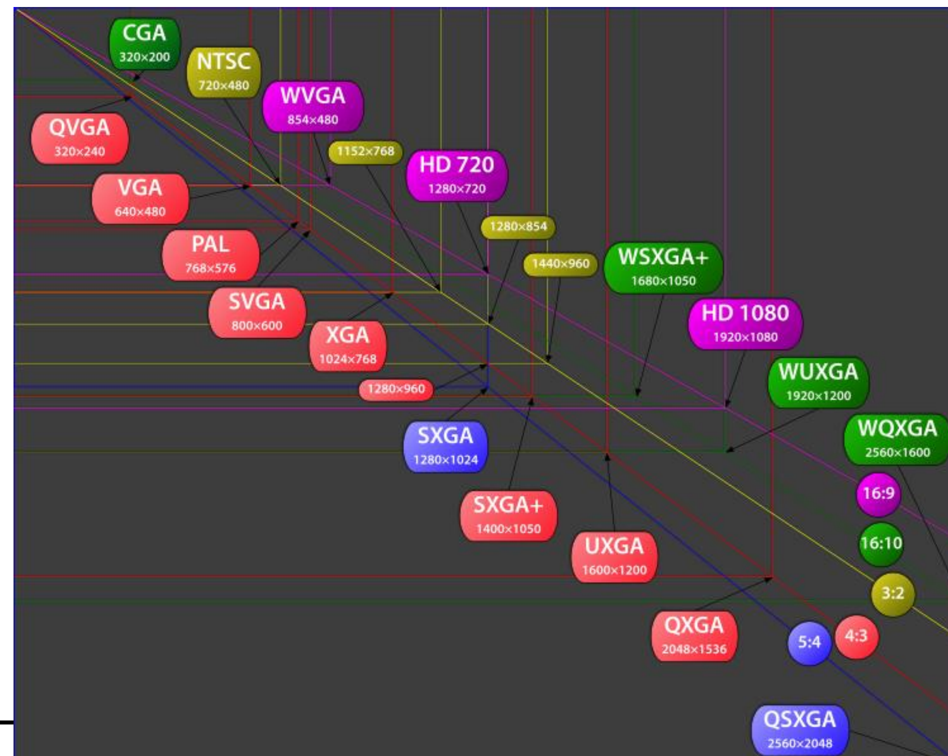
Computer Graphics

- Programmable Shading in OpenGL -

Philipp Slusallek

History: Graphics HW

- **Graphics in the ‘80ies**
  - Framebuffer was a designated memory area in RAM
  - „HW“: Set individual pixels directly via memory access
    - Peek & poke, getpixel & putpixel, …
    - MDA ('81: text only but 720x350 resolution, monochrome, 4 kB of RAM!)
      - Character code was index into bit pattern in ROM for each character
    - CGA ('81: 160x200:
      16 colors w/ tricks;
      320x200: 4 col;
      640x200: 2 col)
    - EGA ('85: 640x350: 16 from 64 col, CGA mode)
    - VGA ('90: 640x480: 16 col @ table with 2^18 col, 320x200: 256 col), with BIOS extension
  - Everything done on the CPU
    - Except for driving the display output



History: Graphics HW (II)

SGI Infinite Reality Geometry Engine (approx. 51x48 cm)

- **Pre-GPU graphics acceleration**
  - SGI, Evans & Sutherland
  - Introduced concepts like vertex transformation and texture mapping
- **First-generation GPUs (-1998)**
  - NVIDIA TNT2, ATI Rage, Voodoo3
  - Vertex transformation on CPU, limited set of math operations
- **Second-generation GPUs (1999-2000)**
  - GeForce 256, GeForce2, Radeon 7500, Savage3D
  - Transform & lighting, more configurable, not programmable
- **Third-generation GPUs (2001)**
  - GeForce3, GeForce4 Ti, Xbox, Radeon 8500
  - Vertex programmability, pixel-level configurability
- **Fourth-generation GPUs (2002)**
  - GeForce FX series, Radeon 9700 and on
  - Vertex-level and pixel-level programmability (limited)
- **Eighth-generation GPUs (2007)**
  - Geometry shaders, feedback, *unified shaders*, *CUDA* (!) …
- **…**

History: Graphics HW (III)

Generation	Year	Product	Process	Transistors	Antialiasing fill rate	Polygon rate
1st	1998	RIVA TNT	0.25μ	7 M	50 M	6 M
1st	1999	RIVA TNT2	0.22μ	9 M	75 M	9 M
2nd	1999	GeForce 256	0.22μ	23 M	120 M	15 M
2nd	2000	GeForce2	0.18μ	25 M	200 M	25 M
3rd	2001	GeForce3	0.15μ	57 M	800 M	30 M
3rd	2002	GeForce4 Ti	0.15μ	63 M	1,200 M	60 M
4th	2003	GeForce FX	0.13μ	125 M	2,000 M	200 M
8th	2007	GeForce 8800 (GT100)	0.09μ	681 M	36,800 M	13,800 M
8th	2008	GeForce 280 (GT200)	0.065μ	1,400 M	48,200 M	??
9th	2009	GeForce 480 (GF100)	0.04μ	3,000 M	42,000 M	??

