

Real-time Graphics

1. Graphics Pipeline, Shaders

Graphics Pipeline

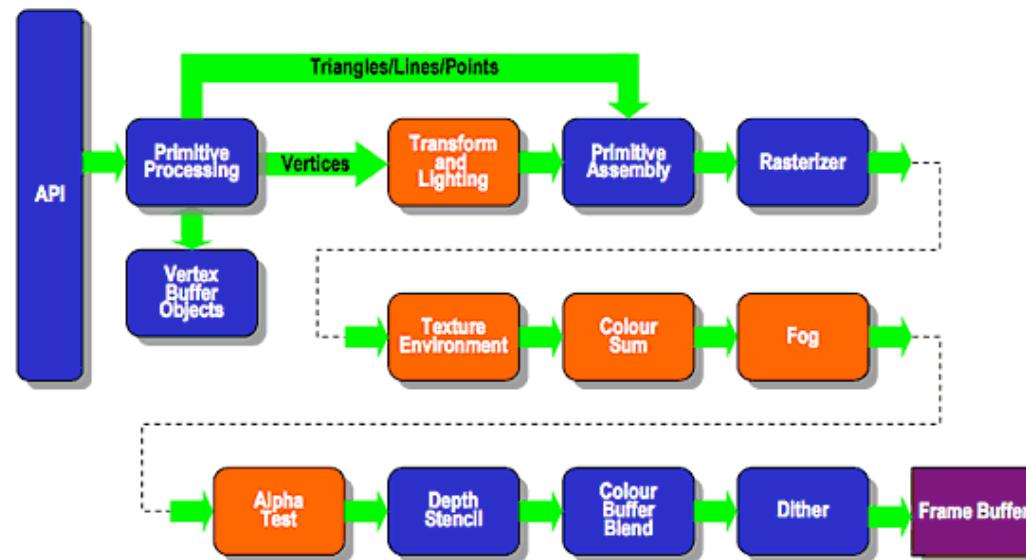
- Rasterization-based real-time rendering
- Supported by common hardware - graphics cards
- Input = 3D representation of scene
- Output = 2D raster image
- Stream processing
- Another ways of rendering: raytracing, global illumination (radiosity), REYES, ...



OpenGL Graphics Pipeline

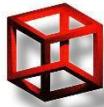
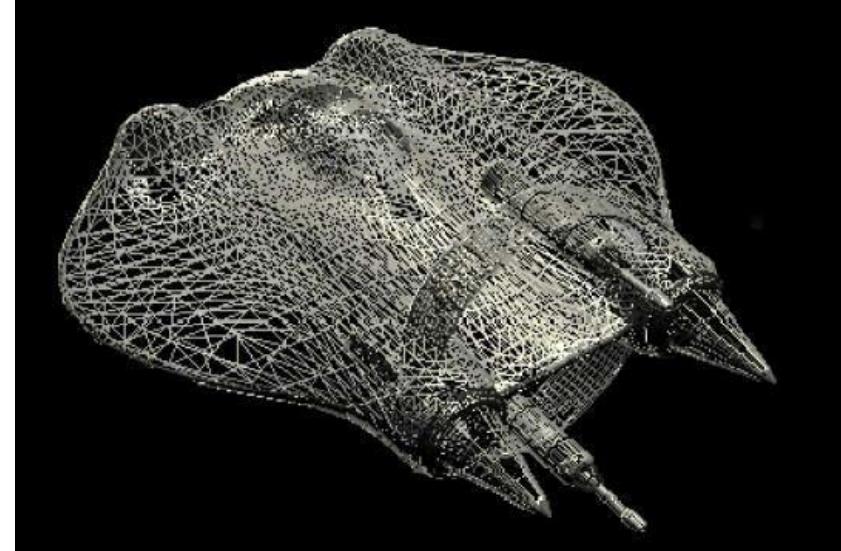
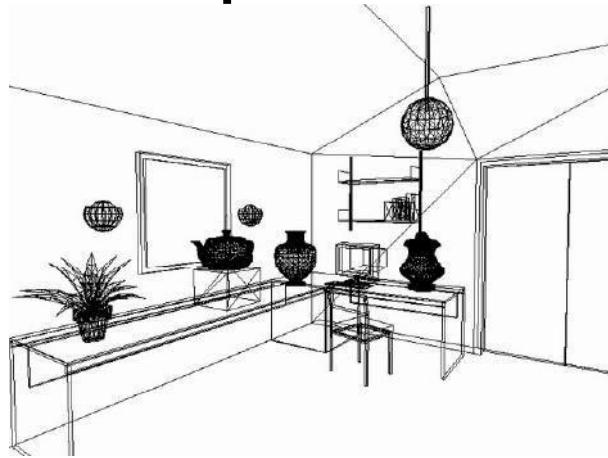
- Fixed pipeline - prerequisite for this course
- Knowledge of extensions mechanism
- Some parts are depreciated

Existing Fixed Function Pipeline



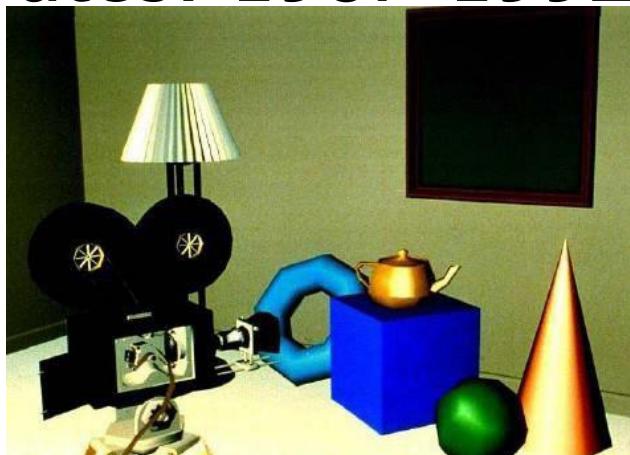
First generation

- Vertex: transform, clip, project
- Pixel: color interpolation of lines
- Frame buffer: overwrite
- Dates: prior to 1987



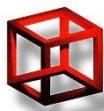
Second generation

- Vertex: lighting generation
- Pixel: depth interpolation, triangles
- Frame buffer: depth buffer, blending
- Dates: 1987-1992



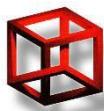
Third generation

- Vertex: texture coordinate transformation
- Pixel: texture coordinate interpolation, texture evaluation and filtering
- Dates: 1992 - 2000



Fourth generation

- Programmable shading: Vertex, Pixel, Geometry
- Multi GPU (SLI, Crossfire), Full floating point, GPGPU



Real-time Graphics
Martin Samuelčík

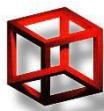
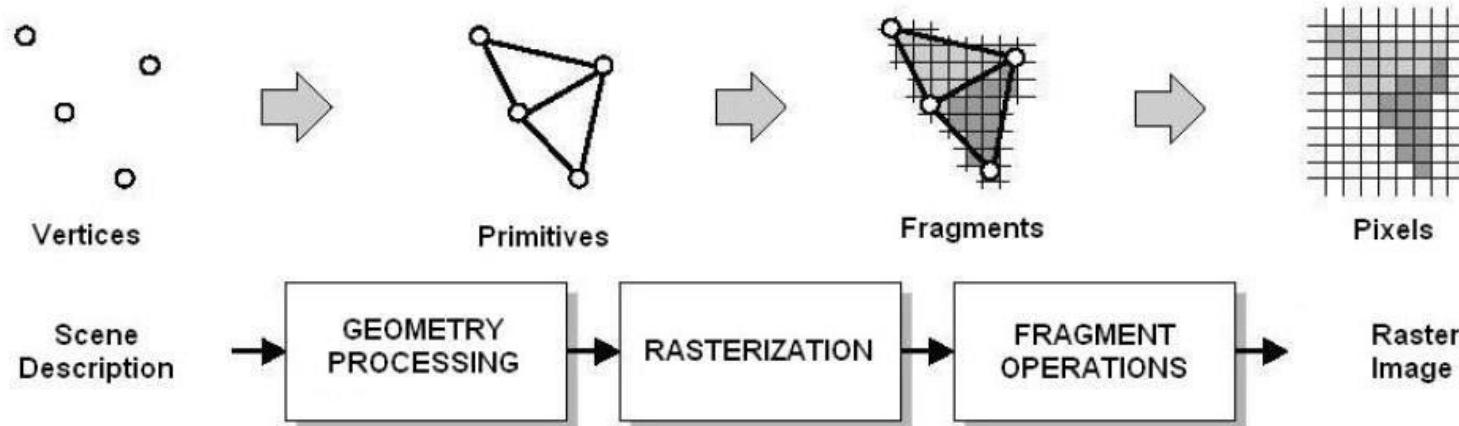
Fifth generation

- Multi cores
- Merging CPU and GPU
 - IBM Cell (8+1cores)
 - AMD Fusion (CPU+GPU)
 - Intel Larabee – canceled
- Programming
 - Tesselation shader
 - DX11 (Compute shader)
 - CUDA, AMD Brook
 - OpenCL



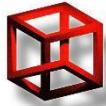
Parts of pipeline

- Application – user settings, scene description
- Geometry – transformations, clipping, projection
- Rasterization – computations, texturing
- Fragments – tests, blending, operations



