Realtime Computer Graphics on GPUs

Introduction

Jan Kolomazník

Department of Software and Computer Science Education
Faculty of Mathematics and Physics
Charles University in Prague

Real-time Algorithms

Real-time Algorithms

- Time Constrains:
  - Hard limit
  - Soft limit
- CG examples:
  - Video frame rate
    - Cinema – 24 Hz
    - TV – 25 (50) Hz, 30 (60) Hz
  - Video games – 30–60 Hz
    - Virtual reality – frame rate doubled
  - Haptic rendering – 1 kHz

Real-time Algorithms

- Time Constrains:
  - Hard limit
  - Soft limit
- CG examples:
  - Video frame rate
    - Cinema – 24 Hz
    - TV – 25 (50) Hz, 30 (60) Hz
  - Video games – 30–60 Hz
    - Virtual reality – frame rate doubled
  - Haptic rendering – 1 kHz

Real-time Algorithms

- Time Constrains:
  - Hard limit
  - Soft limit
- CG examples:
  - Video frame rate
    - Cinema – 24 Hz
    - TV – 25 (50) Hz, 30 (60) Hz
  - Video games – 30–60 Hz
    - Virtual reality – frame rate doubled
  - Haptic rendering – 1 kHz

Real-time Algorithms

- Time Constrains:
  - Hard limit
  - Soft limit
- CG examples:
  - Video frame rate
    - Cinema – 24 Hz
    - TV – 25 (50) Hz, 30 (60) Hz
  - Video games – 30–60 Hz
    - Virtual reality – frame rate doubled
  - Haptic rendering – 1 kHz

Real-time Algorithms

- Time Constrains:
  - Hard limit
  - Soft limit
- CG examples:
  - Video frame rate
    - Cinema – 24 Hz
    - TV – 25 (50) Hz, 30 (60) Hz
  - Video games – 30–60 Hz
    - Virtual reality – frame rate doubled
  - Haptic rendering – 1 kHz

How to achieve speed

- Optimal algorithm (time complexity ?)
- Approximations vs. precision requirements
- Tuning for specific hardware
- Specialized tools for hot spots – GPUs

How to achieve speed

- Optimal algorithm (time complexity ?)
- Approximations vs. precision requirements
- Tuning for specific hardware
- Specialized tools for hot spots – GPUs

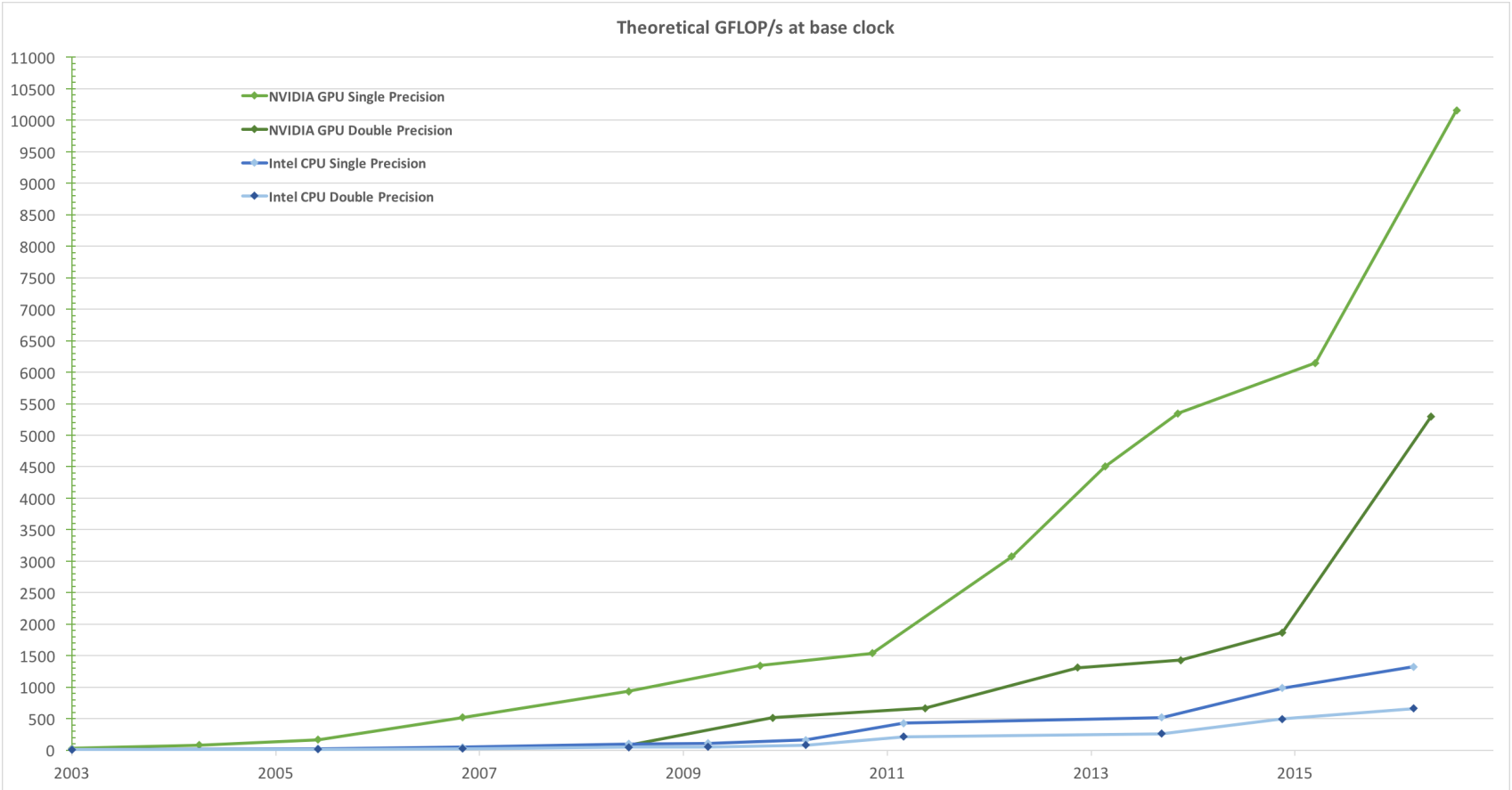
How to achieve speed

- Optimal algorithm (time complexity ?)
- Approximations vs. precision requirements
- Tuning for specific hardware
- Specialized tools for hot spots – GPUs

