GPU Shading

CMSC 435/634

So what is real-time shading?

- More realistic appearance
  - Bumps, anisotropic surfaces, PRT, …
- Non-realistic appearance
  - Cartoon, sketch, illustration, …
- Animated appearance
  - Character skin, water, clouds
- Visualization
  - Data on surfaces, Volume rendering, …
- General computation

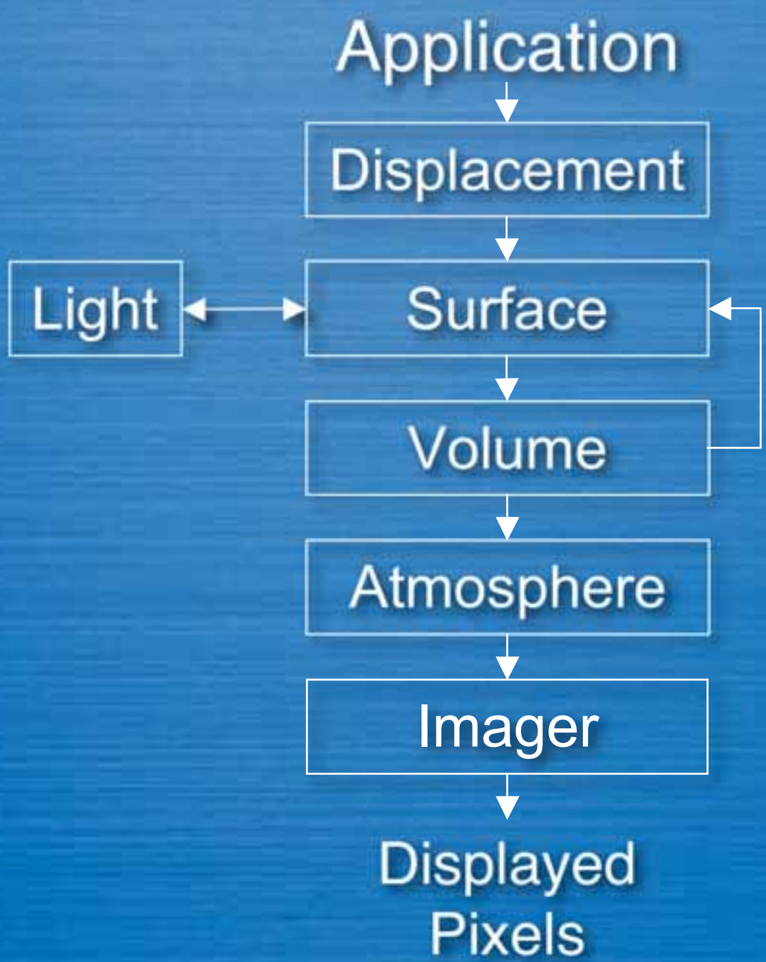
Put another way…

Non-real time vs. Real time

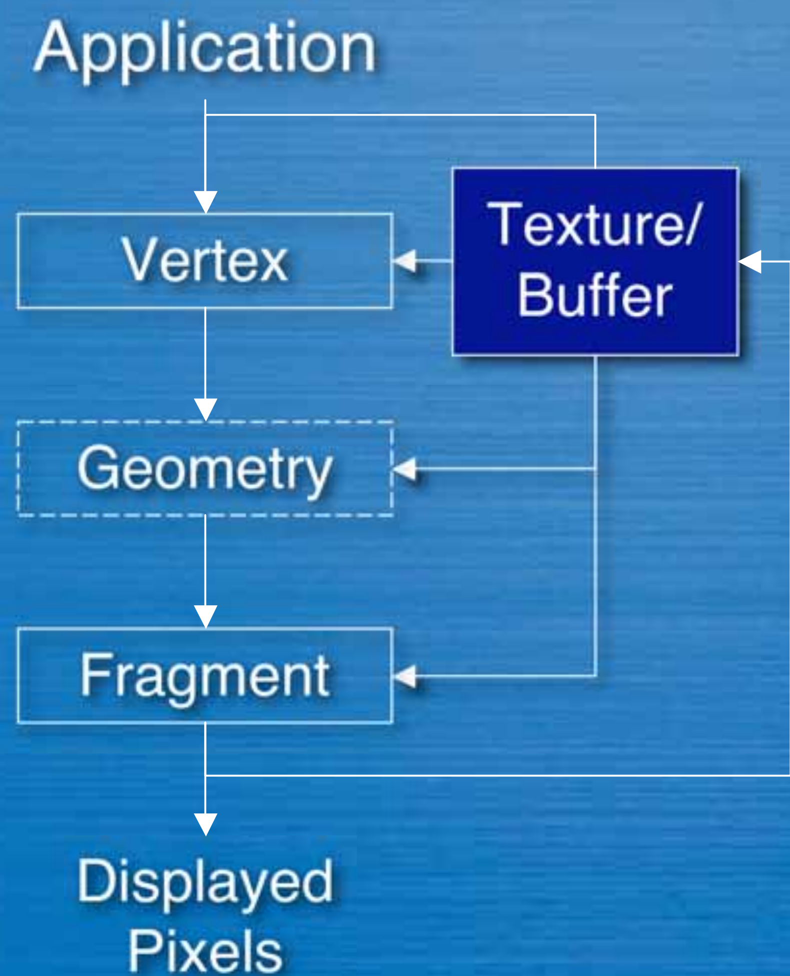
- Not real-time
  - General CPU
  - Seconds to hours per frame
  - Thousands of lines
  - “Unlimited” computation, texture, memory, …
- Real-time
  - Graphics hardware
  - Tens of frames per second
  - Thousands of instructions
  - Limited computation, texture, memory, …

Non-real time vs. Real-time

- Non-real time



- Real-time



History (not real-time)

- Testbed (1981)
- Shade Trees (1984)
- Image Synthesizer (1985)
- RenderMan (1990)

History (real-time)

- Custom HW (1998)
- Multi-pass standard HW (2000)
- Register combiners (2000)
- Vertex programs (2001)
- Compiling to mixed HW (2001)
- Fragment programs (2002)
- Standardized languages (2003-2004)
- Geometry shaders (2006)

Choices

- OS: Windows, Mac, Linux
- GPU: ATI, NVIDIA
- API: DirectX, OpenGL
- Language: HLSL, GLSL, Cg
- Compiler: DirectX, OpenGL, Cg, ASHLI
- Runtime: CgFX, ASHLI, OSG (& others), sample code

Major Commonalities

- Vertex & Fragment/Pixel
- C-like, if/while/for
- Structs & arrays
- Float + small vector and matrix
  - Swizzle & mask (a.xyz = b.xxw)
- Common math & shading functions

Procedure I/O

- Vertex
  - In: [position, normal, matrices, texture coordinates, …]
  - Out: position, [arbitrary others]
- Fragment
  - In: position, [arbitrary others]
  - Out: color, [depth, data]

Major Differences

- Profiles vs. required feature set
- “Virtualization”
- Generate low-level vs. direct compilation

