
    // set model view matrix
    GL.Frustum( -1.0f, 1.0f, -1.0f, 1.0f, 1.0f, 15.0f );
    GL.Translate( 0, 0, -2 );
    GL.Rotate( r, 0, 1, 0 );
    // render primitives
    GL.Begin( PrimitiveType.Triangles );
    GL.Color3( 1.0f, 0, 0 ); GL.Vertex3(  0.0f, -0.3f,  1.0f );
    GL.Color3( 0, 1.0f, 0 ); GL.Vertex3( -1.0f, -0.3f, -1.0f );
    GL.Color3( 0, 0, 1.0f ); GL.Vertex3(  1.0f, -0.3f, -1.0f );
    GL.End();
    r += 0.1f;
}
```

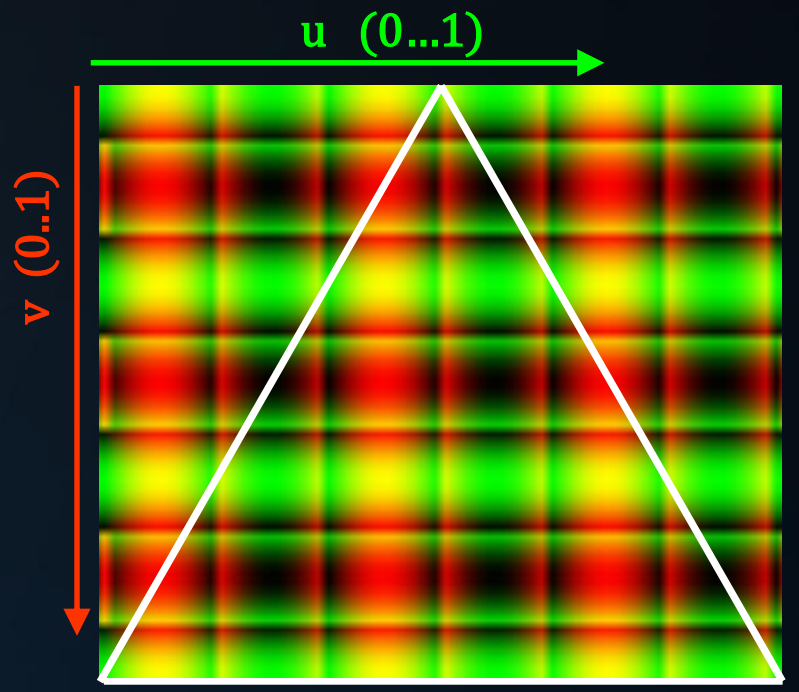
Apply perspective to:
A translated object:
That we rotated.
Here are it’s original vertices.

OpenGL

OpenGL Basics

```
static int textureID;

public void TickGL()
{
    GL.BindTexture( TextureTarget.Texture2D, textureID );
    GL.Begin( PrimitiveType.Triangles );
    GL.TexCoord2( 0.5f, 0 ); GL.Vertex2(  0.0f,  1.0f );
    GL.TexCoord2(    0, 1 ); GL.Vertex2( -1.0f, -1.0f );
    GL.TexCoord2(    1, 1 ); GL.Vertex2(  1.0f, -1.0f );
    GL.End();
}
```