How to achieve speed

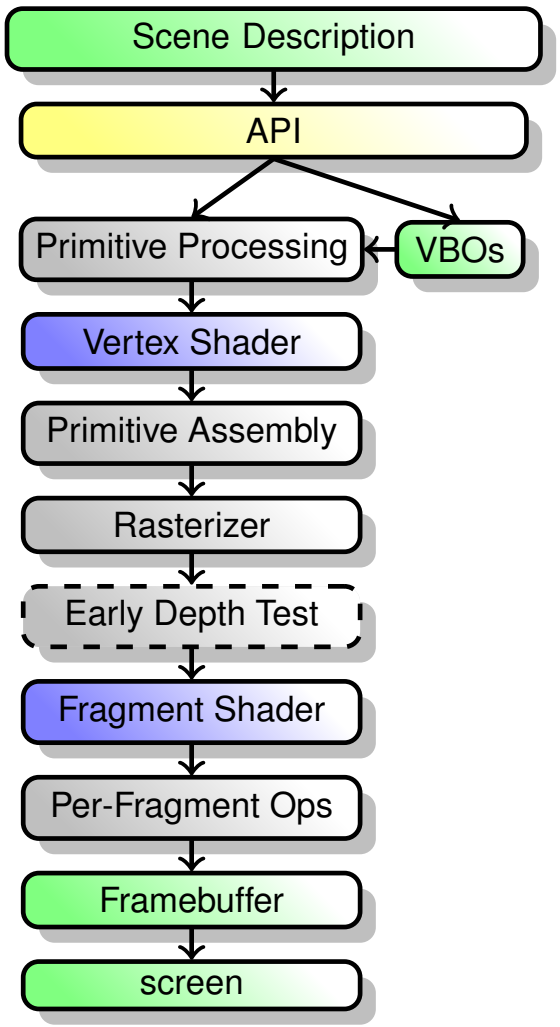
- Optimal algorithm (time complexity ?)
- Approximations vs. precision requirements
- Tuning for specific hardware
- Specialized tools for hot spots – GPUs

Why GPU?


Theoretical GFLOP/s at base clock
NVIDIA GPU Single Precision
NVIDIA GPU Double Precision
Intel CPU Single Precision
Intel CPU Double Precision
11000
10500
10000
9500
9000
8500
8000
7500
7000
6500
6000
5500
5000
4500
4000
3500
3000
2500
2000
1500
1000
500
0
2003
2005
2007
2009
2011
2013
2015

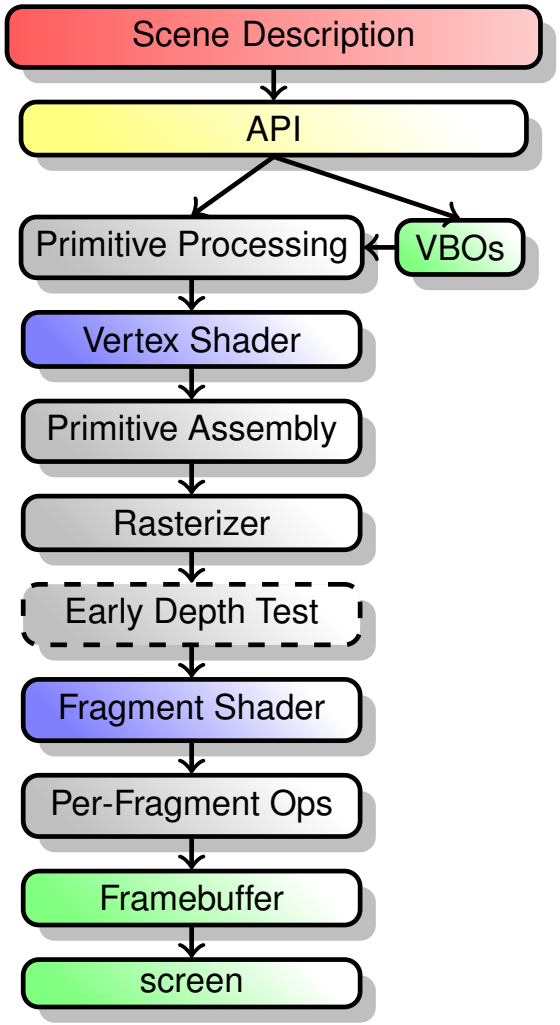
Programmable Pipeline

Programmable Pipeline



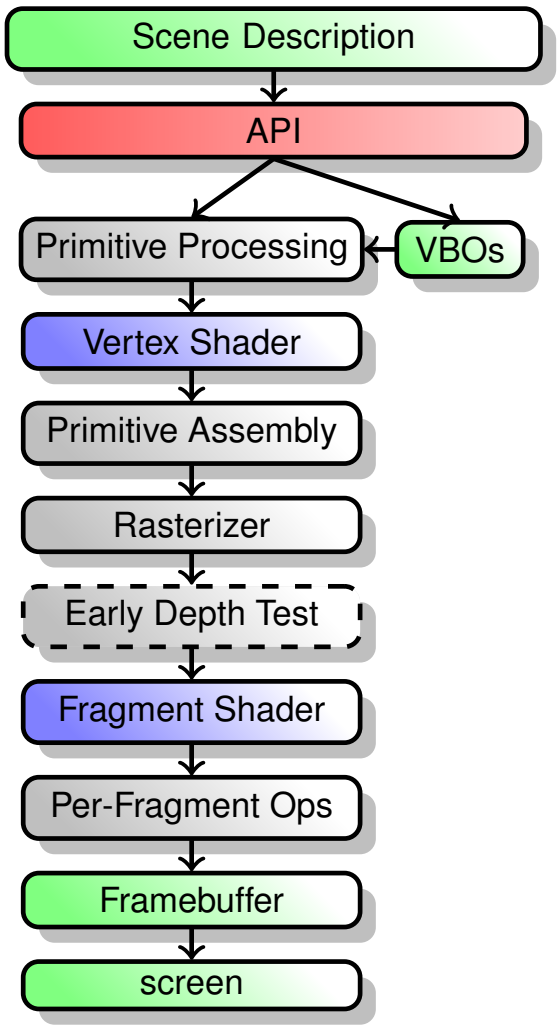
- Boundary representation – vertices, edges, faces
- Comunication (draw schedule, data transfers): App $\leftrightarrow$ GPU driver $\leftrightarrow$ GPU
- Prepare inputs for the shader processing
- Transform vertices into *clip space* ($[x, y, z] \in (-1, 1)^3$), bundle attributes (colors, normals) for further processing
- Prepare primives for rasterization
- Process primitives from clip space, rasterize into *screen space* – generate *fragments*
- Custom fragment processing – lighting, texturing, ...
- Depth (Z-buffer) and stencil tests, color blending
- Output image, Z-buffer, stencil buffer, double buffering, rendering to texture
- Output on screen

