GPU programming (shaders)

CGG MFF UK Praha

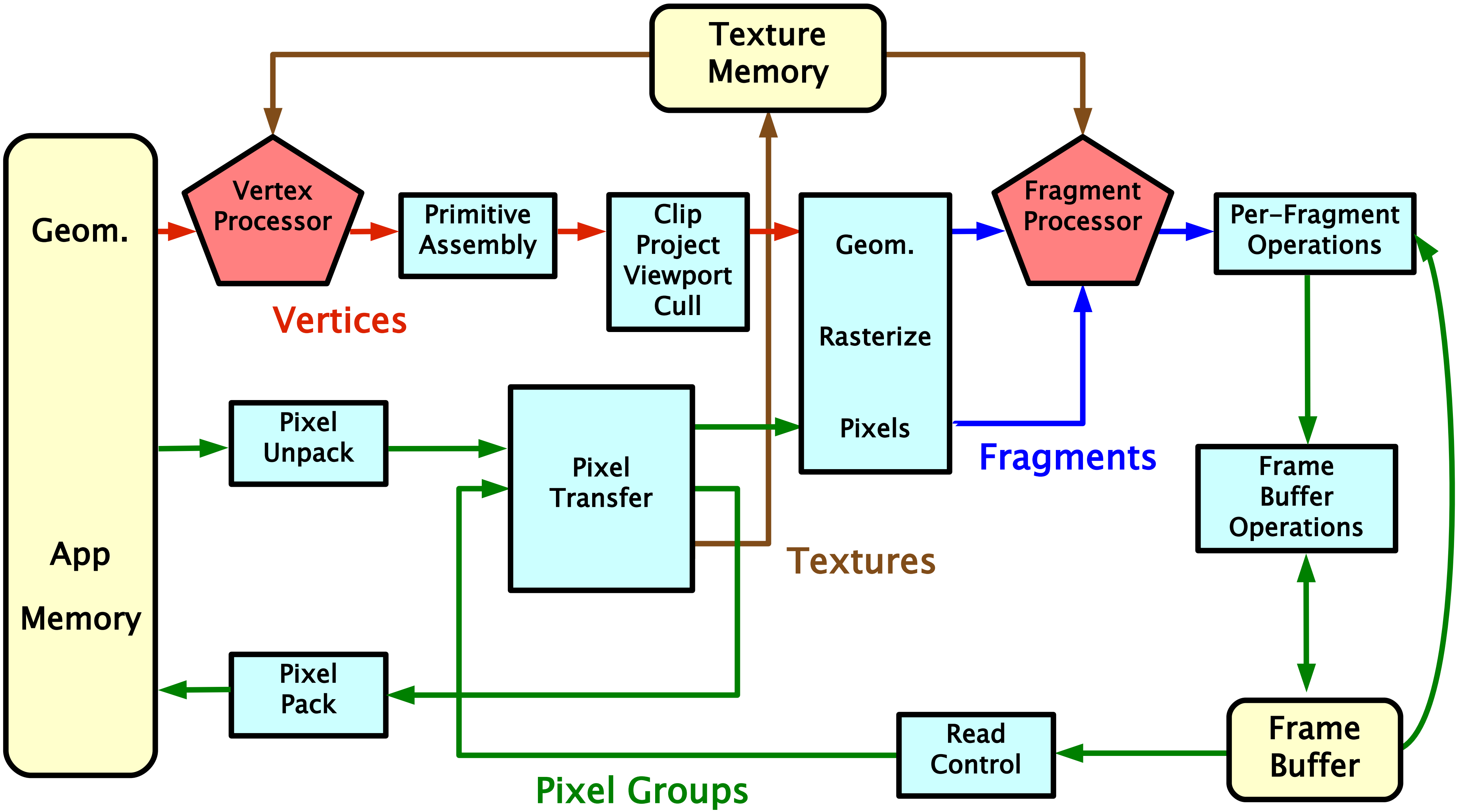
pepca@cgg.mff.cuni.cz

http://cgg.mff.cuni.cz/~pepca/

Content

- programmable GPU: architecture
- vertex shaders
- fragment shaders
- texture shaders (history)
- GPU programming
  - low-level: assembler
  - high-level languages: GLSL, Cg, HLSL