1000x (in 10 years)

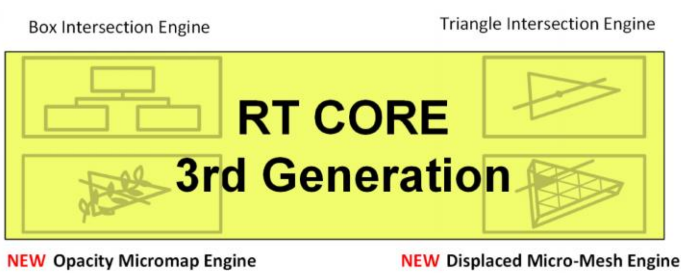
History: Graphics HW (III)

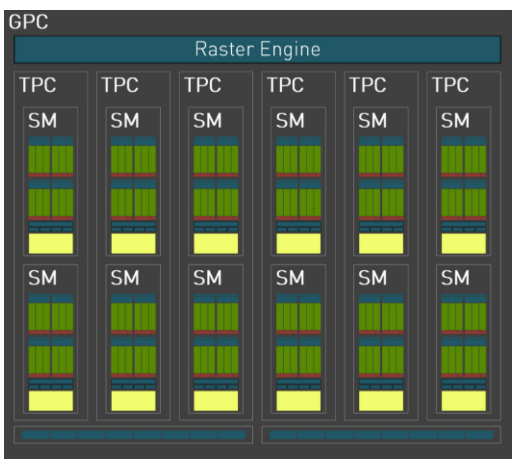
- **Today (Nvidia Blackwell, Flagship GB 202, RTX 50)**
  - Discrete graphics card via high-speed link
    - e.g. PCIe-5.0 x16: up to 63 GB/s transfer rate
  - Autonomous, high-performance GPU (more powerful than CPU)
    - 21,760 SIMD processors (32 in each of 680 Streaming Processors/SPs)
    - Up to 32GB of local GDDR7 RAM
    - 1,792 GB/s memory bandwidth (512 bits wide)
    - 104.8 TFLOPS 16bit floats
    - 104.8 TFLOPS single precision (SP) + 1.6 TFLOPS doubles (DP)
    - 1,676 TFLOPS in FP16 via 680 Tensor Cores (RTX), 16x of SPs
    - 380 TFLOPS in 170 RT Cores
    - Dedicated ray tracing HW unit (BVH traversal & tri. interpol & intersect)
    - Total of 92 Billion transistors at 575 Watts (!!)
  - Performs all low-level tasks & a lot of high-level tasks
    - Clipping, rasterization, hidden surface removal, … + Ray Tracing
    - Procedural geometry, shading, texturing, animation, simulation, …
    - Video rendering, de- and encoding, deinterlacing, …
    - Full programmability at several pipeline stages
    - Deep Learning & Matrix-Multiply (sparse x2): Training and Inference

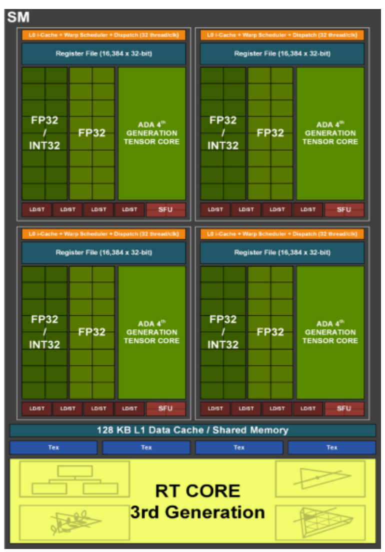
Nvidia GB202 (AD102) GPU

GB202 (Blackwell Architecture)

AD102 (Ada Architecture)







Shading Languages (in general)

- **Small program fragments (a la plug-ins)**
  - Compute certain aspects of the rendering process
  - Executing at innermost loops, must be extremely efficient
  - Executed at each intersection (in ray tracing) and other events
- **Typical shaders (e.g. from RenderMan)**
  - Material/surface shaders: compute reflected color
  - Light shaders: compute light emission towards a given position
  - Volume shader: compute interaction in a participating medium
  - Displacement shader: compute changes to the geometry
  - Camera shader: compute rays for each pixel
- **Shading languages & material definitions**
  - RenderMan-SL (the "mother of all shading languages")
  - OSL: Open Shading Language (OSS, by Larry Gritz et al.)
  - Currently no widely deployed portable material format, yet
    - But MaterialX (OSS) & Material Definition Language (MDL, Nvidia)

