Lecture 1

Introduction to Computer Graphics and GPU Programming

Hanspeter Pfister
pfister@merl.com
Eric Chan
chan@merl.com

Offline Rendering 5 Years Ago

Shrek, PDI Dreamworks

Interactive 5 Years Ago

Quake 3, id software

Modern Offline Rendering

Cars, Pixar

Modern Interactive Rendering

Project Gotham Racing

Modern Offline Rendering

Starship Troopers 2, Tippett Studio

Modern Interactive Rendering

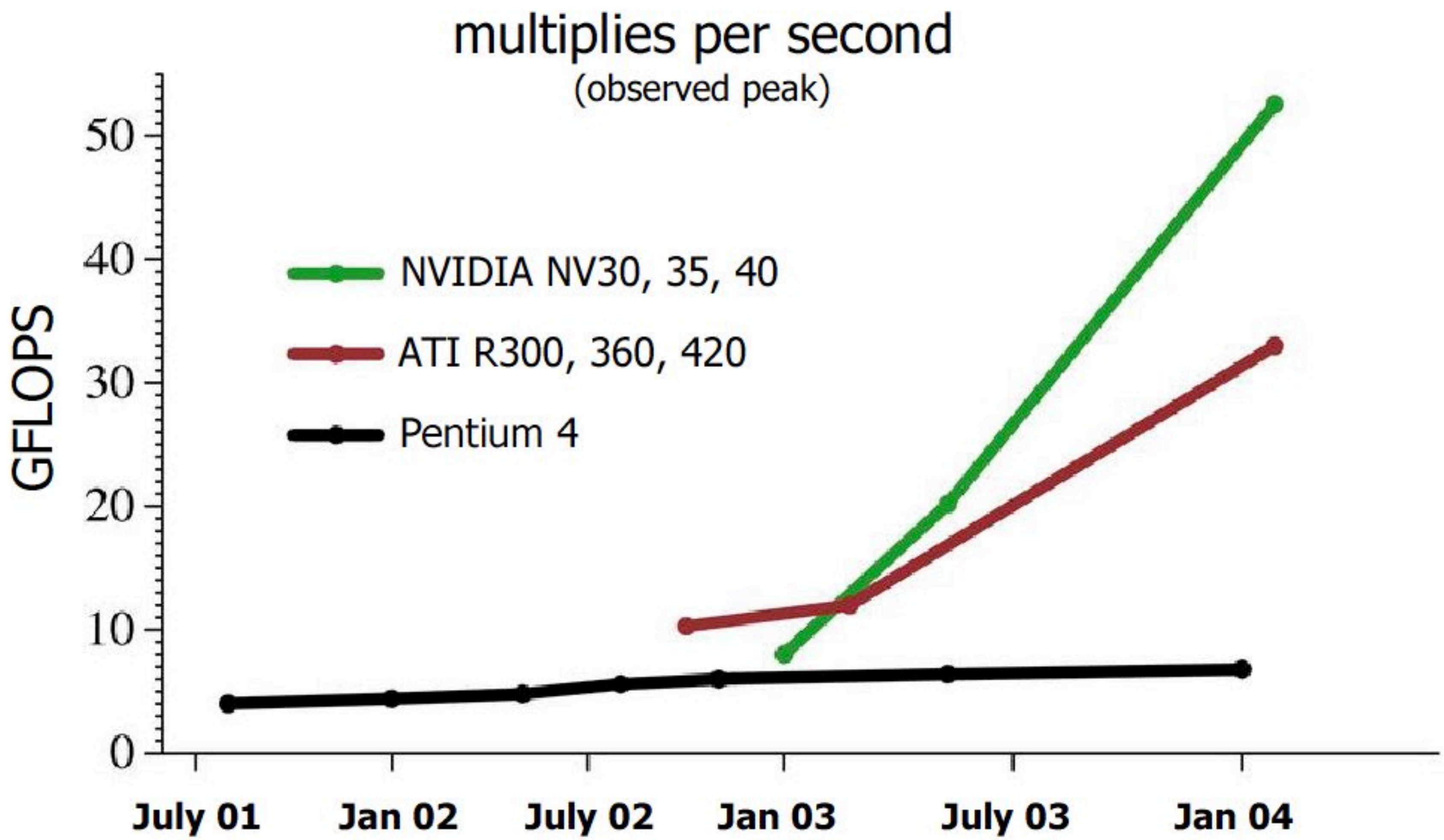
I-8, Insomniac Games

What happened in the past 5 years?

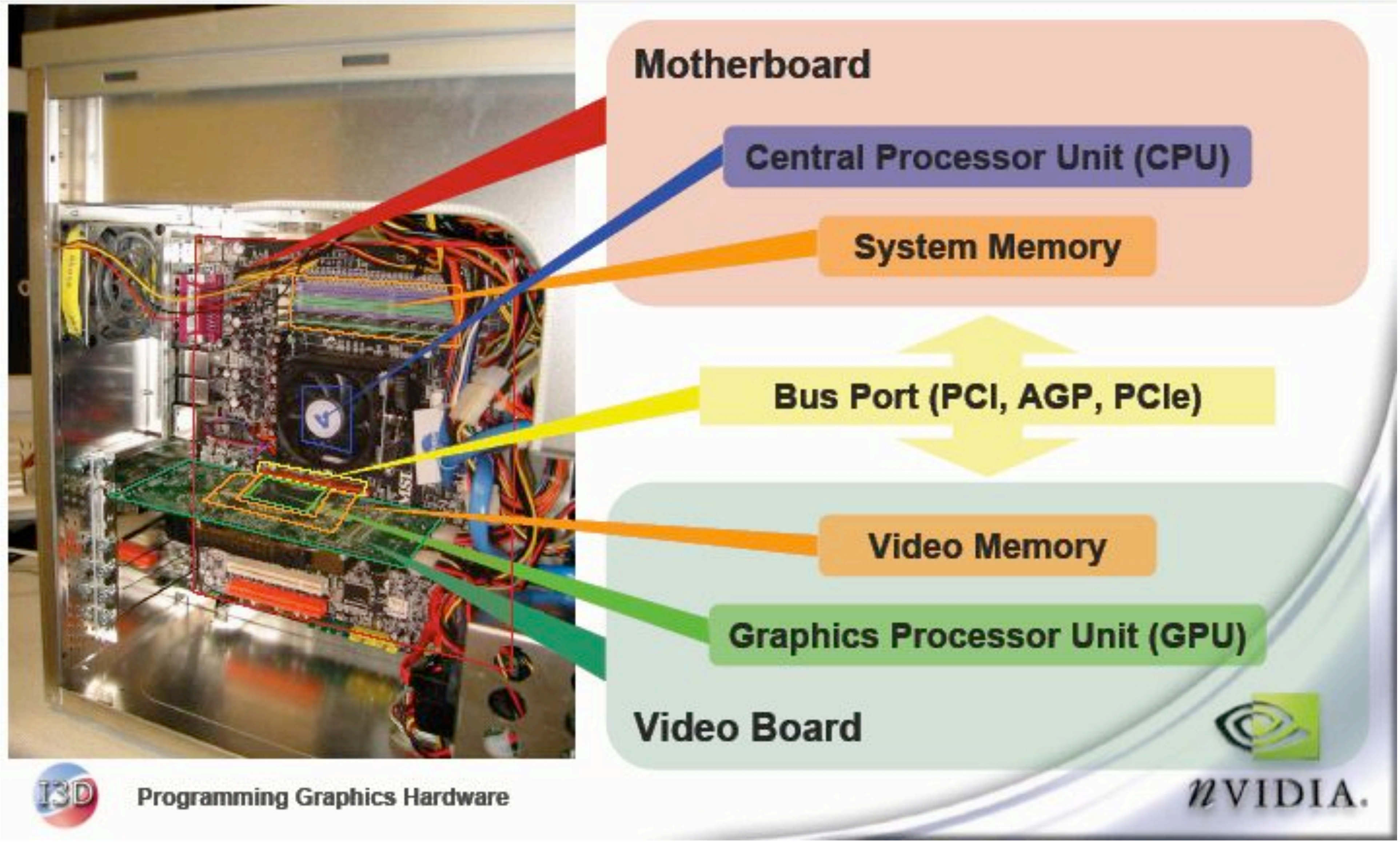
- Interactive is delivering near-offline quality 1,000,000x faster
- GPUs have taken advantage of semiconductor trends to deliver performance
- GPU strengths/weaknesses have sparked innovation in algorithms and software