Programmable pipeline scheme



History

- NVIDIA **GeForce 3** (2001)
  - 1st programmable GPU, **DirectX 8.0**: VS,PS 1.0-1.1
  - in the same year: ATI **Radeon 8500**, Microsoft **Xbox**, NVIDIA **GeForce 4 Titanium**
  - very limited shaders (fragment shader looked more like configuration script, only a couple of assembly instructions), texture shaders
  - OpenGL: **ARB_vertex_program** extension (universal), fragment shading depends on manufacturer
- DirectX 8.1
  - PS 1.2-1.4 (ATI only), VS remains 1.1, Radeon 9000

History II

- NVIDIA **GeForce FX** (2002) – **CineFX** architecture
- ATI **Radeon 9500-9700**
  - **DirectX 9.0**: VS, PS 2.0
  - program size increased: 256 instructions
  - “constant” memory (uniform variables): 256 vectors
  - more data formats, FPU type “half”, etc.
  - **OpenGL**: equivalent functionality via extensions

- 2004: **DirectX 9.0c**: **VS**, **PS 3.0**
  - NVIDIA **GeForce 6800**, **6xxx** (**NV4x** chips), PCI-E bus
  - ATI still is not VS, PS 3.0 compatible (II/2005)

Shader model 3.0

- GPUs come close to universal computing units (CPU)
- large number of instructions (thousands, virt. no limit)
  - conditional jumps, loops, subroutines, recursion, ..
  - vertex shader can access texture memory (“vertex texturing”, 4 texture units)
  - **OpenGL**: equivalent functionality via NV_* extensions
- **NV4x** progress in HW:
  - twice more texture accesses in one cycle
  - FP16 & FP32 universally usable, HDR graphics (128bpp)
  - MRT (multiple render targets), 16x anisotropic filtering, SLI (more GPUs), HW geometric instancing, better anti-aliasing, ...

SW shaders

- **Pixar** since 1989: **RenderMan** shaders
  - universal definition of local lighting model
  - textures, noise, ...
  - very flexible thanks to SW implementation
  - inspiration for HW designers and 3D APIs
- **“Toy Story”** movie (1995)
  - first public demonstration of RenderMan capabilities
  - “final rendering” took months of CPU time on Sun RISC workstations (117 SPARCstation farm)
  - ~ 1300 different shaders !

Example: RenderMan shader (texture)

```
surface
turbulence ( float Kd =.8, Ka =.2 )
{
  float a, scale, sum;
  float IdotN;
  point M;
    /* convert to texture coordinate system  */
  M = transform( "marble", P );
  scale = 1;
  sum = 0;
  a = sqrt( area(M) );
  while ( a < scale )
  {
    sum += scale * float noise( M/scale );
    scale *= 0.5;
  }
  Oi = sum;
  Ci = Cs * Oi * (Ka + Kd * I.N * I.N / (I.I * N.N) );
}
```

More shader examples (light models)

