Topics

Bump mapping In OpenGL

Blending and Composition

Mapping Methods

- Texture mapping
- Environmental (reflection) mapping
  - Variant of texture mapping
- Bump mapping
  - Solves flatness problem of texture mapping

Recall the Example of Modeling an Orange

Consider modeling an orange

Texture map a photo of an orange onto a surface

- Captures dimples
- Will not be correct if we move viewer or light
- We have shades of dimples rather than their correct orientation

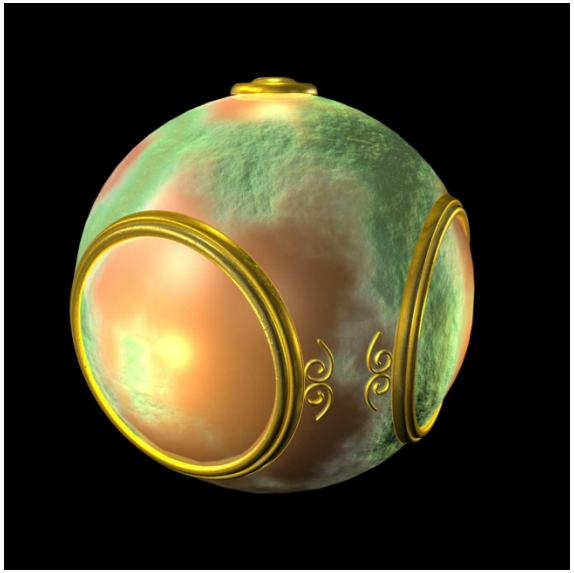
Ideally we need to perturb normal across surface of object and compute a new color at each interior point

Bump Mapping

Perturb normal for each fragment before applying lighting

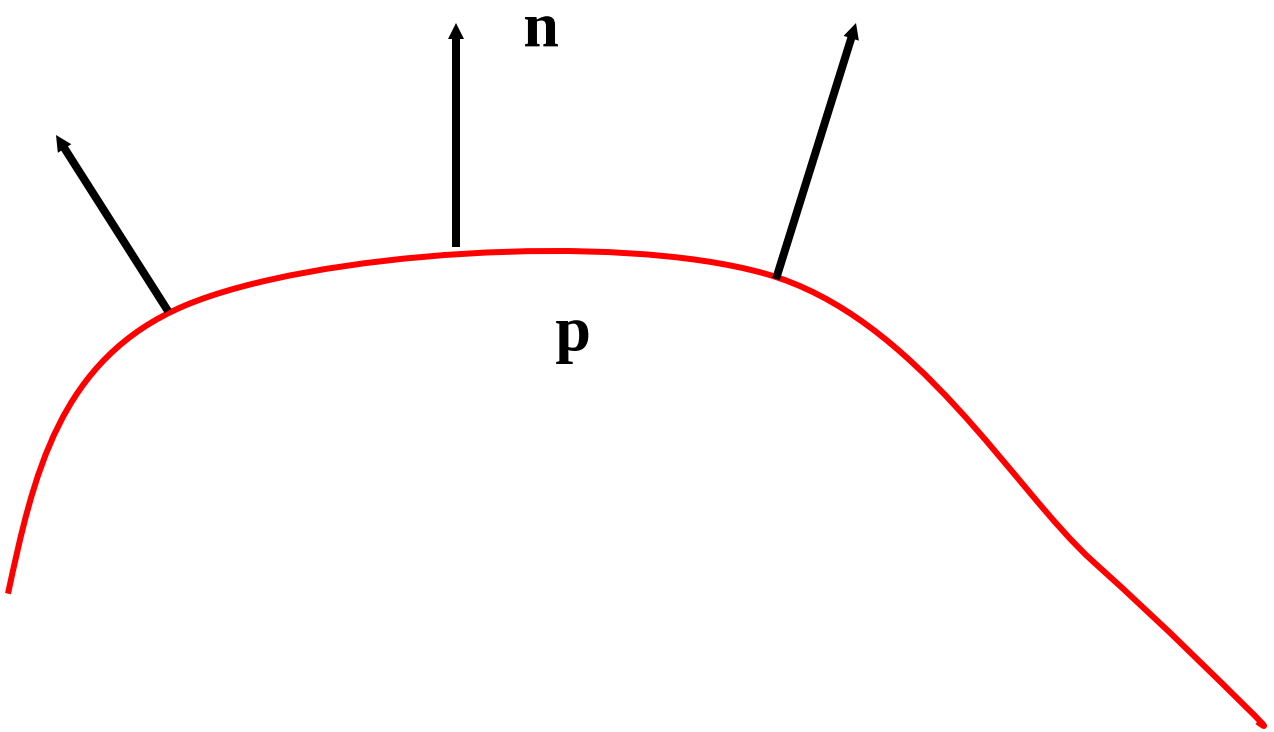
- Add noise to the normal or
- Store perturbation as textures and lookup a perturbation value in a texture map