Matt Pharr, Neoptica

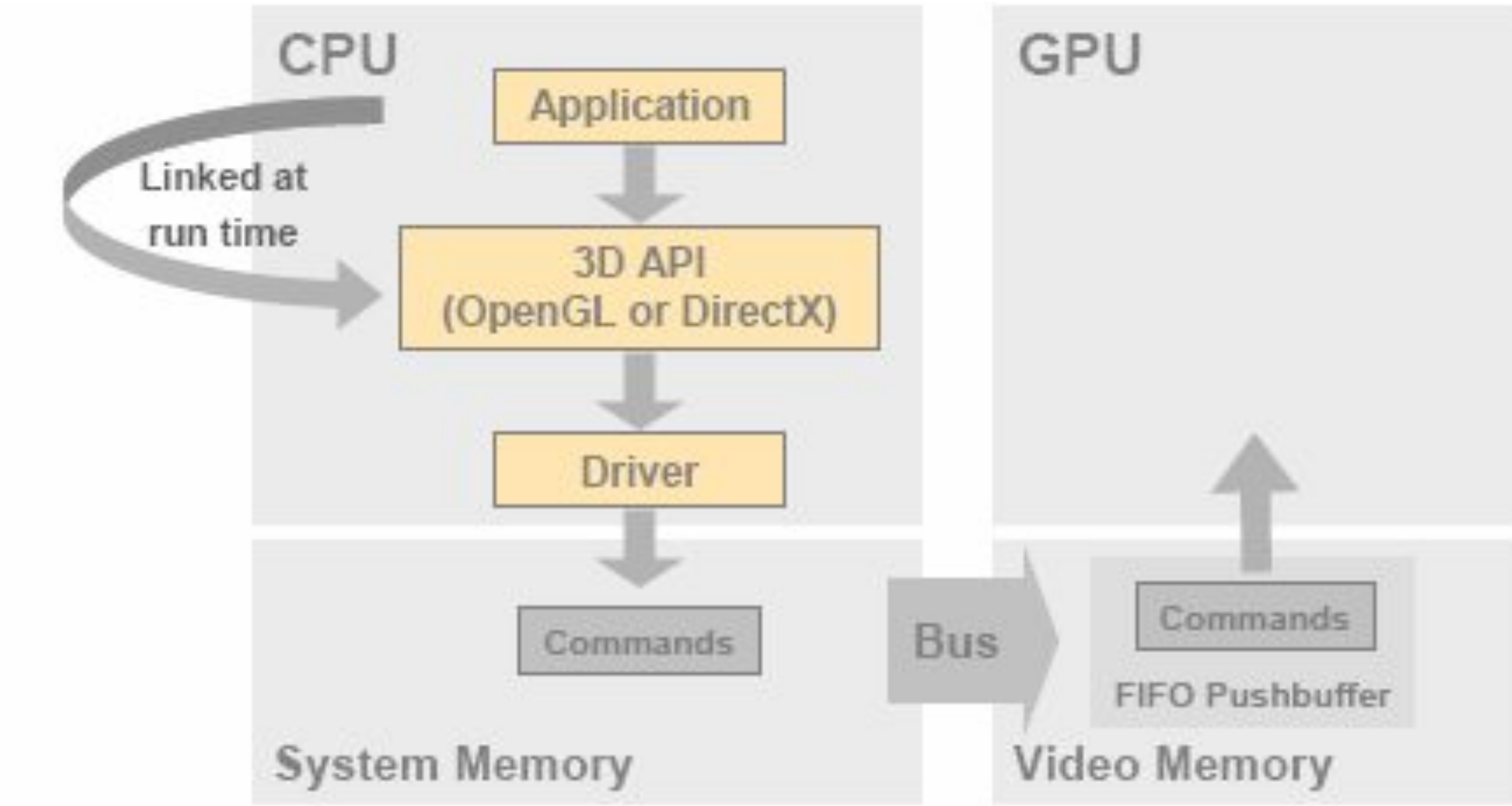
Recent Trends



PC Architecture

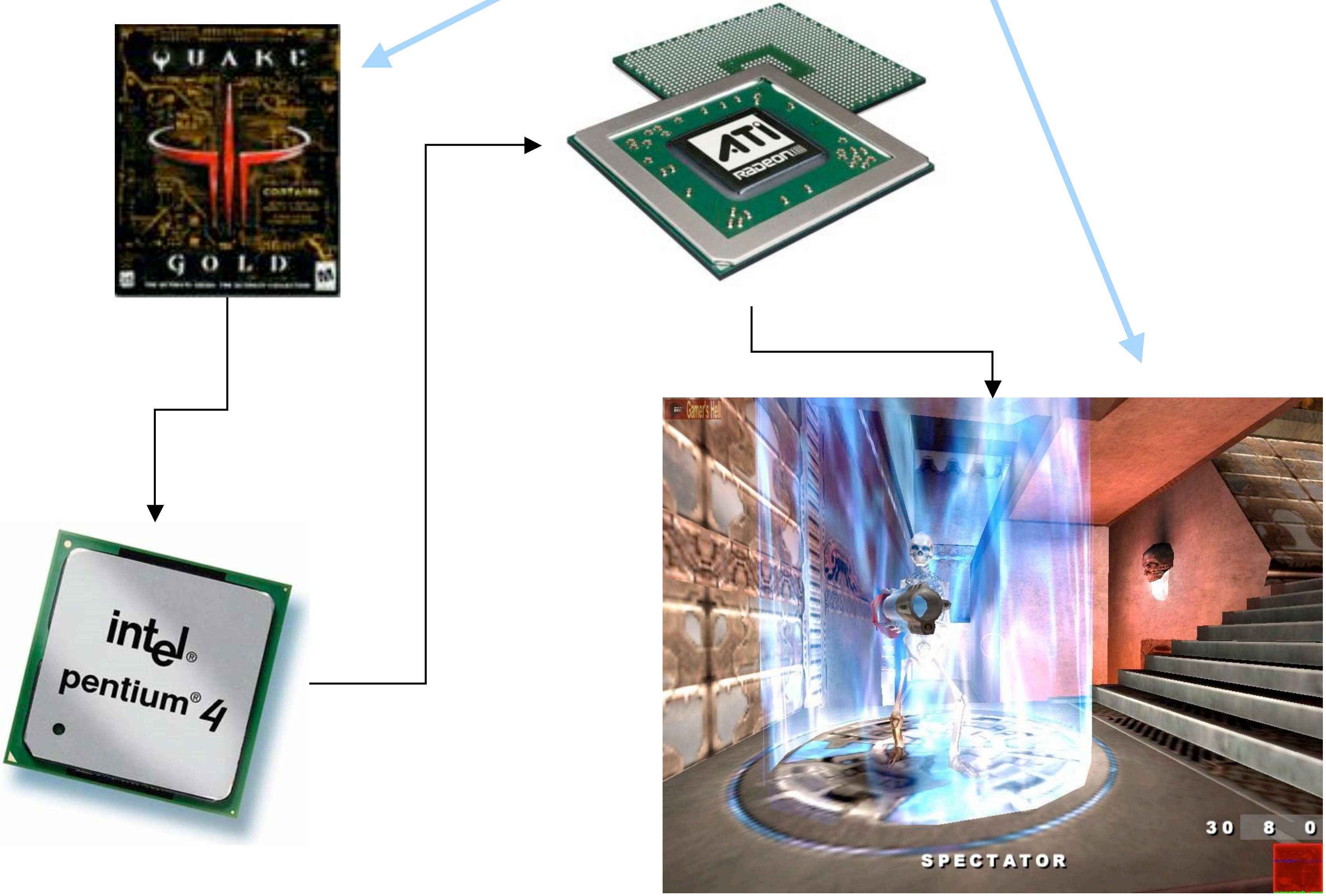


GPU Programming

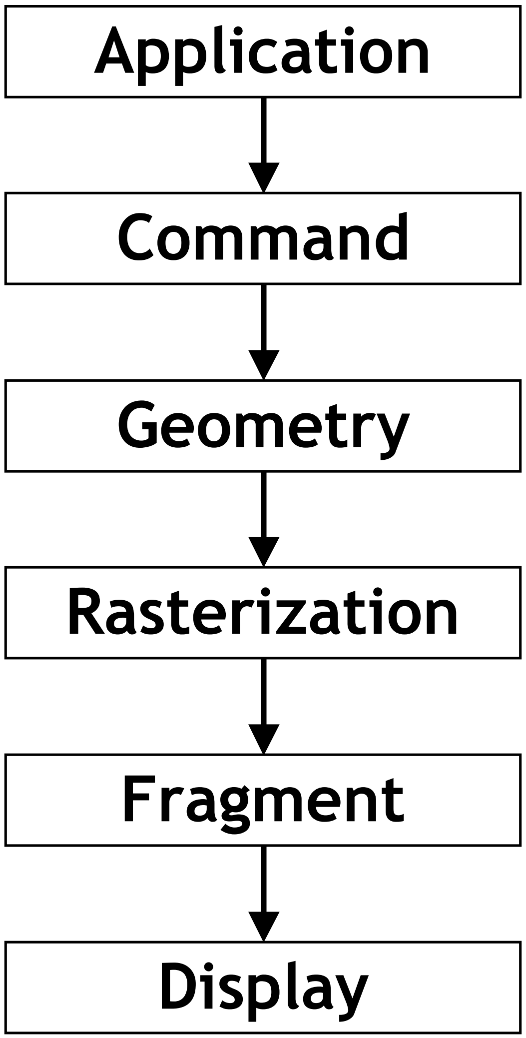


Nvidia, EG 2004

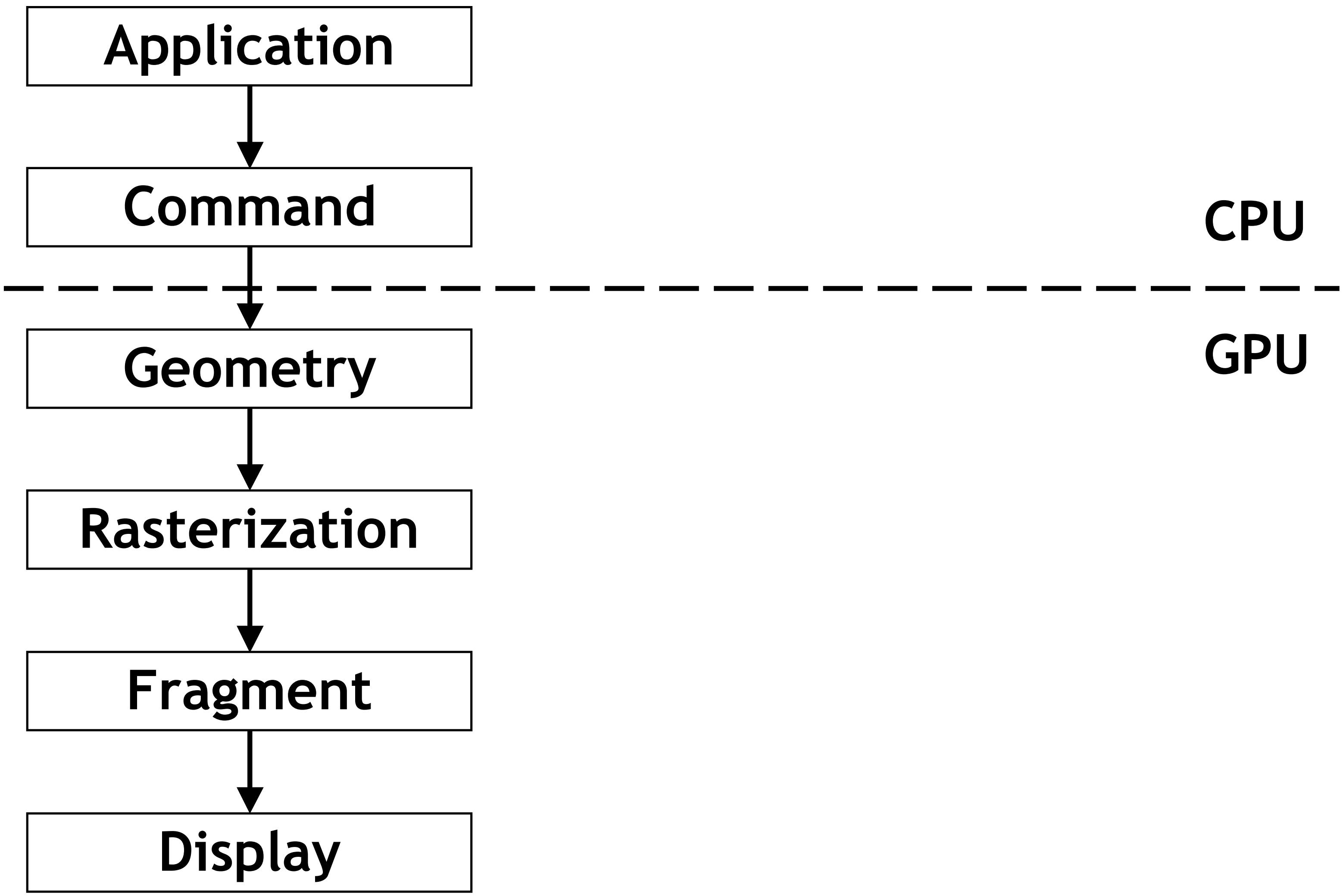
How to get from here to here?