History of Shading Languages

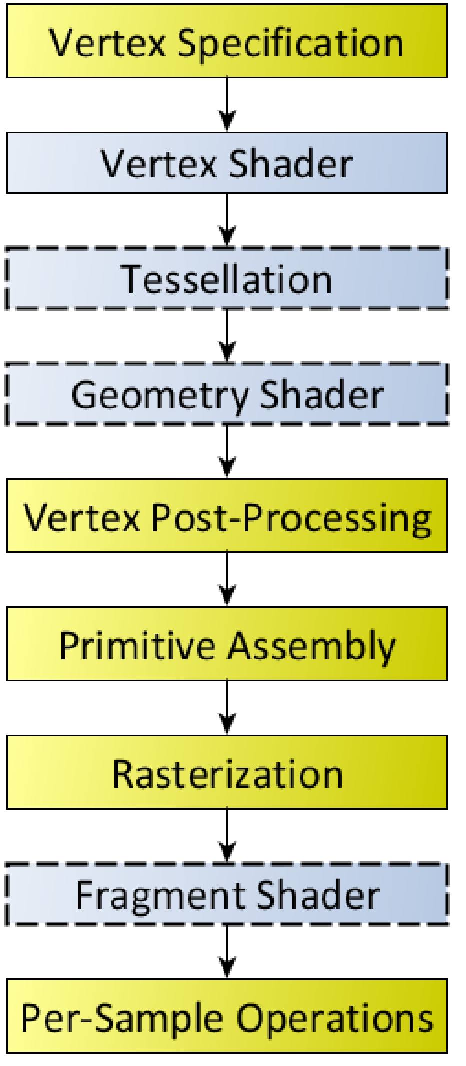
- **Rob Cook: shade trees (1984 @ LucasFilm)**
  – Flexible connection of function blocks
- **Ken Perlin: The Image Synthesizer (1985)**
  – Deep pixels (pixels with more than color data)
  – Control structures, noise function
- **Pat Hanrahan: RenderMan (1988 @ Pixar)**
  – RenderMan is still a widely used shading language
  – Mostly for offline and high-quality rendering

- **Later: Realtime shading languages**
  – RTSL (Stanford, initial work from 1999, lead to Cg)
  – Cg (NVIDIA, 2001, cross platform)
  – HLSL (Microsoft, 2002, with DirectX 9)
  – GLSL (Khronos, 2002, with OpenGL 1.4)
  – SPIR-V (Khronos, low-level, binary instructions for Vulkan/OpenCL)

GLSL

- **OpenGL Shading Language**
- **Syntax somewhat similar to C**
- **Supports overloading**
- **Used at different stages of the rendering pipeline**

- **Also Compute Shaders that work in a very different context**
  - No specified input
  - Different programming model (a la OpenCL but more restricted)
- **Recently, also “Mesh Shader”**
  - In a compute shaders context but submitting vertex input



GLSL: Data Types

- **Three basic data types in GLSL:**
  - float, bool, int – just like in C, uint: unsigned int
  - Allows for constructor syntax (vec3 a = vec3(1.0, 2.0, 3.0))
- **Vectors with 2, 3 or 4 components, declared as:**
  - {, b, i, u}vec{2,3,4}: a vector of 2, 3 or 4 floats, bools, ints, unsigned
- **Matrices**
  - mat2, mat3, mat4: square matrices
  - mat2x2, mat2x3, mat2x4, mat3x2 to mat4x4: explicit size
- **Sampler (texture access)**
  - {, i, u}sampler{1D, 2D, 3D, Cube, 2DRect, 1DArray, 2DArrray, Buffer, 2DMS, 2DMSArray}
    - float, int, unsigned: texture access return type (vec4, ivec4, uvec4)
    - Different types of textures:
      - Array: texture array, MS: multi-sample, buffer: buffer texture
    - Cannot be assigned, set by OpenGL, passed to same type of parameter
- **Structures: as in C**
- **Arrays: full types**

Storage/Interpolation Qualifiers

- **Storage qualifiers**
  - const
    - Compile time constant
  - in, centroid in (read only)
    - Input into a shader, pass by value
    - „Centroid“ interpolates at centroids (not sample positions) in fragment shader
  - out, centroid out
    - Output from a shader, pass by value
  - uniform (read only)
    - Does not change across a primitive, passed from OpenGL API
  - inout (only for „function“ parameter)
    - Passed by reference
- **Interpolation qualifiers (for in/out)**
  - flat: no interpolation
  - smooth: perspective correct interpolation
  - nonperspective: linear in screen space