```
textureID = screen.GenTexture();
GL.BindTexture( TextureTarget.Texture2D, textureID );
uint [] data = new uint[64 * 64];
for( int y = 0; y < 64; y++ ) for( int x = 0; x < 64; x++ ) data[x + y * 64] =
    ((uint)(255.0f * Math.Sin( x * 0.3f )) << 16) +
    ((uint)(255.0f * Math.Cos( y * 0.3f )) << 8);
GL.TexImage2D( TextureTarget.Texture2D, 0, PixelInternalFormat.Rgba, 64, 64, 0,
    OpenTK.Graphics.OpenGL.PixelFormat.Bgra, PixelType.UnsignedByte, data );
```

OpenGL

OpenGL State

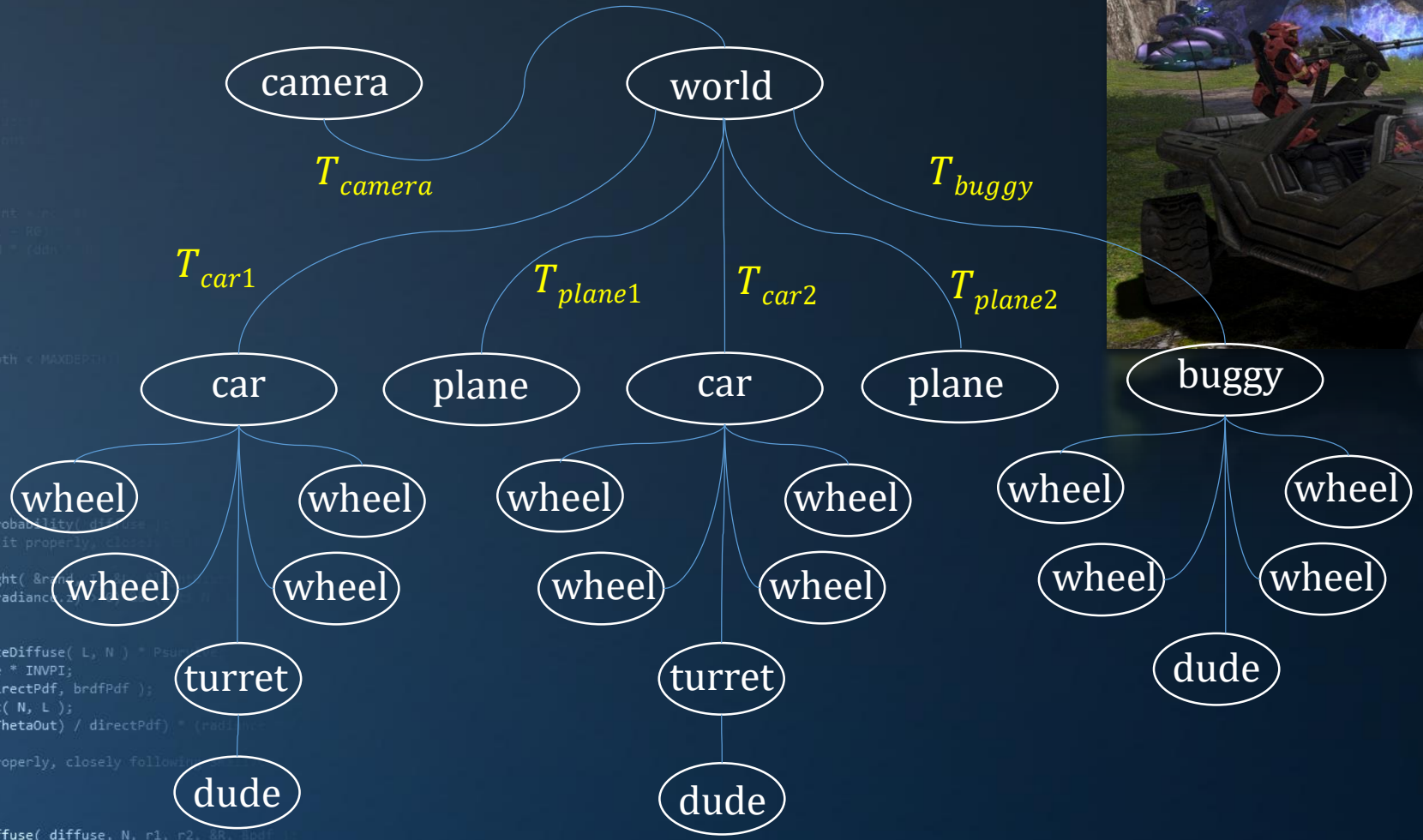
OpenGL is a *state machine*:

- We set a texture, and all subsequent primitives are drawn with this texture;
- We set a color … ;
- We set a matrix … ;
- …

Related to this:

- A scene graph matches this behavior.
- A GPU expects this behavior.

OpenGL


camera
world
T_{camera}
T_{buggy}
T_{car1}
T_{plane1}
T_{car2}
T_{plane2}
car
plane
car
plane
buggy
wheel
wheel
wheel
wheel
wheel
wheel
wheel
wheel
wheel
wheel
wheel
wheel
turret
turret
dude
dude
dude

Today’s Agenda:

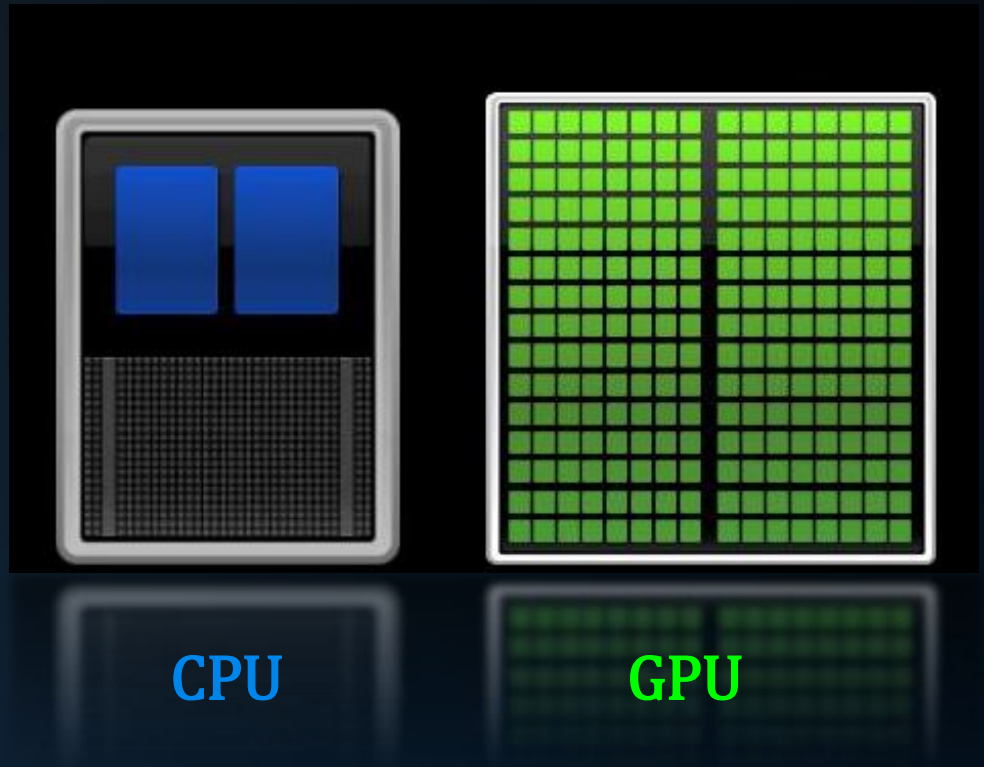
- Introduction
- OpenGL
- GPU Model
- Upcoming
- Assignment P3

GPU Model

GPU: Streaming Processor

A GPU is designed to work on many uniform tasks in parallel.

- It has (vastly) more cores
- The cores must all execute identical code
- It does not rely on caches

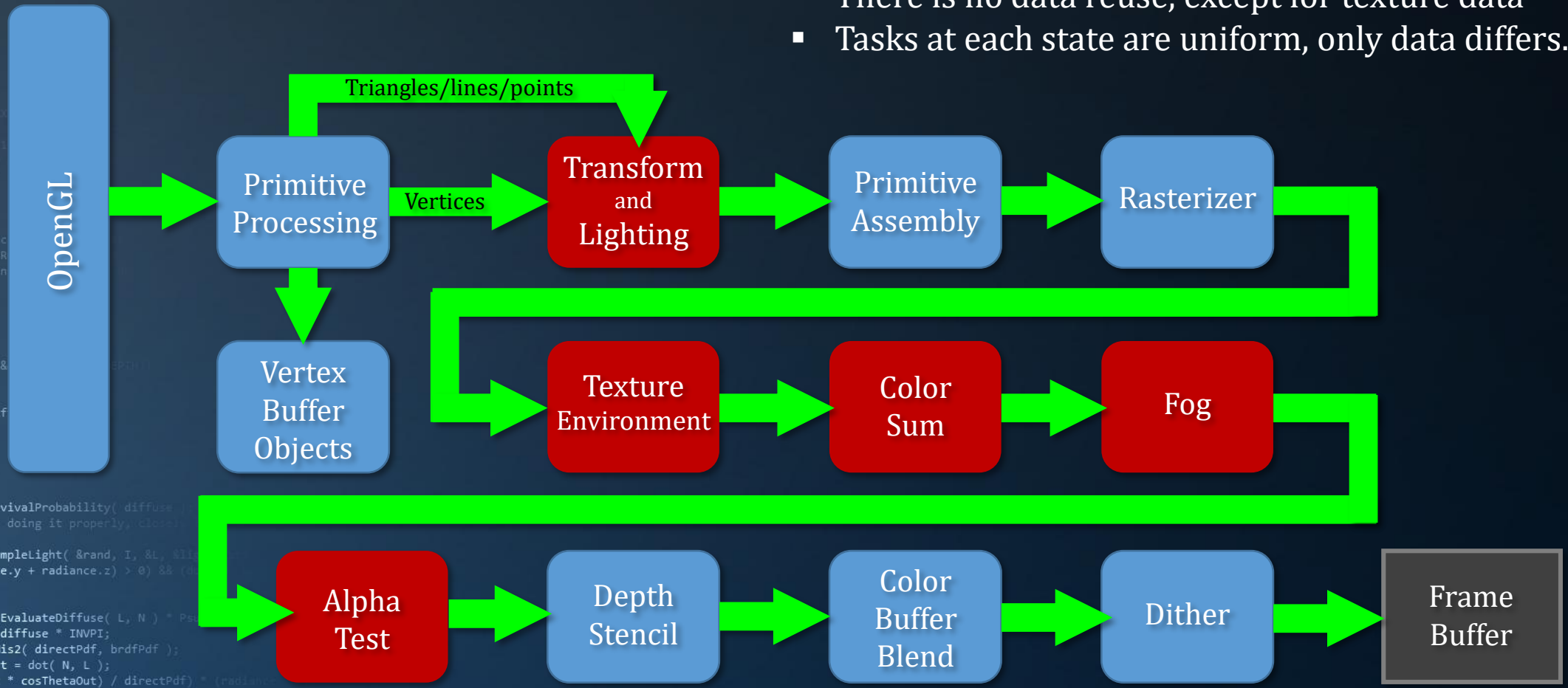


A CPU is optimized to execute a few complex tasks.

- It uses caches to benefit from patterns in data access
- It uses complex cores to maximize throughput for complex algorithms.

GPU Model

- Thousands of primitives and vertices enter the pipeline
- There is no data reuse, except for texture data
- Tasks at each state are uniform, only data differs.



GPU Model

Switching State

A state change requires:

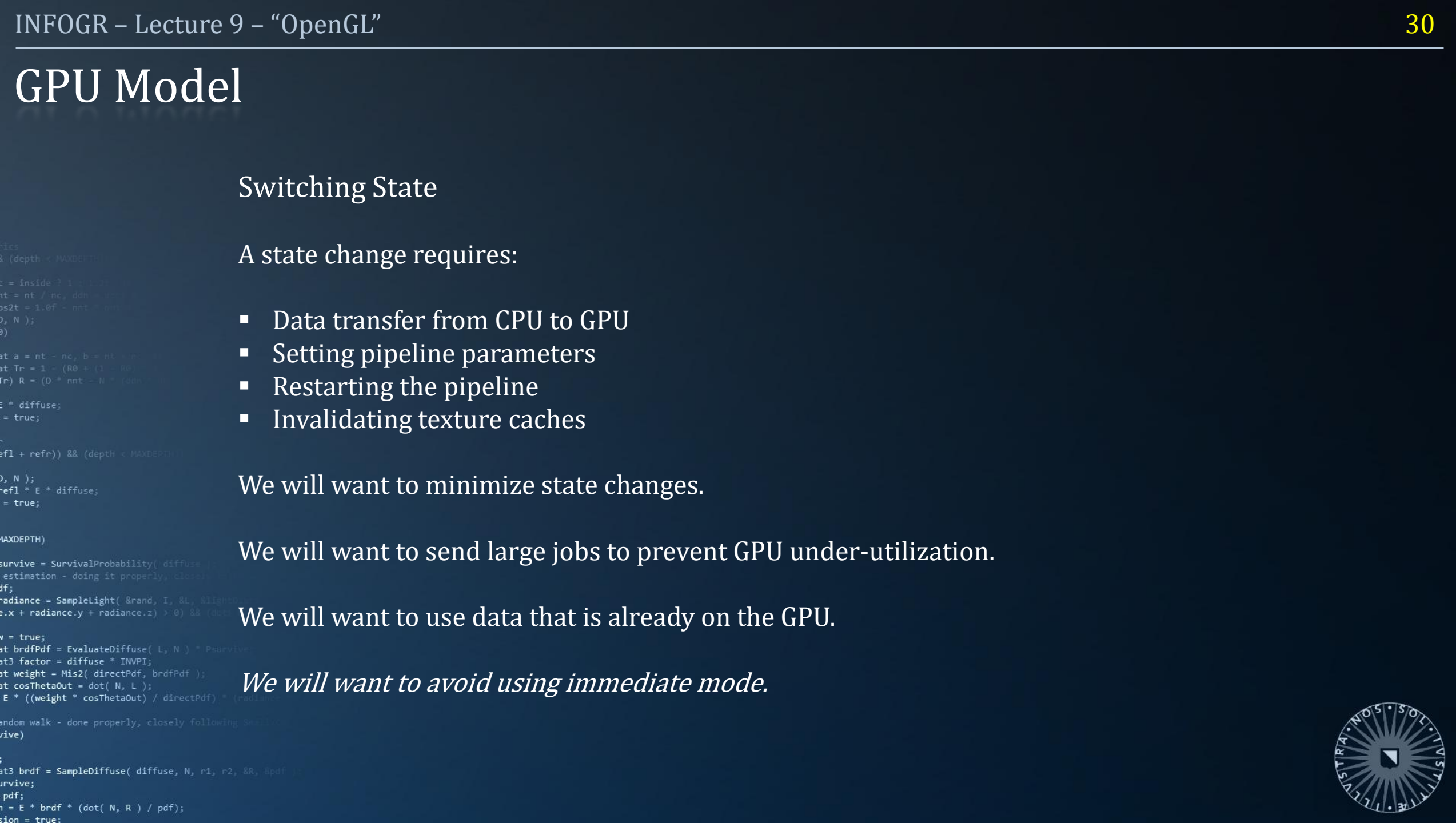
- Data transfer from CPU to GPU
- Setting pipeline parameters
- Restarting the pipeline
- Invalidating texture caches

We will want to minimize state changes.

We will want to send large jobs to prevent GPU under-utilization.

We will want to use data that is already on the GPU.

We will want to avoid using immediate mode.



GPU Model

Immediate Mode

In immediate mode, everything between GL.Begin() and GL.End() is pushed through the pipeline *right away*.

To make things worse, all parameters are passed from the CPU to the GPU.

We get:

- Expensive data transfer with severe latency
- A tiny render task for the massively parallel graphics processor.

We can improve on this using *retained mode* and *Vertex Buffer Objects*.

GPU Model

Retained Mode

In retained mode, we create a ‘list’ of commands:

```
GL.NewList( out listID, ListMode.Compile );
    GL.Begin( PrimitiveType.Triangles );
    GL.Vertex2(  0.0f,  1.0f );
    GL.Vertex2( -1.0f, -1.0f );
    GL.Vertex2(  1.0f, -1.0f );
    GL.End();
GL.EndList();
```

We can now execute the list of commands using GL.CallList:

```
GL.CallList( listID );
```

‘Compiling’ here means: optimizing for fast execution on the GPU.

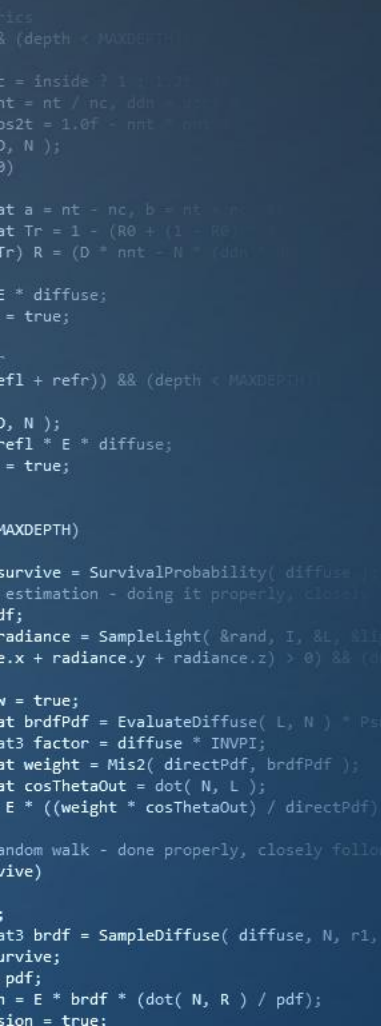
GPU Model

Vertex Buffer Objects

VBOs allow us to store vertex data in GPU memory.

```
static int vboID;
public void TickGL()
{
    GL.BindBuffer( BufferTarget.ArrayBuffer, vboID );
    GL.DrawArrays( PrimitiveType.Triangles, 0, 9 );
}
```

```
GL.GenBuffers( 1, out vboID );
float [] vertexData = { 0, 1, depth, -1, -1, depth, 1, -1, depth };
GL.BindBuffer( BufferTarget.ArrayBuffer, vboID );
GL.BufferData<float>( BufferTarget.ArrayBuffer, (IntPtr)(vertexData.Length * 4),
    vertexData, BufferUsageHint.StaticDraw );
GL.EnableClientState( ArrayCap.VertexArray );
GL.VertexPointer( 3, VertexPointerType.Float, 9, 0 );
```



GPU Model

Vertex Buffer Objects

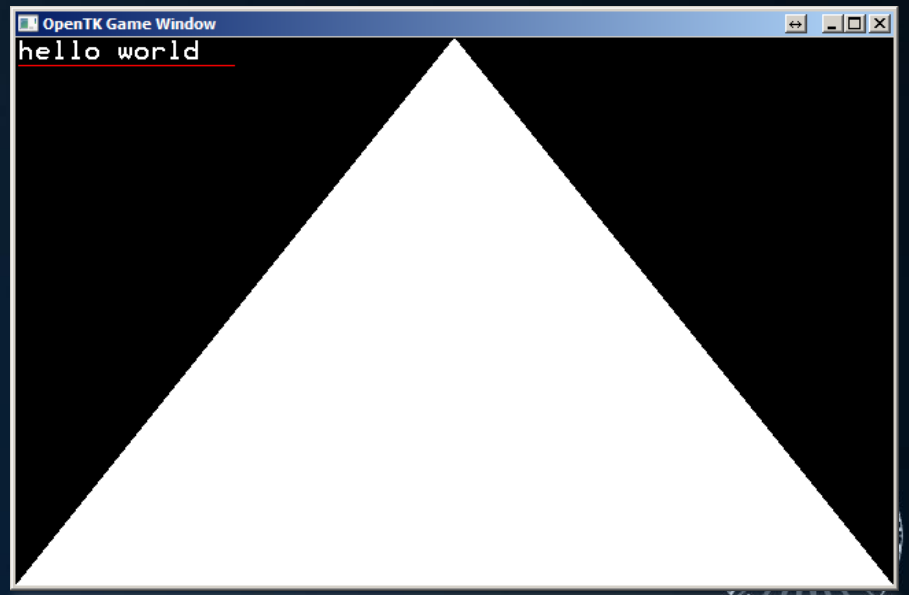
```
static int vboID;
public void TickGL()
{
    GL.BindBuffer( BufferTarget.ArrayBuffer, vboID );
    GL.DrawArrays( PrimitiveType.Triangles, 0, 9 );
}
```

equals:

```
public void TickGL()
{
    GL.Begin( PrimitiveType.Triangles );
    GL.Vertex2(  0.0f,  1.0f );
    GL.Vertex2( -1.0f, -1.0f );
    GL.Vertex2(  1.0f, -1.0f );
    GL.End();
}
```

Colors / texture coordinates / etc.:

1. Pass additional data in the vertex array, enable it using GL.EnableClientState.
2. Start using shaders, see P1.



GPU Model

Optimizing GPU Usage

We don’t send individual commands to the GPU, but we batch them in lists.

We don’t send the texture to the GPU for each frame, we just use the ‘ID’ of a texture, managed by OpenGL.

We now also don’t send vertex data to the GPU anymore: the data is already on the device.

Problem:

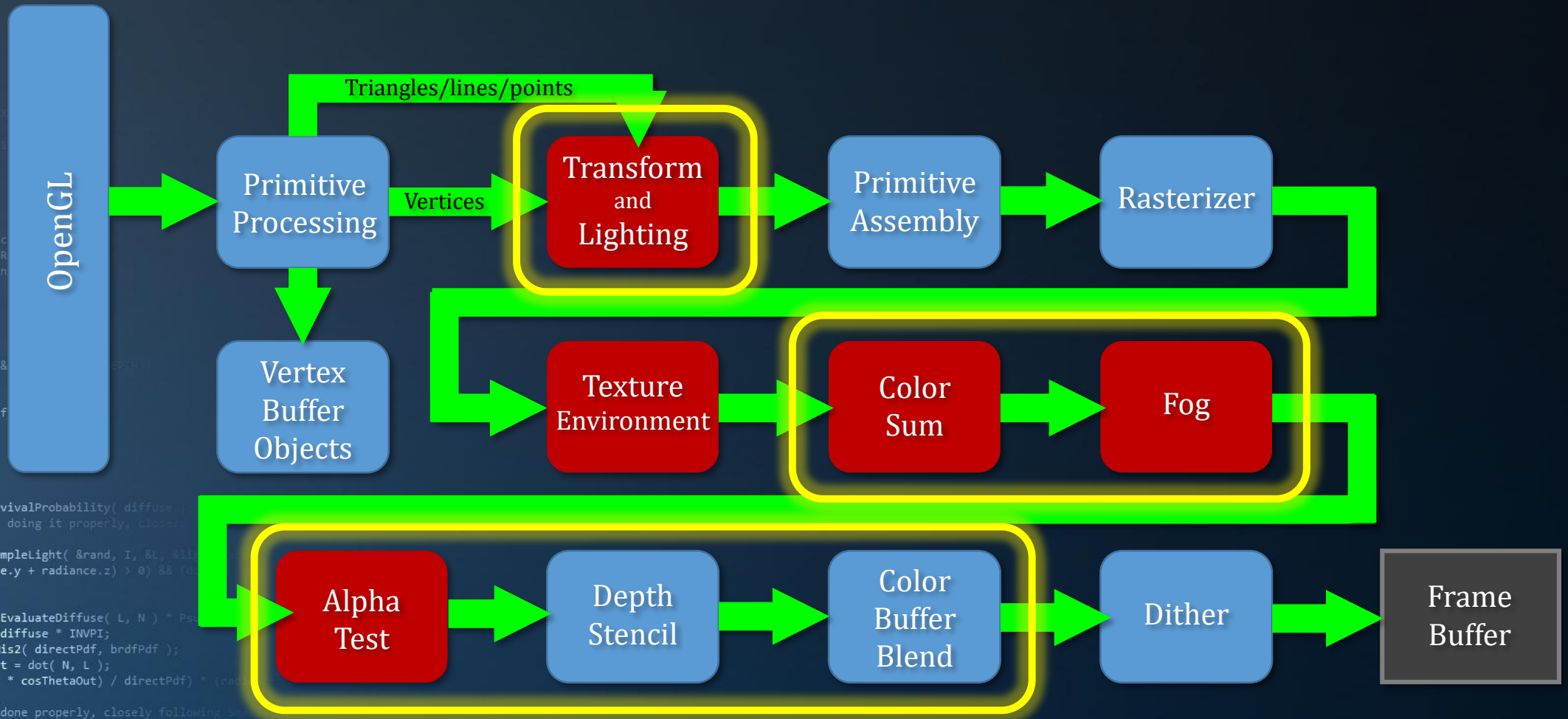
What happens when we want to change variable depth?

```
GL.NewList( out listID, ListMode.Compile );
    GL.Begin( PrimitiveType.Triangles );
    ...
GL.End();
GL.EndList();
```

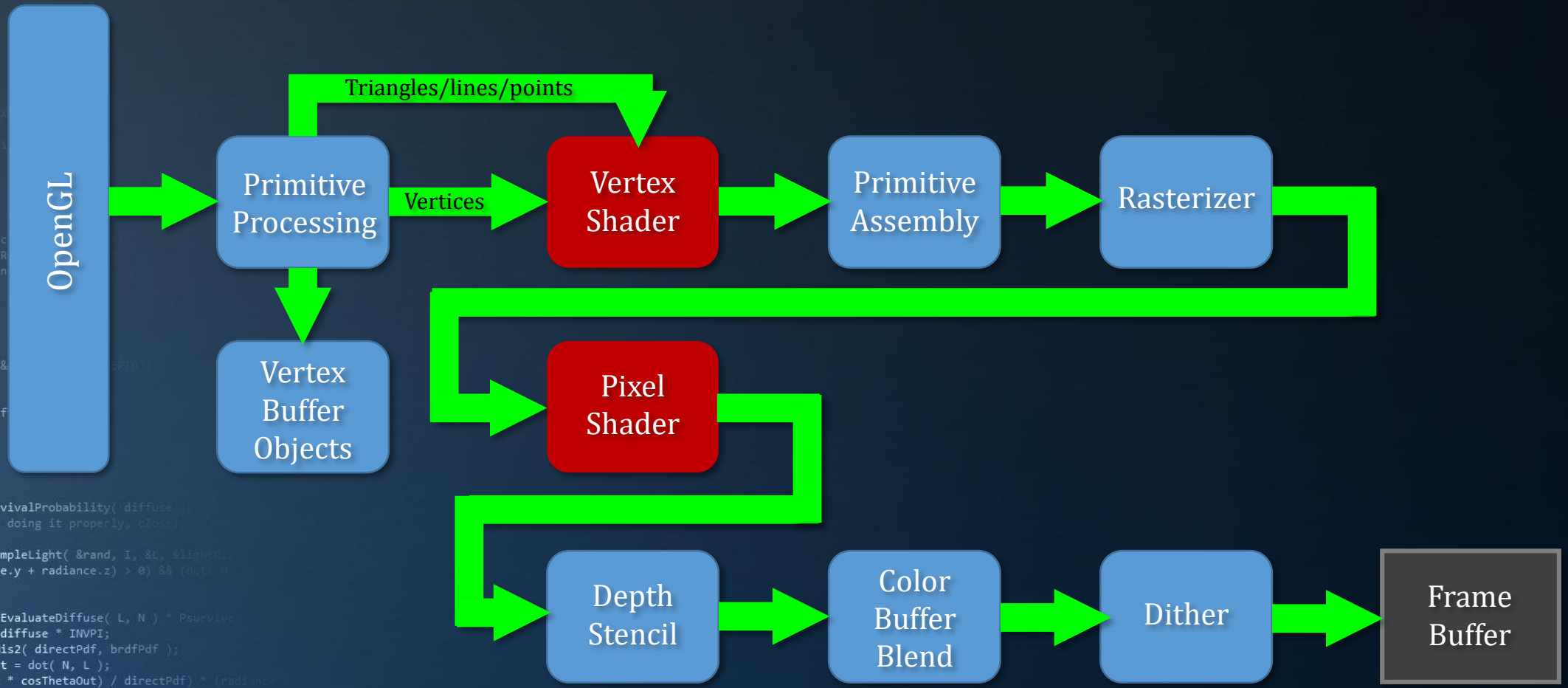
```
textureID = screen.GenTexture();
GL.BindTexture( TextureTarget.Texture2D, textureID );
uint [] data = new uint[64 * 64];
...
GL.TexImage2D( TextureTarget.Texture2D, ... , data );
```