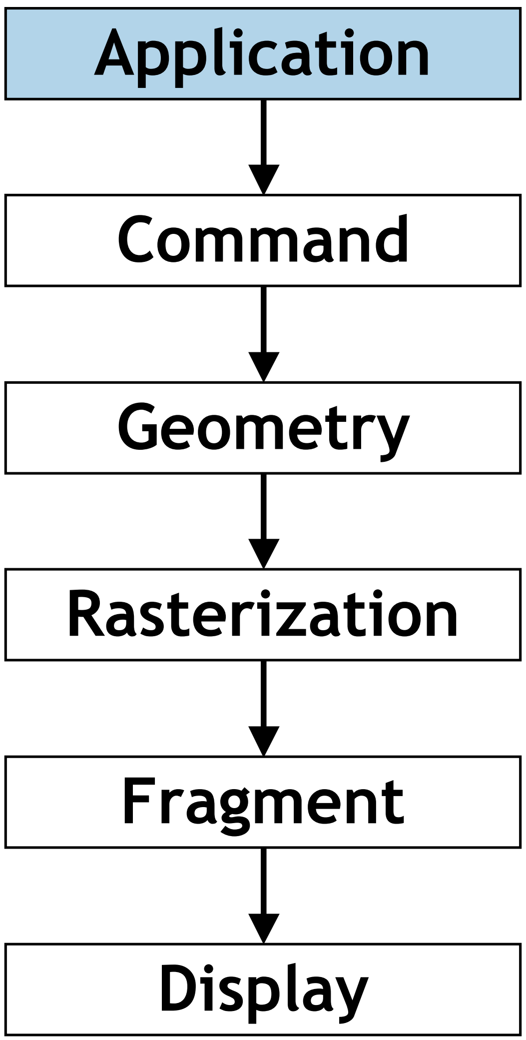
The Graphics Pipeline



The Graphics Pipeline

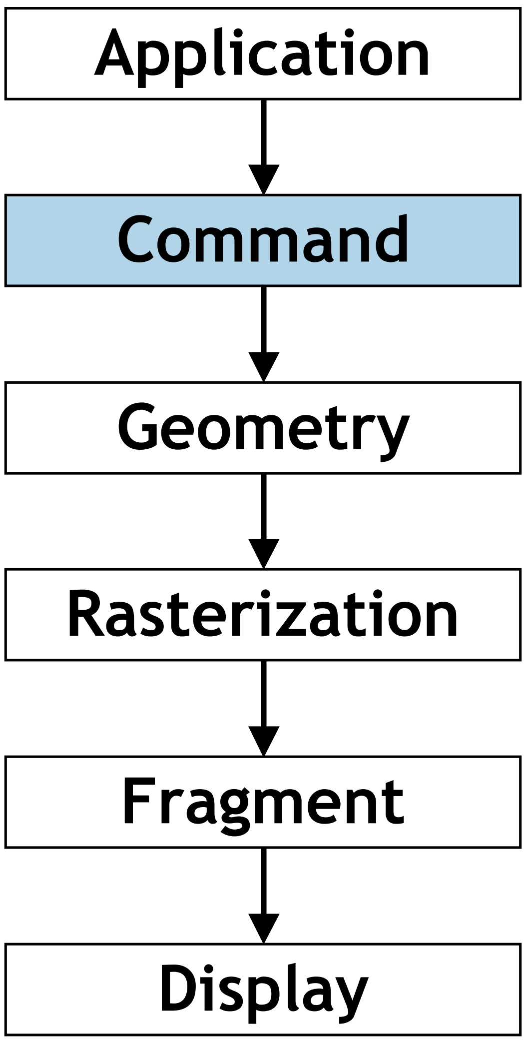


The Graphics Pipeline



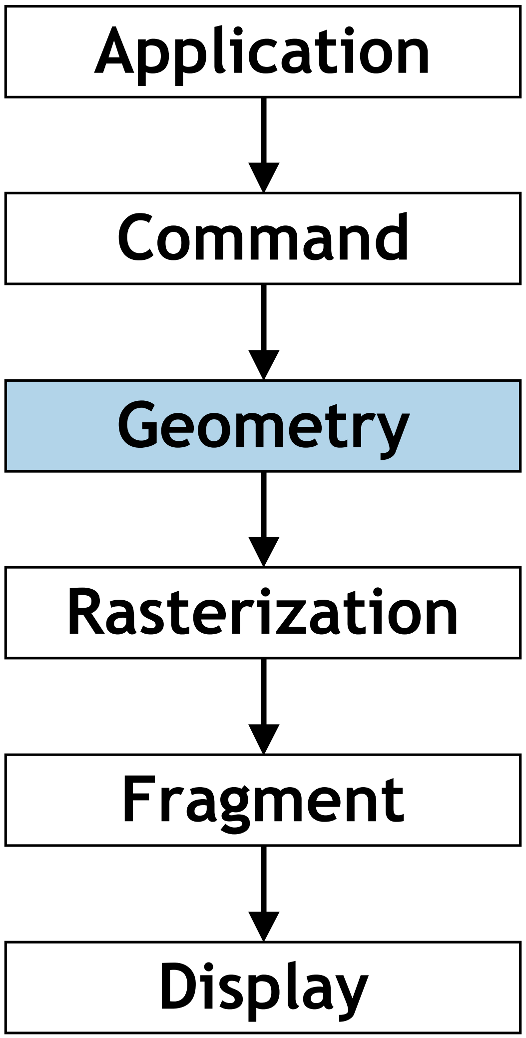
- Quake 3:
- define game behavior
- networking
- user input events
- sound processing
- game AI
- game physics

The Graphics Pipeline



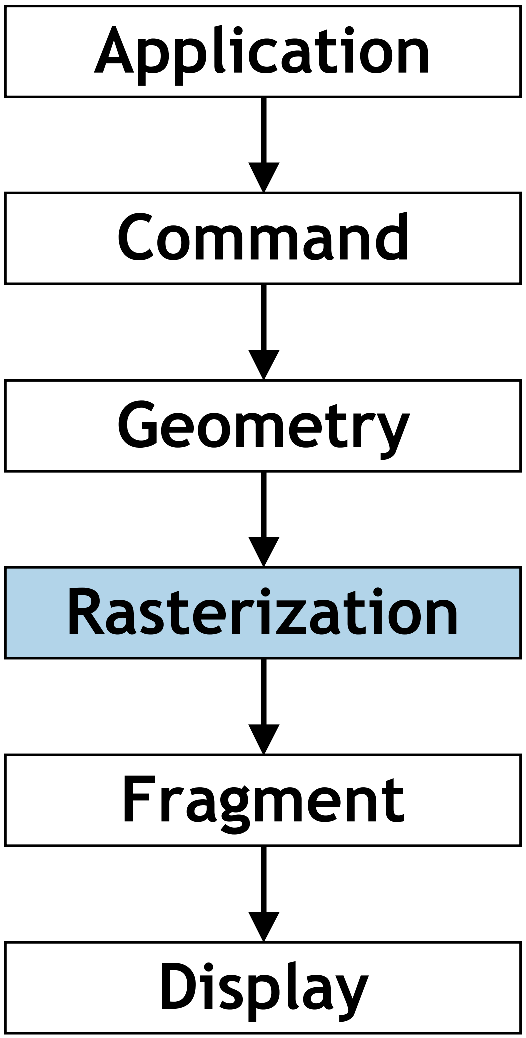
- Quake 3:
- send OpenGL commands
- OpenGL driver:
- process GL command stream
- talk to GPU

The Graphics Pipeline



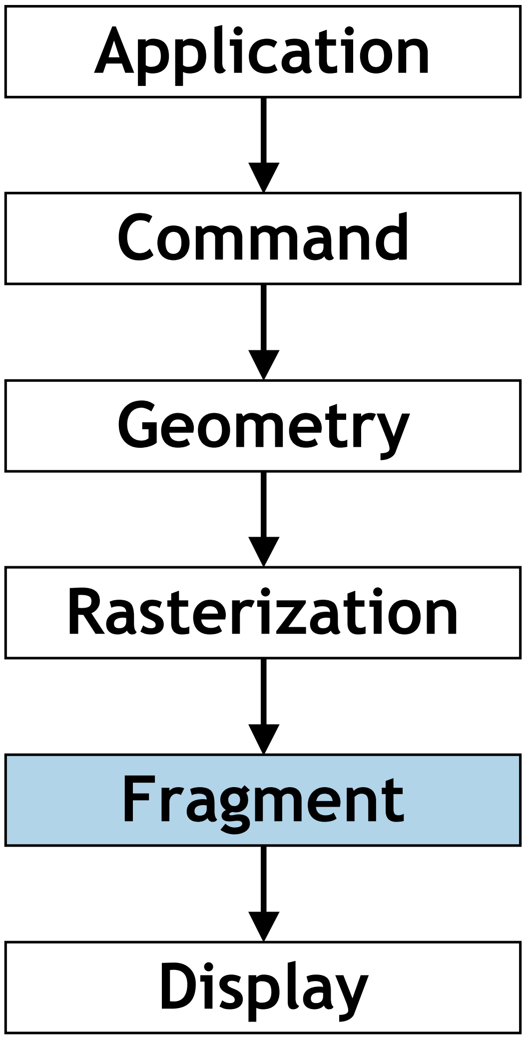
- GPU:
- vertex transformations
- vertex lighting
- clipping
- primitive assembly

The Graphics Pipeline



- GPU:
- convert triangles to fragments
- tex coordinate interpolation
- color interpolation