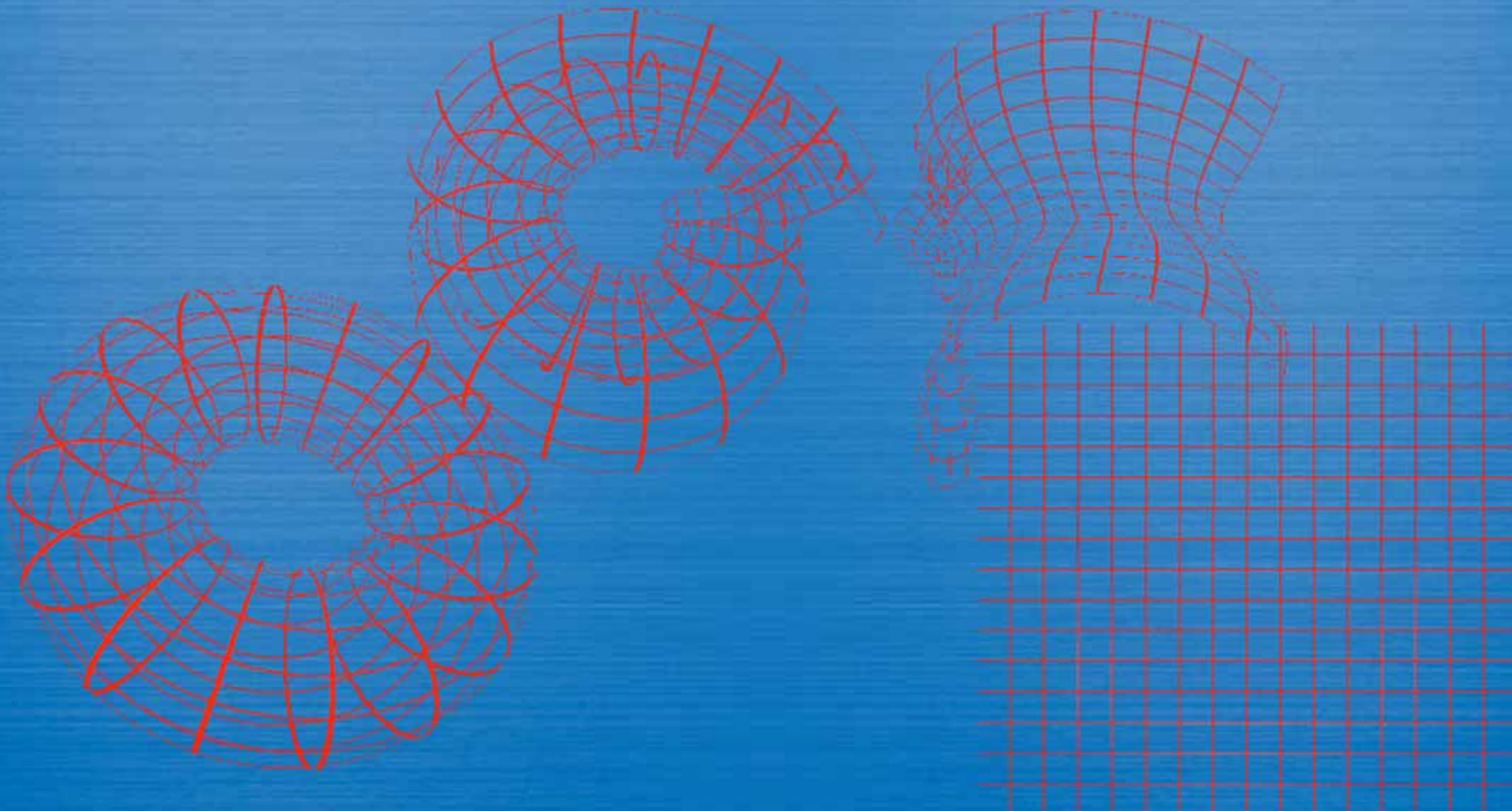
Notable Minor Differences

- :NORMAL vs. predefined & attribute
- half, fixed
- float3 vs. vec3
- mul(matrix, matrix) vs. matrix*matrix

Some OpenGL Code

- OpenGL
- GLSL / vertex & fragment *program*
- Low-level / vertex & fragment *shader*
- C interface

Blend Positions



High-level code

```
void main() {
    float Kin = gl_Color.r;          // key input

    // screen position from vertex and texture
    vec4 Vp = ftransform();
    vec4 Tp = vec4(gl_MultiTexCoord0.xy*1.8-.9, 0.,1.);

    // interpolate between Vp and Tp
    gl_Position = mix(Tp,Vp,pow(1.-Kin,8.));

    // copy to output
    gl_TexCoord[0] = gl_MultiTexCoord0;
    gl_TexCoord[1] = Vp;
    gl_TexCoord[3] = vec4(Kin);
}
```

