Computer Graphics

spring, 2013

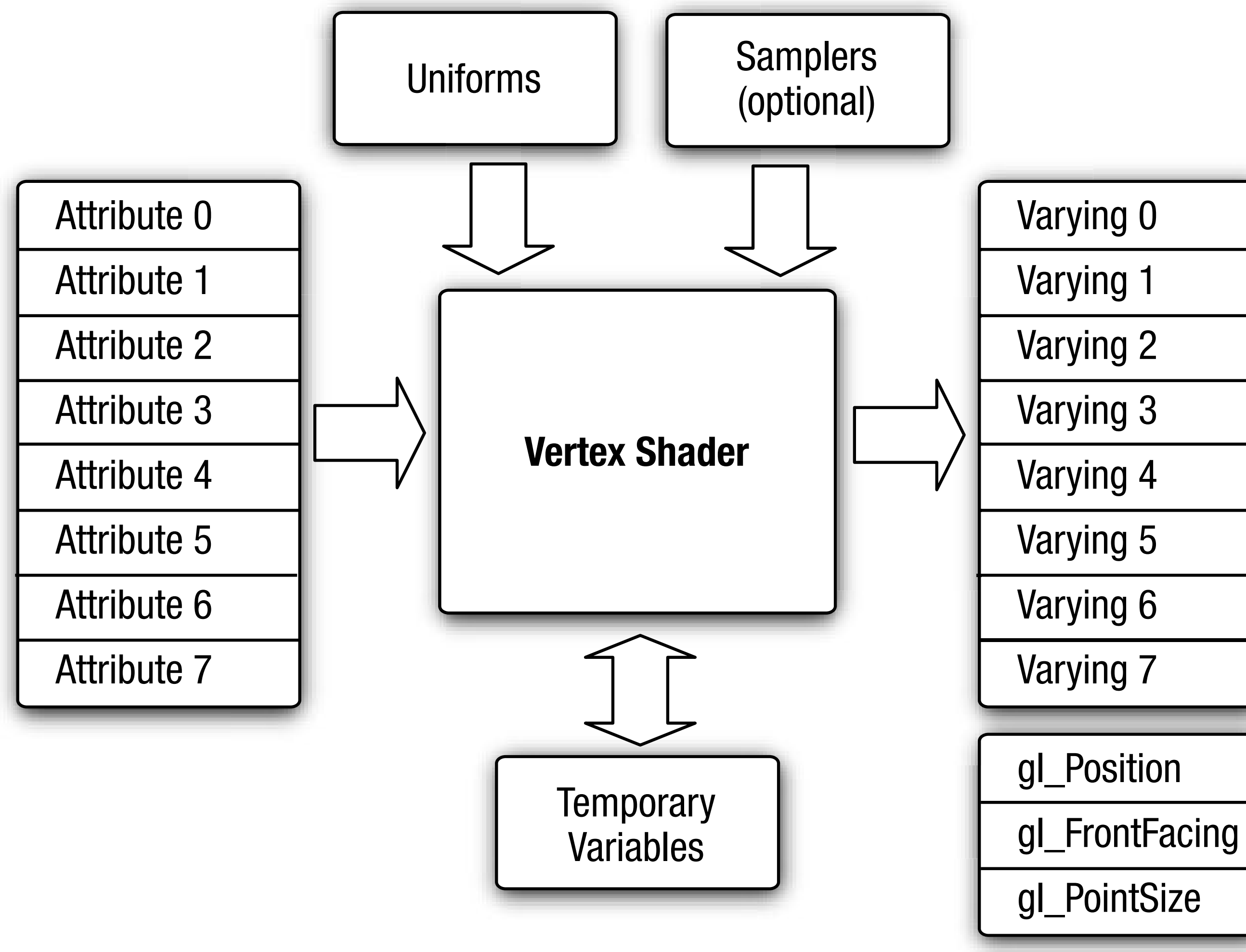
Chapter 8
Vertex Shaders

Vertex Shaders

- Vertex-based operations
  - Transformation by model view & projection matrices
  - Lighting computation to generate per-vertex diffuse & specular colors
  - Generating/transforming texture coordinates
  - Vertex skinning

Vertex Shaders

- Inputs
  - Attributes
  - Uniforms
  - Shader program
- Output
  - Varying variables including built-ins such as gl_Position, gl_FrontFacing, and gl_PointSize