Programmable Pipeline



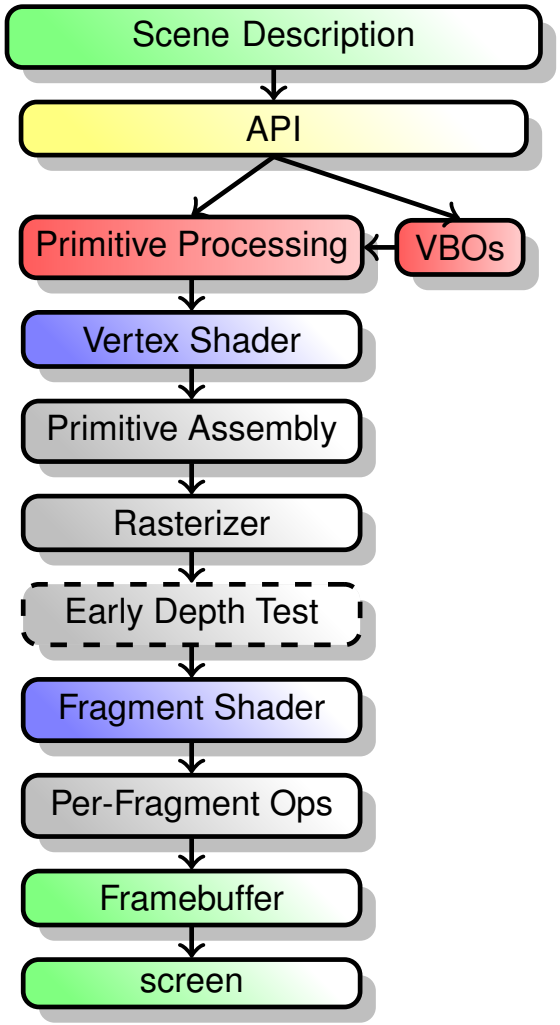
- Boundary representation – vertices, edges, faces
- Comunication (draw schedule, data transfers): App $\leftrightarrow$ GPU driver $\leftrightarrow$ GPU
- Prepare inputs for the shader processing
- Transform vertices into *clip space* ($[x, y, z] \in (-1, 1)^3$), bundle attributes (colors, normals) for further processing
- Prepare primives for rasterization
- Process primitives from clip space, rasterize into *screen space* – generate *fragments*
- Custom fragment processing – lighting, texturing, ...
- Depth (Z-buffer) and stencil tests, color blending
- Output image, Z-buffer, stencil buffer, double buffering, rendering to texture
- Output on screen

Programmable Pipeline



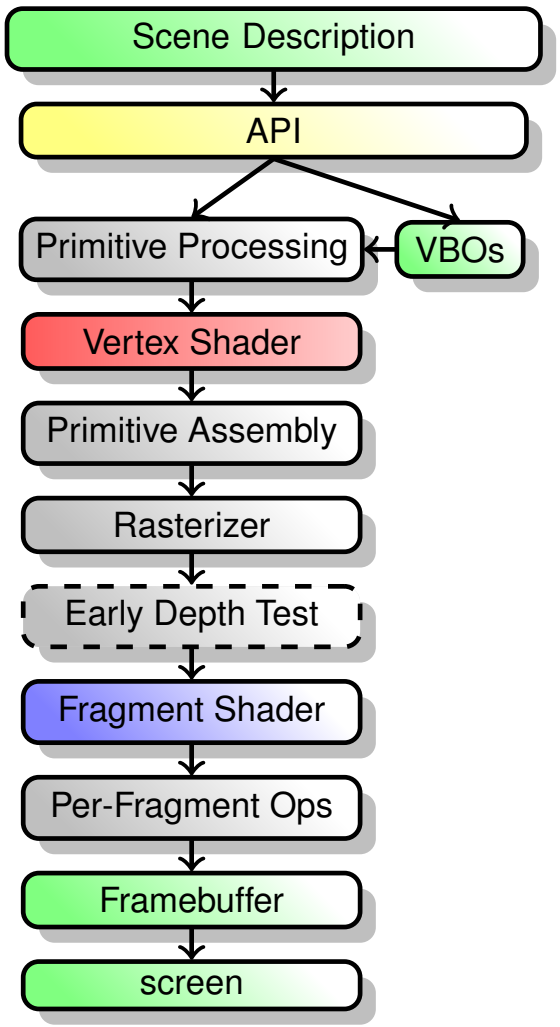
- Boundary representation – vertices, edges, faces
- Comunication (draw schedule, data transfers): App $\leftrightarrow$ GPU driver $\leftrightarrow$ GPU
- Prepare inputs for the shader processing
- Transform vertices into *clip space* ($[x, y, z] \in (-1, 1)^3$), bundle attributes (colors, normals) for further processing
- Prepare primives for rasterization
- Process primitives from clip space, rasterize into *screen space* – generate *fragments*
- Custom fragment processing – lighting, texturing, ...
- Depth (Z-buffer) and stencil tests, color blending
- Output image, Z-buffer, stencil buffer, double buffering, rendering to texture
- Output on screen

Programmable Pipeline



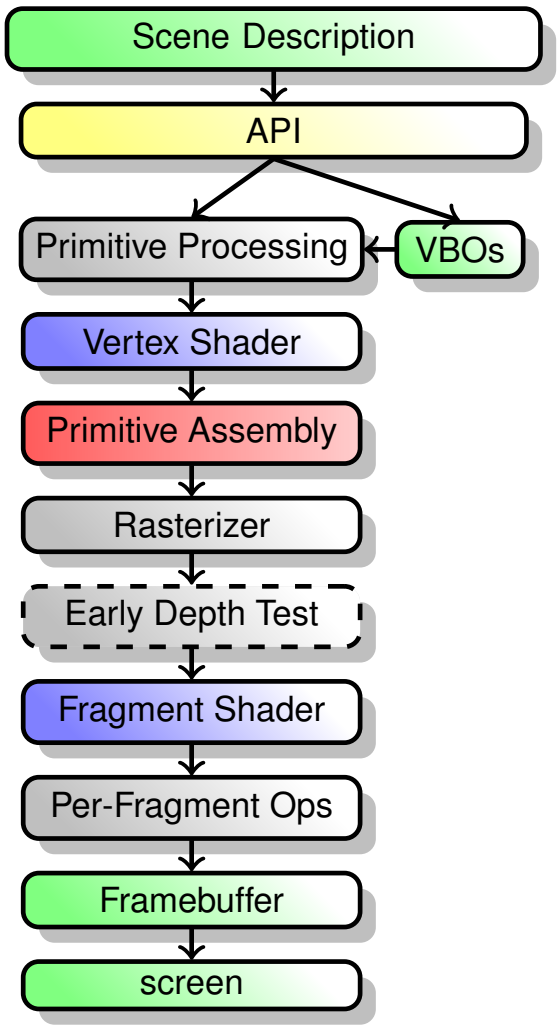
- Boundary representation – vertices, edges, faces
- Comunication (draw schedule, data transfers): App $\leftrightarrow$ GPU driver $\leftrightarrow$ GPU
- Prepare inputs for the shader processing
- Transform vertices into *clip space* ($[x, y, z] \in (-1, 1)^3$), bundle attributes (colors, normals) for further processing
- Prepare primives for rasterization
- Process primitives from clip space, rasterize into *screen space* – generate *fragments*
- Custom fragment processing – lighting, texturing, ...
- Depth (Z-buffer) and stencil tests, color blending
- Output image, Z-buffer, stencil buffer, double buffering, rendering to texture
- Output on screen