Low-level code

```
!!ARBvp1.0
  # screen position from vertex
TEMP Vp;
DP4 Vp.x, state.matrix.mvp.row[0], vertex.position;
DP4 Vp.y, state.matrix.mvp.row[1], vertex.position;
DP4 Vp.z, state.matrix.mvp.row[2], vertex.position;
DP4 Vp.w, state.matrix.mvp.row[3], vertex.position;
  # screen position from texture
TEMP Tp;
MAD Tp, vertex.texcoord,{1.8,1.8,0,0},{-.9,-.9,0,1};
  # interpolate
MAD Tp, Tp, -vertex.color.x, Tp;
MAD result.position, Tp, vertex.color.x, Tp;
  # copy to output
MOV result.texcoord[0], vertex.texcoord;
MOV result.texcoord[1], Vp;
MOV result.texcoord[2], vertex.color.x;
END
```

Using high-level code

- Create shader object
  `S = glCreateShader (GL_VERTEX_SHADER)`
  - Vertex or Fragment (or Geometry)
- Load shader into object
  `glShaderSource(S, n, shaderArray, lenArray)`
  - Array of strings
  - NULL `lenArray` or 0 length = `\0` terminated
- Compile object
  `glCompileShader(S);`
  - Can check errors

Using high-level code (2)

- Create program object

```
P = glCreateProgram ()
```

- Attach all shader objects

```
glAttachObject(P, S)
```

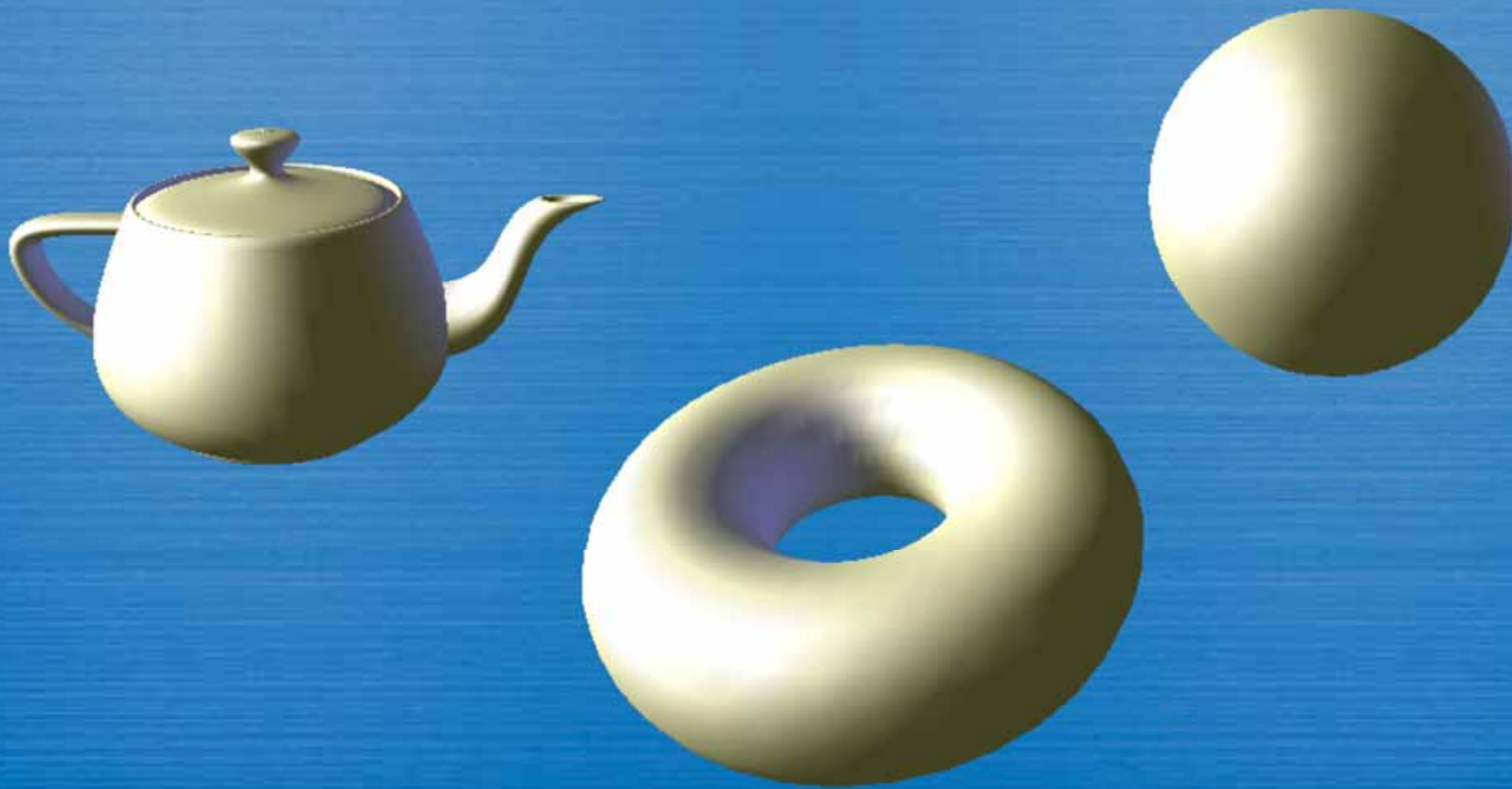
  - Vertex, Fragment or both
- Link together

```
glLinkProgram(P)
```