Shader Input & Output

- **Variable names and types of connected shaders (e.g. Vertex → Fragment) must match**
  - But no sampler & no arrays (except for vertex shader in)
  - Vertex shaders cannot have structures (but arrays)
  - Geometry shader must have all variables as arrays
    - Receives an entire primitive
    - "in float a[ ]" for an output "out float a" from the vertex shader
  - int and uint must be "flat" for a fragment shader (no interpolation)
  - Fragment shader cannot have matrix or structure output
- **Interface blocks**
  - in/out/uniform InterfaceName { …} instance_name;
  - Groups together related variables
  - InterfaceName is used for name lookup from OpenGL
    - InterfaceName.VariableName
- **Advanced: Layout qualifiers**
  - Used to specify characteristics of geometry shaders
    - Primitive type of input and output, max number of output primitives, ...

Vertex Shader Input/Output

- **Predefined vertex shader variables**

```
in int gl_VertexID;          // Implicit index of vertex in vertex-array call
in int gl_InstanceID;        // Instance ID passed by instance calls
in int gl_DrawID;            // Draw ID from MultiDraw commands

out gl_PerVertex
{
   vec4  gl_Position;        // Homogeneous position of vertex
   float gl_PointSize;       // Size of point in pixels
   float gl_ClipDistance[];  // Distance from clipping planes > 0 == valid,
                                think WEC
};
```

  – Vertex geometry output data must be in the OpenGL canonical view volume:
    - Given by $-w \leq \{x, y, z\} \leq w$

Geometry Shader Input/Output

- **Predefined geometry shader variables**

```
in gl_PerVertex
{
   vec4  gl_Position;
   float gl_PointSize;
   float gl_ClipDistance[];
} gl_in[];

in int gl_PrimitiveIDIn; // # of primitives processed so far in input

out gl_PerVertex
{
   vec4  gl_Position;
   float gl_PointSize;
   float gl_ClipDistance[];
};

out int gl_PrimitiveID;
out int gl_Layer;         // Specifies layer of frame buffer to write to
```

Fragment Shader Input/Output

- **Predefined fragment shader variables**

```
in vec4   gl_FragCoord;       // (x, y, z, 1/w) for (sub-)sample
in bool   gl_FrontFacing;     // Primitive is front facing
in float  gl_ClipDistance[];  // Linearly interpolated
in vec2   gl_PointCoord;      // 2D coords within point sprite
in int    gl_PrimitiveID;     // As before

out float gl_FragDepth;       // Computed depth value
```

- **Output „fragment colors“**
  - Shader can have any number of output values, e.g.
    `out vec4 color;`
  - Single color output goes to the default frame buffer
  - The application can assign additional values to *compatible* buffers defined through glDrawBuffers()
  - Assignment of output variables to draw buffers can be done by glBindFragDataLocation(program, colorNumber, name);

GLSL Operations

- **Vector component access**
  - {x,y,z,w}, {r,g,b,a} and {s,t,p,q} provide equivalent access to vecs
  - LHS: no repetition, defines type for assignment
  - RHS: arbitrary set, defines result type
  - Example:

```
vec4 pos = vec4(1.0, 2.0, 3.0, 4.0);
vec4 swiz = pos.wzyx;     // swiz = (4.0, 3.0, 2.0, 1.0)
vec4 dup = pos.xxyy;      // dup = (1.0, 1.0, 2.0, 2.0)
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)
```

- **All vector and matrix operations act component wise**
  - Except multiplication involving a matrix:
    - Results in correct vec/mat, mat/vec, mat/mat multiply from LinAlg

Control Flow

- **Usual C/C++ control flow, but …**
- **`discard`**
  - Statement allowed only in fragment shader
  - Fragment is thrown away, does not reach frame buffer

- **Everything is executed on a SIMD processor**
  - Should make sure that control flow is as similar as possible
- **Some texture functions require implicit derivatives**
  - Computed from a 2x2 pixel “quad” through divided differences
  - Required to be in control flow only containing uniform conditions

Functions

- **Example**

```
vec4 toonify(in float intensity)
{
    vec4 color;

    if      (intensity > 0.98) color = vec4(0.8,0.8,0.8,1.0);
    else if (intensity > 0.50) color = vec4(0.4,0.4,0.8,1.0);
    else if (intensity > 0.25) color = vec4(0.2,0.2,0.4,1.0);
    else                       color = vec4(0.1,0.1,0.1,1.0);

    return(color);
}
```