Programmable Pipeline



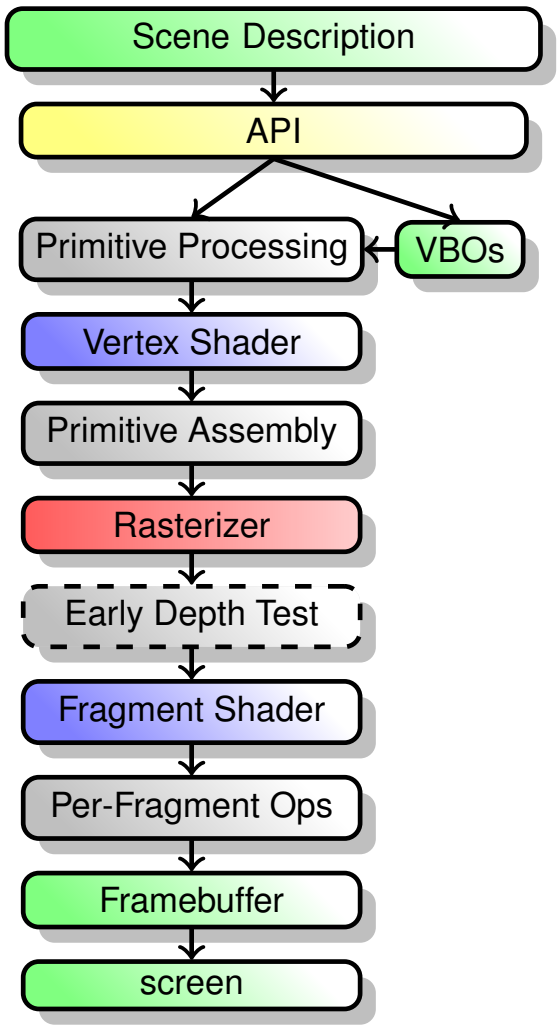
- Boundary representation – vertices, edges, faces
- Comunication (draw schedule, data transfers): App $\leftrightarrow$ GPU driver $\leftrightarrow$ GPU
- Prepare inputs for the shader processing
- Transform vertices into *clip space* ($[x, y, z] \in (-1, 1)^3$), bundle attributes (colors, normals) for further processing
- Prepare primives for rasterization
- Process primitives from clip space, rasterize into *screen space* – generate *fragments*
- Custom fragment processing – lighting, texturing, ...
- Depth (Z-buffer) and stencil tests, color blending
- Output image, Z-buffer, stencil buffer, double buffering, rendering to texture
- Output on screen

Programmable Pipeline



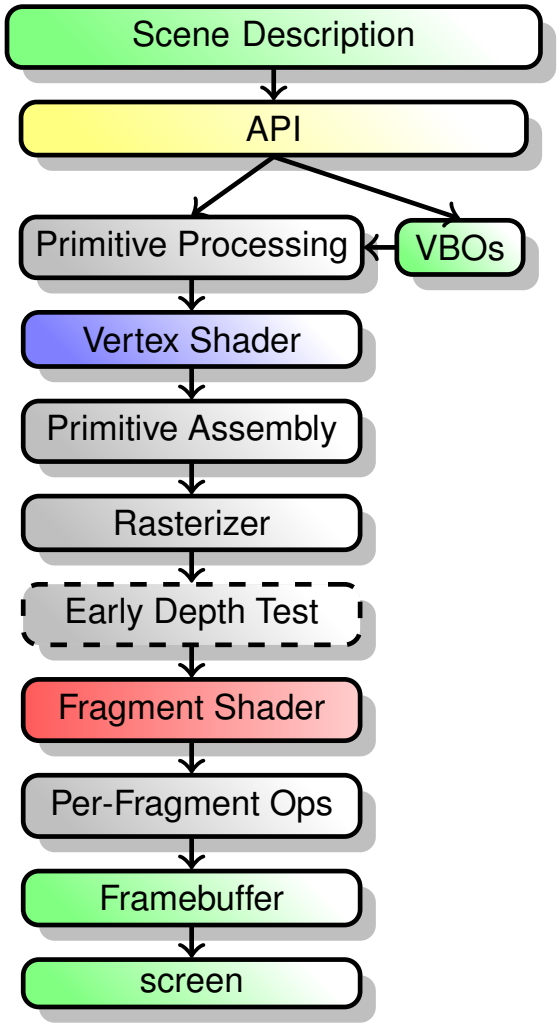
- Boundary representation – vertices, edges, faces
- Comunication (draw schedule, data transfers): App $\leftrightarrow$ GPU driver $\leftrightarrow$ GPU
- Prepare inputs for the shader processing
- Transform vertices into *clip space* ($[x, y, z] \in (-1, 1)^3$), bundle attributes (colors, normals) for further processing
- Prepare primives for rasterization
- Process primitives from clip space, rasterize into *screen space* – generate *fragments*
- Custom fragment processing – lighting, texturing, ...
- Depth (Z-buffer) and stencil tests, color blending
- Output image, Z-buffer, stencil buffer, double buffering, rendering to texture
- Output on screen