- Use

```
glUseProgram (P)
```

Vertex Lighting



Vertex Lighting

```
void main() {
  // convert shading-related vectors to eye space
  vec4 P = gl_ModelViewMatrix*gl_Vertex;
  vec4 E = gl_ProjectionMatrixInverse*vec4(0,0,-1,0);
  vec3 V = normalize(E.xyz*P.w-P.xyz*E.w);
  vec3 N = normalize(gl_NormalMatrix*gl_Normal);
  …
```

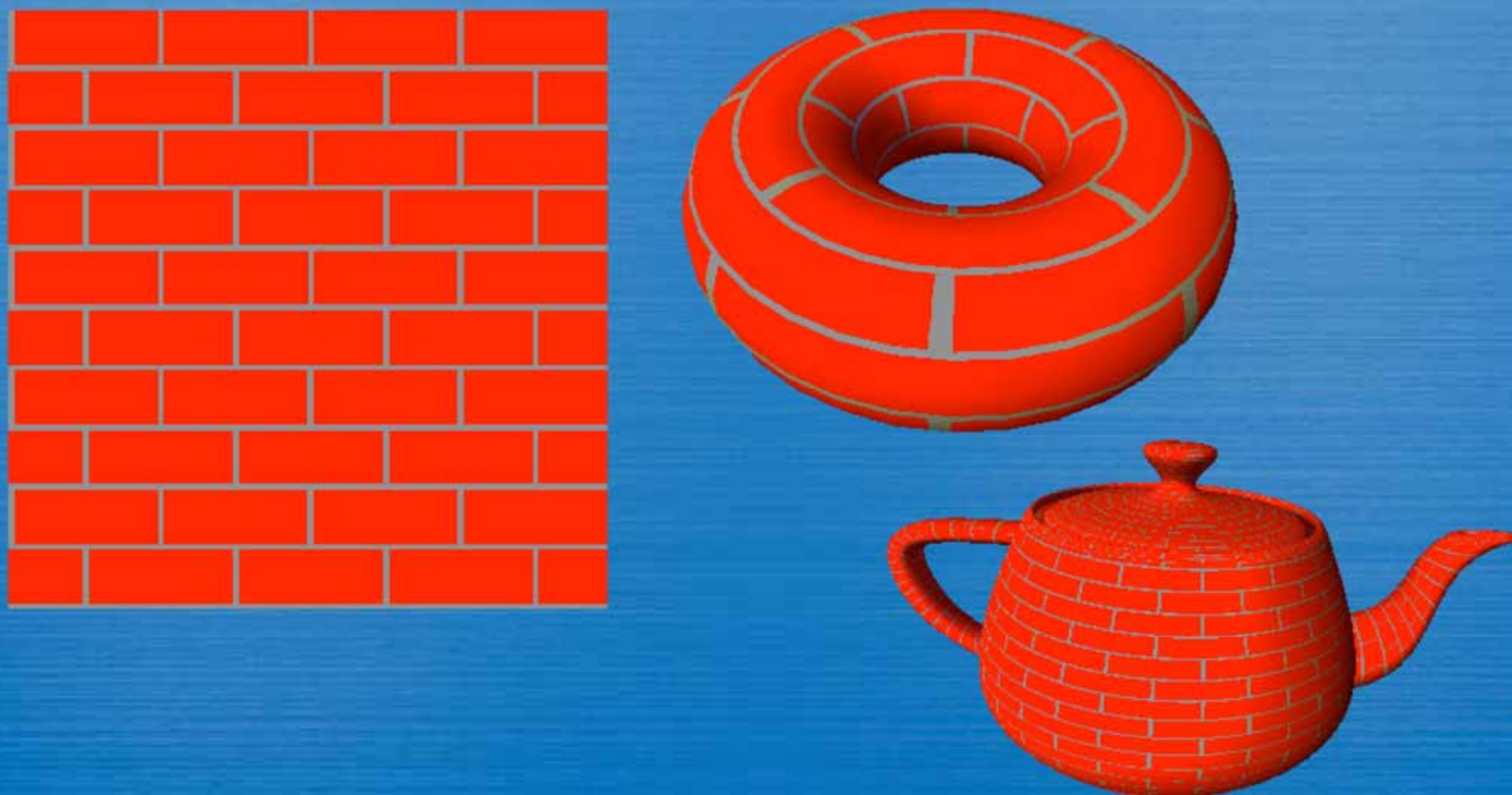
Vertex Lighting

```
…
// accumulate contribution from each light
gl_FrontColor = gl_FrontMaterial.emission;
for(int i=0; i<gl_MaxLights; i++) {
  vec3 L = normalize(gl_LightSource[i].position.xyz*P.w
                   - P.xyz*gl_LightSource[i].position.w);
  vec3 H = normalize(L+V);
  float diff = dot(N,L);

  gl_FrontColor += gl_FrontLightProduct[i].ambient;
  if (diff > 0.) {
    gl_FrontColor += gl_FrontLightProduct[i].diffuse * diff;
    gl_FrontColor += gl_FrontLightProduct[i].specular *
                     max(pow(dot(N,H),
                            gl_FrontMaterial.shininess),0.);
  }
}
…
```