Bump mapping must be performed in shaders

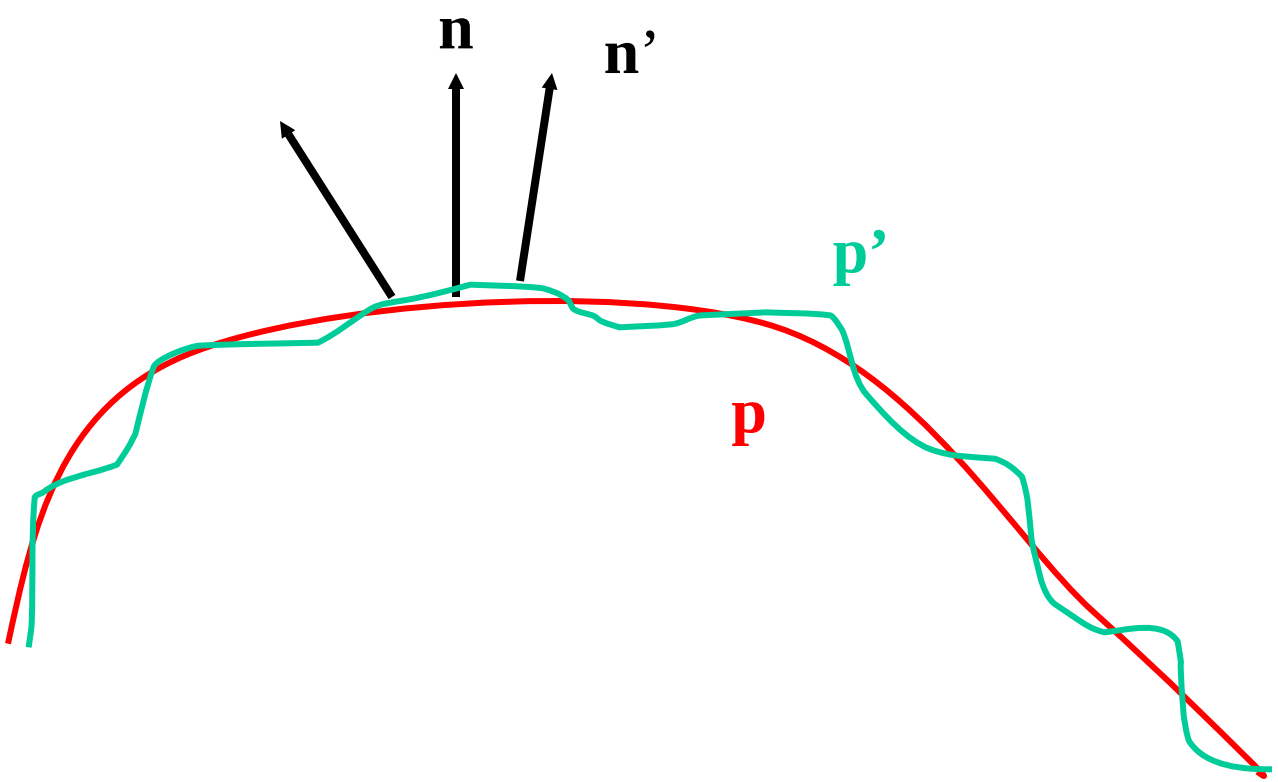


Bump Mapping (Blinn)

Consider a smooth surface



Rougher Version



Finding Bump Maps

Assume a point P on a parametric surface

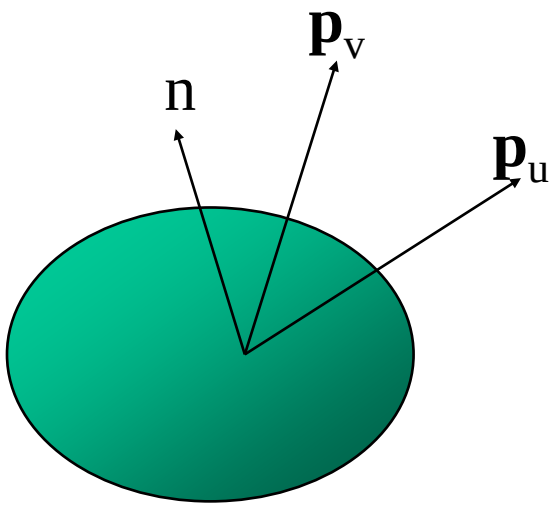
$$\mathbf{p}(u,v) = [x(u,v), y(u,v), z(u,v)]^T$$