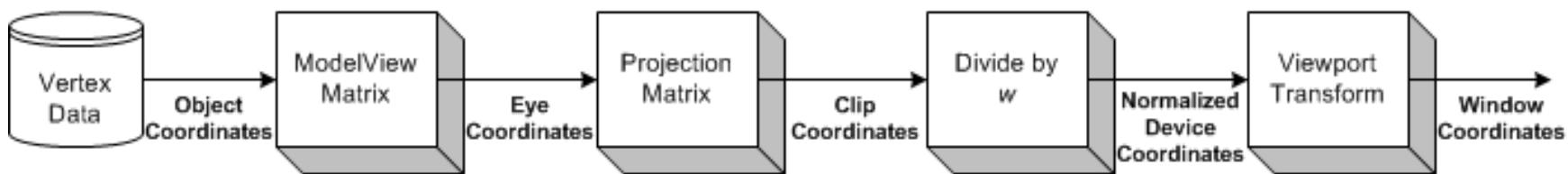
Application stage

- Scene description
- User full control
- Scene complexity – LOD, clipping
- Scene dynamics (physics)
- Handling – mouse, keyboard

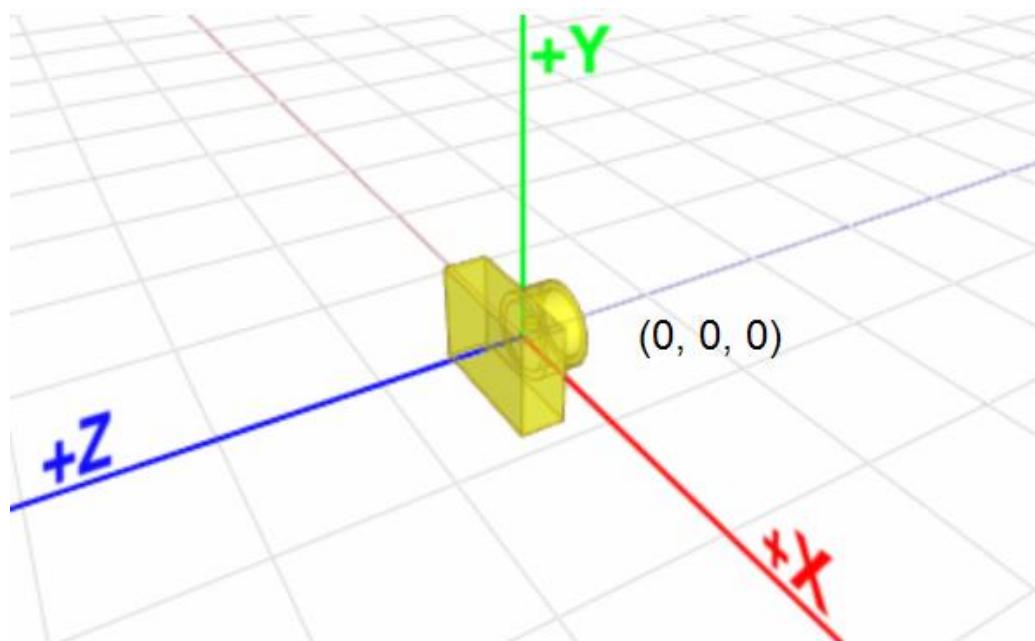
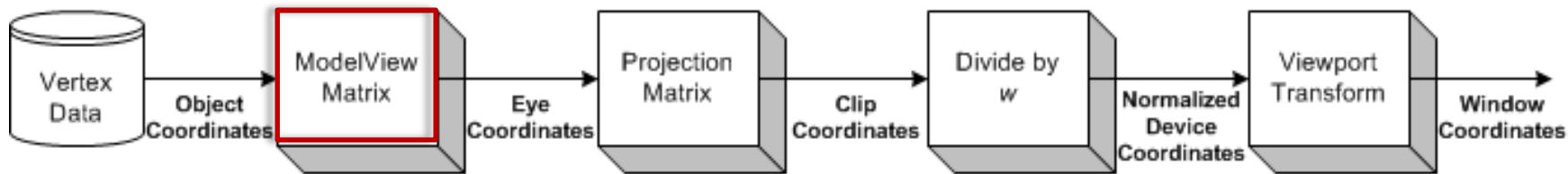


Geometry stage

- Per-primitive operations
 - Model & view transform
 - Per-vertex lightning & shading
 - Projection
 - Clipping
 - Screen mapping

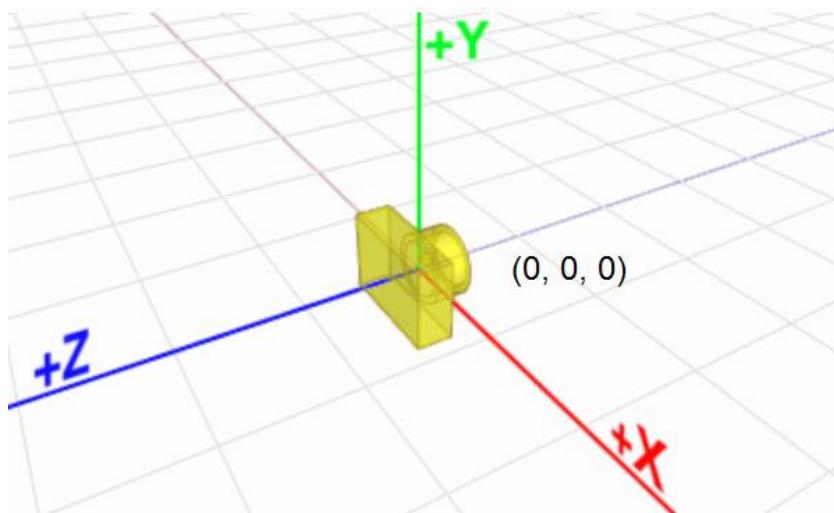
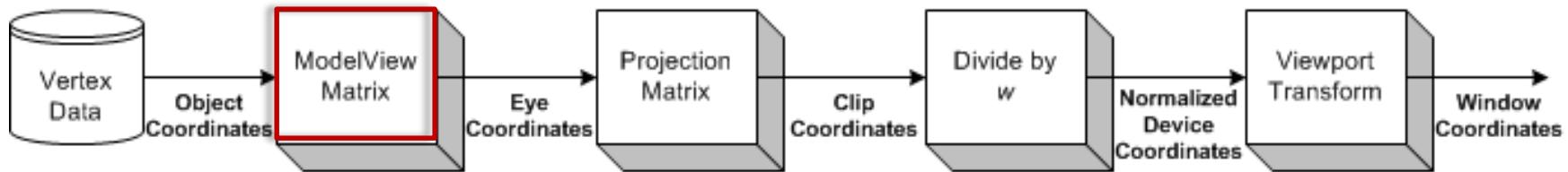


Geometry stage

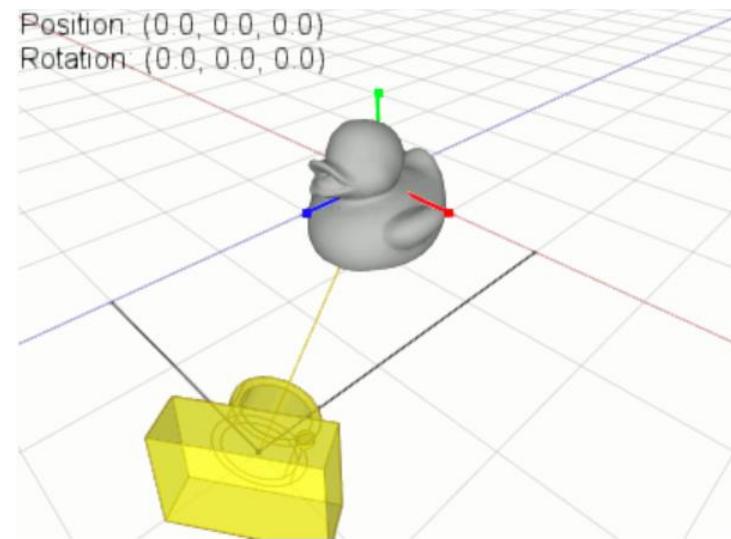


OpenGL camera is always at origin and facing to -Z in eye space

Geometry stage

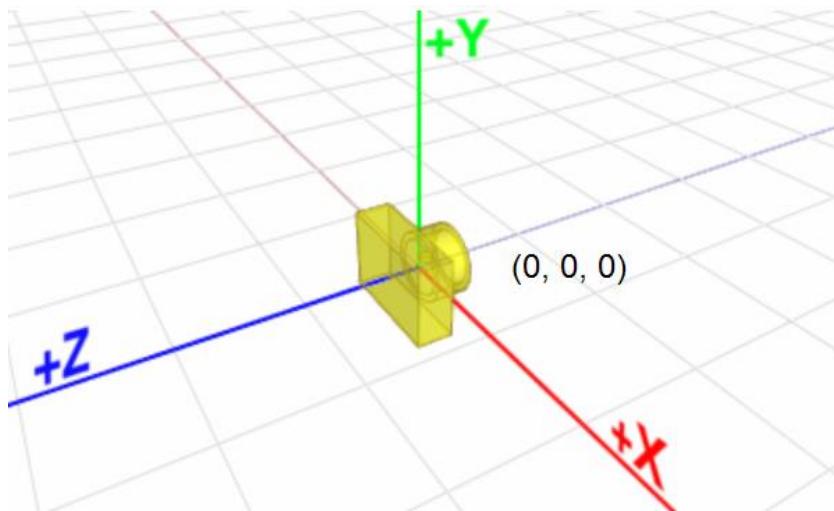
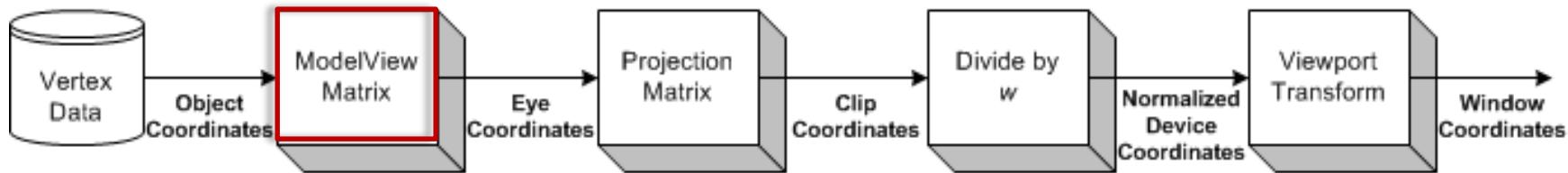