The Graphics Pipeline



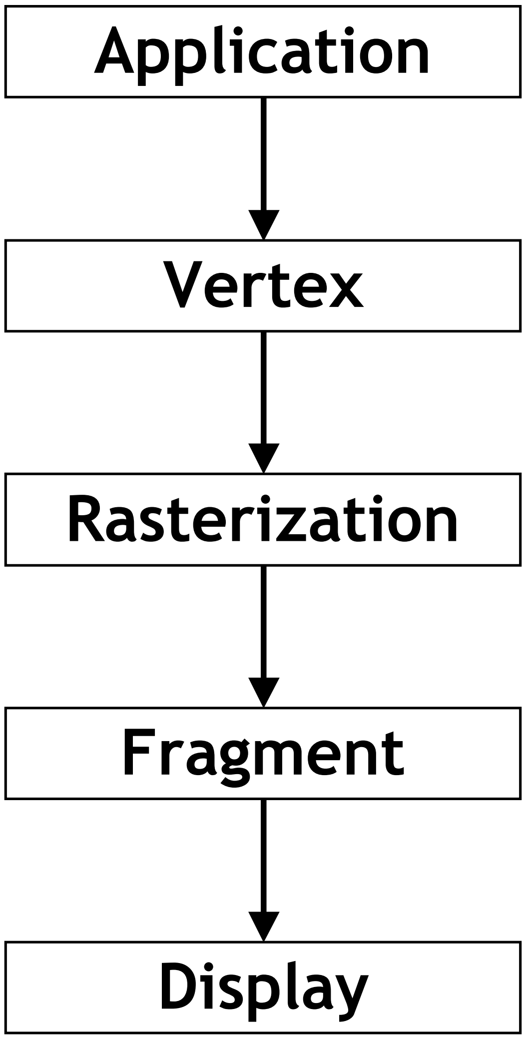
- GPU:
- texturing
- depth test
- color blending

The Graphics Pipeline

Application
↓
Command
↓
Geometry
↓
Rasterization
↓
Fragment
↓
Display

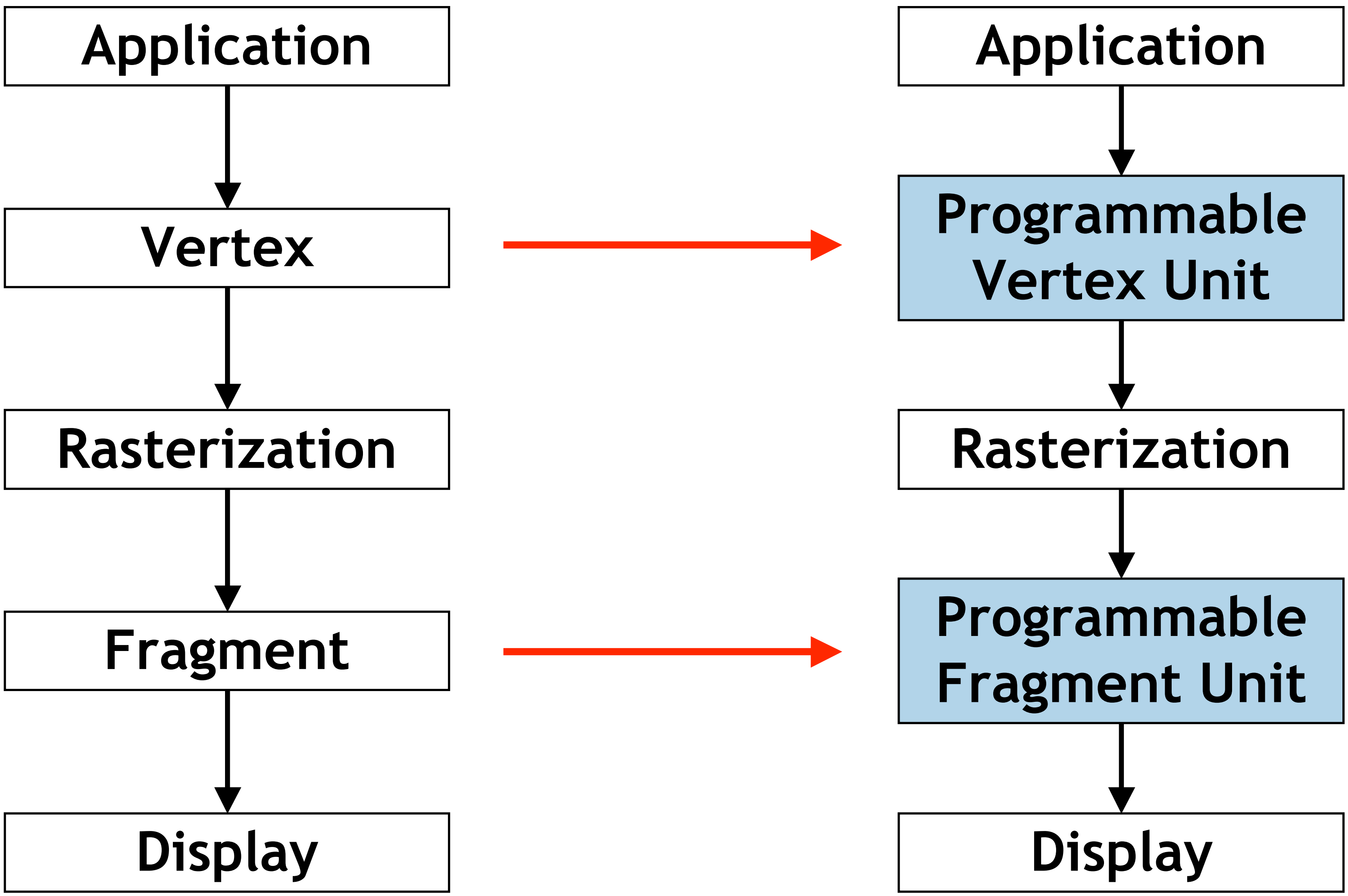
id software

Upgrading the pipeline

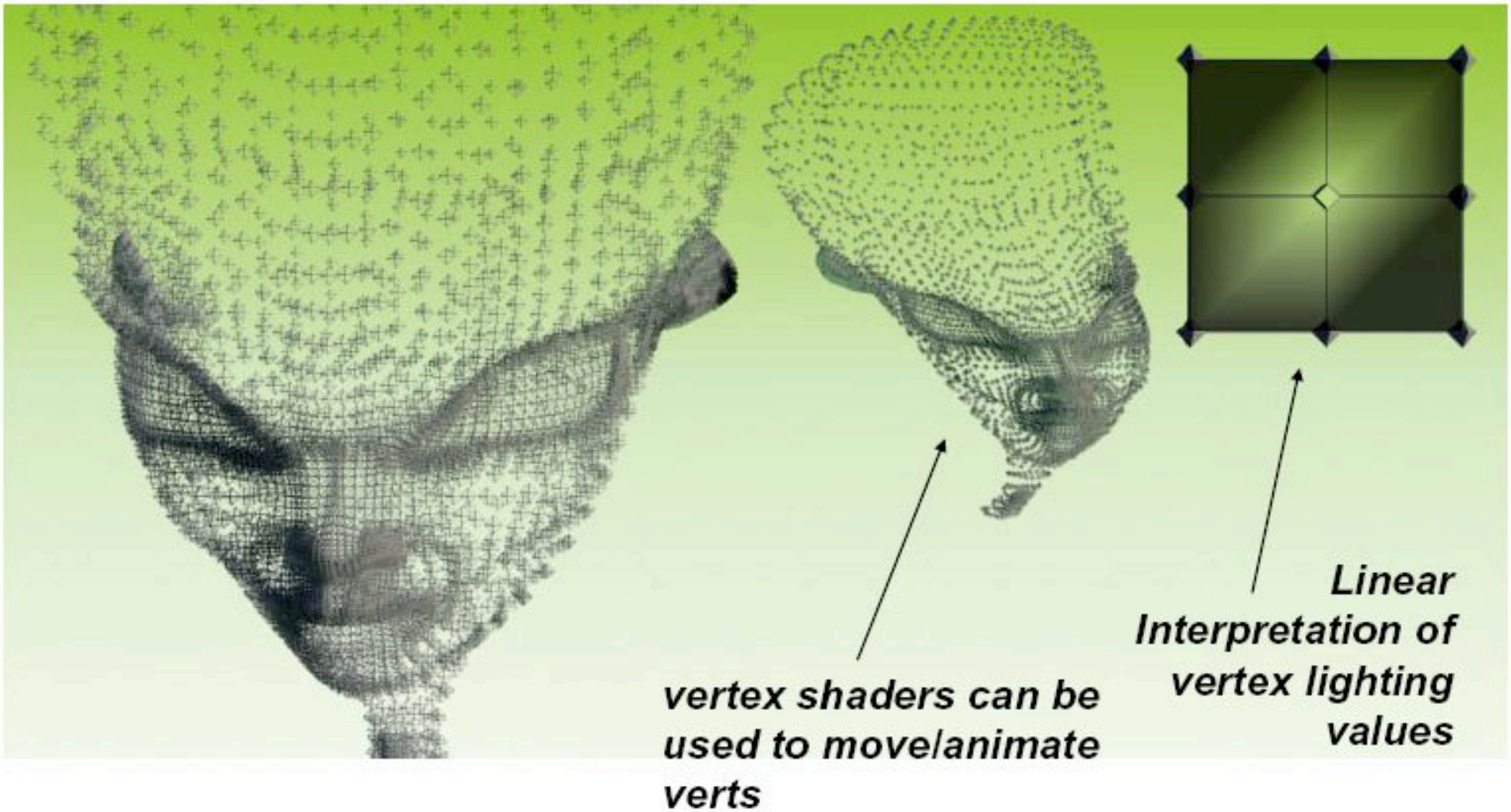


- Traditional pipeline:
- fixed-function
- configurable using GL states
- limited features