The surface normal at point P is

$$\mathbf{n} = (\mathbf{p}_u \times \mathbf{p}_v) / |\mathbf{p}_u \times \mathbf{p}_v|$$

$$\mathbf{p}_u = [\partial x/\partial u, \partial y/\partial u, \partial z/\partial u]^T$$

$$\mathbf{p}_v = [\partial x/\partial v, \partial y/\partial v, \partial z/\partial v]^T$$



$\mathbf{p}_u$ and $\mathbf{p}_v$ determines a tangent plane

Displacement Function

Intuitively, we can create a perturbed point along the normal direction with a displacement *d*

$$\mathbf{p}' = \mathbf{p} + d(u,v)\,\mathbf{n}$$

d(u,v) is the bump or displacement function, $|d(u,v)| << 1$

However, this process is slow

Perturbed Normal

Instead, we can calculate perturbed normal

$$\mathbf{n'} = \mathbf{p'}_u \times \mathbf{p'}_v$$

$$\mathbf{p'}_u = \mathbf{p}_u + (\partial d/\partial u)\mathbf{n} + d(u,v)\mathbf{n}_u$$

$$\mathbf{p'}_v = \mathbf{p}_v + (\partial d/\partial v)\mathbf{n} + d(u,v)\mathbf{n}_v$$

$$\mathbf{n}_u = [\partial nx/\partial u, \partial ny/\partial u, \partial nz/\partial u]^T$$

$$\mathbf{n}_v = [\partial nx/\partial v, \partial ny/\partial v, \partial nz/\partial v]^T$$

If d is small, we can neglect last term

Approximating the Perturbed Normal

$$\mathbf{n'} = \mathbf{p'}_u \times \mathbf{p'}_v$$

$$\approx \mathbf{n} + (\partial d/\partial u)\mathbf{n} \times \mathbf{p}_v + (\partial d/\partial v)\mathbf{n} \times \mathbf{p}_u$$

The vectors $\mathbf{n} \times \mathbf{p}_v$ and $\mathbf{n} \times \mathbf{p}_u$ lie in the tangent plane

→ The normal is displaced in the tangent plane

→ $\mathbf{n'}$, $\mathbf{p'}_u$ and $\mathbf{p'}_v$ form a local coordinate space – ***Tangent Space***

Tangent Space and Normal Matrix

However, n', p'_u and p'_v may be not unit vectors and not orthogonal to each other

Need to get an orthogonal basis

Normalized normal: $\mathbf{m} = \frac{\mathbf{n}'}{|\mathbf{n}'|}$

Tangent vector: $\mathbf{t} = \frac{\mathbf{p}'_u}{|\mathbf{p}'_u|}$

Binormal vector: $\mathbf{b} = \mathbf{m} \times \mathbf{t}$

A transformation matrix is used to transform the view and light to tangent space

$$\mathbf{M} = [\mathbf{t} \quad \mathbf{b} \quad \mathbf{m}]^{\mathbf{t}}$$

Normal Maps

Suppose that we start with a function d(u,v)