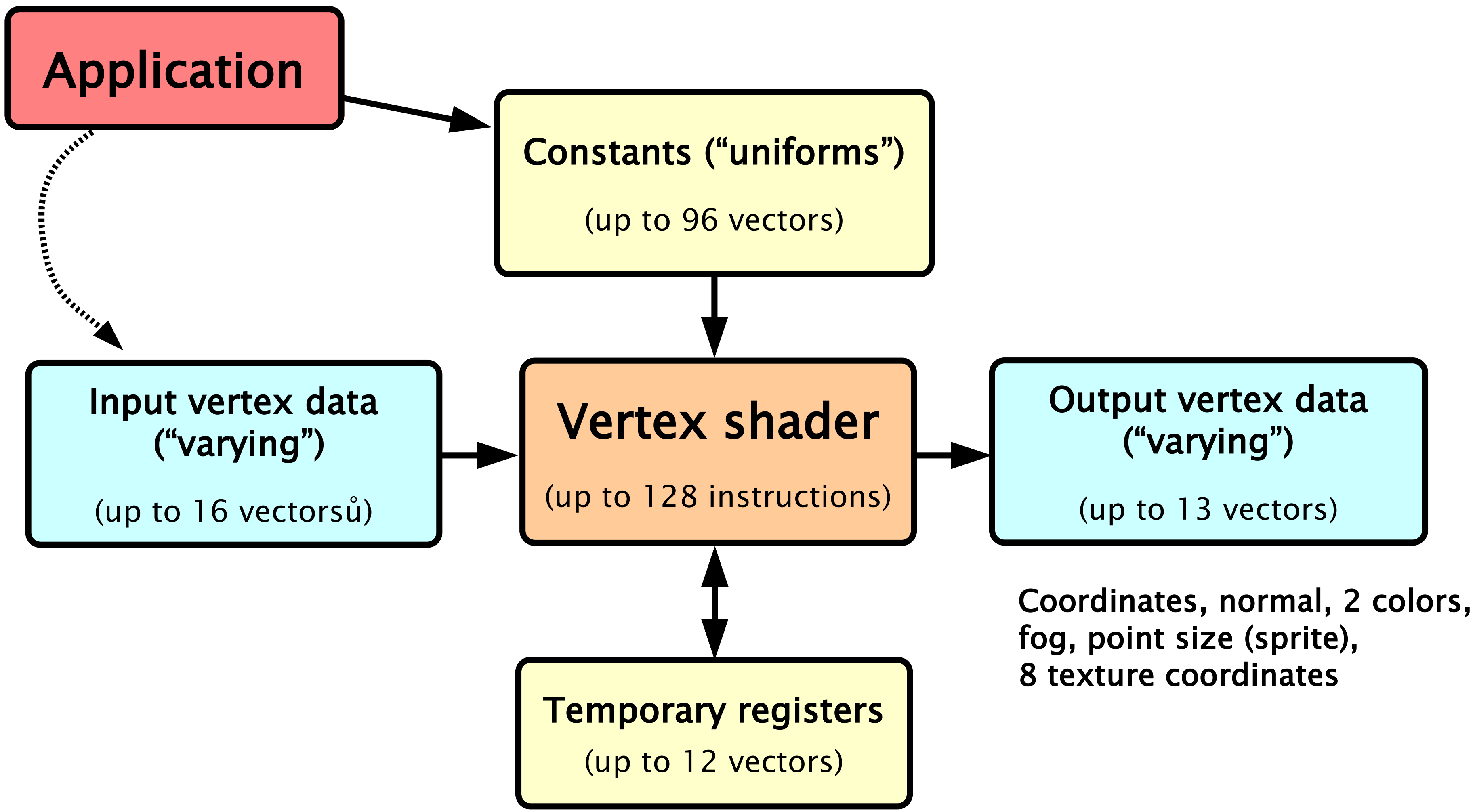
```
light
phong ( float intensity = 1.0;
        color color = 1;
        float size = 2.0;
        point from = point "shader" (0,0,0);
        point to = point "shader" (0,0,1); )
{
  uniform point R = normalize( to - from );
  solar( R, PI/2 )
    Cl = intensity * color * pow( R.L/length(L), size );
}


light
reflection ( string texturename = "";
             float intensity = 1.0 )
{
  solar()
    Cl = intensity * color environment( texturename, -L );
}
```

Vertex processor

- replaces the **vertex processing unit** in FFP
  - vertex coordinate transform
  - normal vector transform and normalization
  - computing/transformation of texturing coordinates
  - lighting vectors
  - setup of material attributes
- **cannot modify**
  - **number of vertices**
    - partial solution: primitive degeneration
  - type / topology of geometric primitives