```
GL.GenBuffers( 1, out vboID );
float [] vertexData = { 0, 1, depth, -1, -1, depth, 1, -1, depth };
GL.BindBuffer( BufferTarget.ArrayBuffer, vboID );
GL.BufferData<float>( BufferTarget.ArrayBuffer, ... , vertexData, ... );
GL.EnableClientState( ArrayCap.VertexArray );
GL.VertexPointer( 3, VertexPointerType.Float, 12, 0 );
```

GPU Model


OpenGL
Triangles/lines/points
Primitive Processing
Vertices
Transform and Lighting
Primitive Assembly
Rasterizer
Vertex Buffer Objects
Texture Environment
Color Sum
Fog
Alpha Test
Depth Stencil
Color Buffer Blend
Dither
Frame Buffer

GPU Model


OpenGL
Primitive Processing
Triangles/lines/points
Vertices
Vertex Shader
Primitive Assembly
Rasterizer
Vertex Buffer Objects
Pixel Shader
Depth Stencil
Color Buffer Blend
Dither
Frame Buffer

GPU Model

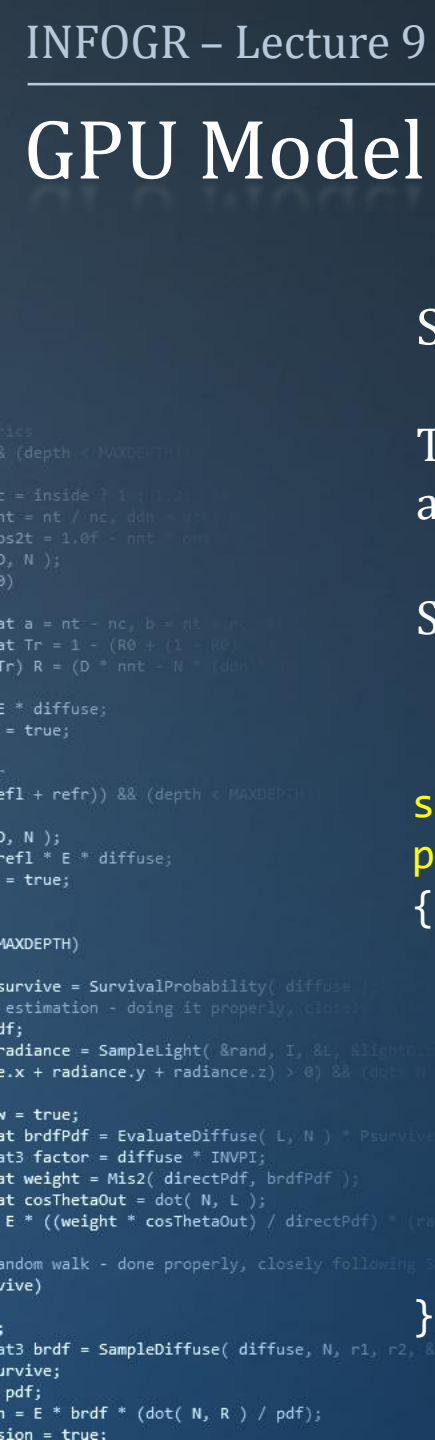
Shaders: Consequences

The ‘transform & lighting’ stage is programmable,
as in: ‘it must be programmed’.

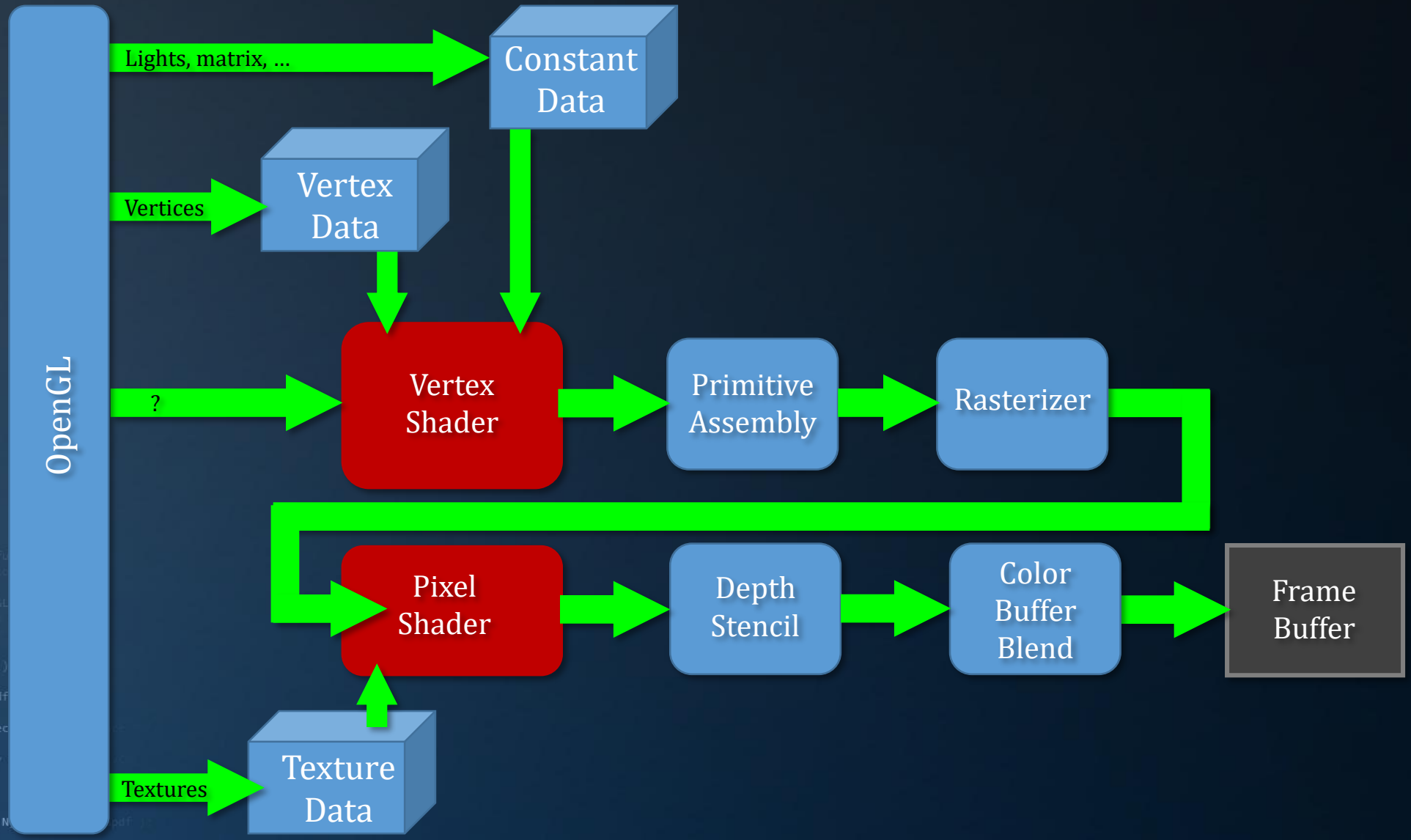
So, what happened here?

OpenGL emulates a fixed function pipeline to support old OpenGL code.