OpenGL camera is always at origin and facing to -Z in eye space

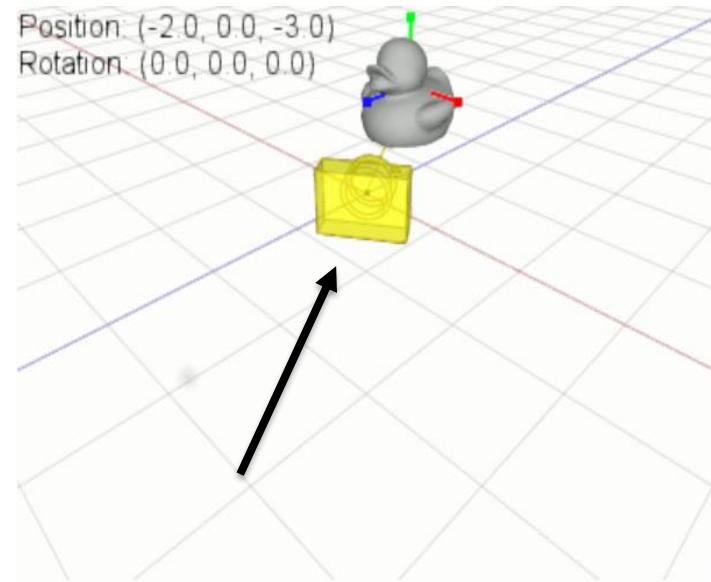


OpenGL camera's lookAt() transformation

Geometry stage

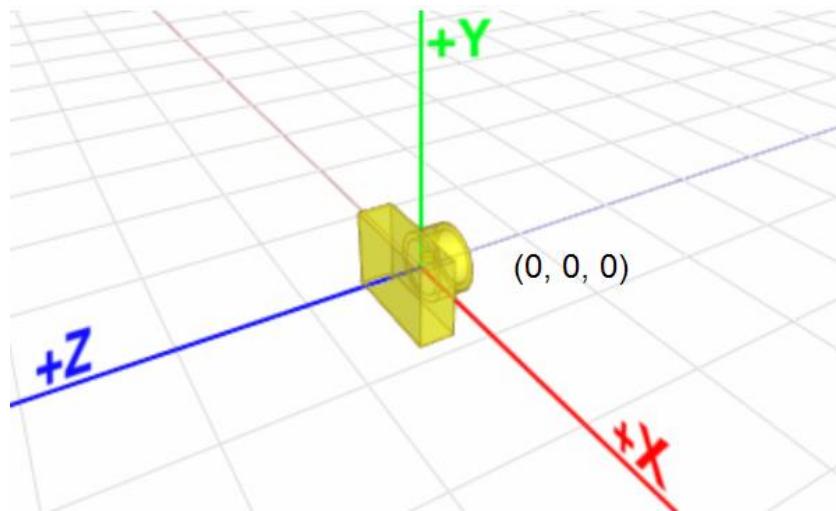
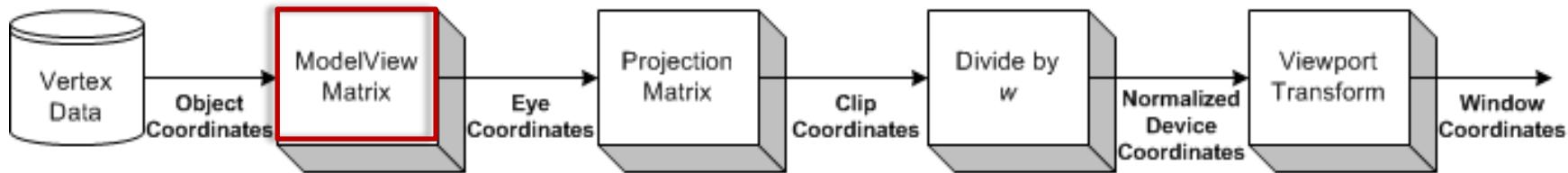


OpenGL camera is always at origin and facing to -Z in eye space

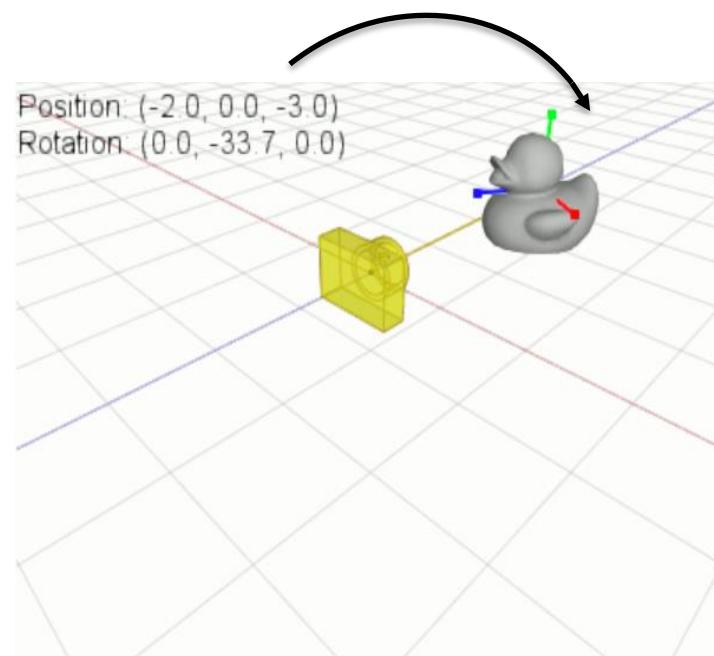


OpenGL camera's lookAt() transformation

Geometry stage

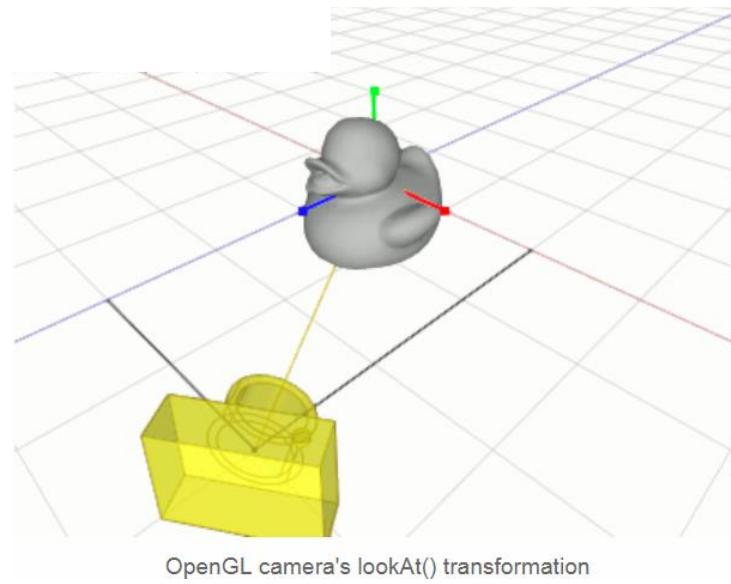
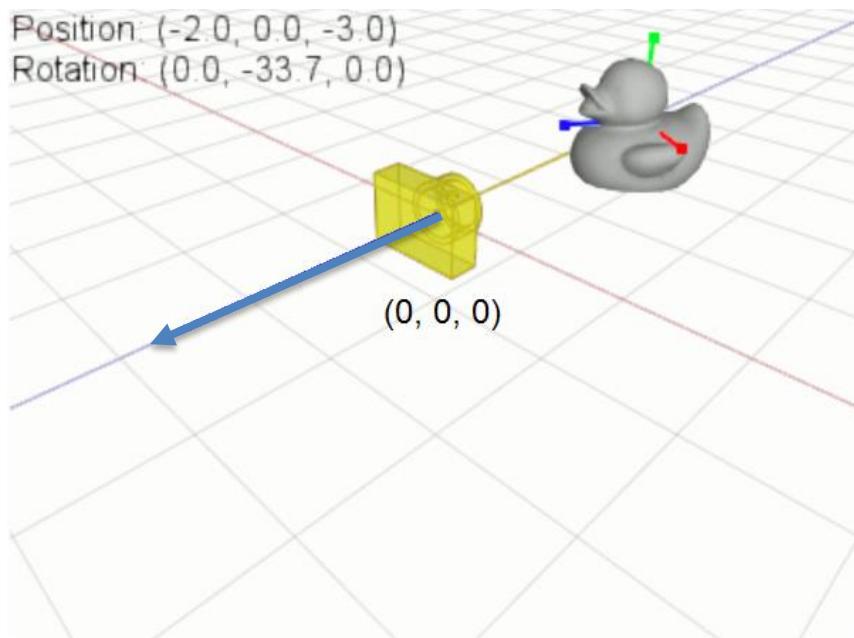
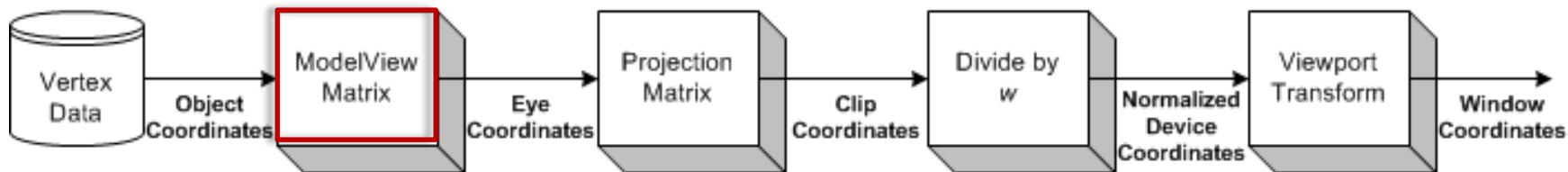