We can sample it to form an array $D=[d_{ij}]$

Then $\partial d/\partial u \approx d_{ij} - d_{i-1,j}$

and $\partial d/\partial v \approx d_{ij} - d_{i,j-1}$

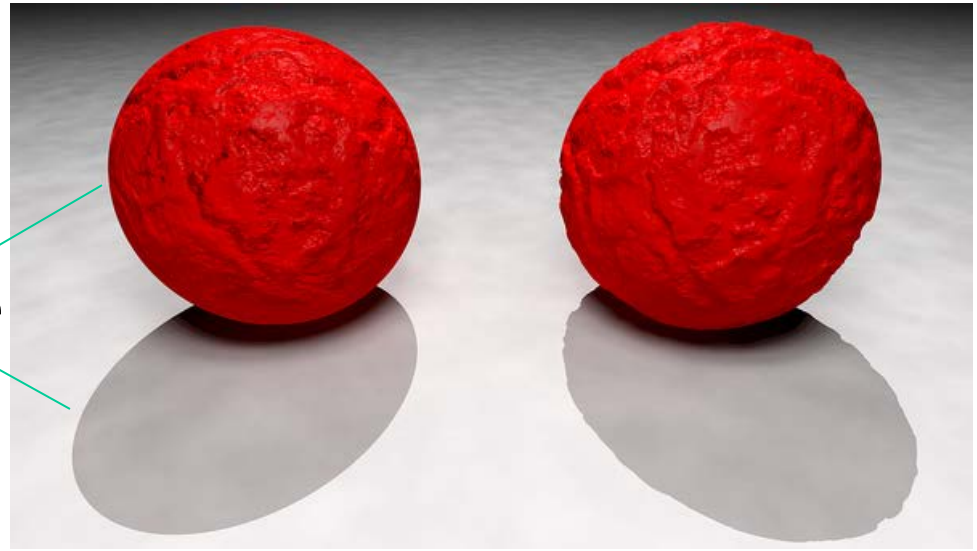
The arrays $\partial d/\partial u$ and $\partial d/\partial v$ can be precomputed and stored as a texture called ***normal map*** used by fragment shader with a smapler.

Example of Normal Maps

Bump Mapping vs Geometric Model

Which one is from bump mapping?

Bump mapping does not modify the shape of the object

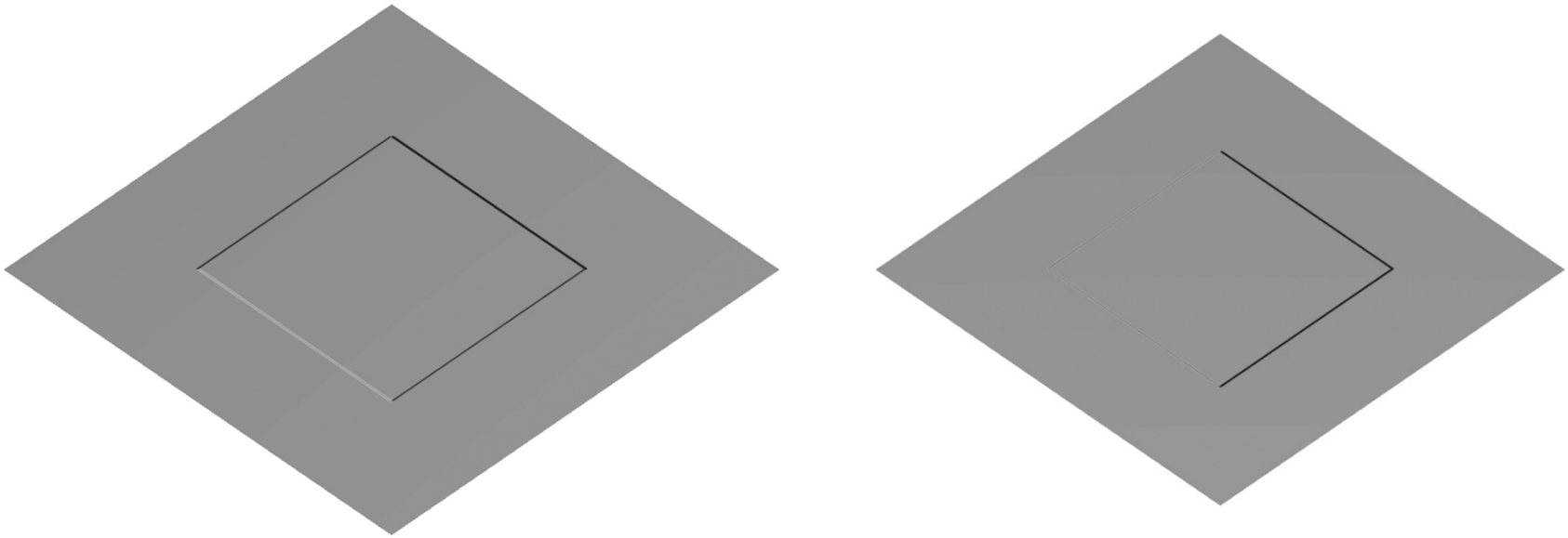


Bump mapping Geometric model

https://en.wikipedia.org/wiki/Bump_mapping

Example

Single Polygon and a Rotating Light Source



How to do this?