Built-in Varying

- gl_Position
  - Vertex position in clip coordinates
  - Used for clipping and then transformed to screen coords
  - highp float
- gl_PointSize
  - Size of the point sprite in pixels
  - mediump float
- gl_FrontFacing
  - Not directly written by the vertex shader
  - boolean type

Built-in Uniform

- gl_DepthRange
  - gl_DepthRangeParameters type
  - ```
    struct gl_DepthRangeParameters
    {
        highp float near; // near Z
        highp float far; // far Z
        highp float diff; // far - near
    }
    ```
    ```

Built-in Constants

- gl_MaxVertexAttribs
  - mediump int, >=8
- gl_MaxVertexUniformVectors
  - mediump int, >=128
- gl_MaxVaryingVectors
  - mediump int, >=8
- gl_MaxVertexTextureImageUnits
  - mediump int, >=0
- gl_MaxCombinedTextureImageUnits
  - mediump int, >=8

Precision Qualifiers

- Default precision in the vertex shader is highp
  - c.f) No default precision in the fragment shader
- Guideline
  - highp for position, normal, & texcoords
  - mediump for color & light computation

Limitations

- Length of vertex shader
  - No way to query the # of instructions
  - Compilation error if exceeded
- # of temporary variables
  - No requirement
  - Compilation error if exceeded

Limitations (cont'd)

- Flow control
  - Only one loop index in a for loop
  - The loop index must be initialized to a constant integral expression
  - The condition in the for loop should be in the form
    - loop_index <, <=, >, >=, !=, == constant_expression
  - The loop index can be modified in the for loop only as
    - loop_index--, ++
    - loop_index -=, += constant_expression
  - The loop index can be passed as a read-only argument to functions inside the for loop

Limitations: Examples

valid

```
const int numLights = 4;

int i, j;
for (i=0; i<numLights; i++)
{
   …
}

for (j=4; j>0; j--)
{
   …
   foo(j);  // argument to function foo that takes j
            // is declared with the in qualifier.
}
```

invalid

```
uniform int numLights;

int i;
for (i=0; i<numLights; i++)  // conditional expression is
                             // not constant
{
   …
}

for (i=0; i<8; i++)
{
   i = foo();   // return value of foo() cannot be
                // assigned to loop index i
}

for (j=4; j>0;)
{
   …
   j--;   // loop index j cannot be modified
          // inside for loop
}
```

Limitations (cont'd)

- while & do-while are not requirements -- may not be supported
- Conditional statements
  - Different value for condition --> divergent flow control --> serialized --> slow performance
  - Same condition value for all vertices/ fragments recommended (e.g. uniform expression)

Limitations

- Array indexing
  - Uniforms (except samplers) -- fully supported including dynamic indexing
  - Samplers -- constant integral expressions only (literal value, const integer var, constant expression)
  - Attribs -- constant integral expressions only

of Uniforms Used

- Uniform storage is used to store...
  - Uniforms
  - Const vars
  - Literal values
  - Implementation-specific constants
- Packing rules applied
- No constant propagation assumed for literal values --> using const vars recommended

```
#define NUM_TEXTURES   2

uniform mat4  tex_matrix[NUM_TEXTURES];        // texture matrices
uniform bool  enable_tex[NUM_TEXTURES];        // texture enables
uniform bool  enable_tex_matrix[NUM_TEXTURES]; // texture matrix
                                               // enables

attribute vec4  a_texcoord0;  // available if enable_tex[0] is true
attribute vec4  a_texcoord1;  // available if enable_tex[1] is true

varying vec4    v_texcoord[NUM_TEXTURES];

v_texcoord[0] = vec4(0.0, 0.0, 0.0, 1.0);
// is texture 0 enabled
if (enable_tex[0])
{
   // is texture matrix 0 enabled
   if(enable_tex_matrix[0])
      v_texcoord[0] = tex_matrix[0] * a_texcoord0;
   else
      v_texcoord[0] = a_texcoord0;
}

v_texcoord[1] = vec4(0.0, 0.0, 0.0, 1.0);
// is texture 1 enabled
if (enable_tex[1])
{
   // is texture matrix 1 enabled
   if(enable_tex_matrix[1])
      v_texcoord[1] = tex_matrix[1] * a_texcoord1;
   else
      v_texcoord[1] = a_texcoord1;
}
```