OpenGL camera is always at origin and facing to -Z in eye space

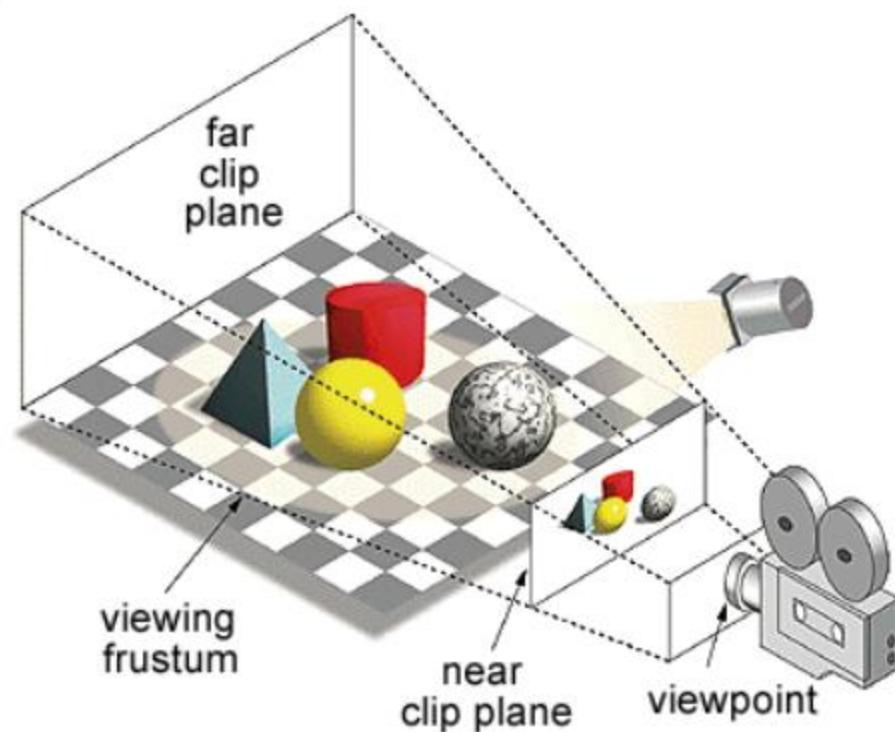
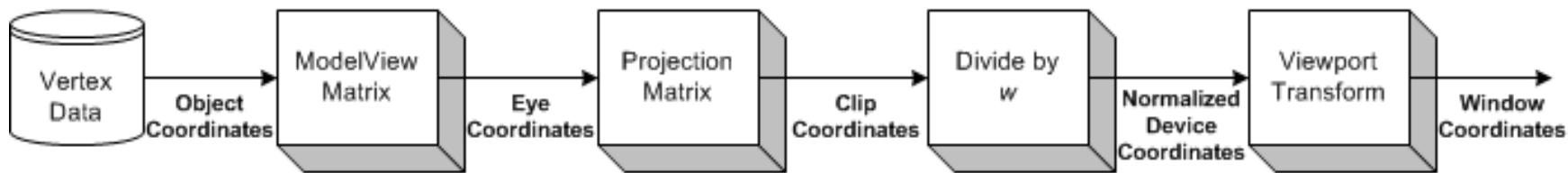


OpenGL camera's lookAt() transformation

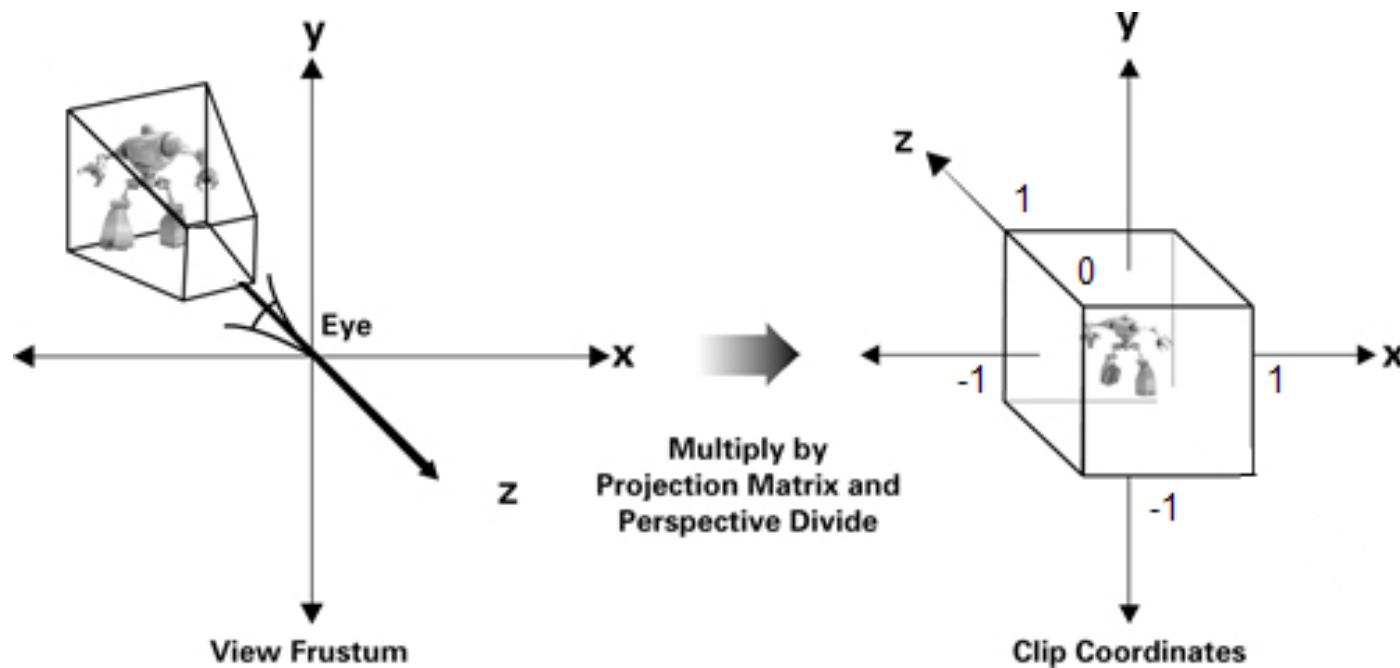
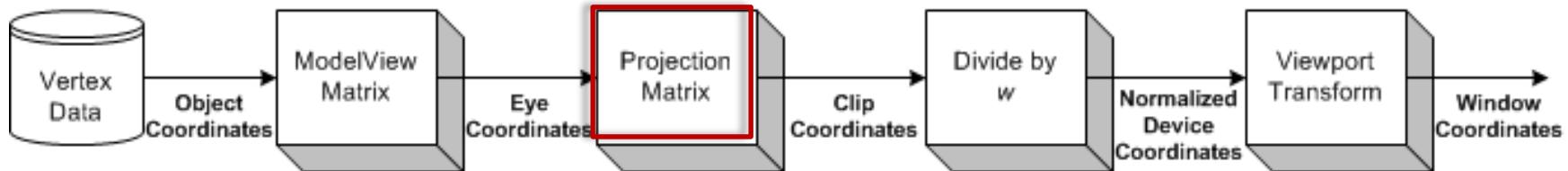
Geometry stage



Geometry stage

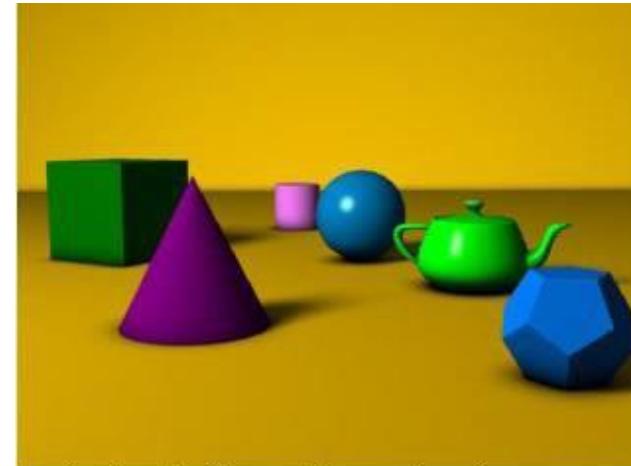


Geometry stage



Fragment tests

- Ownership
- Scissor test
- Alpha test
- Stencil test
- Depth test
- Alpha blending
- Logical operations