  – No recursion or other fancy things

Shader Library

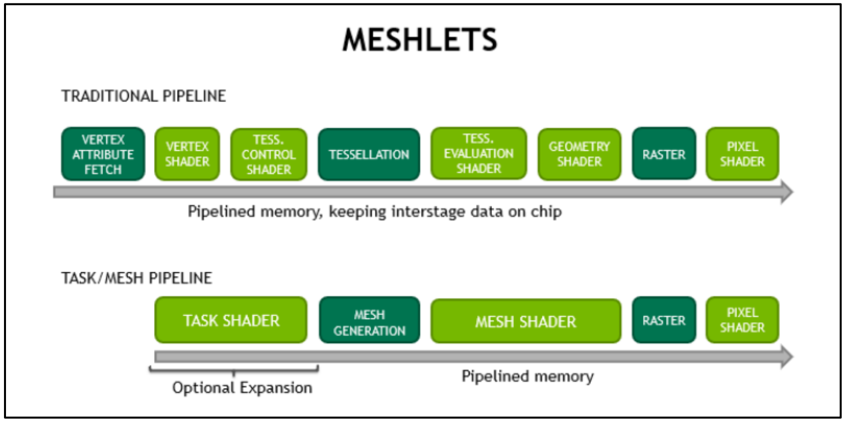
- **Typical math library**
  - sin, cos, pow, min/max, …
  - clamp, max, dot, cross, normalize, …
- **Shader specific**
  - faceforward(N, I, Nref): returns N iff dot(Nref, I) < 0, -N otherwise
  - reflect(I, N): reflects I at plane with normalized normal N
  - refract(I, N, eta): refracts at normalized N with refraction index eta
  - smoothstep(begin, end, x): Hermite interpolation between 0 and 1
  - mix(x, y, a): affine interpolation
  - noise1() to noise4(): Perlin-style noise
  - …

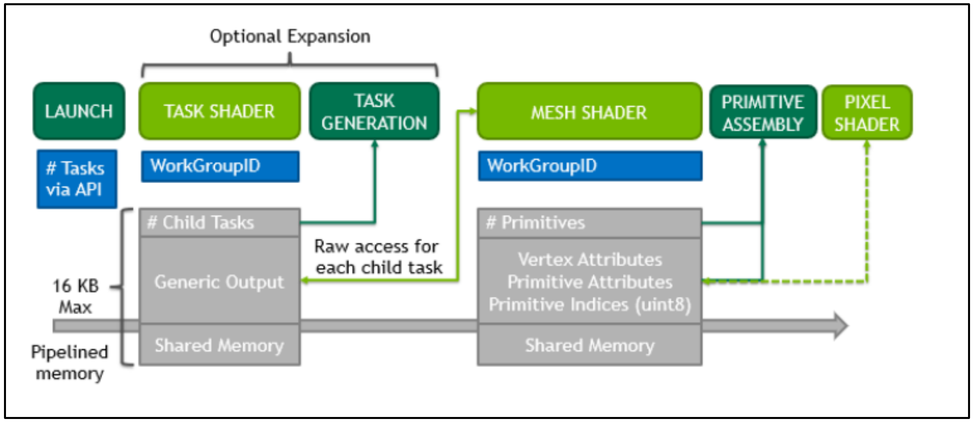
Shader Library: Texturing & Deriv.

- **Huge list of texture functions**
  - Direct and homogeneous projection sampling
  - With and without LOD (MIP-mapping)
  - With and without offset in texture coordinates
  - With and without derivatives in texture space
  - Fetch with integer coords (no interpolation/filtering)
  - Combinations of the above
- **Derivatives**
  - `dFdx(e), dFdy(e)`: derivatives of expression e with respect to window coordinates
    - Approximated with divided differences on “quads”: piecewise linear
  - `fwidth(e) = abs(dFdx(e)) + abs(dFdy(e))`
    - Approximate filter width

Changes with New Mesh Shaders

- **Mesh Shader Concept**
  - Move most scene processing into *compute shaders* on GPU
  - **Task shader:** A programmable unit that operates in workgroups and allows each to emit (or not) mesh shader workgroups
  - **Mesh shader:** A programmable unit that operates in workgroups and allows each to generate primitives
  - **Workgroup:** A number of jobs submitted for parallel execution (on a GPU) with some shared memory