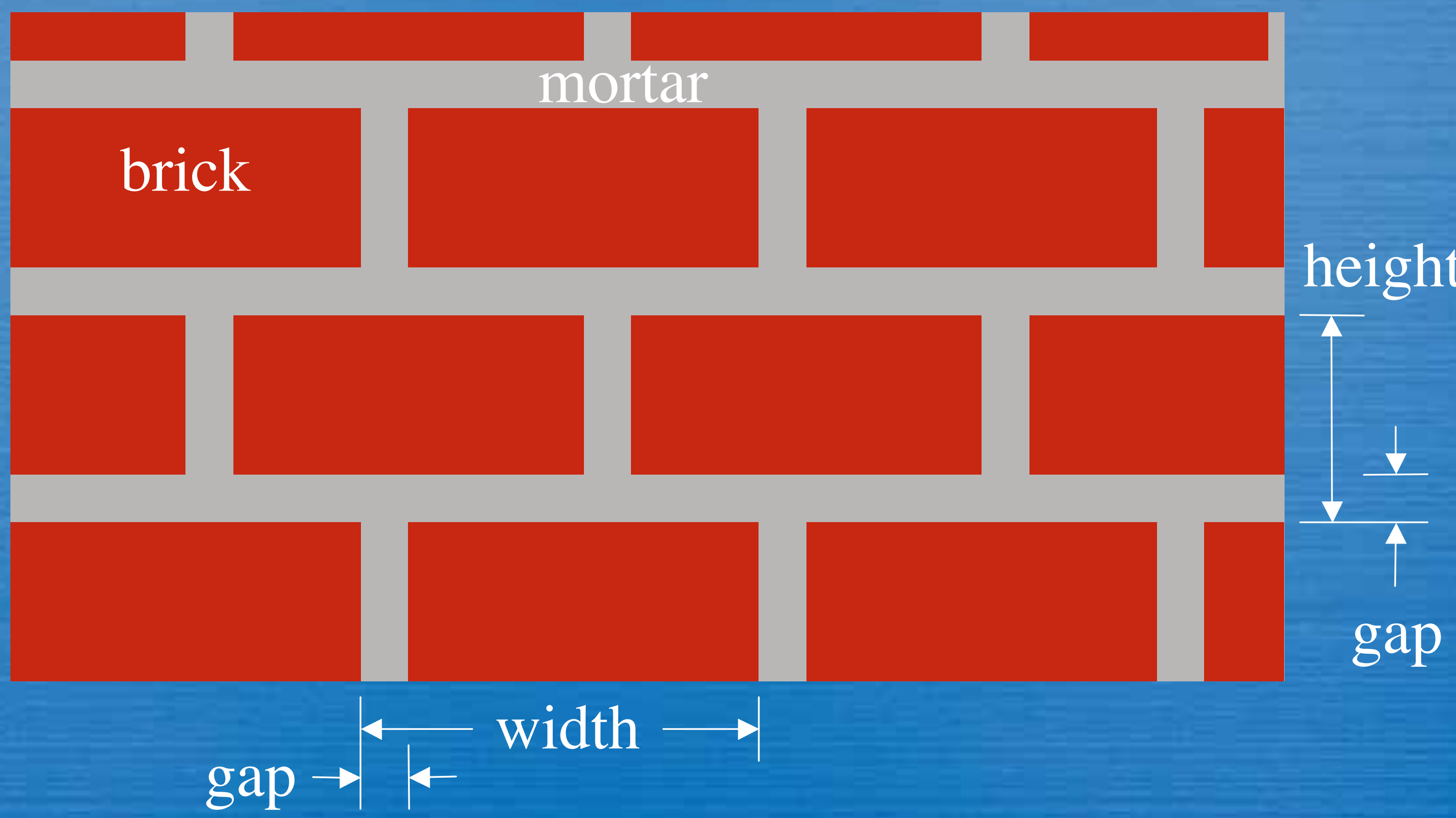
Vertex Lighting

```
  …
  // standard texture coordinate and position stuff
  gl_TexCoord[0] = gl_TextureMatrix[0]*gl_MultiTexCoord0;
  gl_Position = ftransform();
}
```