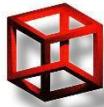
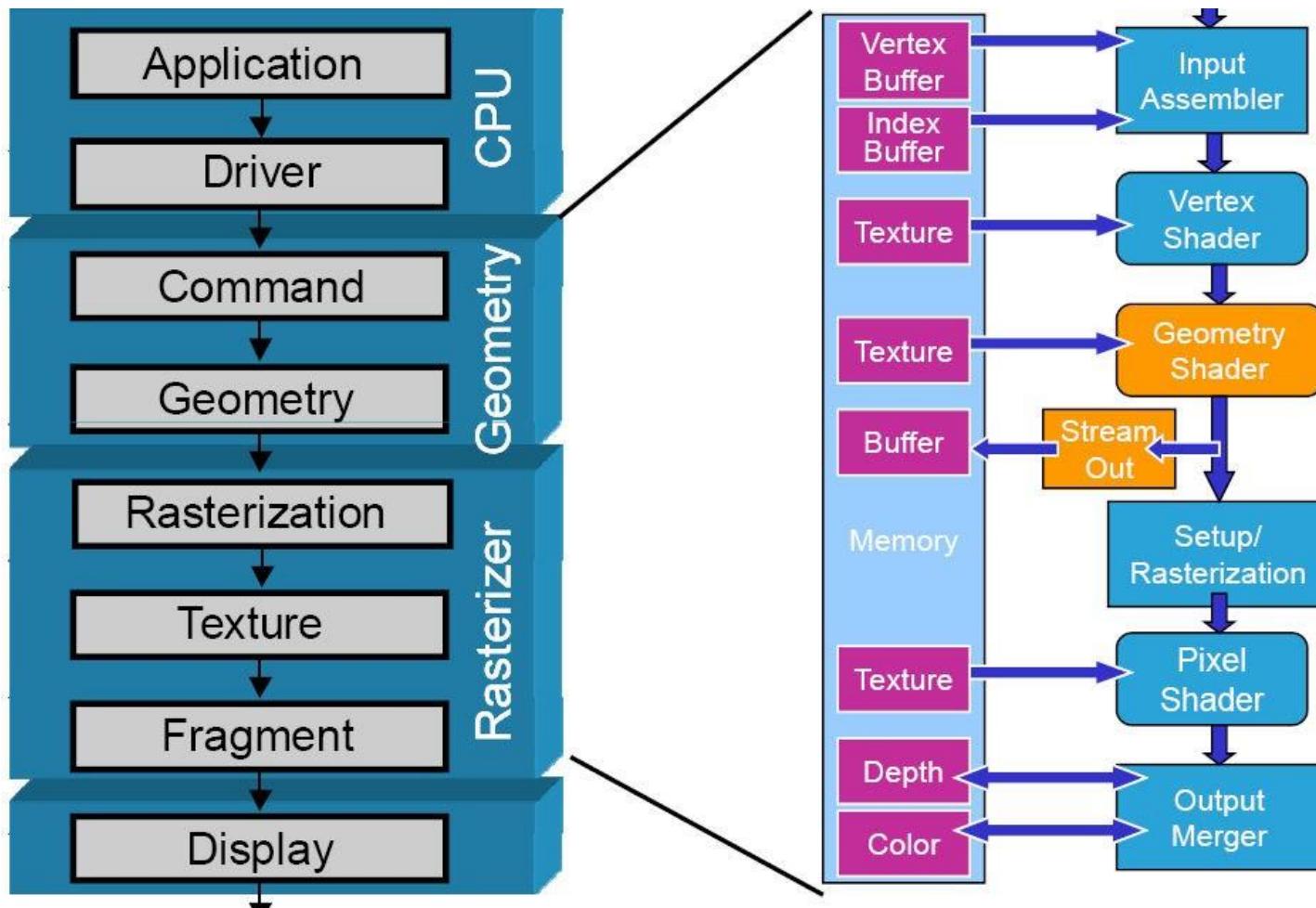
A simple three dimensional scene