VS 1.1 environment



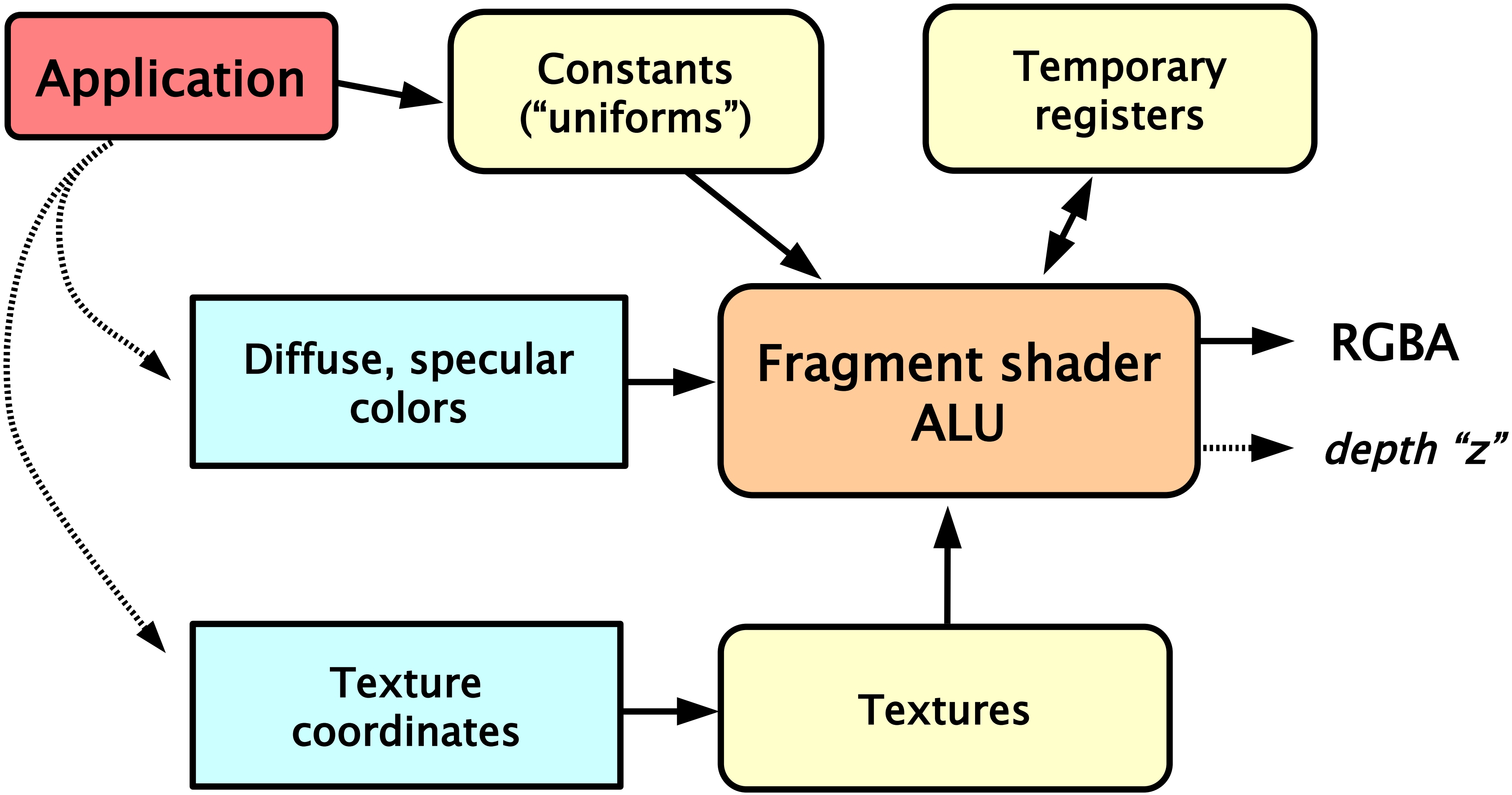
VS 2.0, VS 3.0

- **VS 2.0**: NVIDIA GeForce FX, ATI Radeon 9500
  - quantitative improvements only
  - more instructions (256), more uniforms (256)
- **VS 3.0**: NVIDIA GeForce 6xxx
  - **texture memory** access (“vertex texturing”)
    - e.g. for “displacement mapping”
    - pre-computed complicated function (noise, ..)
    - true constant and big data
  - virtually **unlimited instruction number** (≥32k)
    - would not be actually used (performance)