Programmable Pipeline



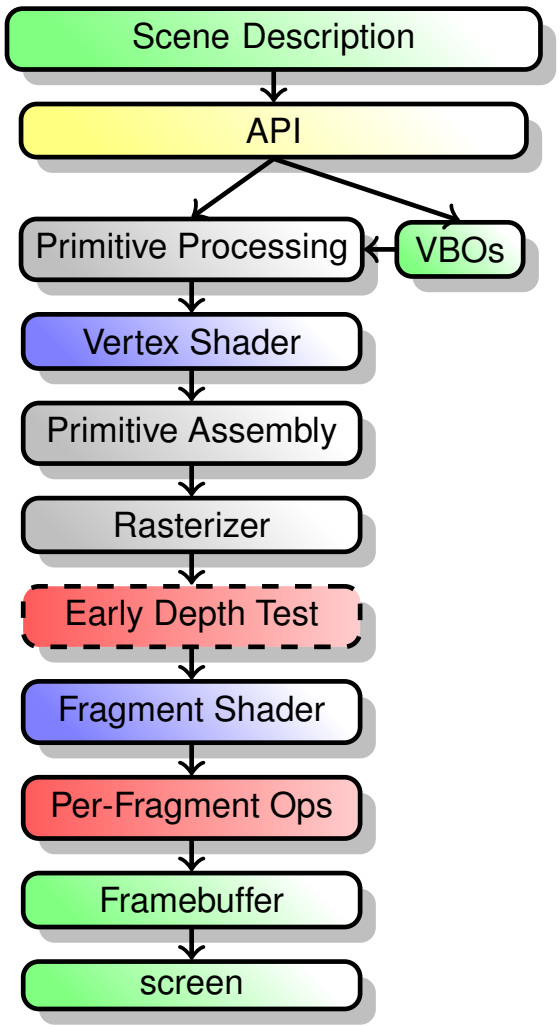
- Boundary representation – vertices, edges, faces
- Comunication (draw schedule, data transfers): App $\leftrightarrow$ GPU driver $\leftrightarrow$ GPU
- Prepare inputs for the shader processing
- Transform vertices into *clip space* ($[x, y, z] \in (-1, 1)^3$), bundle attributes (colors, normals) for further processing
- Prepare primives for rasterization
- Process primitives from clip space, rasterize into *screen space* – generate *fragments*
- Custom fragment processing – lighting, texturing, ...
- Depth (Z-buffer) and stencil tests, color blending
- Output image, Z-buffer, stencil buffer, double buffering, rendering to texture
- Output on screen

Programmable Pipeline



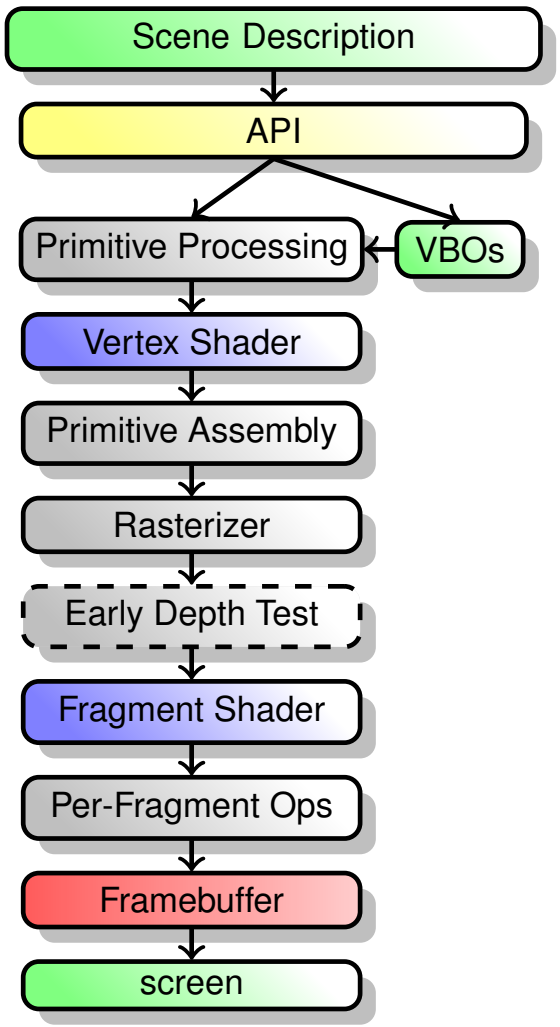
- Boundary representation – vertices, edges, faces
- Comunication (draw schedule, data transfers): App $\leftrightarrow$ GPU driver $\leftrightarrow$ GPU
- Prepare inputs for the shader processing
- Transform vertices into *clip space* ($[x, y, z] \in (-1, 1)^3$), bundle attributes (colors, normals) for further processing
- Prepare primives for rasterization
- Process primitives from clip space, rasterize into *screen space* – generate *fragments*
- Custom fragment processing – lighting, texturing, ...
- Depth (Z-buffer) and stencil tests, color blending
- Output image, Z-buffer, stencil buffer, double buffering, rendering to texture
- Output on screen

Programmable Pipeline



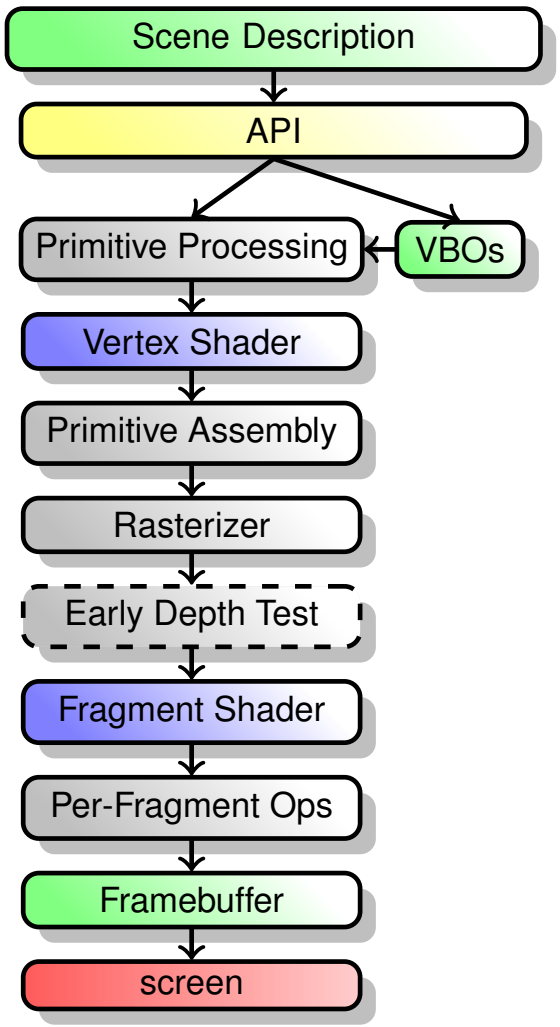
- Boundary representation – vertices, edges, faces
- Comunication (draw schedule, data transfers): App $\leftrightarrow$ GPU driver $\leftrightarrow$ GPU
- Prepare inputs for the shader processing
- Transform vertices into *clip space* ($[x, y, z] \in (-1, 1)^3$), bundle attributes (colors, normals) for further processing
- Prepare primives for rasterization
- Process primitives from clip space, rasterize into *screen space* – generate *fragments*
- Custom fragment processing – lighting, texturing, ...
- Depth (Z-buffer) and stencil tests, color blending
- Output image, Z-buffer, stencil buffer, double buffering, rendering to texture
- Output on screen