Fragment Brick



Brick

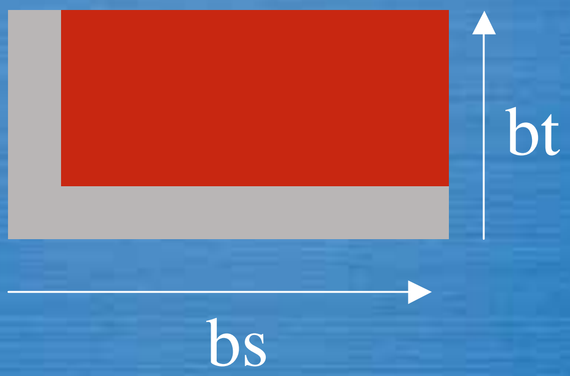


Brick Shader

```
// shader constants, could be passed in to allow modification
float width=.25, height = .1, gap = .01;
vec4 brick = vec4(1.,0.,0.,1.);
vec4 mortar = vec4(.5,.5,.5,1.);

void main() {

  /* … compute brick color … */

  gl_FragColor *= gl_Color;
}
```

Brick Color

- Where am I in my brick?
  - "brick coordinates"




```
/* compute bs and bt brick coordinates */

// pick color for this pixel, brick or mortar
if (bs < gap || bt < gap)
  gl_FragColor = mortar;
else
  gl_FragColor = brick;
```