Z-buffer representation



DirectX 10 pipeline



DirectX 11 pipeline

Input Assembler

Vertex Shader

Hull Shader

Tessellator

Domain Shader

Geometry Shader

Rasterizer

Pixel Shader

Output Merger

Stream Output

Direct3D 10 pipeline

Plus

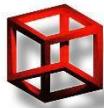
Three new stages for
Tessellation

Plus

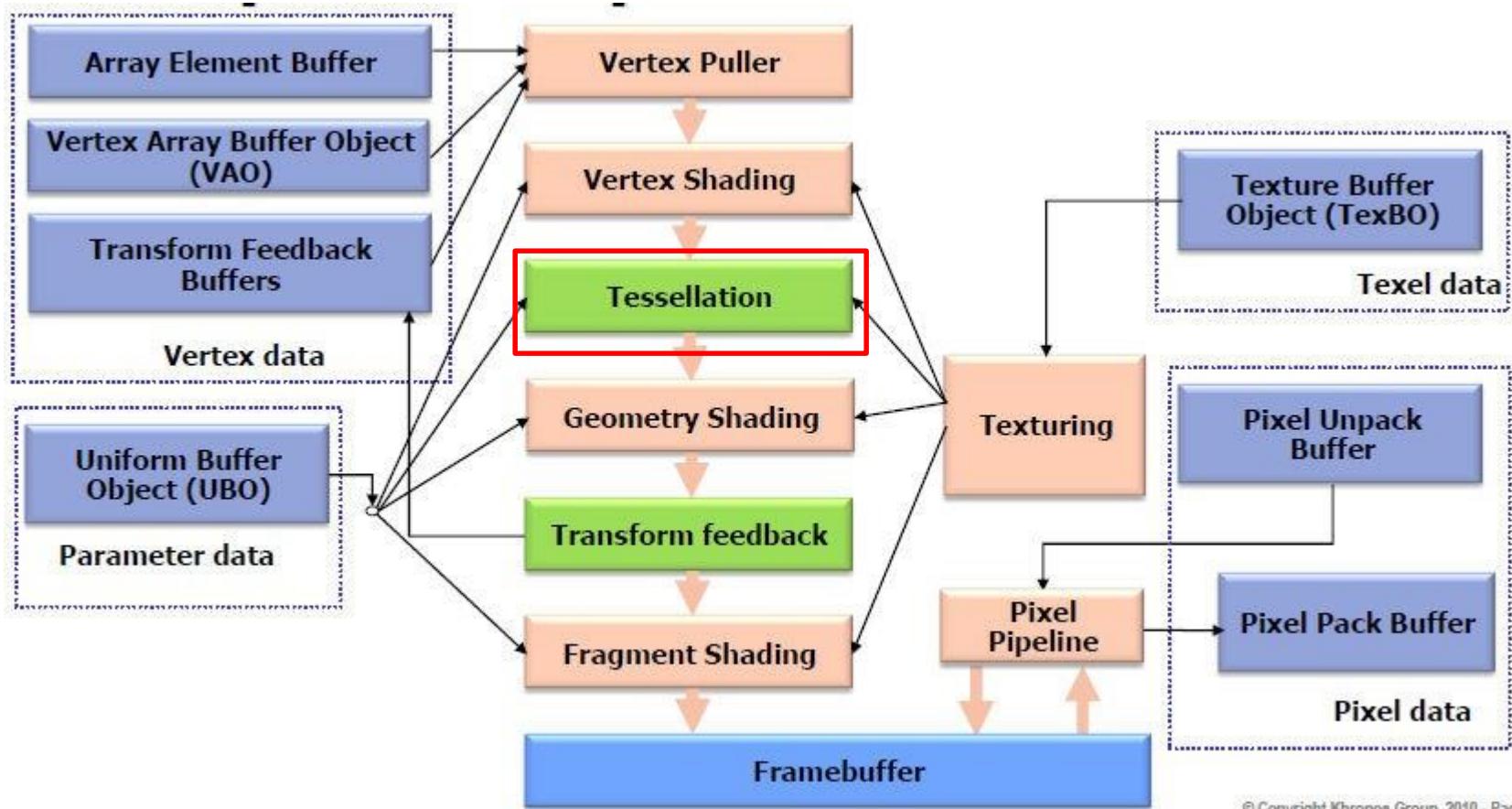
Compute Shader



Compute
Shader



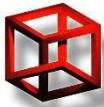
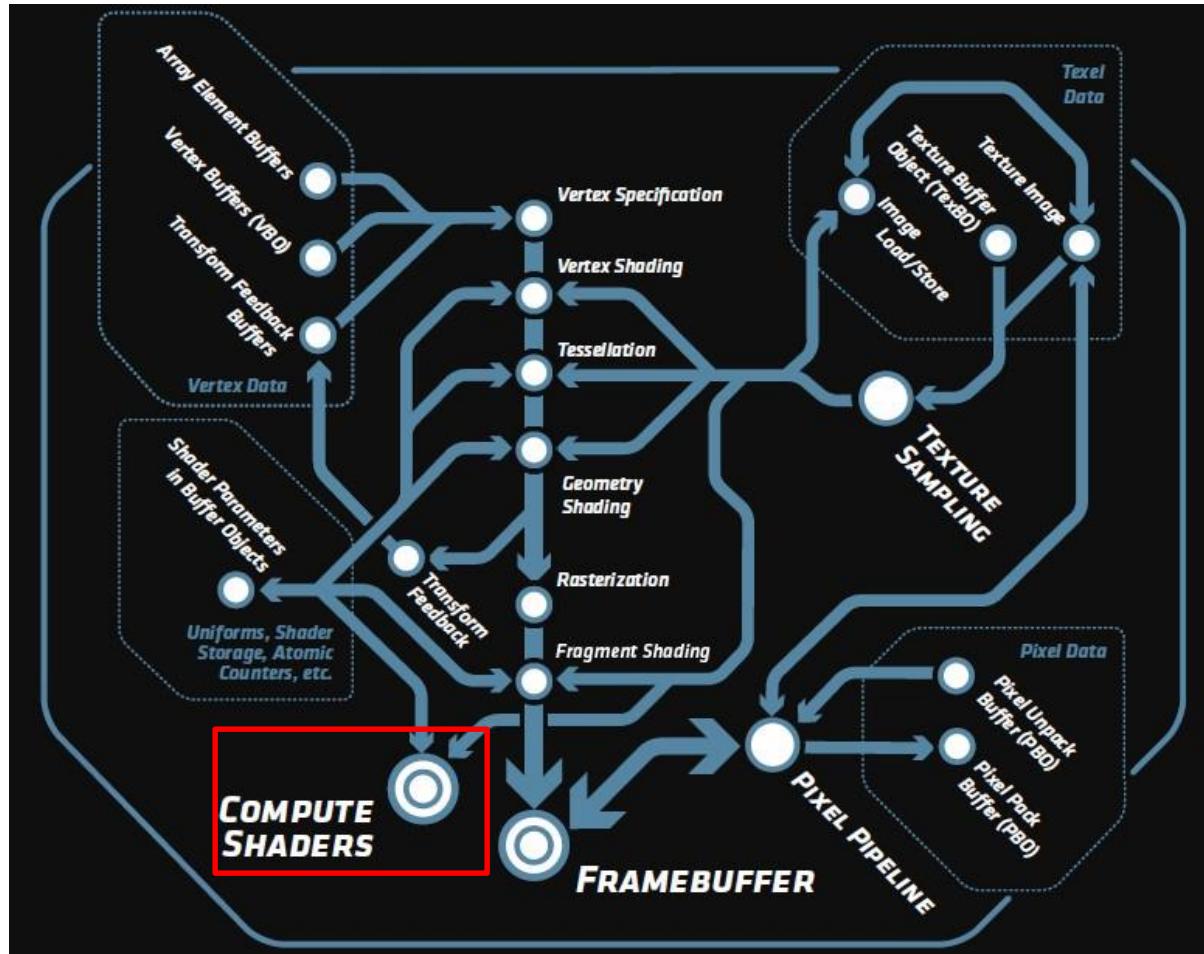
OpenGL 4.0 pipeline



© Copyright Khronos Group, 2010 - Page 34



OpenGL 4.3 pipeline



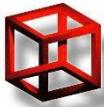
Shaders

- Programs for creating new geometry, changing vertices and shaders
- Take control of processing on GPU
- Move some computation from CPU to GPU



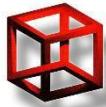
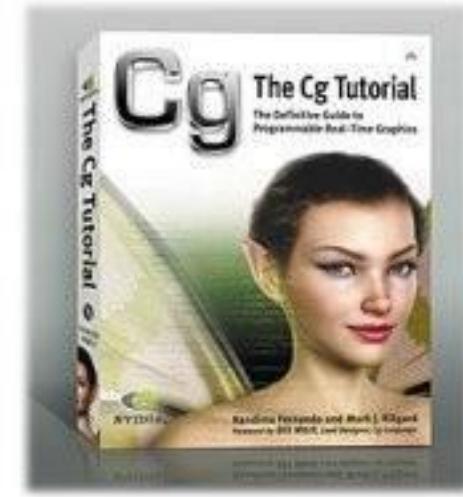
Shader languages

- Assembler - different for ATI, Nvidia
- Source code shaders - ATI
- Cg – C for graphics (Nvidia), HLSL (DirectX)
- GLSL - OpenGL Shading Language
 - OpenGL 2.0 standard
 - ATI, Nvidia



Cg

- Nvidia extensions
- Various profiles
- Compiler, runtime libraries
- Examples – Cg Toolkit
- http://developer.nvidia.com/object/cg_toolkit.html



Cg shaders

```
struct input_data
{
    float4 position : POSITION;
};

struct output_data
{
    float4 position : POSITION;
};

output_data main(input_data IN)
{
    output_data OUT;
    OUT.position = IN.position;
    return OUT;
}
```

Vertex shader

```
struct input_data{
    float2 tc : TEXCOORD0;
};
struct output_data{
    float4 color : COLOR;
};

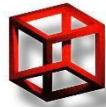
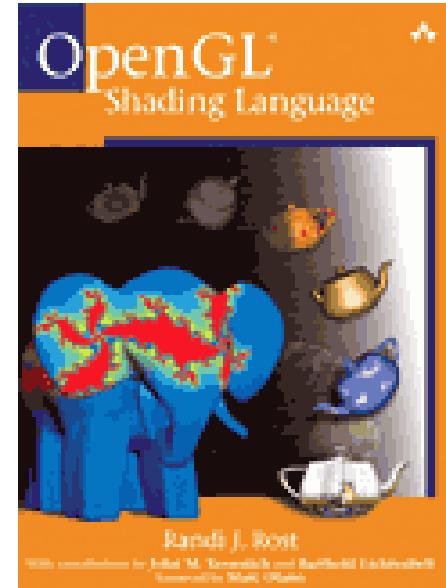
output_data main(input_data IN,
                  uniform sampler2D textureIn )
{
    output_data OUT;
    OUT.color = tex2D(textureIn, IN.tc);
    return OUT;
}
```

Fragment shader



GL shading language

- ANSI C-like language for writing shaders
- <http://www.opengl.org/registry/doc/GLSLangSpec.4.10.6.clean.pdf>
- Extended with mechanisms from C++ and vector and matrix types
- Part of core specification
- Older functionality is deprecated



OpenGL extension for GLSL

- New language for writing shaders
 - `GL_ARB_shading_language_100`, ...
- New shader programs
 - `GL_ARB_fragment_shader`
 - `GL_ARB_vertex_shader`
 - `GL_ARB_geometry_shader4`
 - `GL_ARB_tessellation_shader`, ...
- Management using shader objects
 - `GL_ARB_shader_objects`
- <http://www.opengl.org/registry/>



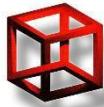
GLSL basic types

- void
- float vec2 vec3 vec4
- mat2 mat3 mat4
- int ivec2 ivec3 ivec4
- bool bvec2 bvec3 bvec4
- samplerND samplerCube
- samplerShadowD



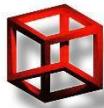
GLSL type qualifiers

- Const – compile-time constant
- Attribute – for passing data for vertex
- Uniform – global variable, read-only
- Varying – vertex->fragment sh. data
- In – parameters passed into function
- Out – passed out of function
- Inout – in & out of function



GLSL scalar constructors

- `int(bool)` // converts a Boolean value to an int
- `int(float)` // converts a float value to an int
- `float(bool)` // converts a Boolean value to a float
- `float(int)` // converts an integer value to a float
- `bool(float)` // converts a float value to a Boolean
- `bool(int)` // converts an integer value to a Boolean



GLSL matrix constructors

```
vec3(float)      // initializes each component of a vec3 with the float
vec4(ivec4)      // makes a vec4 from an ivec4, with component-wise conversion

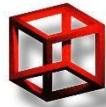
vec2(float, float)           // initializes a vec2 with 2 floats
ivec3(int, int, int)         // initializes an ivec3 with 3 ints
bvec4(int, int, float, float) // initializes with 4 Boolean conversions

vec2(vec3) // drops the third component of a vec3
vec3(vec4) // drops the fourth component of a vec4

vec3(vec2, float) // vec3.x = vec2.x, vec3.y = vec2.y, vec3.z = float
vec3(float, vec2) // vec3.x = float, vec3.y = vec2.x, vec3.z = vec2.y
vec4(vec3, float)
vec4(float, vec3)
vec4(vec2, vec2)           mat2(vec2, vec2);
                           mat3(vec3, vec3, vec3);
                           mat4(vec4, vec4, vec4, vec4);

                           mat2(float, float,
                                 float, float);
mat2(float)
mat3(float)
mat4(float)                 mat3(float, float, float,
                           float, float, float,
                           float, float, float);

                           mat4(float, float, float, float,
                                 float, float, float, float,
                                 float, float, float, float,
                                 float, float, float, float);
```



GLSL vectors & matrices

- Component access:
 - {x,y,z,w}, {r,g,b,a}, {s,t,p,q}
 - Allows access to multiple components
 - Allows access using indexing []

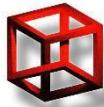
```
vec4 pos = vec4(1.0, 2.0, 3.0, 4.0);
pos.xw = vec2(5.0, 6.0);      // pos = (5.0, 2.0, 3.0, 6.0)
pos.wx = vec2(7.0, 8.0);      // pos = (8.0, 2.0, 3.0, 7.0)
pos.xx = vec2(3.0, 4.0);      // illegal - 'x' used twice
pos.xy = vec3(1.0, 2.0, 3.0); // illegal - mismatch between vec2 and vec3
```

- Simple addition and multiplication:
 - vec3 u, v, w; w = u + v; w = u * v;
 - mat3 m, n, o; m = n * o; m = n + o



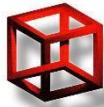
GLSL built-in functions

- Angle and Trigonometry (radians, degrees, sin, cos, tan, asin, acos, ...)
- Exponential (pow, exp, log, sqrt, ...)
- Math (abs, floor, mod, min, max, clamp, mix, step, mod, ...)
- Geometric (length, distance, dot, cross, normalize, ftransform, reflect, ...)
- Matrix (matrixCompMult)
- Vector Relational (lessThan, greaterThan, equal, any, all, ...)
- Texture Lookup (textureND, textureDLod, textureDProj, textureCube, shadowD, ...)
- Noise (noise1, noise2, ...)
- Fragment processing (discard, dFdx, dFdy, fwidth, ...)