The problem is that we want to apply the perturbation at all points on the surface

Cannot solve by vertex lighting (unless polygons are very small)

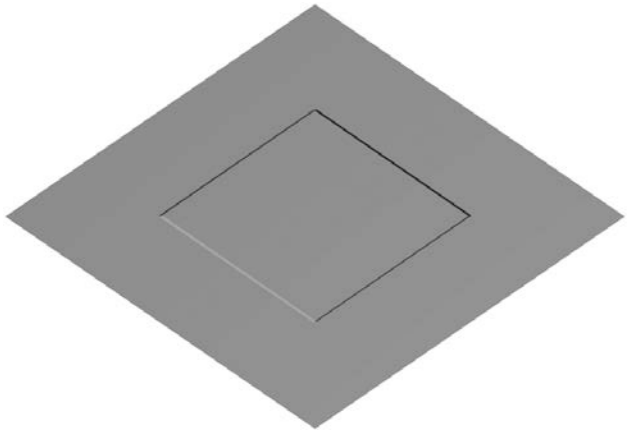
Really want to apply to every fragment in a fragment shader

Angel's Example of Applying Bump Mapping on A Simple Polygon

```
// Define the object – the simple polygon
point4 points[6];
point2 tex_coord[6];
void mesh() // the polygon is formed by two triangles
{
  point4 vertices[4] = { point4(0.0, 0.0, 0.0, 1.0),
                         point4(1.0, 0.0, 0.0, 1.0),
                         point4(1.0, 0.0, 1.0, 1.0),
                         point4(0.0, 0.0, 1.0, 1.0)
                       };
  points[0] = vertices[0]; tex_coord[0] = point2(0.0, 0.0);
  points[1] = vertices[1]; tex_coord[1] = point2(1.0, 0.0);
  points[2] = vertices[2]; tex_coord[2] = point2(1.0, 1.0);
  points[3] = vertices[2]; tex_coord[3] = point2(1.0, 1.0);
  points[4] = vertices[3]; tex_coord[4] = point2(0.0, 1.0);
  points[5] = vertices[0]; tex_coord[5] = point2(0.0, 0.0);
}
```

Angel's Example of Applying Bump Mapping on A Simple Polygon

```
// Generate the displacement – a small square in the center
const int N = 256;
float data[N+1][N+1];
vec3 normals[N][N];
for(int i = 0; i < N+1; i++)
    for(int j = 0; j < N+1; j++)
            data[i][j]=0.0;
for(int i = N/4; i < 3*N/4; i++)
    for(int j = N/4; j < 3*N/4; j++)
            data[i][j] = 1.0;
```



```
// Generate the Normal map
for(int i = 0;i < N; i++)
    for(int j = 0;j < N; j++)
    {
            vec4 n = vec3(data[i][j] - data[i+1][j], 0.0, data[i][j] -data[i][j+1]);
            normals[i][j] = 0.5*normalize(n) + 0.5;
    }
```

Angel's Example of Applying Bump Mapping on A Simple Polygon

The surface normal of a flat polygon is a constant → the tangent vector is constant and can be any vectors on the plane of the polygon

We need a ***normal matrix***, which is the inverse transpose of the upper-left 3x3 submatrix of the model view matrix

The normal matrix transforms a vector in the object frame to the eye frame and can be precomputed.

Angel's Example of Applying Bump Mapping on A Simple Polygon – Vertex Shader

```
in vec2 texcoord;
in vec4 vPosition;

uniform vec3 Normal;
uniform vec4 LightPosition;
uniform mat4 ModelView;
uniform mat4 Projection;
uniform mat4 NormalMatrix;
uniform vec3 objTangent; // tangent vector

out vec3 L; /* light vector in texture-space coordinates */
out vec3 V; /* view vector in texture-space coordinates */
out vec2 st; /* texture coordinates */
```

Angel's Example of Applying Bump Mapping on A Simple Polygon – Vertex Shader

```
void main()
{
    gl_Position = Projection*ModelView*vPosition;
    st = texcoord;

    vec3 eyePosition = vec3(ModelView*vPosition);
    vec3 eyeLightPos = LightPosition.xyz;

    /* normal, tangent, and binormal in eye coordinates */
    vec3 N = normalize(NormalMatrix*Normal);
    vec3 T = normalize(NormalMatrix*objTangent);
    vec3 B = cross(N, T);

   /* Change the light vector to the tangent space */
   L.x = dot(T, eyeLightPos-eyePosition);
   L.y = dot(B, eyeLightPos-eyePosition);
   L.z = dot(N, eyeLightPos-eyePosition);
   L = normalize(L);
...
```

Angel's Example of Applying Bump Mapping on A Simple Polygon – Vertex Shader

```
…

/* Change the view vector to the tangent space */

V.x = dot(T, -eyePosition);

V.y = dot(B, -eyePosition);

V.z = dot(N, -eyePosition);

V = normalize(V);

}
```