Brick Coordinates

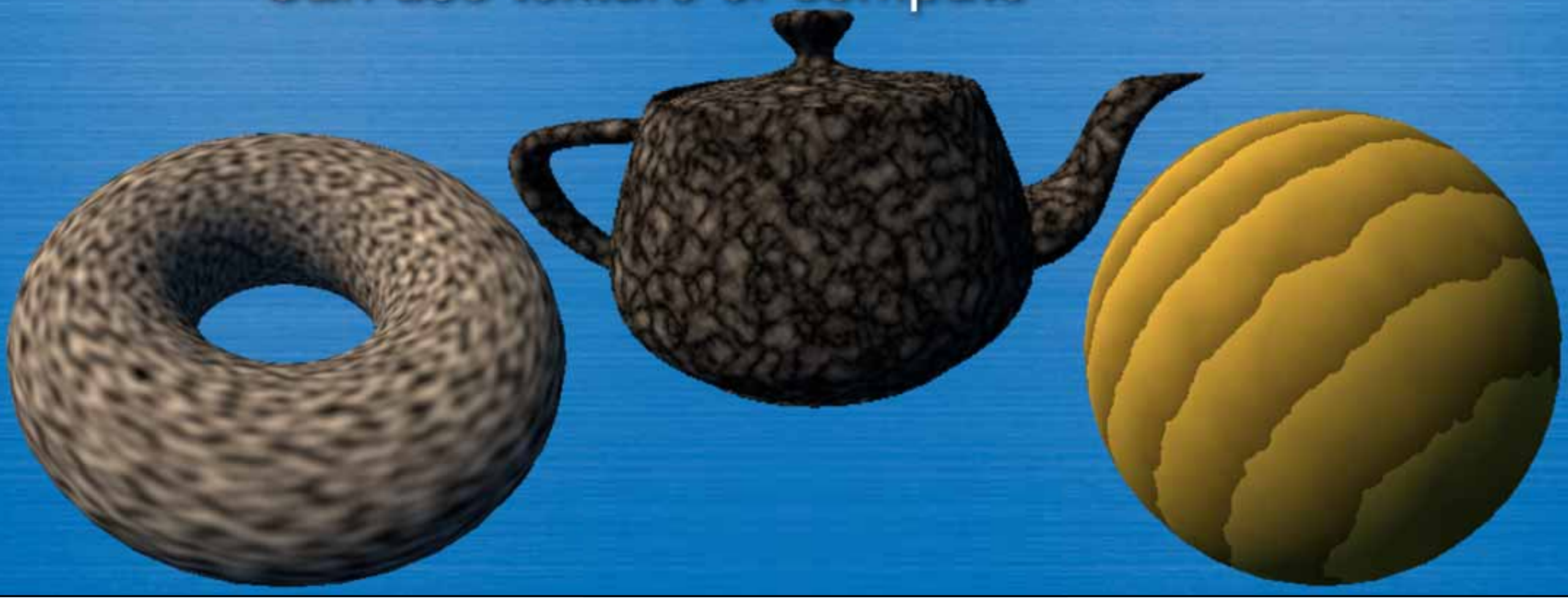
```
// find row and column for this pixel
float bs = gl_TexCoord[0].x, bt = gl_TexCoord[0].y;

// offset even rows by half a column
if (mod(bt,2.*height)<height)
  bs += width/2.;

// wrap texture coordinates to get “brick coordinates”
bs = mod(bs,width);
bt = mod(bt,height);
```

Shader Design Strategies

- Learn and adapt from RenderMan
  - Noise
  - Layers
- Multiple Passes
- *Baked* computation

Noise

- Controlled, repeatable randomness
  - Noise functions generally not implemented
  - Can use texture or compute



Layers

- Incremental
  - Easier to write
  - Easier to visually debug
- See Steve May's RManNotes
  - http://accad.osu.edu/~smay/RManNotes/

Multiple Passes

- Uses
  - Non-local communication
  - Exceed resource constraints
- Methods
  - Projection
  - Geometry Images
  - Texture Atlas

Baked Computation

- Texture = arbitrary vector-valued function of 1-3 variables
- Often cheaper to precompute & look up
  - Noise textures
  - Precomputed radiance transfer
  - BRDF factorizations
  - …

Precomputation Tricks

- Fix some degrees of freedom
  - E.g. Isotropic BRDFs only
- Factor into several functions
- Project input to another space
  - Tangent space
  - World space
- Project output to another space
  - Spherical harmonics
  - Wavelets

Advanced Uses

- Visualization
- Approximations to global illumination
- Surfaces with volume shells
- Point-based rendering
- Geometry shading