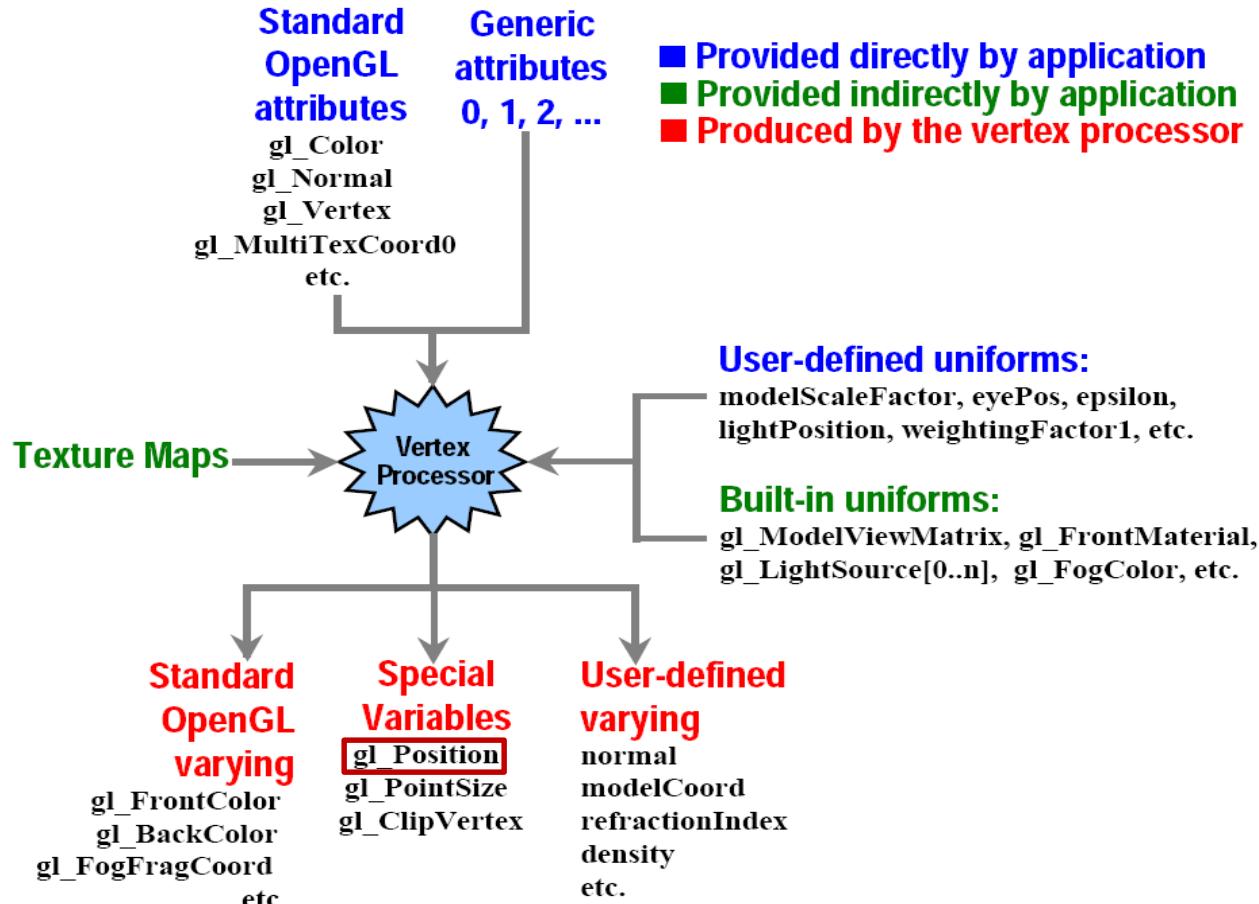


GLSL built-in variables

- For access of data
- Part from OpenGL state
- User defined data
- Variables for input and output of shaders
- Based on fixed functionality pipeline



GLSL VS built-in variables



GLSL GS built-in variables

- Input:

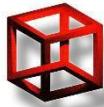
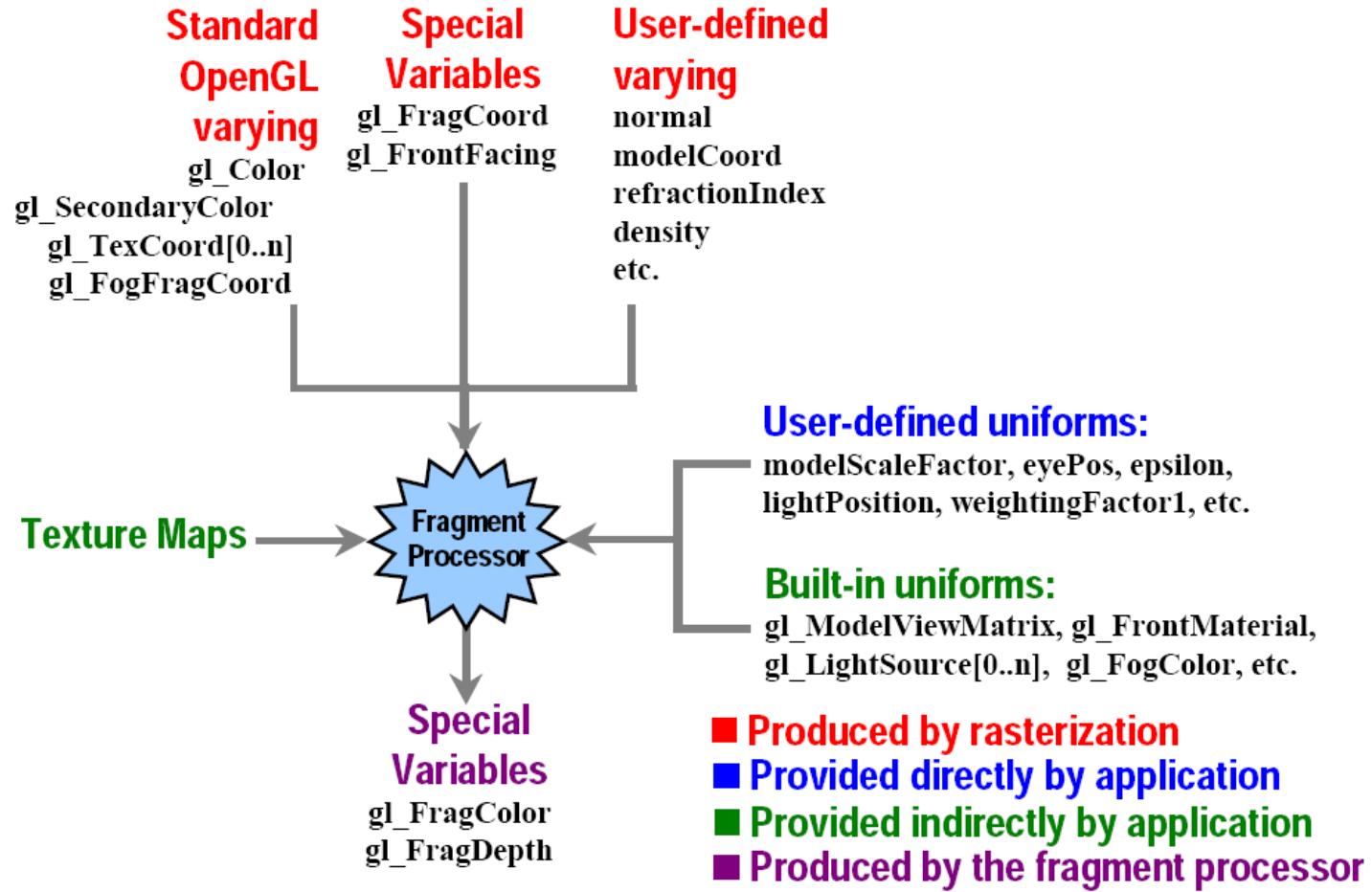
- varying in vec4 gl_FrontColorIn[gl_VerticesIn]
- varying in vec4 gl_BackColorIn[gl_VerticesIn]
- varying in vec4 gl_FrontSecondaryColorIn[gl_VerticesIn]
- varying in vec4 gl_BackSecondaryColorIn[gl_VerticesIn]
- varying in vec4 gl_TexCoordIn[gl_VerticesIn][]
- varying in vec4 gl_PositionIn[gl_VerticesIn]
- varying in float gl_PointSizeIn[gl_VerticesIn]
- varying in vec4 gl_ClipVertexIn[gl_VerticesIn]

- Output:

- varying vec4 gl_FrontColor
- varying vec4 gl_BackColor
- varying vec4 gl_FrontSecondaryColor
- varying vec4 gl_BackSecondaryColor
- varying vec4 gl_TexCoord[]
- varying out vec4 gl_FrontColor
- varying out vec4 gl_BackColor
- varying out vec4 gl_FrontSecondaryColor
- varying out vec4 gl_BackSecondaryColor
- varying out vec4 gl_TexCoord[];