```
#define NUM_TEXTURES   2

const int  c_zero = 0;
const int  c_one  = 1;

uniform mat4  tex_matrix[NUM_TEXTURES];        // texture matrices
uniform bool  enable_tex[NUM_TEXTURES];        // texture enables
uniform bool  enable_tex_matrix[NUM_TEXTURES]; // texture matrix
                                               // enables
attribute vec4  a_texcoord0;  // available if enable_tex[0] is true
attribute vec4  a_texcoord1;  // available if enable_tex[1] is true

varying vec4    v_texcoord[NUM_TEXTURES];

v_texcoord[c_zero] = vec4(float(c_zero), float(c_zero),
                          float(c_zero), float(c_one));
// is texture 0 enabled
if(enable_tex[c_zero])
{
   // is texture matrix 0 enabled
   if(enable_tex_matrix[c_zero])
      v_texcoord[c_zero] = tex_matrix[c_zero] * a_texcoord0;
   else
      v_texcoord[c_zero] = a_texcoord0;
}

v_texcoord[c_one] = vec4(float(c_zero), float(c_zero),
                         float(c_zero), float(c_one));
// is texture 1 enabled
if(enable_tex[c_one])
{
   // is texture matrix 1 enabled
   if(enable_tex_matrix[c_one])
      v_texcoord[c_one] = tex_matrix[c_one] * a_texcoord1;
   else
      v_texcoord[c_one] = a_texcoord1;
}
```

Examples

- Transforming vertex position with a matrix
- Lighting computations to generate per-vertex diffuse and specular color
- Texture coordinate generation
- Vertex skinning

Transformation

```
// uniforms used by the vertex shader
uniform mat4 u_mvp_matrix; // matrix to convert P from
                           // model space to clip space.

// attributes input to the vertex shader
attribute vec4 a_position; // input position value
attribute vec4 a_color;    // input vertex color

// varying variables – input to the fragment shader
varying vec4 v_color; // output vertex color