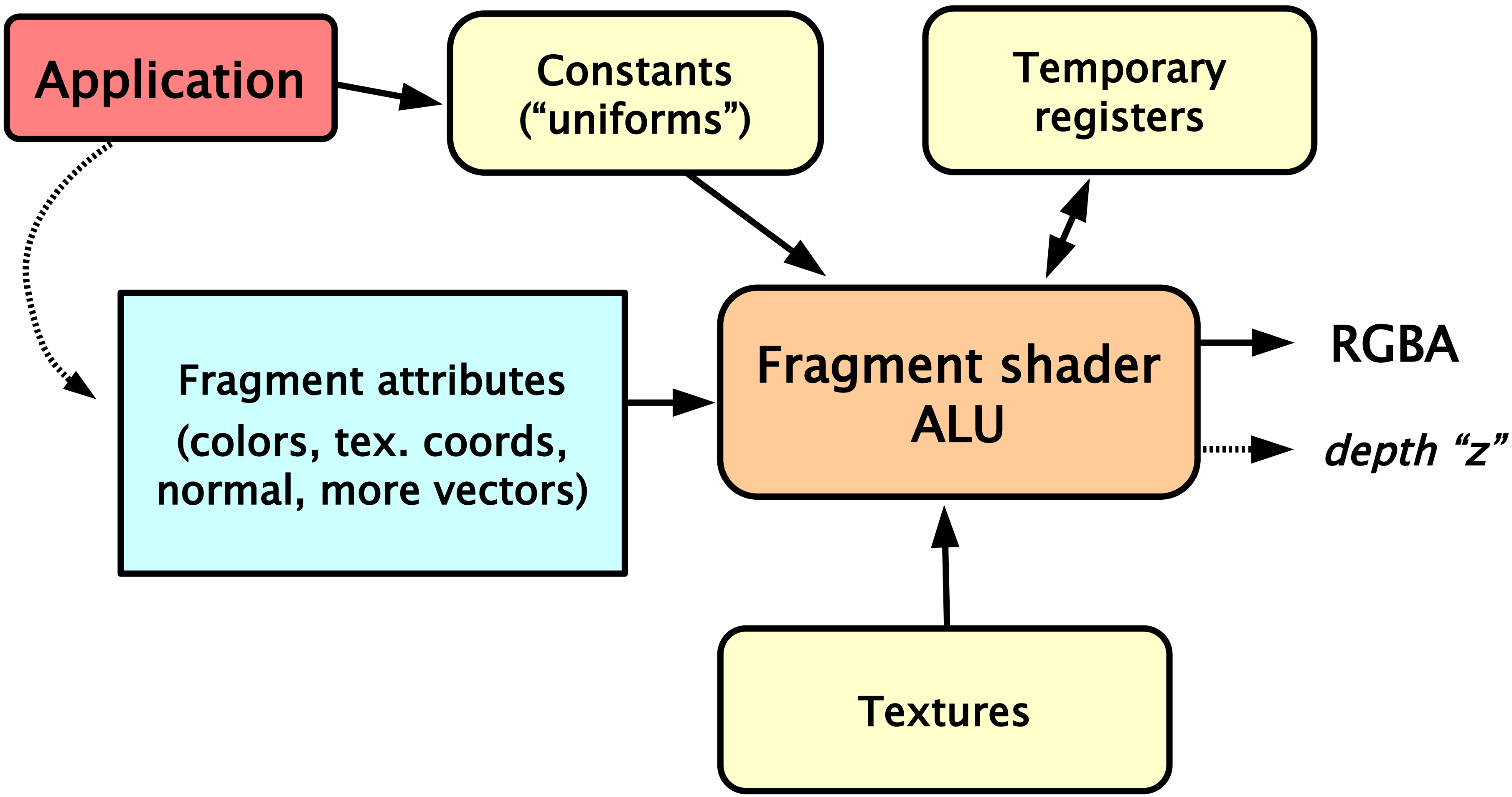
Fragment processor

- replaces **fragment processing unit** in FFP
  - arbitrary arithmetic on fragment attributes
  - texture data fetch and application (color, etc.)
  - fog computation
  - output fragment color synthesis
  - fragment depth can be modified
- **cannot modify**
  - **number of fragments** (except for the “discard” op.)
  - **fragment position** within the viewport [x,y]