GLSL FS built-in variables



GLSL – vertex, fragment

```
const vec4 AMBIENT = vec4( 0.9, 0.9, 0.1, 1.0 );
const vec4 SPECULAR = vec4( 1.0, 1.0, 1.0, 1.0 );
uniform vec4 light;

varying vec4 Ca;
varying vec4 Cd;
varying vec4 Cs;

varying vec4 V_eye;
varying vec4 L_eye;
varying vec4 N_eye;

void main(void)
{
    V_eye = gl_ModelViewMatrix * gl_Vertex;
    L_eye = (gl_ModelViewMatrix * light) - V_eye;
    N_eye = vec4(gl_NormalMatrix * gl_Normal, 1.0);

    gl_Position = gl_ProjectionMatrix * V_eye;
    V_eye = -V_eye;

    Ca = AMBIENT;
    Cd = gl_Color;
    Cs = SPECULAR;
}
```

```
varying vec4 Ca;
varying vec4 Cd;
varying vec4 Cs;

varying vec4 V_eye;
varying vec4 L_eye;
varying vec4 N_eye;

vec3 reflect(vec3 N, vec3 L)
{
    return 2.0*N*dot(N, L) - L;
}

void main(void)
{
    vec3 V = normalize(vec3(V_eye));
    vec3 L = normalize(vec3(L_eye));
    vec3 N = normalize(vec3(N_eye));

    float diffuse = clamp(dot(L, N), 0.0, 1.0);

    vec3 R = reflect(N, L);
    float specular = pow(clamp(dot(R, V), 0.0, 1.0), 16);

    gl_FragColor = Ca + (Cd*diffuse) + (Cs*specular);
}
```



Geometry shader example

```
#version 120
#extension GL_EXT_geometry_shader4: enable
uniform float FpNum;
void main()
{
    int num = int( FpNum + 0.99 );
    float dt = 1. / float(num);
    float t = 0.;
    for( int i = 0; i <= num; i++ )
    {
        float omt = 1. - t;
        float omt2 = omt * omt;
        float omt3 = omt * omt2;
        float t2 = t * t;
        float t3 = t * t2;
        vec4 xyzw = omt3 * gl_PositionIn[0].xyzw +
            3. * t * omt2 * gl_PositionIn[1].xyzw +
            3. * t2 * omt * gl_PositionIn[2].xyzw +
            t3 * gl_PositionIn[3].xyzw;
        gl_Position = xyzw;
        EmitVertex();
        t += dt;
    }
    EndPrimitive();
}
```

```
void main()
{
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}

void main()
{
    gl_FragColor = vec4( 0., 1., 0., 1. );
}
```

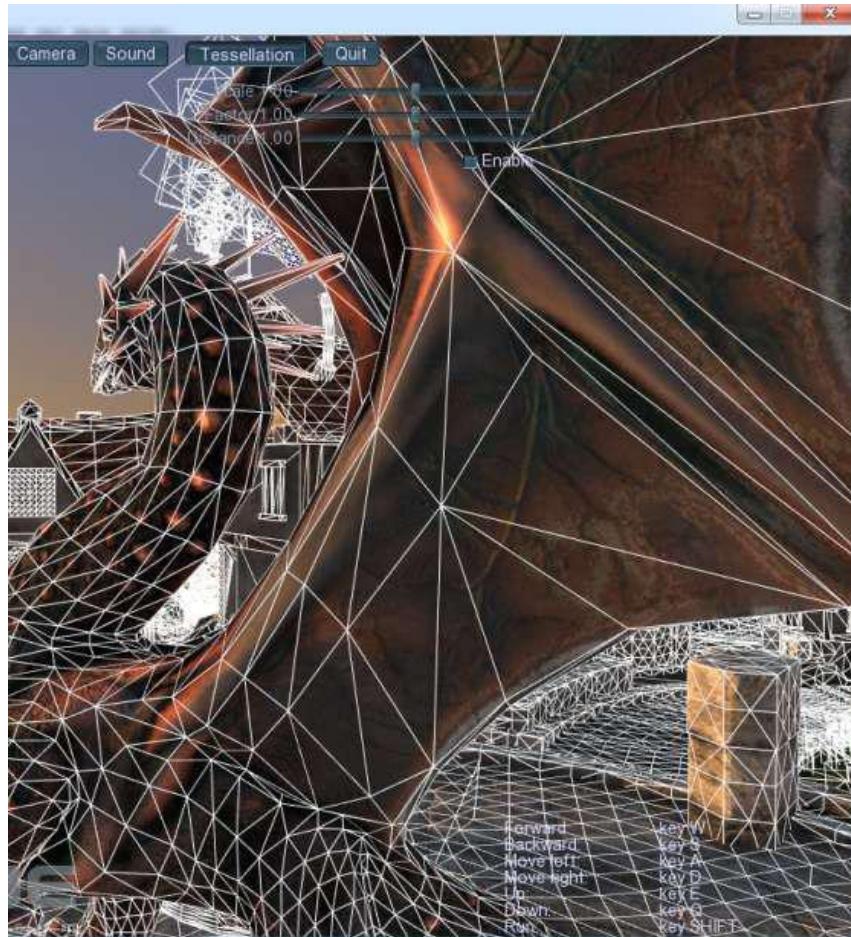


Tesselation shaders

- Works on patch – given by set of vertices and per-patch attributes
- Tessellation control shader transforms per-vertex data and per-patch attr.
- Tessellator decomposes patch into set of new primitives based on tess. level
- Tessellation evaluation shader computes position and attributes of new generated vertices



Tessellation shaders



Real-time Graphics
Martin Samuelčík

Shaders management

- Shader objects – shaders with unique identifier
- Creation: `glCreateShaderObject(type)`, type:
 - `GL_VERTEX_SHADER`
 - `GL_GEOMETRY_SHADER`
 - `GL_FRAGMENT_SHADER`, ...
- Setting source: `glShaderSource(shaderID, numStrings, strings, length)`
- Compilation: `glCompileShader(shaderID)`



Shaders management

- Shader programs - container for shader objects, set of shaders that are linked together
- Creation: `prog = glCreateProgramObject()`
- Adding shader: `glAttachObject(programID, shaderID)`
- Linking: `glLinkProgram(programID)`
- Set as current:
`glUseProgramObject(programID)`



Management example

```
GLhandle g_programObj;
GLhandle g_vertexShader;
GLhandle g_fragmentShader;

g_vertexShader = glCreateShaderObjectARB( GL_VERTEX_SHADER );
unsigned char *vertexShaderAssembly = readShaderFile( "vertex_shader.vert" );
vertexShaderStrings[0] = (char*)vertexShaderAssembly;
glShaderSource( g_vertexShader, 1, vertexShaderStrings, NULL );
glCompileShader( g_vertexShader );
delete vertexShaderAssembly;

g_fragmentShader = glCreateShaderObject( GL_FRAGMENT_SHADER );
unsigned char *fragmentShaderAssembly = readShaderFile( "fragment_shader.frag" );
fragmentShaderStrings[0] = (char*)fragmentShaderAssembly;
glShaderSource( g_fragmentShader, 1, fragmentShaderStrings, NULL );
glCompileShader( g_fragmentShader );
delete fragmentShaderAssembly;

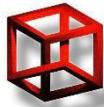
g_programObj = glCreateProgramObject();
glAttachObject( g_programObj, g_vertexShader );
glAttachObject( g_programObj, g_fragmentShader );

glLinkProgram( g_programObj );
glGetObjectParameteriv( g_programObj, GL_OBJECT_LINK_STATUS, &bLinked );
```



Passing variables

- From application to shaders, based on location of variable in shader program:
 - Glint `glGetAttribLocation(GLhandle program, const GLchar* name);`
 - Glint `glGetUniformLocation(GLhandle program, const GLchar * name);`
 - void `glUniform{1|2|3|4}{f|i}(GLint location, TYPE val);`
 - void `glUniform{1|2|3|4}{f|i}v(GLint location, GLuint count, const TYPE * vals);`
 - void `glUniformMatrix{2|3|4|}fv(GLint location, GLuint count, GLboolean transpose, const GLfloat * vals);`
 - void `glVertexAttrib{1|2|3|4}{s|f|d}(GLuint index, TYPE val);`
 - void `glVertexAttrib{1|2|3|4}{s|f|d}v(GLuint index, const TYPE * vals);`
- Possibility to sent array of attributes or uniforms



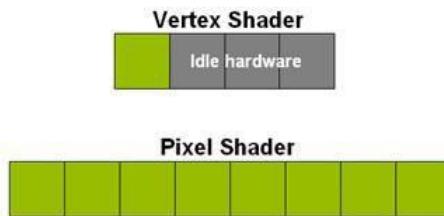
Unified architecture

- Consistent instruction set across all shader types
- Flexible use of the graphics hardware

Why unify?

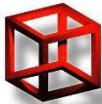


Why unify?



Tools

- Debugging GLSL: your way, gDEBugger, glslDevil, Fx Composer(+Shader Debugger)
- Extensions: GLEE, GLEW
- Render Monkey -
<http://developer.amd.com/archive/gpu/rendermonkey/pages/default.aspx>
- Shader Designer -
<http://www.opengl.org/sdk/tools/ShaderDesigner/>
- Books:
<http://www.opengl.org/documentation/books/>



Questions?