Programmable Pipeline

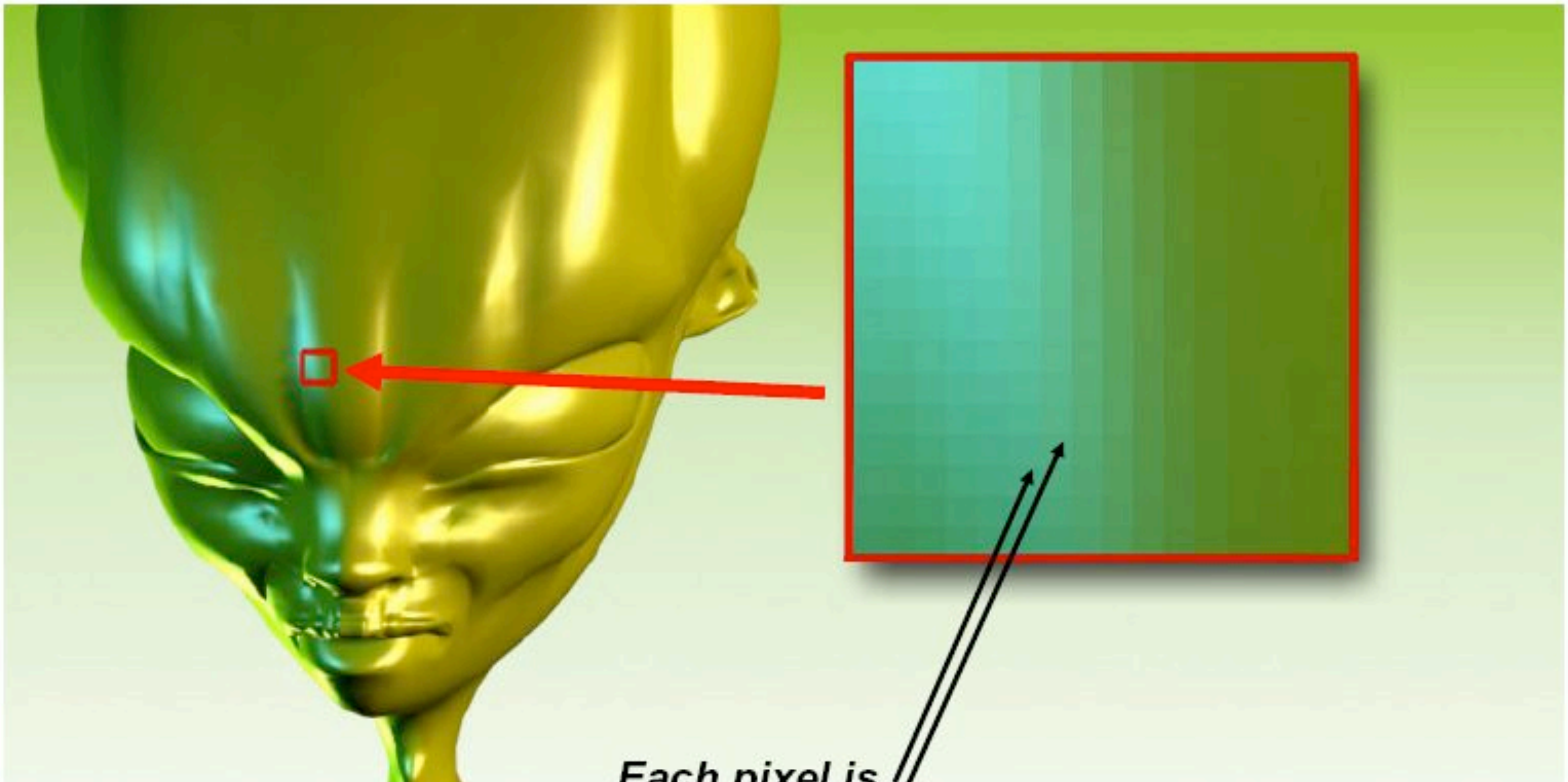


Vertex Shaders



Vertex Shaders are both flexible and quick

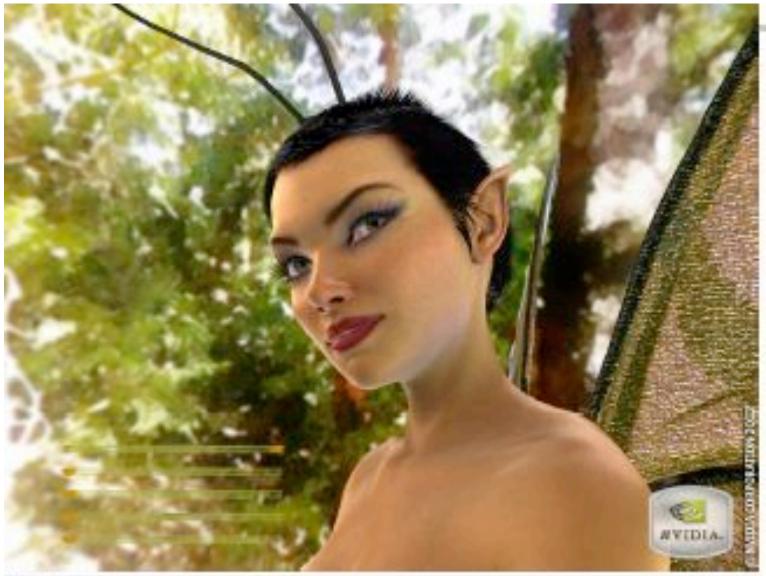
Pixel Shaders

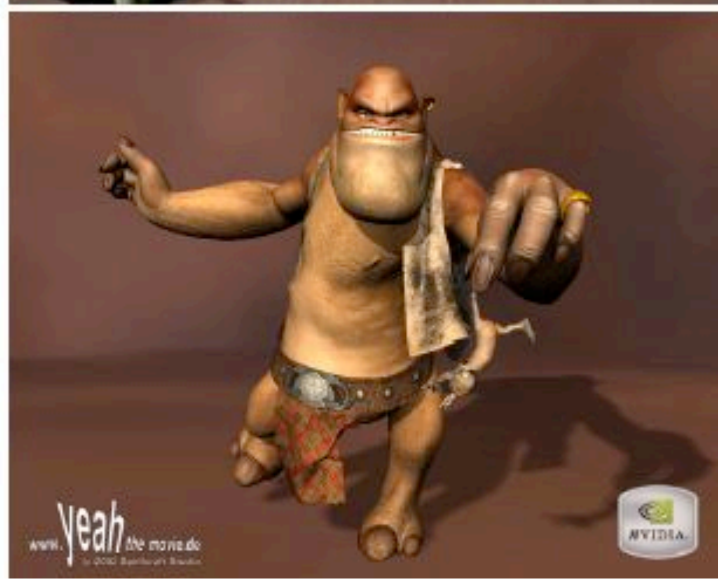


Each pixel is calculated individually

Pixel shaders have limited or no knowledge of neighbouring pixels

Results





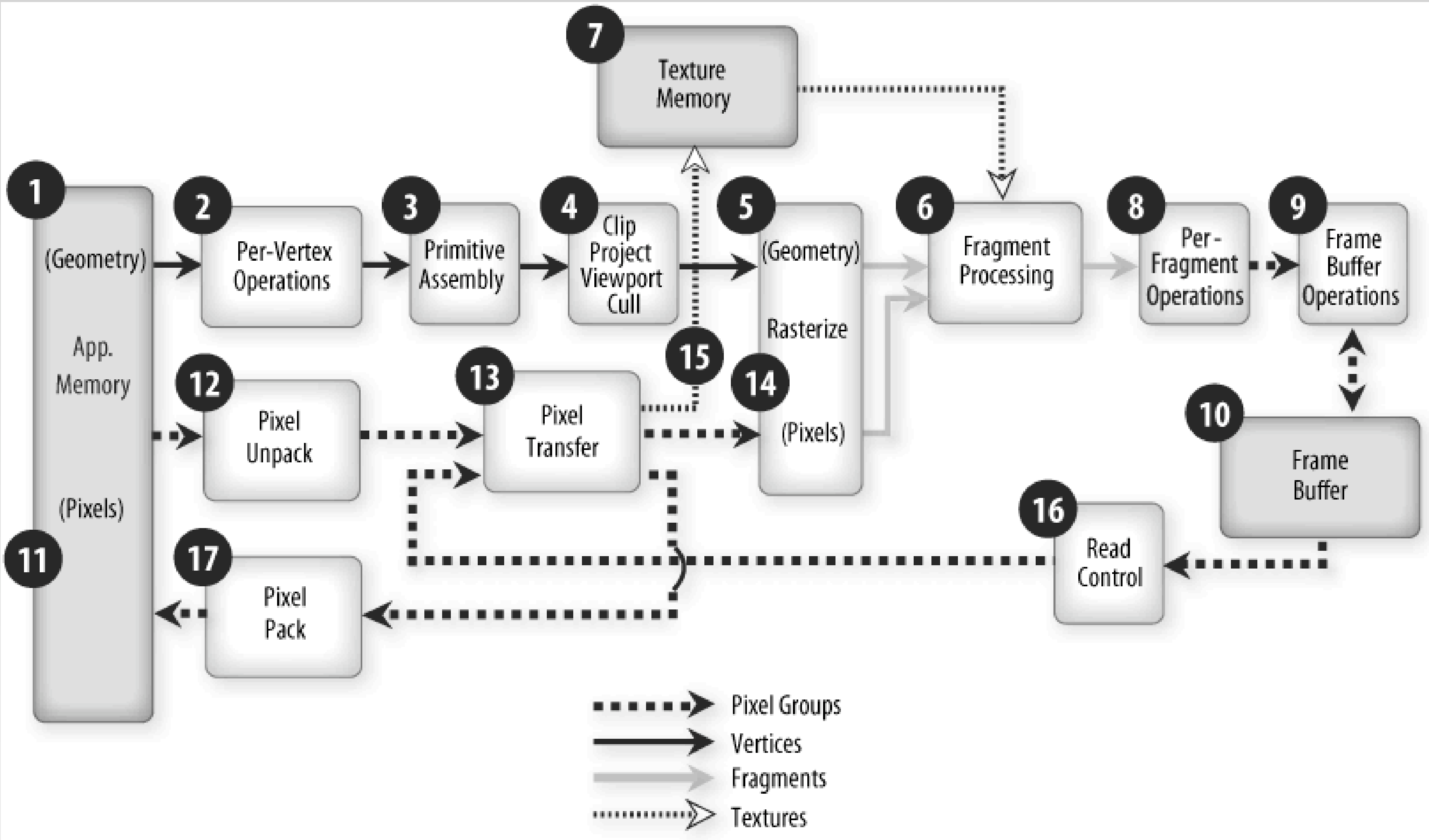
GLSL Programming

Technical Game Development II

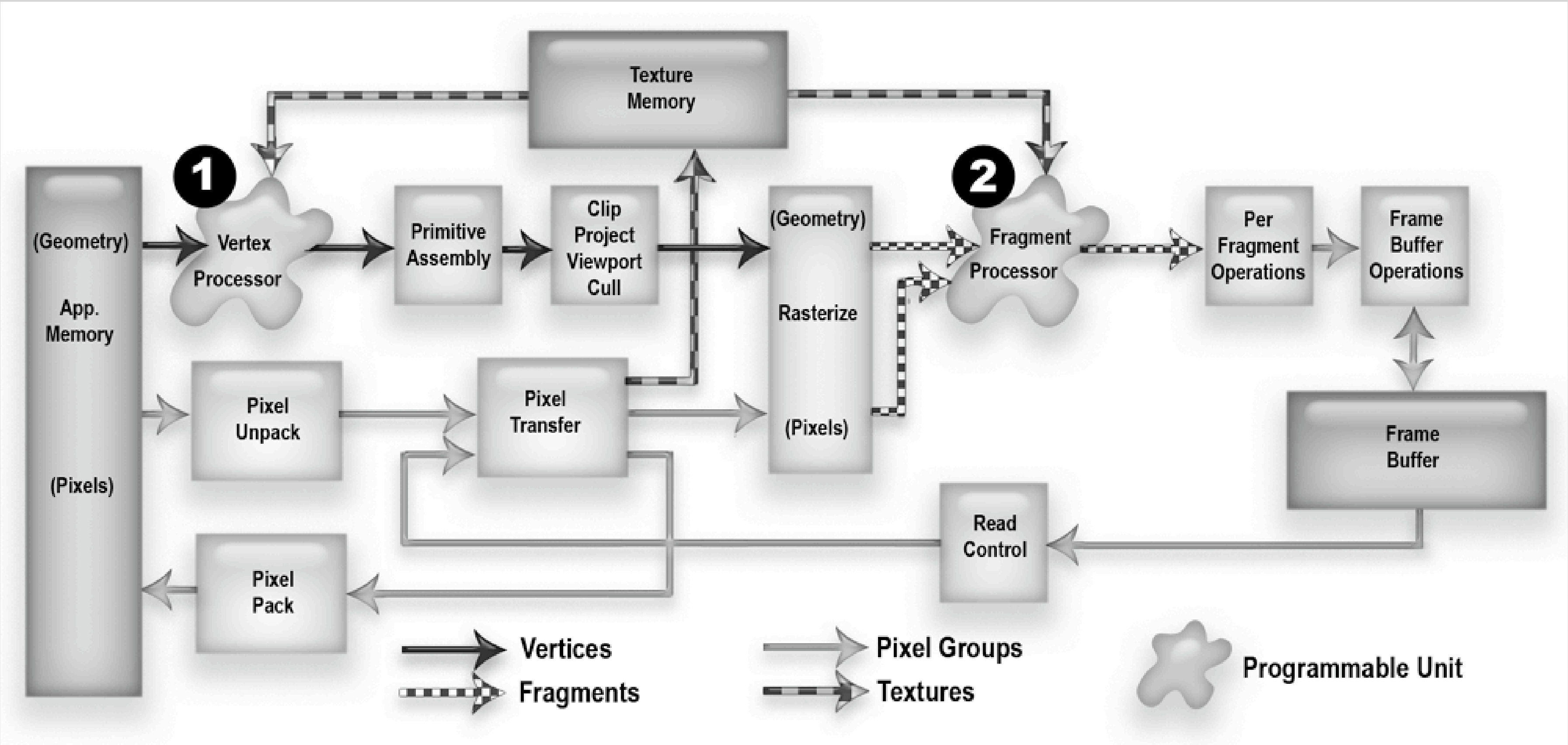
Professor Charles Rich
Computer Science Department
rich@wpi.edu

Reference: Rost, OpenGL Shading Language, 2nd Ed., AW, 2006
"The Orange Book"

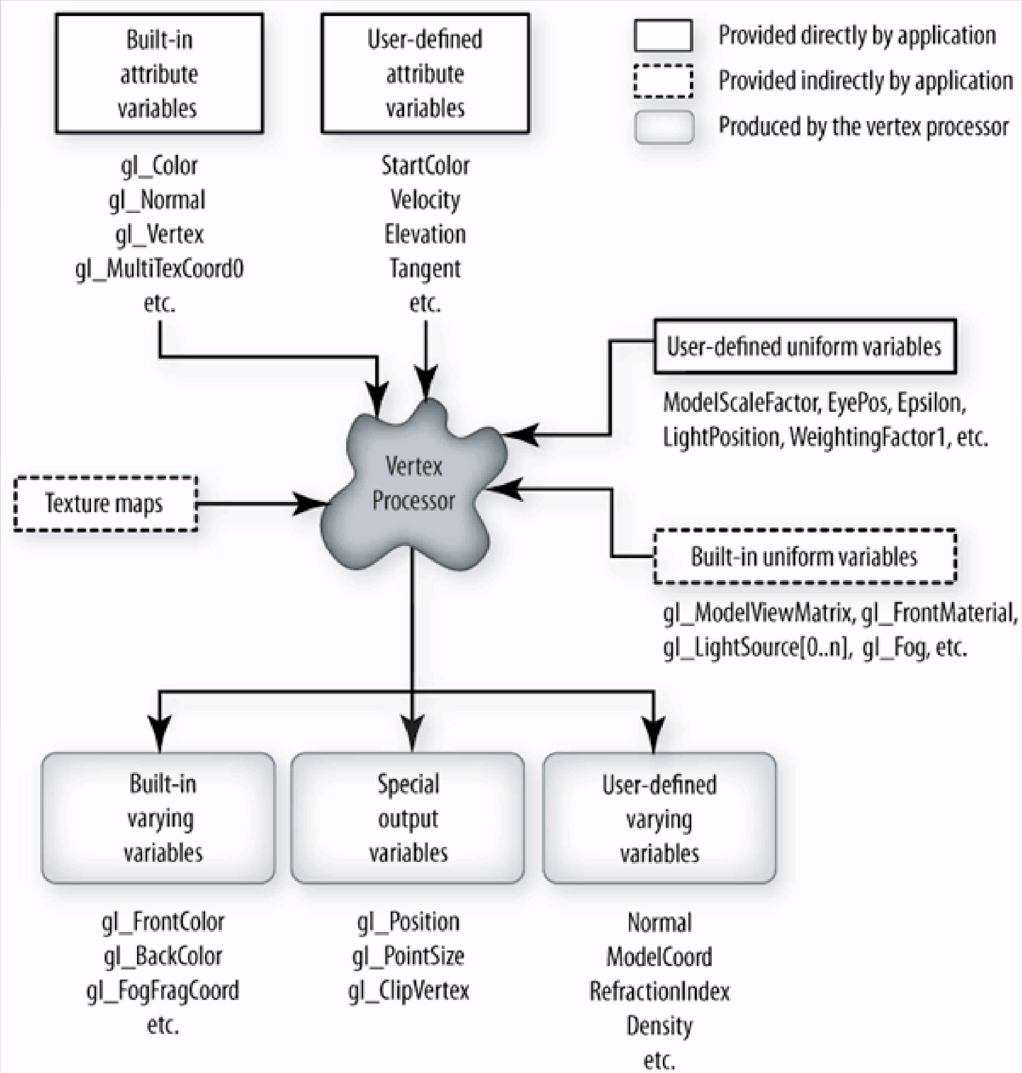
The OpenGL Pipeline



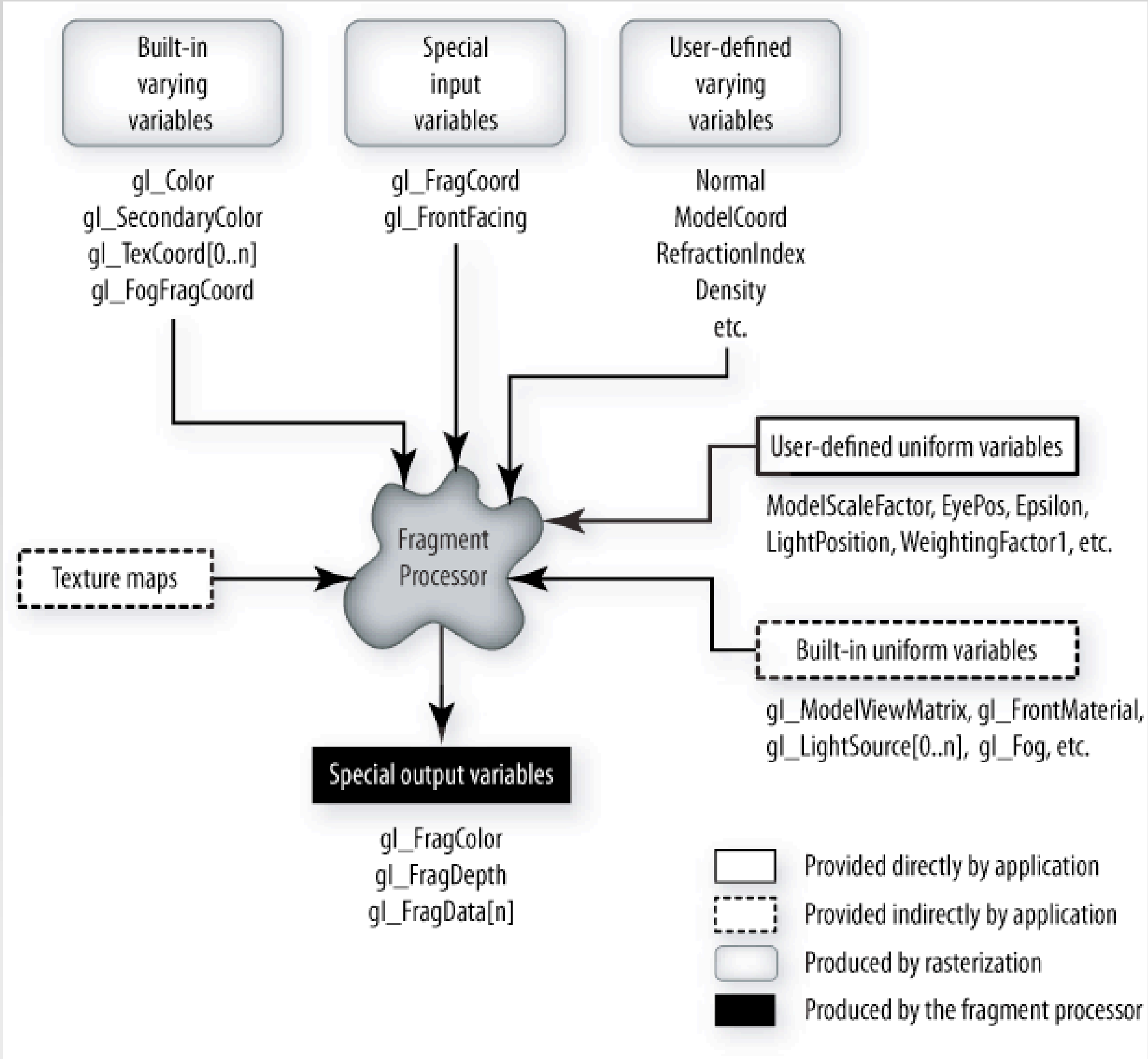
OpenGL Programmable Processors



Vertex Processor



Fragment (Pixel) Processor



GLSL Language

- Similar to C, C++
- Builtin vector and matrix operations:
  - vec2, vec3, vec4
  - mat2, mat3, mat4
- Texture lookup
  - sampler1D, sampler2D, sampler3D

Simple Shader Program Example

- Surface temperature coloring
  - Assume temperature is known at each vertex in model
  - smoothly color surface to indicate temperature at every point (using interpolation

Vertex Shader

```
// parameters read from application (per primitive)
uniform float CoolestTemp;
uniform float TempRange;

// incoming property of this vertex
attribute float VertexTemp;

// to communicate to the fragment shader
varying float Temperature;

void main()
{
    // communicate this vertex's temperature scaled to [0.0, 1.0]
    Temperature = (VertexTemp - CoolestTemp) / TempRange;

    // don't move this vertex
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

Fragment Shader