void main()
{
    v_color = a_color;
    gl_Position = u_mvp_matrix * a_position;
}
```

How to compute u_mvp_matrix?

--> Songho's tutorial

Lighting Computation

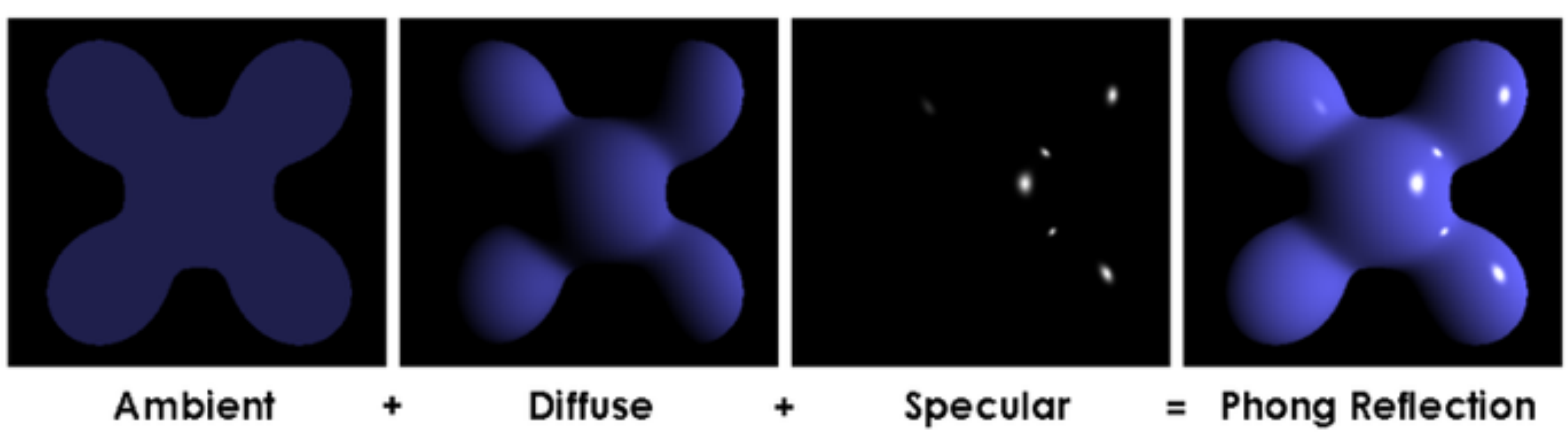
- Blinn-Phong shading model
  - GLES1.1 lighting equation
  - Extension of Phong reflection model

Phong Reflection Model

- Empirical model, local illumination
- Total reflection = ambient + diffuse + specular

$$I_p = k_a i_a + \sum_{\mathrm{m} \in \mathrm{lights}} (k_d(\hat{L}_m \cdot \hat{N}) i_{m,d} + k_s(\hat{R}_m \cdot \hat{V})^{\alpha} i_{m,s}).$$

k_a, k_d, k_s: ambient, diffusive, specular reflection constant
α: shininess constant

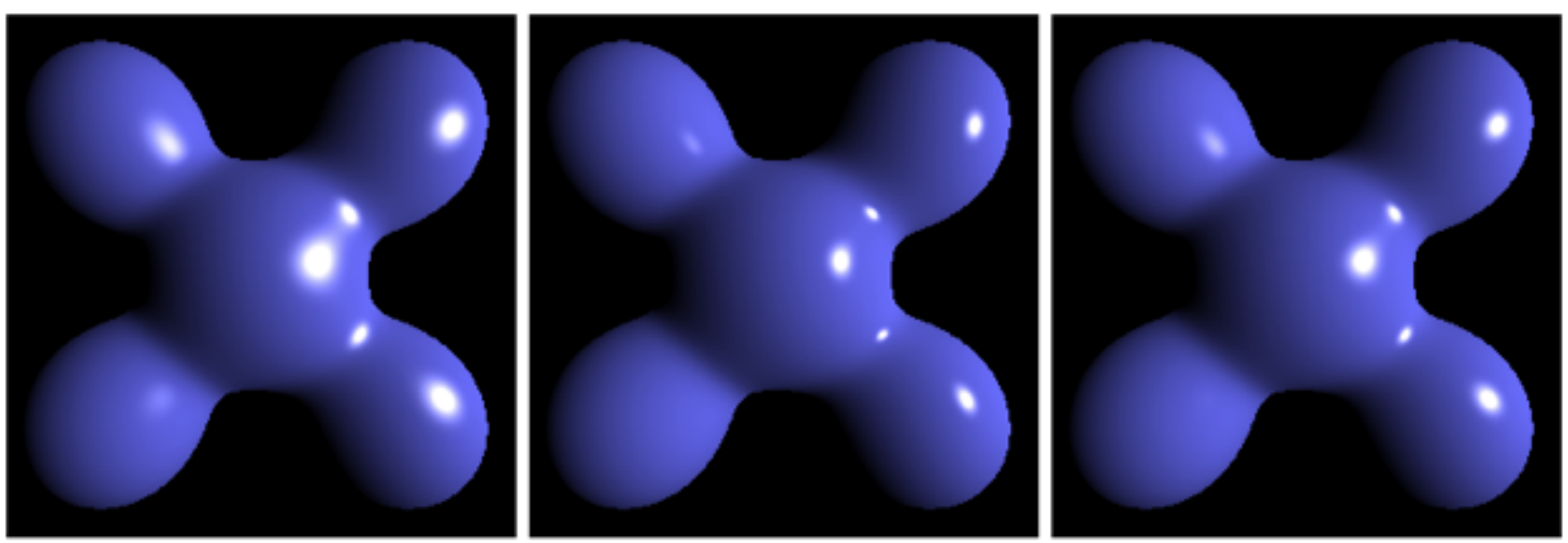


Blinn-Phong Reflection Model

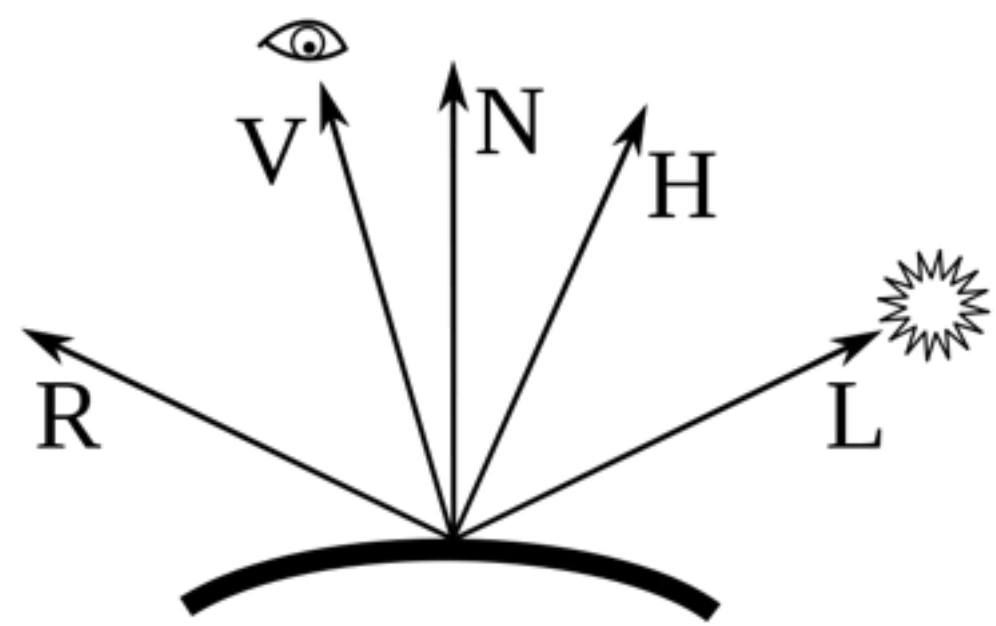
- R is expensive to compute

$$\hat{R}_m = 2(\hat{L}_m \cdot \hat{N})\hat{N} - \hat{L}_m$$

- dot(R,V) is approximated with dot(N,H)



Blinn–Phong   Phong   Blinn–Phong (higher exponent)



Samples

- Phong reflection model
  - http://blog.shayanjaved.com/2011/03/13/shaders-android/
- Blinn-Phong reflection model
  - “ComplexLighting” example in the PowerVR SDK -- Examples/Intermediate/ComplexLighting

Directional Light

- Directional light --> w=0
- The viewer is assumed to be at infinity. (w=0)
- Material property tells how much light is reflected.

Directional Light