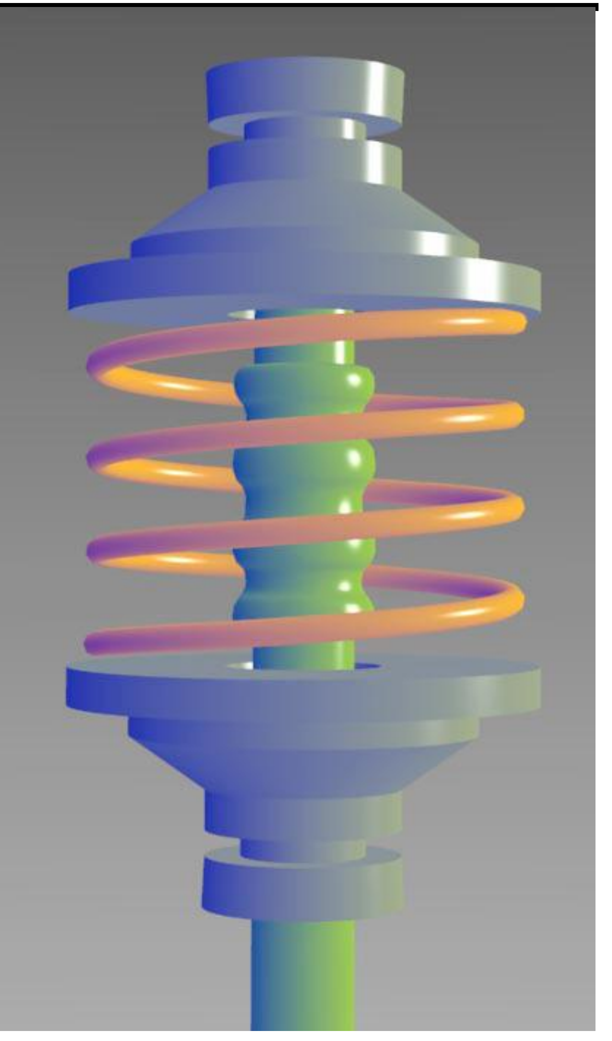




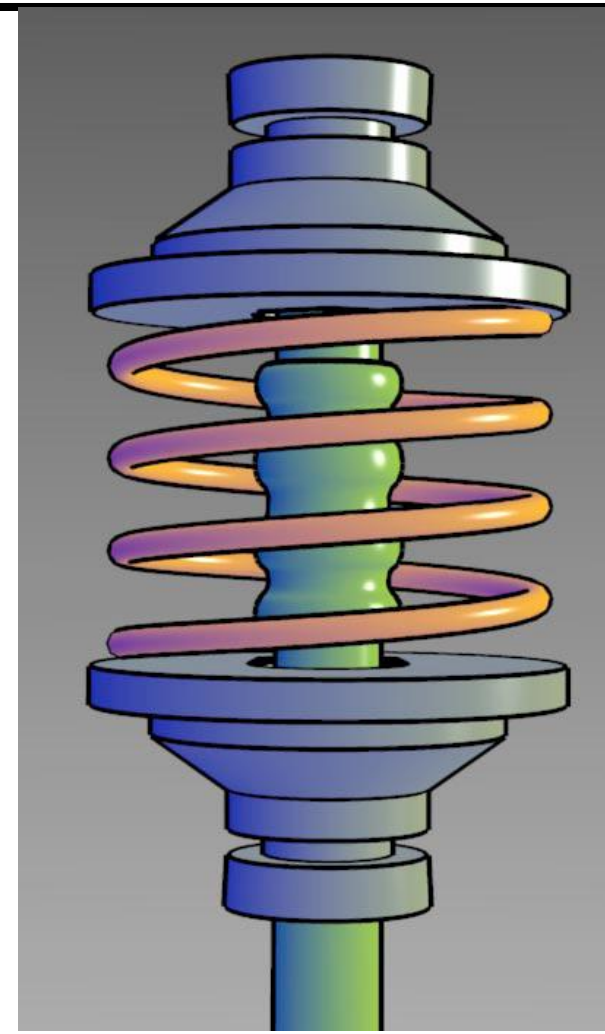
Ex.: Gooch Cool/Warm Shader

- **Vertex shader**

```
uniform vec4 lightPos;
uniform mat4x4 modelview_mat;
uniform mat4x4 modelviewproj_mat;
uniform mat4x4 normal_mat;

in vec3 P;
in vec3 N;

out vec3 normal;
out vec3 lightVec;
out vec3 viewVec;

void main()
{
   // position in clip-space
   gl_Position = modelviewproj_mat * P;
   vec4 vert = modelview_mat * P;

   normal = normal_mat * N;
   lightVec = vec3(lightPos - vert);
   viewVec = -vec3(vert);
}
```