```
// parameters read from application (per primitive)
uniform vec3 CoolestColor;
uniform vec3 HottestColor;

// interpolated value from vertex shader
varying float Temperature;

void main()
{
    // compute a color using built-in mix() function
    vec3 color = mix(CoolestColor, HottestColor, Temperature);

    // set this pixel's color (with alpha blend of 1.0)
    gl_FragColor = vec4(color, 1.0);
}
```

Shader Execution

- Vertex shader is run once per vertex
- Fragment shader is run once per pixel
- Many such executions can happen *in parallel*
- No communication or ordering between executions
  - no vertex-to-vertex
  - no pixel-to-pixel

Moving Vertices in Shader

```
uniform vec3  LightPosition;
uniform vec3  SurfaceColor;
uniform vec3  Offset;
uniform float ScaleIn;
uniform float ScaleOut;
varying vec4  Color;  // color calculation for pixel shader

void main()
{
    vec3 normal = gl_Normal;
    vec3 vertex = gl_Vertex.xyz +
                  noise3(Offset + gl_Vertex.xyz * ScaleIn) * ScaleOut;

    normal = normalize(gl_NormalMatrix * normal);
    vec3 position = vec3(gl_ModelViewMatrix * vec4(vertex,1.0));
    vec3 lightVec = normalize(LightPosition - position);
    float diffuse = max(dot(lightVec, normal), 0.0);

    if (diffuse < 0.125) diffuse = 0.125;

    Color = vec4(SurfaceColor * diffuse, 1.0);
    gl_Position = gl_ModelViewProjectionMatrix * vec4(vertex,1.0);
}
```

Trivial Fragment Shader

```
varying vec4 Color;

void main()
{
    gl_FragColor = Color;
}
```

Loading Shaders in jME

```
Node model = (Node) new BinaryImporter().load(
             getClass().getResource("King_Black.model"));
rootNode.attachChild(model);

GLSLShaderObjectsState shader =
    display.getRenderer().createGLSLShaderObjectsState();

shader.load(getClass().getResource("noise.vert"),
            getClass().getResource("noise.frag"));

shader.setUniform("LightPosition", 0.0f, 0.0f, 4.0f);
shader.setUniform("SurfaceColor", 1.0f, 1.0f, 1.0f);
shader.setUniform("Offset", 0.85f, 0.86f, 0.84f);
shader.setUniform("ScaleIn", 1.0f);
shader.setUniform("ScaleOut", 1.0f);
shader.setEnabled(true);