FS 1.x environment



FS 2.0+ environment



Fragment shader model 2.0

- **FS 2.0**: NVIDIA GeForce FX, ATI Radeon 9500
  - 1st "real" shader model (RenderMan style)
  - arbitrary data (attributes) associated with fragment (perspective-correct interpolation in a rasterizer)
  - arbitrary arithmetic operations (texture coordinates!)
  - texture dependency (texel data can be used in further texture addressing)
  - more instructions (96), more uniforms (256 ?)
- replacement for "texture shaders"

Fragment shader model 3.0

- **FS 3.0**: NVIDIA GeForce 6xxx
  - virtually unlimited length (≥32k)
  - conditional jumps, loops, subroutines, etc.
  - powerful arithmetic (even fast transcendent functions, differential operations)
  - **MRT** (“**Multiple Render Targets**”) – writing to more output buffers simultaneously (more buffers)
    - deferred shading, speedup of multi-pass algorithms, ..
- **computational performance** of **FS** (HW design)
  - many independent “pipelines” (2005/II: 16)

VS ↔ FS cooperation

- **VS obligation:** <u>3D vertex coordinates</u> in “clip space”
  - ⇒ for 3D primitive rasterizing
  - other output varying data are <u>optional</u> (texture coordinates, primary and secondary color, etc.)
  - if <u>FS is not used</u>, output data for FFP are mandatory !
- **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)

Shader languages

- **“low-level”** programming – assembler
  - mandatory in oldest profiles (Cg, HLSL object code)
  - simple instruction set (17 instructions in VS 1.1)
- **higher programming languages**
  - complex GPU architecture (optimization)
  - NVIDIA: **Cg** (“C for graphics”) 2002-2012
  - NVIDIA & Microsoft: **HLSL** (“High Level Shading Language”) since 2003, very similar to Cg
  - OpenGL ARB (orig. 3Dlabs): **GLSL** (“OpenGL Shading Language”) since 2001
  - similar syntax (especially Cg and HLSL)

Cg example (Phong shading)

```
void phongVertex ( float4 position   : POSITION,
                   float3 normal     : NORMAL,

                   out float4 oPosition : POSITION,
                   out float4 color     : COLOR,

                   uniform float4x4 modelViewProj,
                   uniform float3 globalAmbient,
                   uniform float3 lightColor,
                   uniform float3 lightPosition,
                   uniform float3 eyePosition,
                   uniform float3 Ka,
                   uniform float3 Kd,
                   uniform float3 Ks,
                   uniform float  shininess )
{
    // 3D variant of world space vertex position:
  float3 P = position.xyz;
    // 3D variant of world space normal vector:
  float3 N = normal;
    // light direction vector (world space, normalized):
  float3 L = normalize( lightPosition - P );
```

Cg example continued

```
    // max( cos(alpha), 0 ):
  float cosa = max( dot(N,L), 0 );
    // view vector (world space, normalized):
  float3 V = normalize( eyePosition - P );
    // Blinn's half vector (world space, normalized):
  float3 H = normalize( L + V );
    // cos(beta)^shininess:
  float cosb = pow( max( dot(N,H), 0 ), shininess ):
  if ( cosa <= 0 ) cosb = 0;

    // total ambient color:
  float3 ambient = Ka * globalAmbient;
    // total diffuse color:
  float3 diffuse = Kd * lightColor * cosa;
    // total specular color:
  float3 specular = Ks * lightColor * cosb;

    // output values: vertex position in clip space
  oPosition = mul( modelViewProj, position );
    // sum of all color components:
  color.xyz = ambient + diffuse + specular;
  color.w   = 1;
}
```