Ex.: Gooch Cool/Warm Shader



- **Fragment shader**

```
uniform Material
{
    float Ka = 1.0;
    float Kd = 0.8;
    float Ks = 0.9;
    vec3 ambient = vec3(0.2, 0.2, 0.2);
    vec3 spec_col = vec3(1.0, 1.0, 1.0);
    vec3 kCool = vec3(.88, .81, .49);  // Purple
    vec3 kWarm = vec3(.58, .10, .76);  // Orange
} m;

in vec3 normal;
in vec3 lightVec;
in vec3 viewVec;

out vec4 frag_color;

void main()
{
    vec3 norm = normalize(normal);
    vec3 L = normalize(lightVec);
    vec3 V = normalize(viewVec);
    vec3 halfAngle = normalize(L + V);
    float NdotH = clamp(dot(halfAngle, norm), 0.0, 1.0);
    float spec = pow(NdotH, 64.0);
    vec3 Cgooch = mix(mat.kWarm, mat.kCool, 0.5 * dot(L, norm) + 0.5);
    vec3 res = m.Ka * m.ambient + m.Kd * Cgooch + m.spec_col * m.Ks * spec;
    frag_color = vec4(res, 1.0);
}
```

Simple OpenGL Program

- **Discussion of critical pieces of an OpenGL program**
- **Details available at**
  - http://duriansoftware.com/joe/An-intro-to-modern-OpenGL.-Table-of-Contents.html

Simple OpenGL Program

Main program

```
#include <stdlib.h>
#include <GL/glew.h>
#include <GL/glut.h>

int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_RGB|GLUT_DEPTH|GLUT_DOUBLE);
    glutInitWindowSize(400, 300);
    glutCreateWindow("Hello World");
    glutDisplayFunc(&render);
    glutIdleFunc(&update);

    // Initialize the GL Extension Wrangler Library
    glewInit();
    if (!GLEW_VERSION_2_0) {
        fprintf(stderr, "OpenGL 2.0 not available\n");
        return 1;
    }

    // Initialize data structures

    glutMainLoop();
    return 0;
}
```

```
// Called if nothing else to do
static void update (void)
{
    int ms = glutGet(GLUT_ELAPSED_TIME);
    // Do any animation control here

    glutPostRedisplay();
}

// Called if display needs to be rendered
// e.g. due to PostRedisplay() or exposure of window
static void render(void)
{
    // Should ideally just be done once
    glEnable(GL_DEPTH_TEST);
    glEnable(GL_CULL_FACE);
    glClearColor(1.0f, 1.0f, 1.0f, 1.0f);

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // real rendering code

    glutSwapBuffers();
}
```

Simple OpenGL Program

Vertex Shader

```
#version 110

uniform mat4 p_matrix, mv_matrix;

attribute vec3 position, normal;
attribute vec2 texcoord;
attribute float shininess;
attribute vec4 specular;

varying vec3 frag_position, frag_normal;
varying vec2 frag_texcoord;
varying float frag_shininess;
varying vec4 frag_specular;

void main()
{
    vec4 eye_position = mv_matrix * vec4(position, 1.0);
    gl_Position = p_matrix * eye_position;
    frag_position = eye_position.xyz;
    frag_normal   = (mv_matrix * vec4(normal, 0.0)).xyz;
    frag_texcoord = texcoord;
    frag_shininess = shininess;
    frag_specular = specular;
}
```

Fragment Shader

```
#version 110

uniform mat4 p_matrix, mv_matrix;
uniform sampler2D texture;

varying vec3 frag_position, frag_normal;
varying vec2 frag_texcoord;
varying float frag_shininess;
varying vec4 frag_specular;

const vec3 light_direction = vec3(0.408248, -0.816497, 0.408248);
const vec4 light_diffuse = vec4(0.8, 0.8, 0.8, 0.0);
const vec4 light_ambient = vec4(0.2, 0.2, 0.2, 1.0);
const vec4 light_specular = vec4(1.0, 1.0, 1.0, 1.0);

void main()
{
    vec3 mv_light_direction = (mv_matrix * vec4(light_direction, 0.0)).xyz,
         normal = normalize(frag_normal),
         eye = normalize(frag_position),
         reflection = reflect(mv_light_direction, normal);

    vec4 frag_diffuse = texture2D(texture, frag_texcoord);
    vec4 diffuse_factor
        = max(-dot(normal, mv_light_direction), 0.0) * light_diffuse;
    vec4 ambient_diffuse_factor
        = diffuse_factor * light_ambient;
    vec4 specular_factor
        = max(pow(-dot(reflection, eye), frag_shininess), 0.0)
            * light_specular;

    gl_FragColor = specular_factor * frag_specular
        + ambient_diffuse_factor * frag_diffuse;
}
```