model.setRenderState(shader);
```

Lots More You Can Do With Shaders

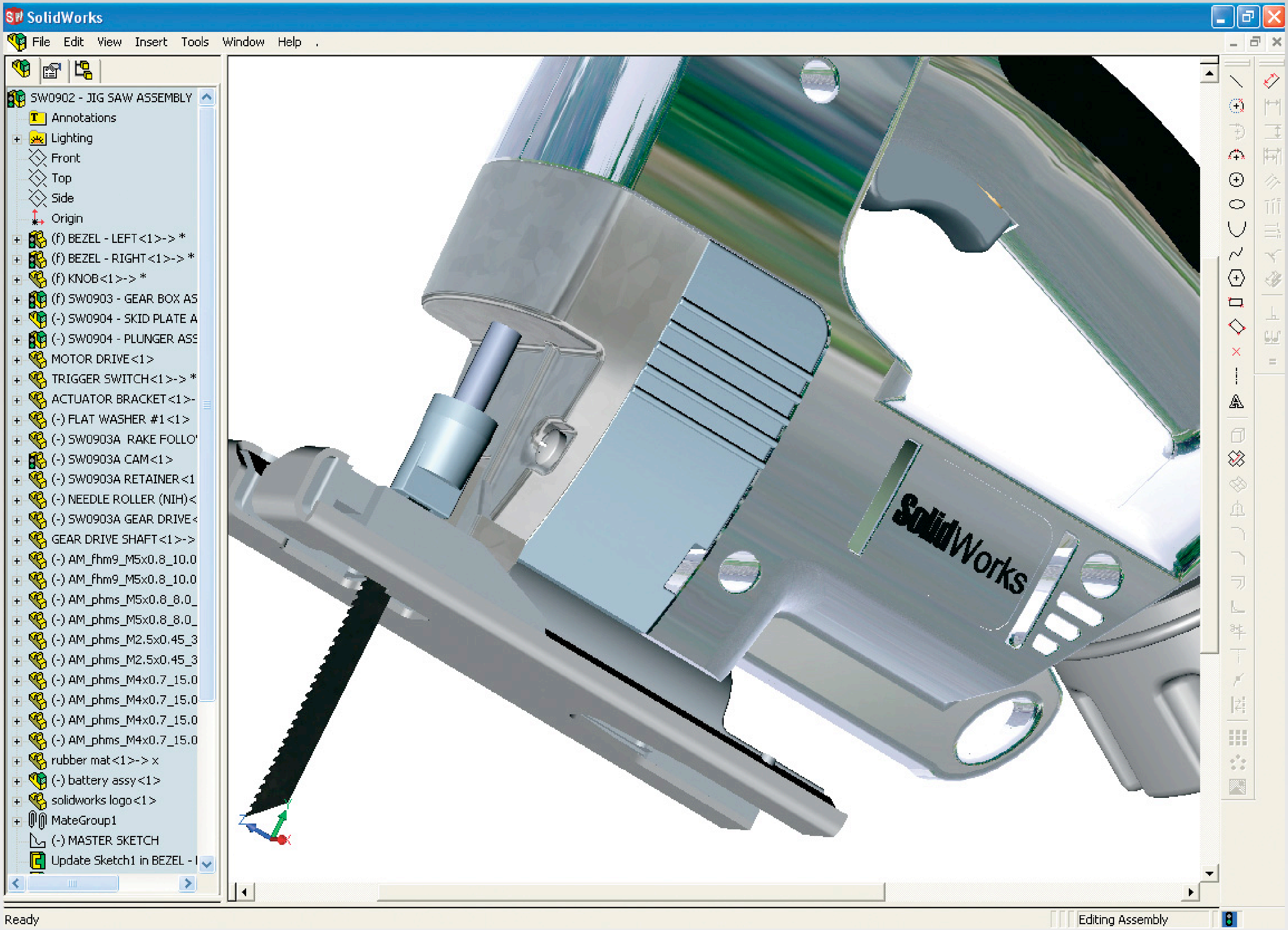
- Procedural Textures
  - patterns (stripes, etc.)
  - bump mapping
- Lighting Effects
- Shadows
- Surface Effects
  - refraction, diffraction
- Animation
  - morphing
  - particles

Lots More ...

- Anti-aliasing
- Non-photorealistic effects
  - hatching, meshes
  - technical illustration
- Imaging
  - sharpen, smooth, etc.
- Environmental effects (RealWorldz)
  - terrain
  - sky
  - ocean

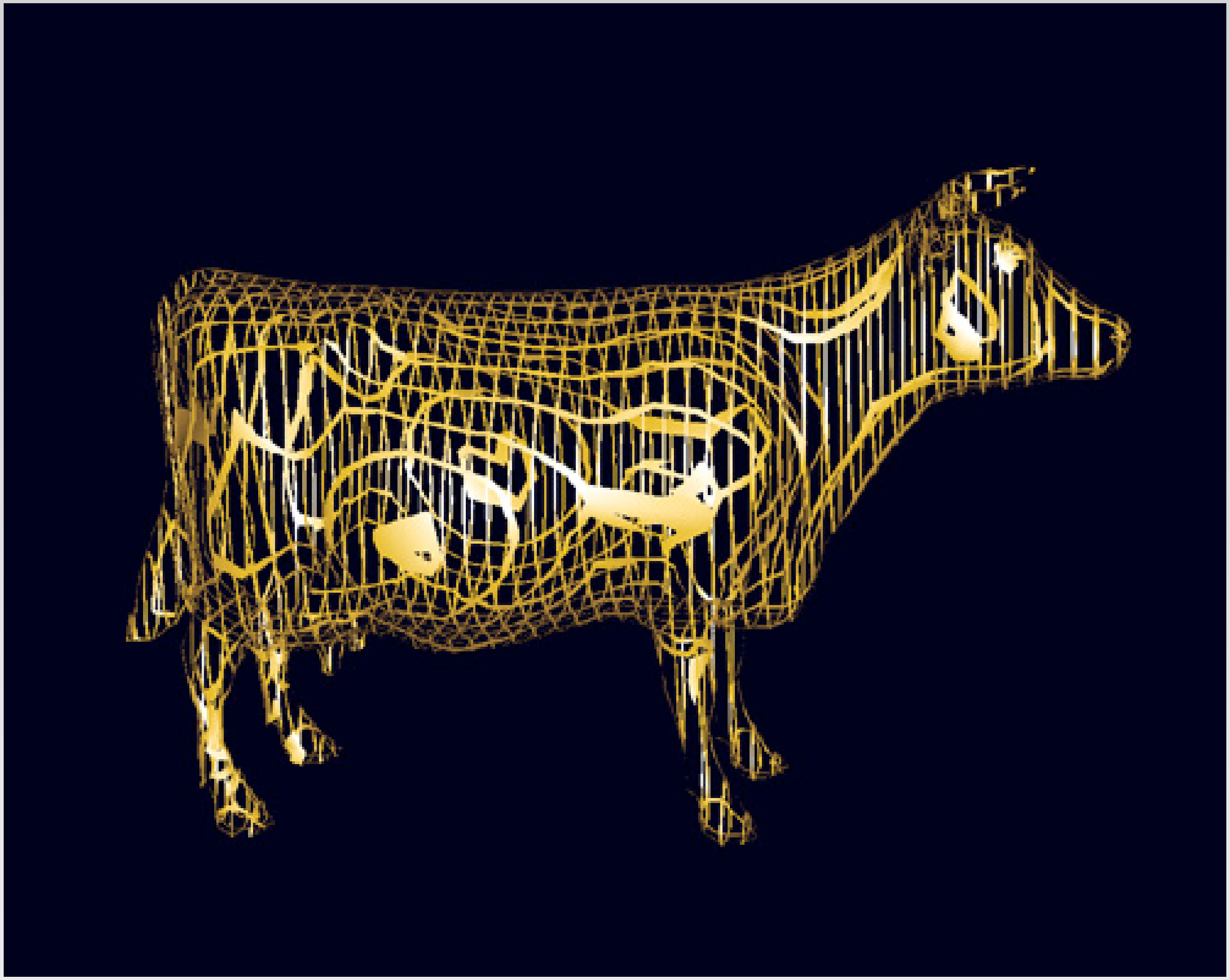
Shader Programming

- Seems to lie on the boundary between art and tech
  - programming is hard-core (parallel algorithms)
  - but intended result is often mostly aesthetic

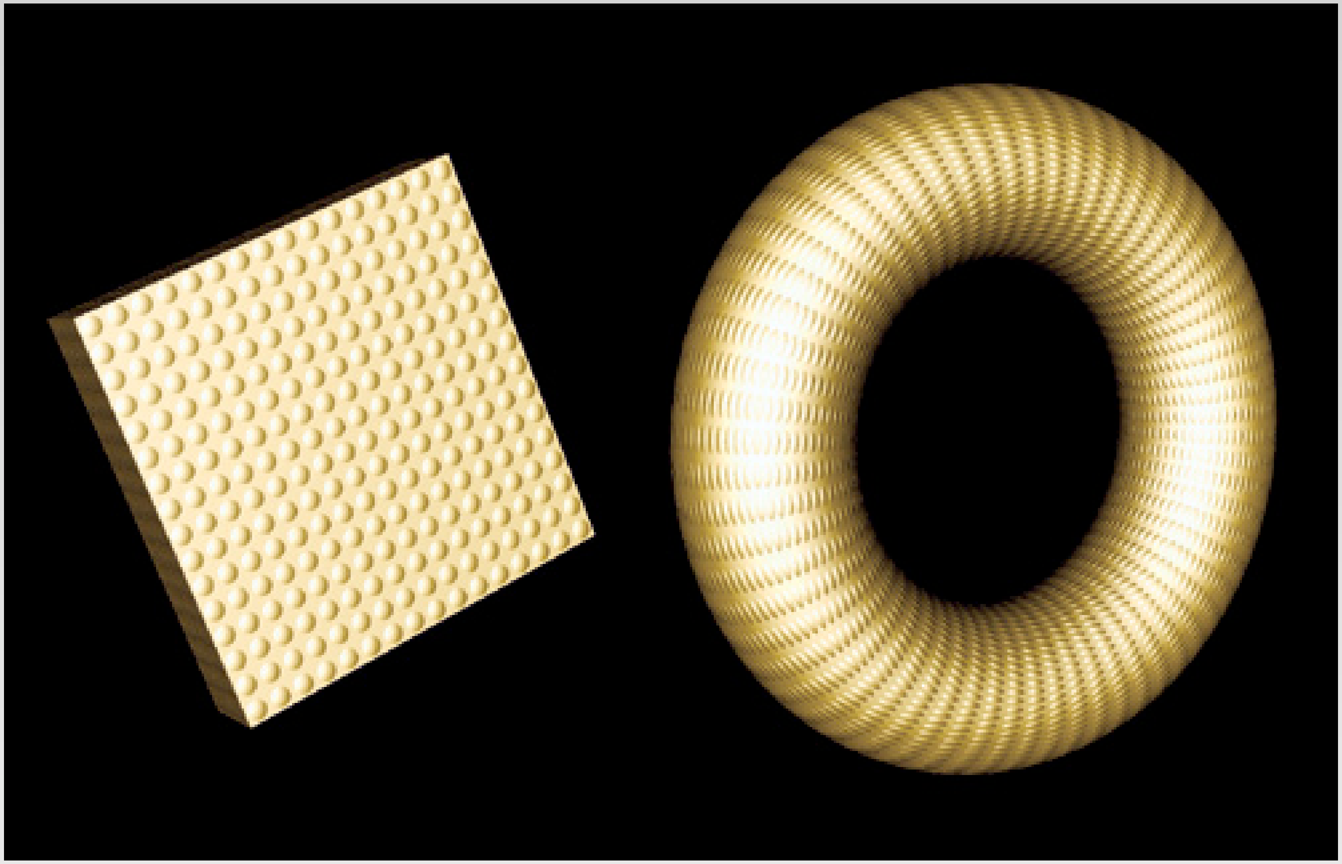


Screen shot of the SolidWorks application, showing a jigsaw rendered with OpenGL shaders to simulate a chrome body, galvanized steel housing, and cast iron blade. (Courtesy of SolidWorks Corporation)

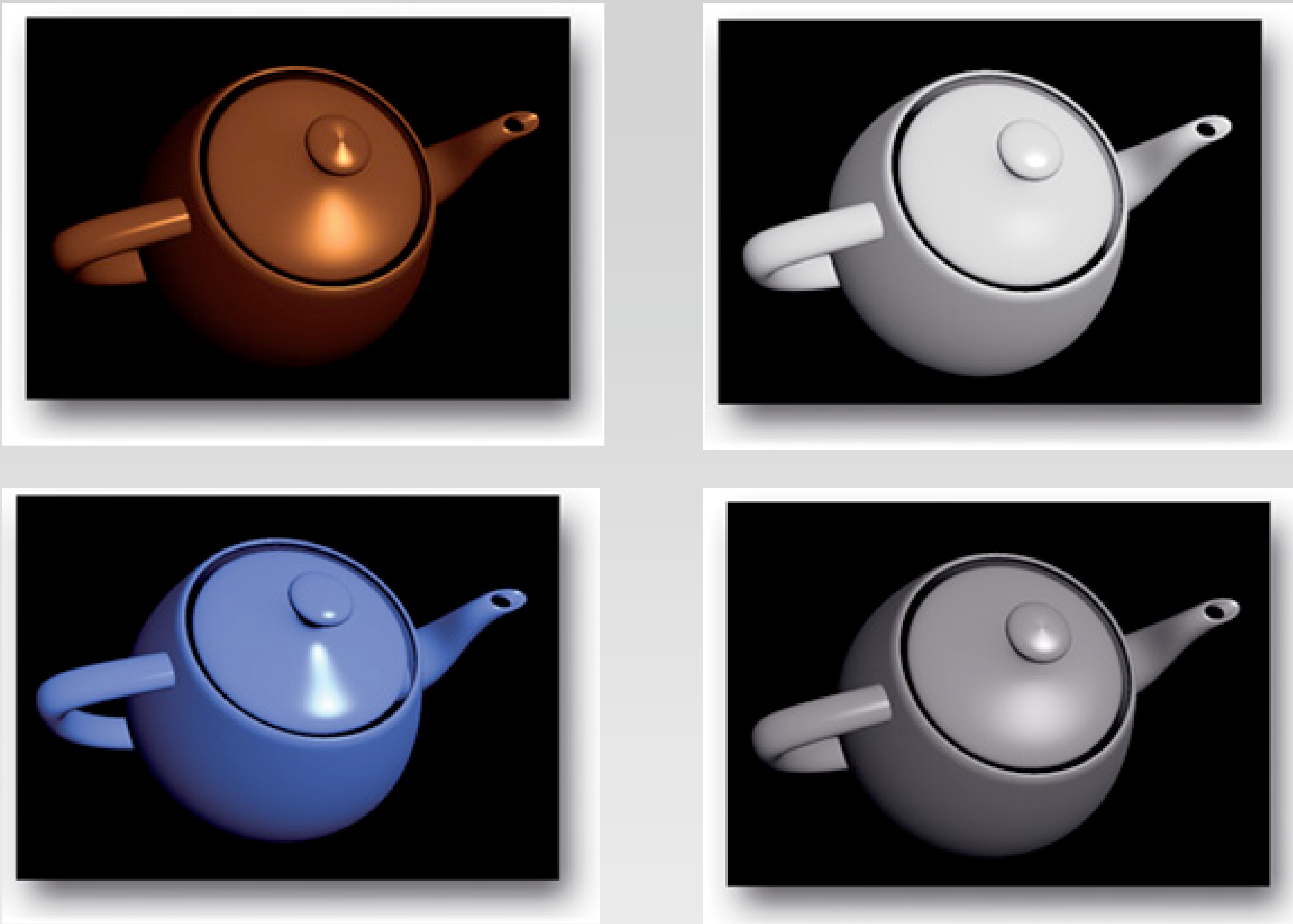
Different glyphs applied to a cube using the glyph bombing shader described in Section 10.6. (3Dlabs, Inc.)



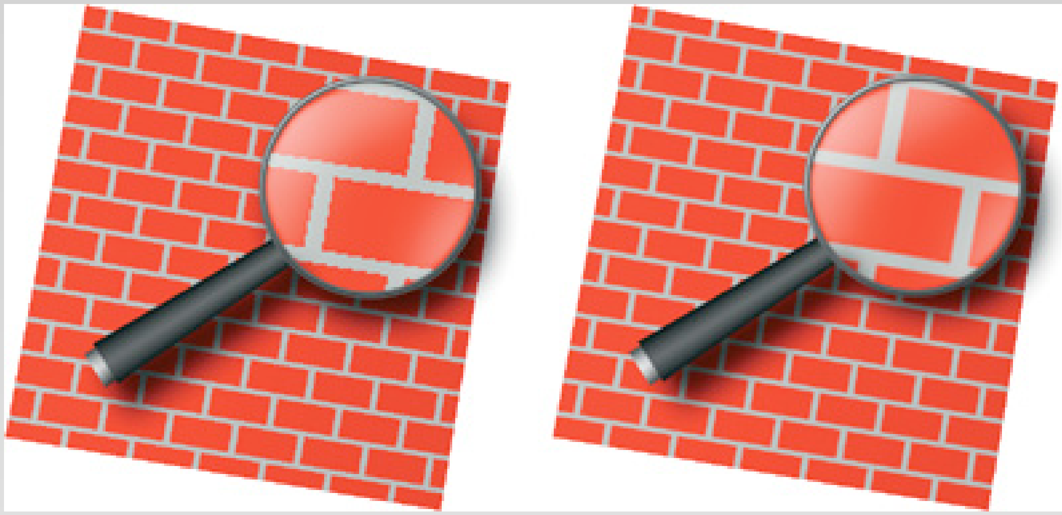
The lattice shader presented in Section 11.3 is applied to the cow model. (3Dlabs, Inc.)



A simple box and a torus that have been bump-mapped using the procedural method described in Section 11.4. (3Dlabs, Inc.)



A variety of materials rendered with Ward's BRDF model (see Section 14.3) and his measured/fitted material parameters.



Brick shader with and without antialiasing. On the left, the results of the brick shader presented in Chapter 6. On the right, results of antialiasing by analytic integration using the brick shader described in Section 17.4.5. (3Dlabs, Inc.)

A variety of screen shots from the 3Dlabs RealWorldz demo. Everything in this demo is generated procedurally using shaders written in the OpenGL Shading Language. This includes the planets themselves, the terrain, atmosphere, clouds, plants, oceans, and rock formations. Planets are modeled as mathematical spheres, not height fields. These scenes are all rendered at interactive rates on current generation graphics hardware