```
struct directional_light {
   vec3  direction;     // normalized light direction in eye space
   vec3  halfplane;     // normalized half-plane vector
   vec4  ambient_color;
   vec4  diffuse_color;
   vec4  specular_color;
};

struct material_properties {
   vec4   ambient_color;
   vec4   diffuse_color;
   vec4   specular_color;
   float  specular_exponent;
};

const float                  c_zero = 0.0;
const float                  c_one = 1.0;

uniform material_properties  material;
uniform directional_light    light;

// normal has been transformed into eye space and is a normalized
// value returns the computed color.
void
directional_light(vec3 normal)
{
   vec4   computed_color = vec4(c_zero, c_zero, c_zero, c_zero);
   float  ndotl;  // dot product of normal & light direction
   float  ndoth;  // dot product of normal & half-plane vector

   ndotl = max(c_zero, dot(normal, light.direction));
   ndoth = max(c_zero, dot(normal, light.halfplane));

   computed_color += (light.ambient_color * material.ambient_color);
   computed_color += (ndotl * light.diffuse_color
                      * material.diffuse_color);

   if (ndoth > c_zero)
   {
      computed_color += (pow(ndoth, material.specular_exponent) *
                         material.specular_color *
                         light.specular_color);
   }

   return computed_color;
}
```

Point & Spot Light

- $w \neq 0$
- $distance\ attenuation = 1 / (K_0 + K_1 \times \| VP_{light} \| + K_2 \times \| VP_{light} \|^2)$
- Spot light
- point light + direction & cutoff angle
- cutoff attenuation

```
struct spot_light {
   vec4    position;            // light position in eye space
   vec4    ambient_color;
   vec4    diffuse_color;
   vec4    specular_color;
   vec3    spot_direction;      // normalized spot direction
   vec3    attenuation_factors; // attenuation factors K0, K1, K2
   bool    compute_distance_attenuation;
   float   spot_exponent;       // spotlight exponent term
   float   spot_cutoff_angle;   // spot cutoff angle in degrees
};

struct material_properties {
   vec4    ambient_color;
   vec4    diffuse_color;
   vec4    specular_color;
   float   specular_exponent;
};

const float                  c_zero = 0.0;
const float                  c_one = 1.0;

uniform material_properties  material;
uniform spot_light           light;