Programmable Pipeline



- Boundary representation – vertices, edges, faces
- Comunication (draw schedule, data transfers): App $\leftrightarrow$ GPU driver $\leftrightarrow$ GPU
- Prepare inputs for the shader processing
- Transform vertices into *clip space* ($[x, y, z] \in (-1, 1)^3$), bundle attributes (colors, normals) for further processing
- Prepare primives for rasterization
- Process primitives from clip space, rasterize into *screen space* – generate *fragments*
- Custom fragment processing – lighting, texturing, ...
- Depth (Z-buffer) and stencil tests, color blending
- Output image, Z-buffer, stencil buffer, double buffering, rendering to texture
- Output on screen

Programmable Pipeline



- Boundary representation – vertices, edges, faces
- Comunication (draw schedule, data transfers): App $\leftrightarrow$ GPU driver $\leftrightarrow$ GPU
- Prepare inputs for the shader processing
- Transform vertices into *clip space* ($[x, y, z] \in (-1, 1)^3$), bundle attributes (colors, normals) for further processing
- Prepare primives for rasterization
- Process primitives from clip space, rasterize into *screen space* – generate *fragments*
- Custom fragment processing – lighting, texturing, ...
- Depth (Z-buffer) and stencil tests, color blending
- Output image, Z-buffer, stencil buffer, double buffering, rendering to texture
- Output on screen

OpenGL

- Open standard
  - OpenGL Architecture Review Board (ARB) 1992-2006
  - Khronos Group 2006–
- Current version 4.6
- Multiplatform, language-independent
- Additional functionality possible by HW vendor extensions
- Open source implementation – *Mesa*

OpenGL Concepts

- API formed by set of functions and integer constants
- Asynchronnous calls (queries)
- GL context – internal global state owning OGL objects
  - Multiple contexts possible (data sharing, etc.)
  - API calls make modification to *current context*
- OGL objects (textures, buffers, framebuffers, shader programs, . . .)
  - Gen/Delete paradigm – glGen*(GLsizei n, GLuint *objects), glDelete(GLsizei n, const GLuint *objects)
  - Bind before usage – glBind*(GLenum target, GLuint object)

Example: VBO and VAO Initialization

```
float vertices[] = {
    -0.5f, -0.5f,
     0.5f, -0.5f,
     0.0f,  0.5f,
};
unsigned int VBO, VAO;
glGenVertexArrays(1, &VAO);
glGenBuffers(1, &VBO);
// bind the Vertex Array Object first,
// then bind and set vertex buffer(s), and then configure vertex attributes(s).
glBindVertexArray(VAO);

glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);

// index, size, type, normalized, stride, pointer offset
glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 2 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);
```

Example: Draw call

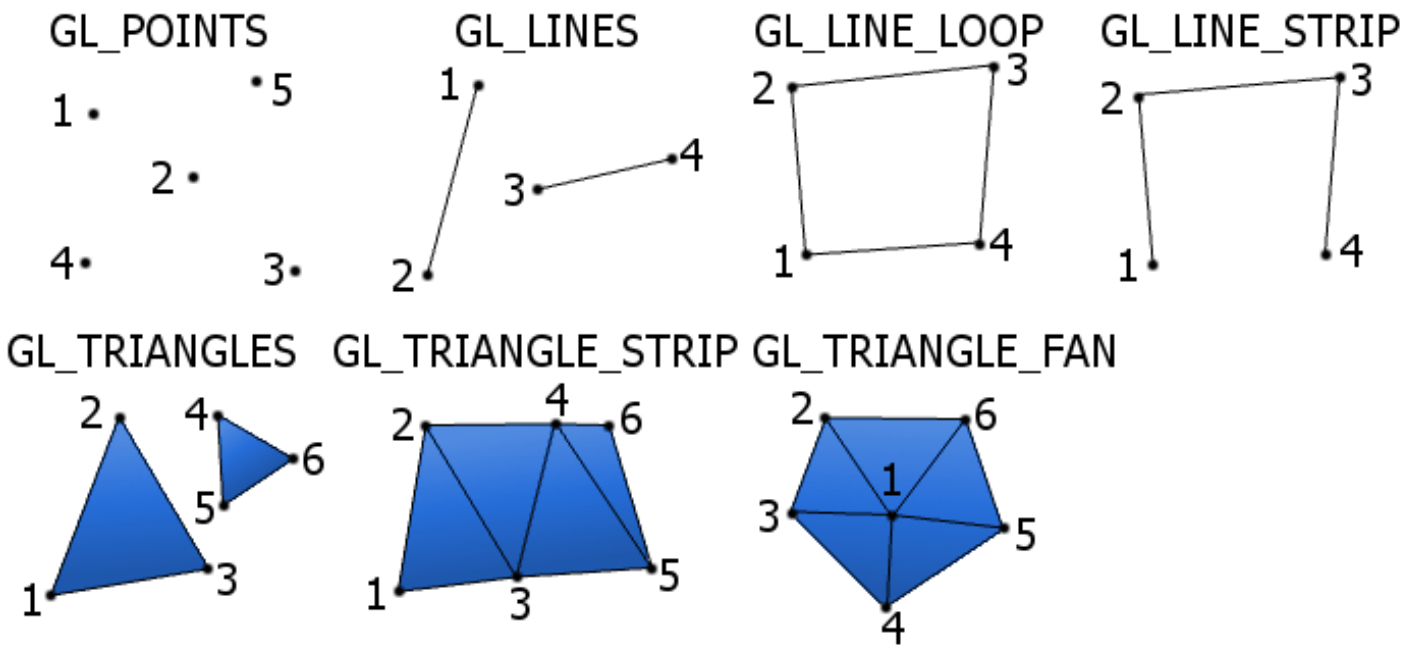
```
glUseProgram(shaderProgram);
glBindVertexArray(VAO);
glDrawArrays(GL_POINTS, 0, 3);
```

```
glUseProgram(shaderProgram);
glBindVertexArray(VAO);
glDrawArrays(GL_LINE_LOOP, 0, 3);
```

```
glUseProgram(shaderProgram);
glBindVertexArray(VAO);
glDrawArrays(GL_TRIANGLES, 0, 3);
```

RASTERIZER

- Decomposition of vector primitives into *fragments*
- Fragment:
  - Raster element – potentialy attributes to pixel color
  - Size: same or smaller (antialiasing) then the target pixel
- Interpolation of vertex attributes – linear, barycentric coordinates
- Triangles sharing two vertices – no overlap, no gap



OpenGL Shading Language (GLSL)

- C based programming language
- Programmable pipeline customization

```
// vertex shader
int vertexShader = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vertexShader, 1, &vertexShaderSource, NULL);
glCompileShader(vertexShader);

// fragment shader
int fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fragmentShader, 1, &fragmentShaderSource, NULL);
glCompileShader(fragmentShader);

// link shaders
int shaderProgram = glCreateProgram();
glAttachShader(shaderProgram, vertexShader);
glAttachShader(shaderProgram, fragmentShader);
glLinkProgram(shaderProgram);
```

VERTEX SHADER

```
#version 430

in vec3 in_vert;


void main() {
    gl_Position = vec4(in_vert, 1.0);
}
```

Fragment Shader

```
#version 430
// Same color for all fragments and all draw calls
const vec3 color = vec3(1.0, 0.0, 0.0);

out vec4 out_color;

void main() {
   out_color = vec4(color, 1.0);
}
```

Use same color for all fragments.