Angel's Example of Applying Bump Mapping on A Simple Polygon – Fragment Shader

```
in vec3 L;
in vec3 V;
in vec2 st;
uniform vec4 DiffuseProduct;
uniform sampler2D texMap;
void main()
{
    vec4 N = texture(texMap, st);
    vec3 NN = normalize(2.0*N.xyz-1.0);
    vec3 LL = normalize(L);
    float Kd = max(dot(NN.xyz, LL), 0.0);
    gl_FragColor = Kd*DiffuseProduct;
}
```

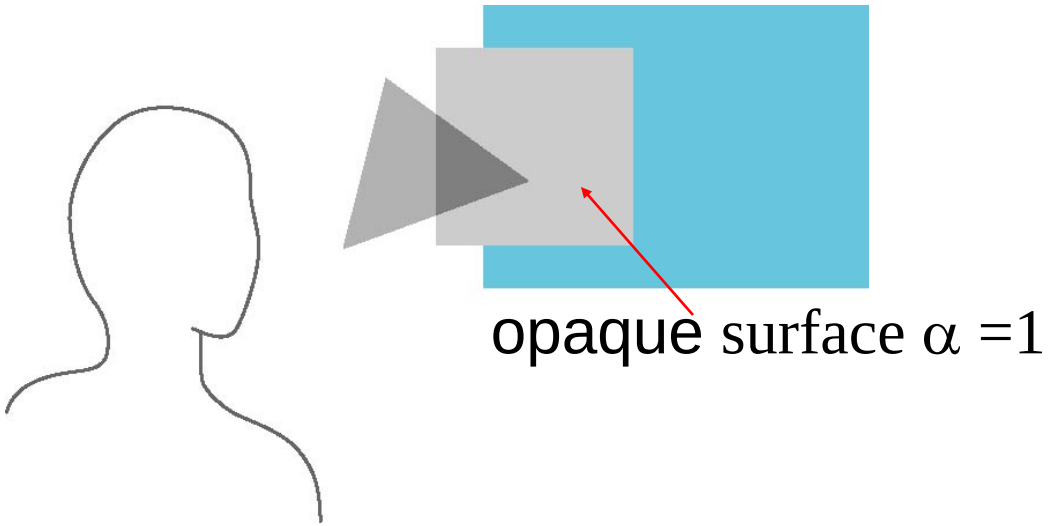
Opacity and Transparency

Opaque surfaces permit no light to pass through

Transparent surfaces permit all light to pass

Translucent surfaces pass some light

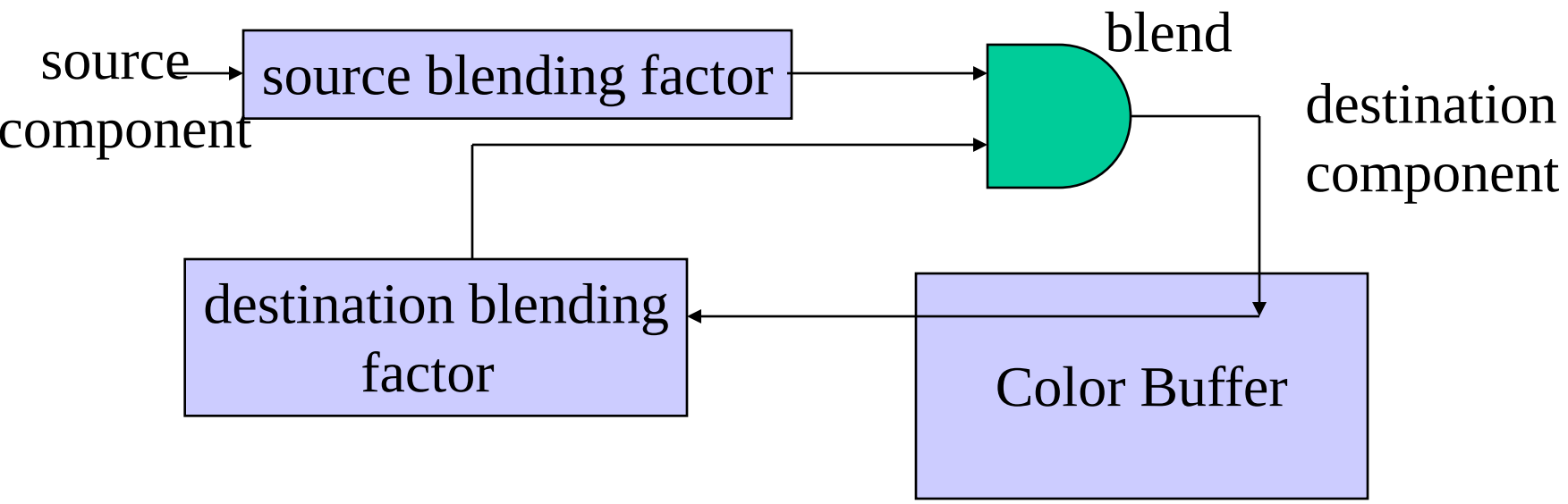
translucency = 1 – opacity (α)