// normal and position are normal and position values in eye space.
// normal is a normalized vector.
// returns the computed color.

vec4
spot_light(vec3 normal, vec4 position)
{
   vec4    computed_color = vec4(c_zero, c_zero, c_zero, c_zero);
   vec3    lightdir;
   vec3    halfplane;
   float   ndotl, ndoth;
   float   att_factor;

   att_factor = c_one;

   // we assume "w" values for light position and
   // vertex position are the same
   lightdir = light.position.xyz - position.xyz;

   // compute distance attenuation
   if(light.compute_distance_attenuation)
   {
      vec3    att_dist;

      att_dist.x = c_one;
      att_dist.z = dot(lightdir, lightdir);
      att_dist.y = sqrt(att_dist.z);
      att_factor = c_one / dot(att_dist, light.attenuation_factors);
   }

   // normalize the light direction vector
   lightdir = normalize(lightdir);

   // compute spot cutoff factor
   if(light.spot_cutoff_angle < 180.0)
   {
      float  spot_factor = dot(-lightdir, light.spot_direction);

      if(spot_factor >= cos(radians(light.spot_cutoff_angle)))
         spot_factor = pow(spot_factor, light.spot_exponent);
      else
         spot_factor = c_zero;

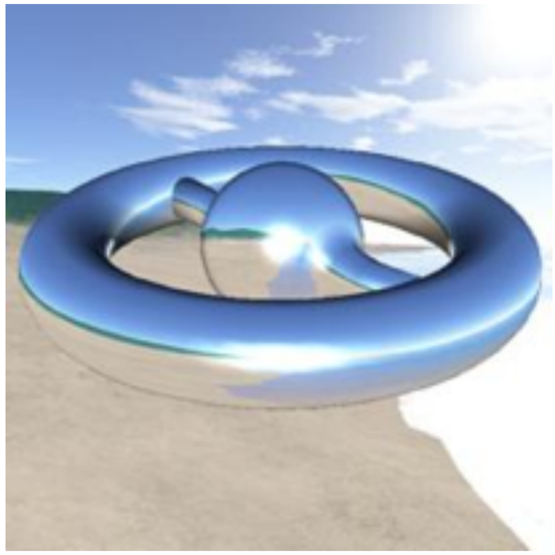
      // compute combined distance & spot attenuation factor
      att_factor *= spot_factor;
   }

   if(att_factor > c_zero)
   {
      // process lighting equation --> compute the light color
      computed_color += (light.ambient_color *
                         material.ambient_color);
      ndotl = max(c_zero, dot(normal, lightdir));
      computed_color += (ndotl * light.diffuse_color *
                         material.diffuse_color);
      halfplane = normalize(lightdir + vec3(c_zero, c_zero, c_one));
      ndoth = dot(normal, halfplane);
      if (ndoth > c_zero)
      {
         computed_color += (pow(ndoth, material.specular_exponent)*
                            material.specular_color *
                            light.specular_color);
      }

      // multiply color with computed attenuation
      computed_color *= att_factor;
   }

   return computed_color;
}
```

Texcoords Generation



- Environment mapping
- Using sphere map or cube map (cube map is less distorted)

```
// position is the normalized position coordinate in eye space
// normal is the normalized normal coordinate in eye space
// returns a vec2 texture coordinate
vec2
sphere_map(vec3 position, vec3 normal)
{
   reflection = reflect(position, normal);
   m = 2.0 * sqrt(reflection.x * reflection.x +
                  reflection.y * reflection.y +
                  (reflection.z + 1.0) * (reflection.z + 1.0));
   return vec2((reflection.x / m + 0.5), (reflection.y / m + 0.5));
}
```

```
// position is the normalized position coordinate in eye space
// normal is the normalized normal coordinate in eye space
// returns the reflection vector as a vec3 texture coordinate
vec3
cube_map(vec3 position, vec3 normal)
{
   return reflect(position, normal);
}
```

Vertex Skinning

- Smooth transition near joints for character animation using <u>skeletons</u>
- Vertices transformed by a combination of multiple matrices stored in a matrix palette

GLES1.1 Fixed Pipeline

- Transform normal & position to eye space (for lighting)
- Rescale or normalization of normal
- Lighting computation for up to 8 lights
- Texcoords transformation up to 2
- Fog factor computation
- Per-vertex user clip plane factor
- Transform position to clip space