Fragment Shader II

```
#version 430

// Same color for all fragments per draw call + default ↩
    value
uniform vec3 color = vec3(1.0, 0.0, 0.0);

out vec4 out_color;

void main() {
   out_color = vec4(color, 1.0);
}
```

Customize color from CPU side

VS $\longleftrightarrow$ FS Obligations

- VS obligation: vertex coordinates in “clip space”
  - for 3D primitive rasterizing
  - other output varying data are optional (texture coordinates, normals, primary and secondary color, etc.)
- VS-FS cooperation
  - GPU is not aware of data semantics:
  - rasterizer unit usually interpolates all the data (perspective correct interpolation)
  - *flat* option (prevents the interpolation)

2D Drawing: Julia Set

```
#version 430

vec4 randomColor(int seed) {
   float r = ((seed * 315 + 714) % 255) / 255.0;
   float g = ((seed * 861 + 346) % 255) / 255.0;
   float b = ((seed * 123 + 1068) % 255) / 255.0;
   return vec4(r, g, b, 1.0);
}
in vec2 v_text;
out vec4 f_color;

uniform vec2 seed = vec2(-0.8, 0.156);

const int maxIter = 100;
void main() {
   int i = 0;
   vec2 z = vec2(3.0 * v_text.x, 2.0 * v_text.y);

   for (; i < maxIter; ++i) {
      float x = (z.x * z.x - z.y * z.y) + seed.x;
      float y = (z.y * z.x + z.x * z.y) + seed.y;
      if ((x * x + y * y) > 4.0) {
          break;
      }
      z.x = x;
      z.y = y;
   }
   f_color = randomColor(i);
}
```

History

SOFTWARE

PHIGS (Programmer's Hierarchical Interactive Graphics Standard)

- 80s – mid 90s
- ANSI, ISO standard
- Slow – consumer hardware accelerated video processing, limited 2D graphics
- Rendering modes:
  - Wireframe
  - Flat
  - Wireframe + Flat (edge visibility)
  - Lit flat
  - Gouraud

VERTEX SHADER – WIREFRAME

```
#version 430
// (Model * View * Projection) matrix
uniform mat4 MVP;

in vec3 in_vert;


void main() {
   // Transform vertex to clip space
   gl_Position = MVP * vec4(in_vert, 1.0);
}
```

SGI

1982 – J. Clark + 7 graduate students
1st generation: Geometry engine – 80s
selected workstations:

- 1983: Iris 1000 – graphical terminal to VAX (Motorola 68k @ 8MHz, Geometry Engine, ..)
- 1984: Iris 2000 (graphical station, Clark GE), Iris 3000
- 1986: Professional Iris 4D (first MIPS processors)
- 1988: Power series, GTX, Personal Iris 4D
- 1991: Iris Indigo (most popular SGI workstation)
- 1992: Iris Crimson

SGI 2

2nd generation: RealityEngine – 90s

- 1992: Indigo R4000 (64bit)
- 1993: Iris Indy (cheap, 3D graphics acceleration option)
- 1993: Onyx (server with RE2)
- 1993: Iris Indigo2 (Extreme graphics)
- 1996: O2 (cheap workstation), InfiniteReality engine
- 1997: Octane (2 CPUs)
- 2000: Octane2 (Vpro graphics)
- 2002: Fuel
- 2003: Ezro
- 2008: Virtu (x86, NVIDIA)

Almost a Revolution

- IrisVision ('91) – IBM PC compatible card
- 24-bit Z-buffer
- Limited support by applications – AutoCAD 12-13, 3D Studio, ...
- Iris GL $\Rightarrow$ Open GL

Consumer HW

- 1st graphic accelerator for home PC
  - 1996: 3Dfx Voodoo 1
  - graphic coprocessor (“pass-through”)
  - Glide API
- 1st SLI card (Scan Line Interleave)
  - 1998: 3Dfx Voodoo2
- NVIDIA
  - 1997: NVIDIA Riva 128
  - 1998: NVIDIA Riva TNT (“TwiNTexel”)
- 1st HW T&L (“transform & lighting”)
  - 1999: NVIDIA GeForce 256

Consumer HW 2

- 2000
  - NVIDIA GeForce2
  - ATI Radeon
- 2001: GPU programming
  - DirectX 8.0 (vertex shaders, fragment shaders, 1.0, 1.1)
  - NVIDIA GeForce3, GeForce3 Titanium
  - DirectX 8.1 (PS 1.2, 1.3, 1.4)
  - ATI Radeon 8500 (TruForm)
- 2002: advanced GPU programming
  - DirectX 9.0 – VS, PS 2.0
  - NVIDIA GeForce4 Titanium
  - ATI Radeon 9000, 9700 [Pro]

Consumer HW 3

- 2003: affordable DX9
  - cheap DirectX 9.0 – compatible cards (VS, PS 2.0)
  - NVIDIA GeForce FX 5200-5800
  - ATI Radeon 9800
- 2004: extended shader programming
  - DirectX 9.0c (VS, PS 3.0), OpenGL 2.0 (at last!)
  - NVIDIA GeForce 6800, 6200, 6600
  - ATI Radeon X800
- 2005: HW advances
  - PCI-Express bus
  - twin GPU systems – NVIDIA: SLI, ATI: CrossFire
  - NVIDIA GeForce 7800
  - ATI Radeon X550, X850

Consumer HW 4

- 2006
  - DirectX 10 (Windows Vista) .. geometry shaders
  - NVIDIA GeForce 7600, 7900
  - ATI Radeon X1800, X1900
- 2007
  - CUDA (NVIDIA) – GPGPU programming in C
  - NVIDIA GeForce 8600, 8800
  - ATI Radeon R600 (HD 2400, 3850)
- 2009
  - OpenGL 3.2, DirectX 11
  - GPU tesselation
  - NVIDIA Fermi

CONSUMER HW 5

- 2010
  - OpenGL 4
  - OpenCL: general computing on GPU, multiplatform
- 2011 - 2017
  - computing servers using many GPU cards (NVIDIA Tesla architecture)
  - OpenGL 4.6
  - DirectX 12 (Windows 10, Xbox One)
  - OpenGL ES for mobile platforms (GLES 3.2)
- 2018 - NVIDIA Volta - HW raytracing support
- other manufacturers
  - Past: Matrox, 3DLabs, S3, PowerVR (Kyro), SiS
  - Intel: very good integrated GPUs