Writing Model

Use A component of RGBA (or RGBα) color to store opacity

During rendering we can expand our writing model to use RGBA values



Blending Equation

We can define source and destination blending factors for each RGBA component

$s = [s_r, s_g, s_b, s_\alpha]$

$d = [d_r, d_g, d_b, d_\alpha]$

Suppose that the source and destination colors are

$b = [b_r, b_g, b_b, b_\alpha]$

$c = [c_r, c_g, c_b, c_\alpha]$

Blend as $c' = [b_r s_r + c_r d_r, b_g s_g + c_g d_g, b_b s_b + c_b d_b, b_\alpha s_\alpha + c_\alpha d_\alpha]$

OpenGL Blending and Compositing

Must enable blending and pick source and destination factors

```
glEnable(GL_BLEND)

glBlendFunc(source_factor,destination_factor)
```

Only certain factors supported, e.g.,

- **GL_ZERO, GL_ONE**
- **GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA**
- **GL_DST_ALPHA, GL_ONE_MINUS_DST_ALPHA**
- See Redbook for complete list

Example

Suppose that we start with the opaque background color $(R_0,G_0,B_0,1)$

- This color becomes the initial destination color

We now want to blend in a translucent polygon with color (R_1,G_1,B_1,α_1)

Select `GL_SRC_ALPHA` and `GL_ONE_MINUS_SRC_ALPHA` as the source and destination blending factors

$$R'_1 = \alpha_1 R_1 + (1- \alpha_1) R_0, \ldots$$ → The composition method discussed earlier

Note this formula is correct if polygon is either opaque or transparent

Clamping and Accuracy

All the components (RGBA) are clamped and stay in the range (0,1)

However, in a typical system, RGBA values are only stored to 8 bits

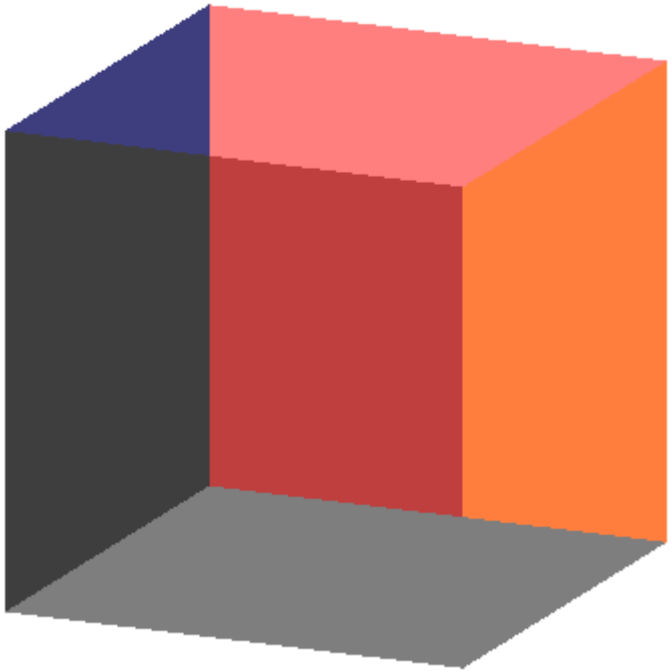
- Can easily lose accuracy if we add many components together
- Example: add together *n* images contributing equally
  - Divide all color components by *n* to avoid clamping
  - Blend with source factor = 1/n, destination factor = 1
  - But division by *n* loses bits

Recent frame buffers supports floating point arithmetic and can avoid the problem

Order Dependency

Is this image correct?

- Probably not
- Polygons are rendered in the order they pass down the pipeline
- Blending functions are order dependent



Opaque and Translucent Polygons

Suppose that we have a group of polygons some of which are opaque and some translucent

How do we use hidden-surface removal?