Simple OpenGL Program

Generating shaders

```
static Gluint
make_shader(GLenum type, const char *filename)
{
    GLint length, shader_ok;
    GLchar *source = file_contents(filename, &length);
    GLuint shader;

    if (!source) return 0;
    shader = glCreateShader(type);
    glShaderSource(shader, 1,
                   (const GLchar**)&source, &length);
    free(source);
    glCompileShader(shader);
    glGetShaderiv(shader, GL_COMPILE_STATUS,
                  &shader_ok);
    if (!shader_ok) {
        fprintf(stderr,
           "Failed to compile %s:\n", filename);
        glDeleteShader(shader);
        return 0;
    }
    return shader;
}
```

Generating the shader program

```
static Gluint
make_program(GLuint vertex_shader, GLuint fragment_shader)
{
    GLint program_ok;

    GLuint program = glCreateProgram();
    glAttachShader(program, vertex_shader);
    glAttachShader(program, fragment_shader);
    glLinkProgram(program);

    glGetProgramiv(program, GL_LINK_STATUS, &program_ok);
    if (!program_ok) {
        fprintf(stderr, "Failed to link shader program:\n");
        glDeleteProgram(program);
        return 0;
    }
    return program;
}

// Getting access to the shader variables
uniform.texture= glGetUniformLocation(program, "texture");
attributes.position= glGetAttribLocation(program, "position");
// …
```

Simple OpenGL Program

Defining a texture

```
static GLuint
make_texture(const char *filename)
{
    GLuint texture;
    int width, height;
    void *pixels = read_imagefile(filename, &width, &height);

    if (!pixels) return 0;
    // Create a texture object and make it the current one
    glGenTextures(1, &texture);
    glBindTexture(GL_TEXTURE_2D, texture);

    // Set parameters (ideally would use a texture sampler here!)
    glTexParameteri(GL_TEXTURE_2D,
       GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D,
       GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D,
       GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_2D,
       GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);

    // Upload the texture data
    glTexImage2D(
        GL_TEXTURE_2D, 0,          /* target, level of detail */
        GL_RGB8,                   /* internal format */
        width, height, 0,          /* width, height, border */
        GL_BGR, GL_UNSIGNED_BYTE,  /* external fmt, type */
        pixels                     /* pixel data */
    );
    free(pixels);
    return texture;
}
```

Simple OpenGL Program

Defining the scene data structure

```
struct flag_vertex {
    GLfloat position[4];
    GLfloat normal[4];
    GLfloat texcoord[2];
    GLfloat shininess;
    GLubyte specular[4];
};
```

Generating and filling the buffers

```
struct flag_vertex *vertex_data= (struct flag_vertex*)
        malloc(FLAG_VERTEX_COUNT * sizeof(struct flag_vertex));
GLushort *element_data= (GLushort*)
        malloc(element_count * sizeof(GLushort));

/* Generate the data */
GLuint vertex_buffer, element_buffer;
glGenBuffers(1, &vertex_buffer);
glGenBuffers(1, &element_buffer);

// Filling the buffers
glBindBuffer(GL_ARRAY_BUFFER, vertex_buffer);
glBufferData(GL_ARRAY_BUFFER,
    vertex_count * sizeof(struct flag_vertex),
    vertex_data, GL_STREAM_DRAW);

glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, element_buffer);
glBufferData(GL_ELEMENT_ARRAY_BUFFER,
    element_count * sizeof(GLushort),
    element_data, GL_STATIC_DRAW);
```

Simple OpenGL Program

Actual Rendering

```
static void render(void)
{
    // Beginning of rendering code (glClear)

    // Activate shader and textures from make_* calls
    glUseProgram(program);

    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_2D, texture);
    glUniform1i(uniform.texture, 0);

    glBindBuffer(GL_ARRAY_BUFFER, vertex_buffer);
    glVertexAttribPointer(attributes.position,
        3, GL_FLOAT, GL_FALSE, sizeof(struct flag_vertex),
        (void*)offsetof(struct flag_vertex, position));
    glVertexAttribPointer(attributes.normal,
        3, GL_FLOAT, GL_FALSE, sizeof(struct flag_vertex),
        (void*)offsetof(struct flag_vertex, normal));
    glVertexAttribPointer(attributes.texcoord,
        2, GL_FLOAT, GL_FALSE, sizeof(struct flag_vertex),
        (void*)offsetof(struct flag_vertex, texcoord));
    glVertexAttribPointer(attributes.shininess,
        1, GL_FLOAT, GL_FALSE, sizeof(struct flag_vertex),
        (void*)offsetof(struct flag_vertex, shininess));
    glVertexAttribPointer(attributes.specular,
        4, GL_UNSIGNED_BYTE, GL_TRUE, sizeof(struct flag_vertex),
        (void*)offsetof(struct flag_vertex, specular));

    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, element_buffer);
    glDrawElements(GL_TRIANGLES, element_count,
        GL_UNSIGNED_SHORT, (void*)0);
```