Shader-assembler example

```
!!ARBvp1.0
  # ARB_vertex_program generated by NVIDIA Cg compiler
PARAM c12 = { 0, 1, 0, 0 };
TEMP R0, R1, R2;
ATTRIB v18 = vertex.normal;
ATTRIB v16 = vertex.position;
PARAM c0[4] = { program.local[0..3] };
PARAM c10 = program.local[10];
PARAM c5 = program.local[5];
PARAM c9 = program.local[9];
PARAM c4 = program.local[4];
PARAM c8 = program.local[8];
PARAM c11 = program.local[11];
PARAM c7 = program.local[7];
PARAM c6 = program.local[6];
  DP4 result.position.x, c0[0], v16;
  DP4 result.position.y, c0[1], v16;
  DP4 result.position.z, c0[2], v16;
  DP4 result.position.w, c0[3], v16;
  ADD R2.xyz, c7.xyzx, -v16.xyzx;
  DP3 R0.x, R2.xyzx, R2.xyzx;
  RSQ R1.w, R0.x;
  ADD R0.yzw, c6.xxyz, -v16.xxyz;
  DP3 R0.x, R0.yzwy, R0.yzwy;
  RSQ R0.x, R0.x;
  MUL R1.xyz, R0.x, R0.yzwy;
...
```

Shaders in OpenGL

- **low-level**: assembler
  - OpenGL extensions
  - different profiles, different approaches (ATI, NVIDIA, 3Dlabs)
- **GLSL language** (2004-)
  - originally 3Dlabs, official since **OpenGL 2.0**
- universal higher language **Cg** (2002-2012)
  - NVIDIA, almost identical to Microsoft's HLSL
  - **"Cg runtime"** needed ("cg.dll" + "cgGL.dll"), two-pass compile system

GLSL example (vertex shader)

```glsl
#version 130
in vec4 position;
in vec3 normal;
in vec2 texCoords;
in vec3 color;
out vec2 varTexCoords;
out vec3 varNormal;
out vec3 varWorld;
out vec3 varColor;
flat out vec3 flatColor;
uniform mat4 matrixModelView;
uniform mat4 matrixProjection;

void main ()
{
  gl_Position  = matrixProjection * matrixModelView * position;
  // propagated quantities:
  varTexCoords = texCoords;
  varNormal    = normal;
  varWorld     = position.xyz;
  varColor     = flatColor = color;
}
```

GLSL example (Phong shader [+txt])

```
#version 130

in vec2 varTexCoords;           // [ s, t ]
in vec3 varNormal;              // world coord. system
in vec3 varWorld;
in vec3 varColor;               // Gouraud color
flat in vec3 flatColor;

uniform bool useShading;
uniform vec3 globalAmbient;
uniform vec3 lightColor;
uniform vec3 lightPosition; // world coord. system
uniform vec3 eyePosition;   // world coord. system
uniform vec3 Ks;
uniform float shininess;

uniform bool useTexture;
uniform sampler2D texSurface;

out vec3 fragColor;             // output = fragment color
```

…

GLSL example continued

```
void main ()
{
  if ( useShading )
  {
    vec3 P = varWorld;
    vec3 N = normalize( varNormal );
    vec3 L = normalize( lightPosition - P );
    vec3 V = normalize( eyePosition - P );
    vec3 H = normalize( L + V );

    float cosb = 0.0;
    float cosa = dot( N, L );
    if ( cosa > 0.0 )
      cosb = pow( max( dot( N, H ), 0.0 ), shininess );

    vec3 kakd;
    if ( useTexture )
      kakd = vec3( texture2D( texSurface, varTexCoords ) );
    else
      kakd = varColor;

    fragColor = kakd * globalAmbient +
                kakd * lightColor * cosa +
                Ks   * lightColor * cosb;
  }
  else
    fragColor = flatColor;
}
```

Sources

- Tomas Akenine-Möller, Eric Haines: ***Real-time rendering, 2nd edition***, A K Peters, 2002, ISBN: 1568811829
- Randima Fernando, Mark J. Kilgard: ***The Cg Tutorial***, Addison-Wesley, 2003, ISBN: 0321194969
- OpenGL ARB: ***OpenGL Programming Guide, 4th edition***, Addison-Wesley, 2004, ISBN: 0321173481
- Randi J. Rost: ***OpenGL Shading Language***, Addison-Wesley, 2004, ISBN: 0321197895
- **http://developer.nvidia.com/**
- **http://www.shadertech.com/**