Opaque polygons block all polygons behind them and affect the depth buffer

Translucent polygons should not affect depth buffer

- Render with `glDepthMask(GL_FALSE)` which makes depth buffer read-only

Sort polygons first to remove order dependency

Fog Effect

We can composite with a fixed color and have the blending factors depend on depth

- Simulates a fog effect

Blend source color C_s and fog color C_f by

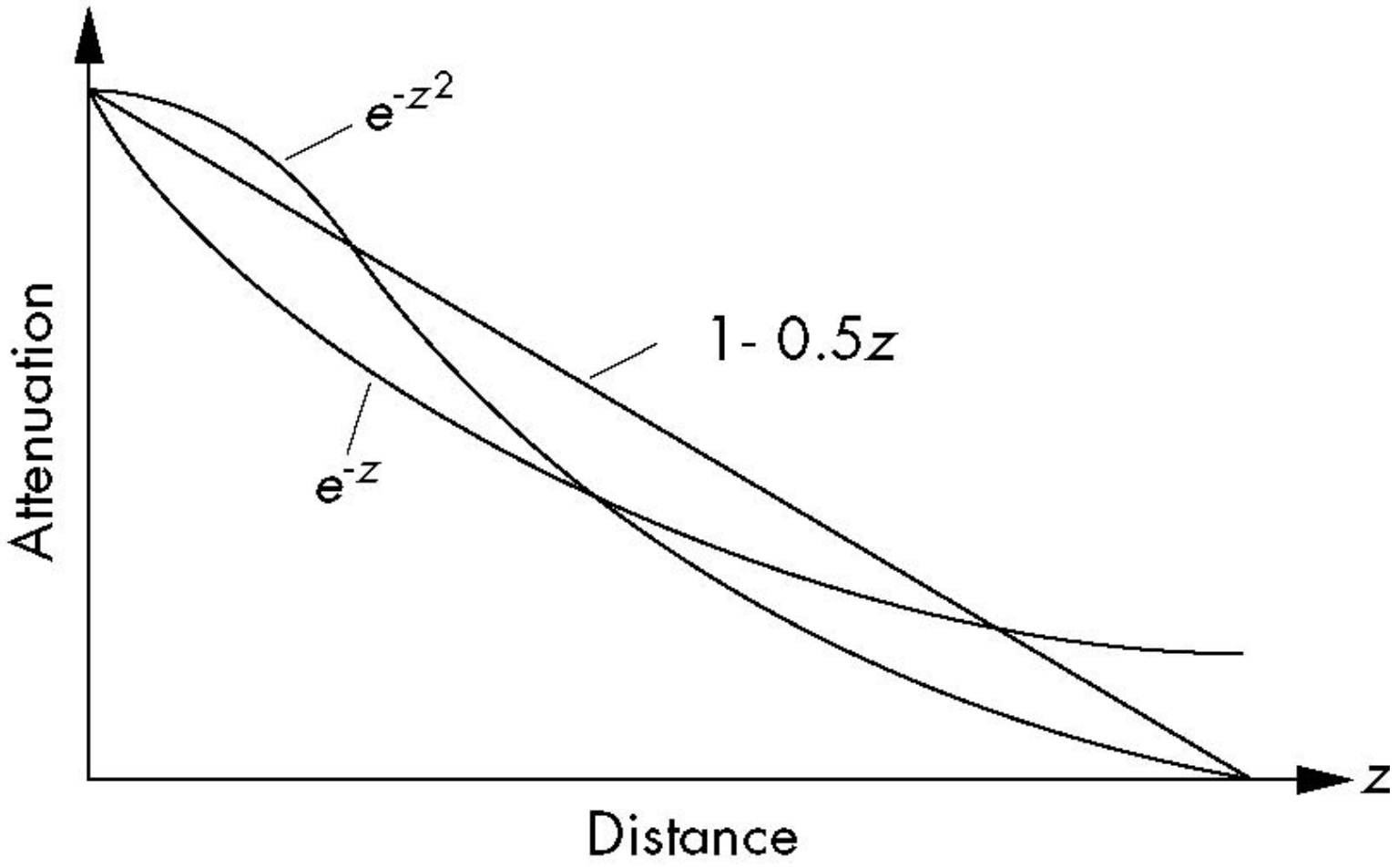
$$C_s' = f\, C_s + (1-f)\, C_f$$

f is the ***fog factor***, which is a function of the depth

- Exponential
- Gaussian
- Linear (depth cueing)

Deprecated but can recreate

Fog Functions



Reading Assignment

Chapter 7.11 and 7.12 in Angel & Shereiner

Chapter 6 and 8 in Shreiner et al: OpenGL Programming Guide (Version 4.3)