```
static float r = 0.0f;
public void TickGL()
{
    GL.Frustum( -1.0f, 1.0f, -1.0f, 1.0f, 1.0f, 15.0f );
    GL.Translate( 0, 0, -2 );
    GL.Rotate( r, 0, 1, 0 );
    GL.Begin( PrimitiveType.Triangles );
    ...
    GL.End();
    r += 0.1f;
}
```



GPU Model


OpenGL
Lights, matrix, …
Constant Data
Vertices
Vertex Data
?
Vertex Shader
Primitive Assembly
Rasterizer
Pixel Shader
Depth Stencil
Color Buffer Blend
Frame Buffer
Textures
Texture Data

GPU Model

OpenGL and the GPU

Digest:

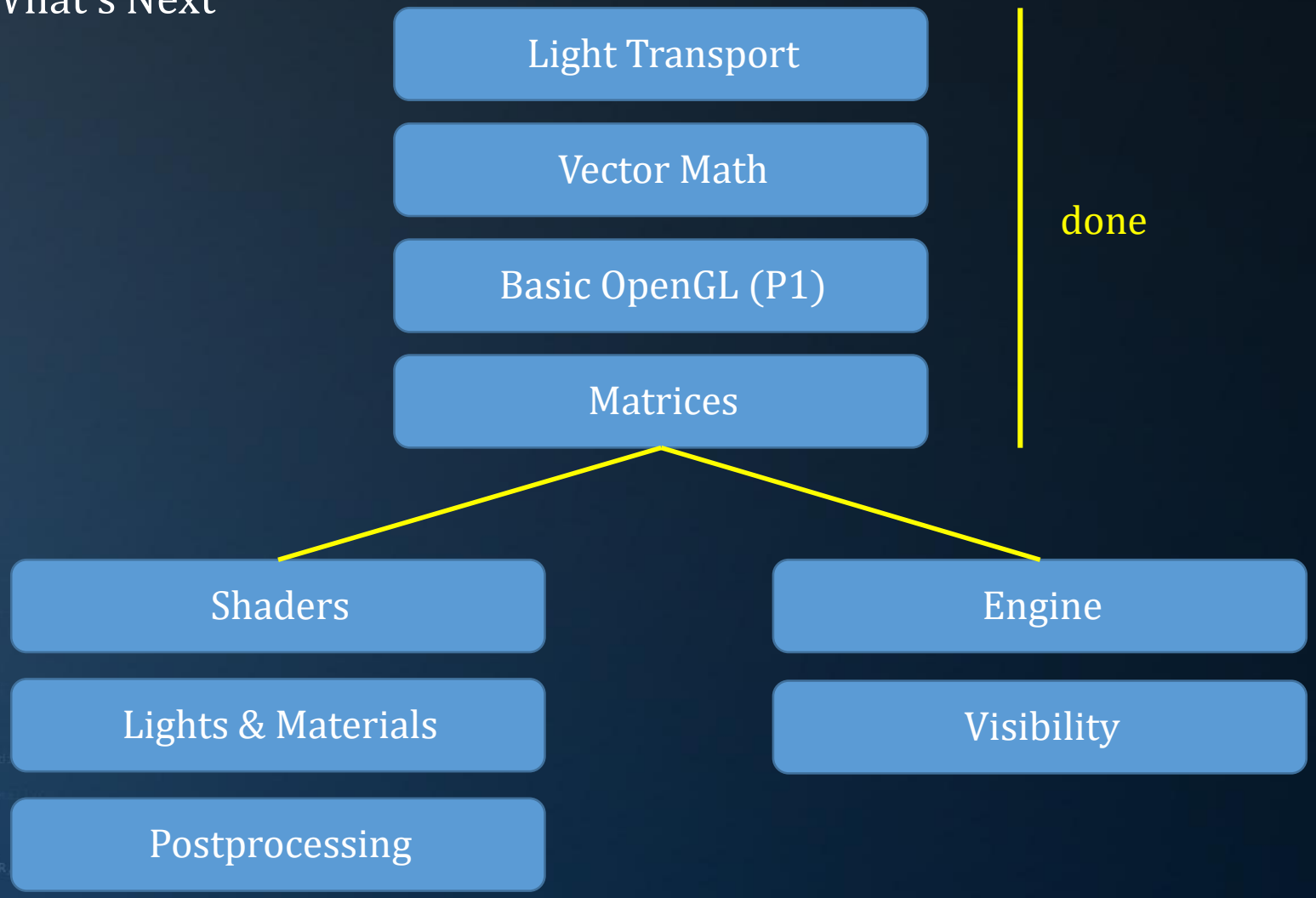
- Modern OpenGL allows us to keep data on the GPU
- Retained mode allows OpenGL to reorder / optimize draw commands
- Vertex and pixel shaders make large parts of ‘classic’ OpenGL redundant

Today’s Agenda:

- Introduction
- OpenGL
- GPU Model
- Upcoming
- Assignment P3

Upcoming

What’s Next



Today’s Agenda:

- Introduction
- OpenGL
- GPU Model
- Upcoming
- Assignment P3

Practical

Assignment P3

New framework! Already done for you:

- Mesh storage and rendering: mesh.cs
- .OBJ file loading: meshLoader.cs
- Shader loading, compilation, binding: shader.cs
- Texture loading: texture.cs
- Render target: renderTarget.cs
- Quad rendering: quad.cs

Assignment goals:

- Implement a scene graph
- Implement proper shaders

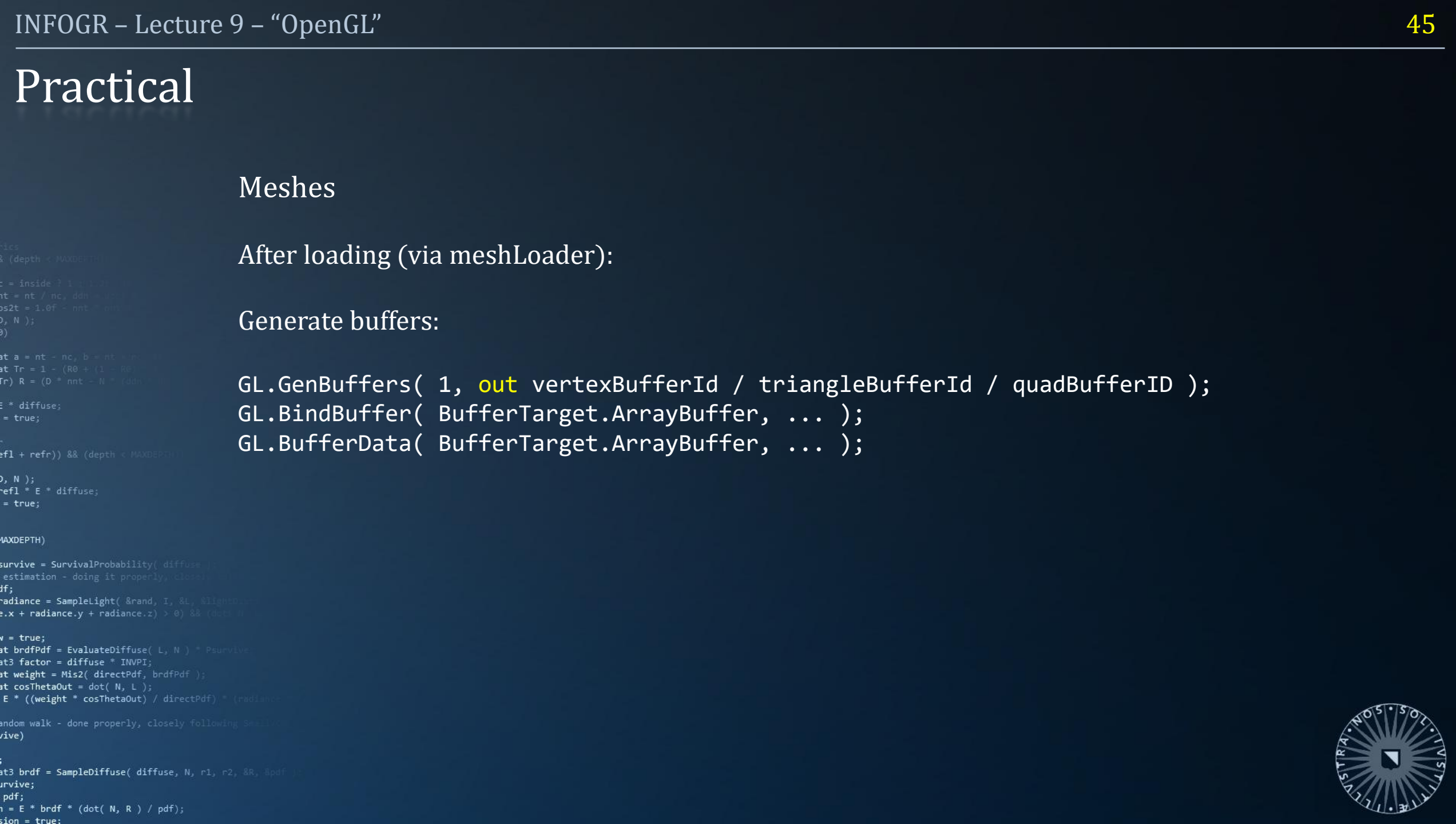
Practical

Meshes

After loading (via meshLoader):

Generate buffers:

```
GL.GenBuffers( 1, out vertexBufferId / triangleBufferId / quadBufferID );
GL.BindBuffer( BufferTarget.ArrayBuffer, ... );
GL.BufferData( BufferTarget.ArrayBuffer, ... );
```



Practical

Meshes

To render, bind the texture to the shader:

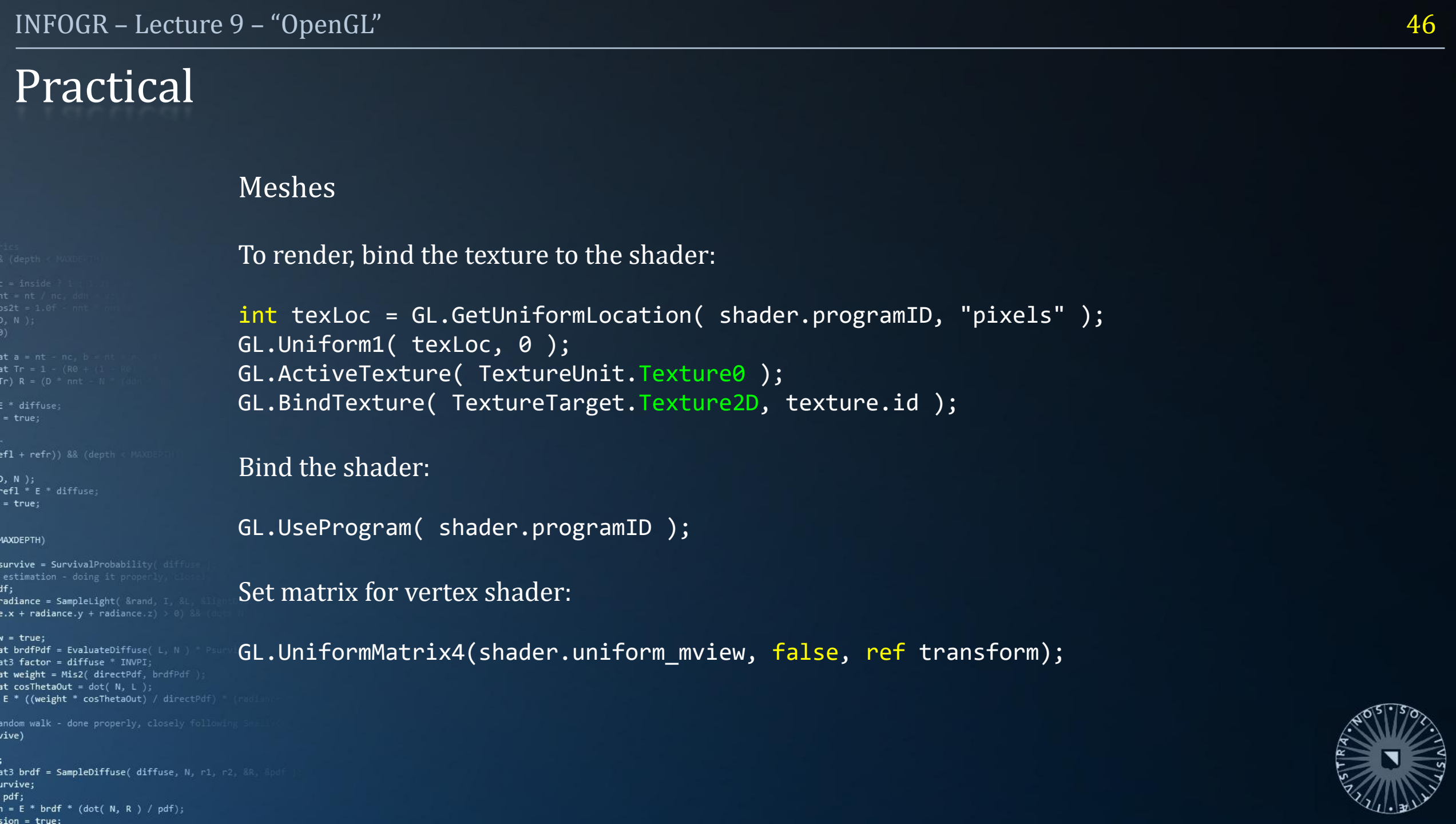
```
int texLoc = GL.GetUniformLocation( shader.programID, "pixels" );
GL.Uniform1( texLoc, 0 );
GL.ActiveTexture( TextureUnit.Texture0 );
GL.BindTexture( TextureTarget.Texture2D, texture.id );
```

Bind the shader:

```
GL.UseProgram( shader.programID );
```

Set matrix for vertex shader:

```
GL.UniformMatrix4(shader.uniform_mview, false, ref transform);
```



Practical

Meshes

Enable VertexArray usage, bind the vertex buffer, specify data layout:

```
GL.EnableClientState( ArrayCap.VertexArray );
GL.BindBuffer( BufferTarget.ArrayBuffer, vertexBufferId );
GL.InterleavedArrays( InterleavedArrayFormat.T2fN3fV3f, ... );
```

Link the vertex attributes to the shader:

```
GL.VertexAttribPointer( shader.attribute_vuvs, 2, VertexAttribPointerType.Float, false, 32, 0 );
GL.VertexAttribPointer( shader.attribute_vnrm, 3, VertexAttribPointerType.Float, true, 32, 2 * 4 );
GL.VertexAttribPointer( shader.attribute_vpos, 3, VertexAttribPointerType.Float, false, 32, 5 * 4 );
```

Enable the attributes:

```
GL.EnableVertexAttribArray( shader.attribute_vpos );
GL.EnableVertexAttribArray( shader.attribute_vnrm );
GL.EnableVertexAttribArray( shader.attribute_vuvs );
```

Practical

Meshes

Bind the index array and render:

```
GL.BindBuffer( BufferTarget.ElementArrayBuffer, triangleBufferId );
GL.DrawArrays( PrimitiveType.Triangles, 0, triangles.Length * 3 );
```

…But, the important part is:

```
public void Render( Shader shader, Matrix4 transform, Texture texture )
{
}
```

Practical

Frame Buffer Object (FBO)

```
public void Bind()
{
    GL.Ext.BindFramebuffer( FramebufferTarget.FramebufferExt, fbo );
    GL.Clear( ClearBufferMask.DepthBufferBit );
    GL.Clear( ClearBufferMask.ColorBufferBit );
}

public void Unbind()
{
    GL.Ext.BindFramebuffer( FramebufferTarget.FramebufferExt, 0 );
}
```

This allows you to *render to a texture*.
Why? So we can put the texture on a screen filling quad, which we can render with a shader.
This enables post processing effects.

Practical

Bringing it all together

In MyApplication.cs:

```
Mesh mesh, floor;
Shader shader;
Shader postproc;
Texture wood;
RenderTarget target;
ScreenQuad quad;
```

We will use this data to:

- Render two meshes (‘mesh’ and ‘floor’)
- using texture ‘wood’ and shader ‘shader’
- onto render target ‘target’,
- which we use to texture ‘quad’ (which is essentially also a mesh).

Practical

```
#version 330

// shader input
in vec2 vUV;
in vec3 vNormal;
in vec3 vPosition;

// shader output
out vec4 normal;
out vec2 uv;
uniform mat4 transform;

// vertex shader
void main()
{
    // transform vertex using supplied matrix
    gl_Position = transform * vec4( vPosition, 1.0 );

    // forward normal and uv coordinate
    normal = transform * vec4( vNormal, 0.0f );
    uv = vUV;
}
```

```
#version 330

// shader input
in vec2 uv;
in vec4 normal;
uniform sampler2D pixels;

// shader output
out vec4 outputColor;

// fragment shader
void main()
{
    outputColor = texture( pixels, uv ) +
            0.5f * vec4( normal.xyz, 1 );
}
```

Practical

```
#version 330

// shader input
in vec2 vUV;
in vec3 vPosition;

// shader output
out vec2 uv;
out vec2 P;

// vertex shader
void main()
{
    uv = vUV;
    P = vec2( vPosition ) * 0.5 + vec2( 0.5, 0.5 );
    gl_Position = vec4( vPosition, 1 );
}
```

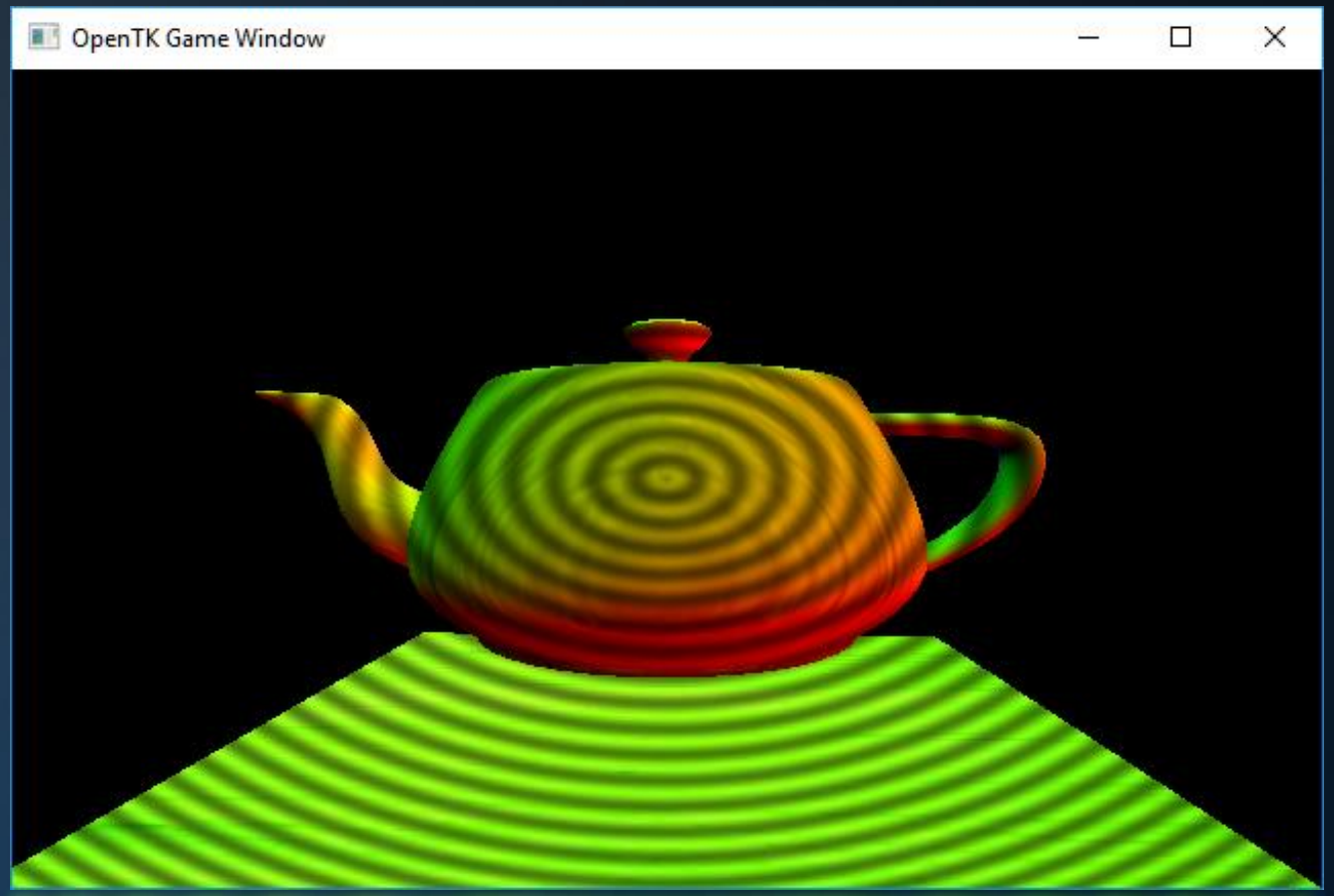
```
#version 330

// shader input
in vec2 P;
in vec2 uv;
uniform sampler2D pixels;

// shader output
out vec3 outputColor;

void main()
{
    outputColor = texture( pixels, uv ).rgb;
    float dx = P.x - 0.5, dy = P.y - 0.5;
    float distance = sqrt( dx * dx + dy * dy );
    outputColor *= sin( distance * 200.0f )
                *  0.25f + 0.75f;
}
```

Practical



INFOGR – Computer Graphics

Jacco Bikker & Debabrata Panja - April-July 2019

END OF lecture 9: “OpenGL”

Next lecture: “Shaders”