OpenGL History I

- OpenGL 1.0 (1992) - First release based on Iris GL
- OpenGL 1.1 (1997) - Texture objects
- OpenGL 1.2 (1998) - 3D textures, BGRA and packed pixel formats
- OpenGL 1.3 (2001) - Multitexturing, multisampling, texture compression
- OpenGL 1.4 (2002) - Depth textures
- OpenGL 1.5 (2003) - Vertex Buffer Object (VBO), Occlusion Queries
- OpenGL 2.0 (2004) - GLSL 1.1, MRT, Non Power of Two textures, Point Sprites, Two-sided stencil
- OpenGL 2.1 (2006) - GLSL 1.2, Pixel Buffer Object (PBO), sRGB Textures
- OpenGL 3.0 (2008) - GLSL 1.3, Texture Arrays, Conditional rendering, Frame Buffer Object (FBO), API Deprecation Mechanism
- OpenGL 3.1 (2009) - GLSL 1.4, Instancing, Texture Buffer Object, Uniform Buffer Object, Primitive restart
- OpenGL 3.2 (2009) - GLSL 1.5, Geometry Shader, Multi-sampled textures
- OpenGL 3.3 (2010) - GLSL 3.30 Backports as much function as possible from the OpenGL 4.0 specification

OpenGL History II

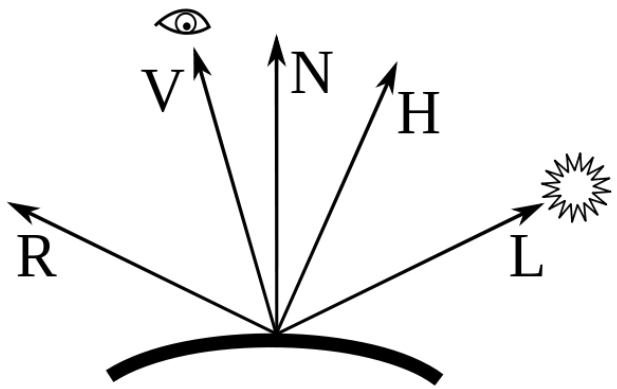
- OpenGL 4.0 (2010) - GLSL 4.00 Tessellation on GPU, shaders with 64-bit precision
- OpenGL 4.1 (2010) - GLSL 4.10 Developer-friendly debug outputs, compatibility with OpenGL ES 2.0
- OpenGL 4.2 (2011) - GLSL 4.20 Shaders with atomic counters, draw transform feedback instanced, shader packing, performance improvements
- OpenGL 4.3 (2012) - GLSL 4.30 Compute shaders leveraging GPU parallelism, shader storage buffer objects, high-quality ETC2/EAC texture compression, increased memory security, a multi-application robustness extension, compatibility with OpenGL ES 3.0
- OpenGL 4.4 (2013) - GLSL 4.40 Buffer Placement Control, Efficient Asynchronous Queries, Shader Variable Layout, Efficient Multiple Object Binding, Streamlined Porting of Direct3D applications, Bindless Texture Extension, Sparse Texture Extension
- OpenGL 4.5 (2014) - GLSL 4.50 Direct State Access (DSA), Flush Control, Robustness, OpenGL ES 3.1 API and shader compatibility, DX11 emulation features
- OpenGL 4.6 (2017) - GLSL 4.60 More efficient geometry processing and shader execution, SPIR-V, anisotropic filtering

OpenGL – Fixed Function Pipeline

- Before shaders
- Hardwired transformations, lighting
- Texturing + multitexturing combination operators

OpenGL FFP – Lighting Model

- Gouraud shading
- Blinn-Phong reflection model
- Hardcoded maximum number of available lights



Phong:

$$I_p = k_a i_a + \sum_{m \in lights} (k_d (L_m \cdot N) i_{m,d} + k_s (R_m \cdot V)^{\alpha} i_{m,s}) \quad (1)$$

$$R_m = 2(L_m \cdot N)N - L_m \quad (2)$$

Blinn-Phong:

$$I_p = k_a i_a + \sum_{m \in lights} (k_d (L_m \cdot N) i_{m,d} + k_s (N \cdot H)^{\alpha} i_{m,s}) \quad (3)$$

$$H = normalize(L + V) \quad (4)$$

Blinn-Phong Shader

```
   float blinn_phong_specular(
      in vec3 normal,
      in vec3 viewDir,
      in vec3 lightDir,
      in float shininess)
{
   vec3 halfDir = normalize(lightDir + viewDir);
   float specAngle = max(dot(halfDir, normal), 0.0);
   return pow(specAngle, shininess*4.0);
}

float phong_specular(
      in vec3 normal,
      in vec3 viewDir,
      in vec3 lightDir,
      in float shininess)
{
   vec3 reflectDir = reflect(-lightDir, normal);
   float specAngle = max(dot(reflectDir, viewDir), 0.0);
   return pow(specAngle, shininess);
}
```

BLINN-PHONG SHADER 2

```
vec3 blinn_phong(
      in vec3 ambientColor,
      in vec3 diffuseColor,
      in vec3 specColor,
      in vec3 normal,
      in vec3 viewDir,
      in vec3 lightColor,
      in vec3 lightDir,
      in float lightPower,
      in float shininess)
{
   float lambertian = max(dot(lightDir, normal), 0.0);
   float specular = 0.0;

   if(lambertian > 0.0) {
      specular = blinn_phong_specular(normal, viewDir, lightDir, shininess);
   }
   return ambientColor +
      (diffuseColor * lambertian * lightColor +
      specColor * specular * lightColor) * lightPower;
}
```

Summary

SUMMARY: OPENGL CALLS

- Shaders:
  glCreateShader(), glDeleteShader(), glShaderSource(),
  glCompileShader(), glAttachShader(), glLinkProgram()
  glProgramUniform*(), glGetUniformLocation(), ...
- Buffer objects:
  glGenBuffers(), glBindBuffer(), glBufferData(),
  glBindVertexArray(), glGenVertexArrays(),
  glVertexAttribPointer(), glEnableVertexAttribArray(),
  glDeleteVertexArrays(), glDeleteVertexArrays()
- Drawing:
  glDrawElements(), glDrawArrays()
- Other:
  glViewPort(), glClearColor(), glClear(), glFinish()

SUMMARY: GLSL DATA TYPES

- Scalars:
  bool, int, uin, float, double
- Vectors:
  bvecn, ivecn, uvecn, vecn, dvecn ($n \in 2, 3, 4$)
  - Swizzling:
    someVec.x, someVec.y, someVec.xyxx
- Matrices:
  matnxm, matn ($n, m \in 2, 3, 4$)
- Arrays and structs
- Storage qualifiers:
  constant, in, out, uniform, ...