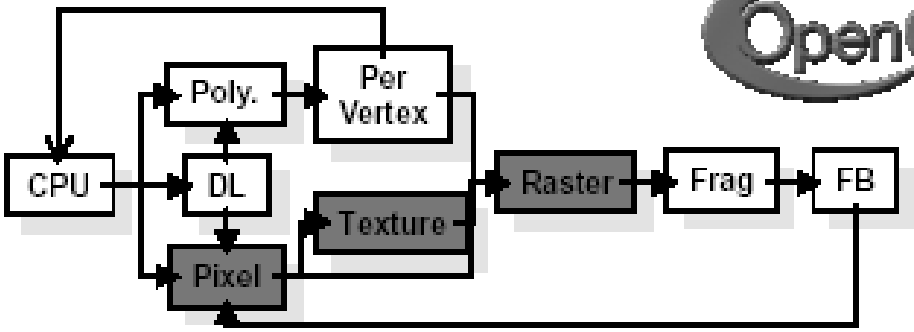
Introduction to OpenGL programming: PART IV

Serena Morigi

A.A.2018/2019

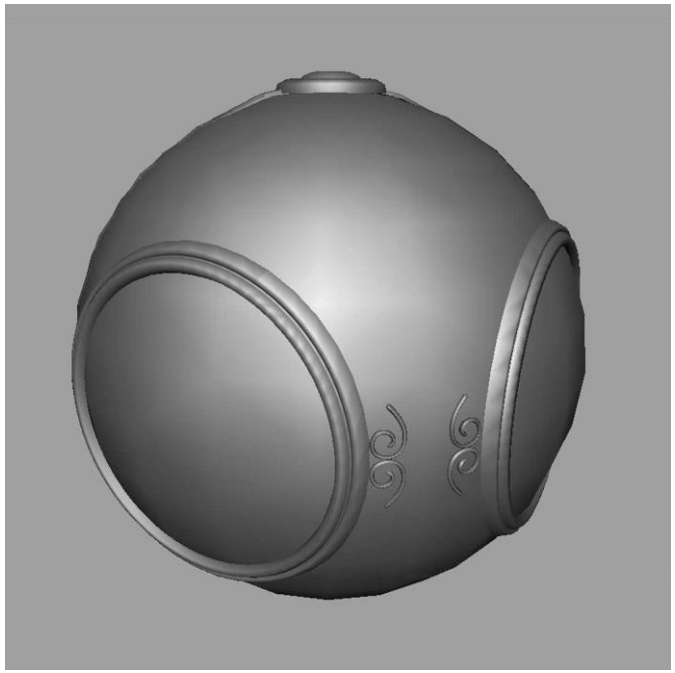
Texture mapping

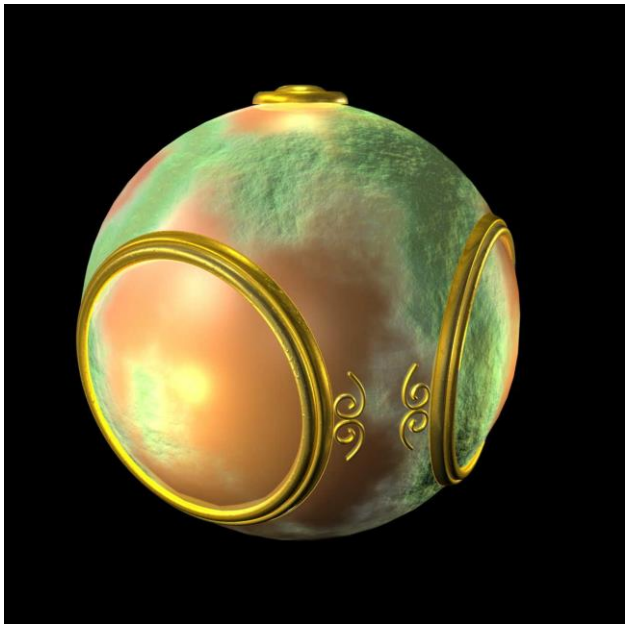


- Apply a 1D,2D or 3D image to a geometric primitive
- Texture mapping is applied at rasterization stage (shading phase)
- Multi-texturing

Use of texture mapping for:

- Simulate different materials
- Reduce the geometric complexity
- Image warping
- Simulate transparency, reflectivity (eg. mirror,..)





Load an image

- Texture images can be stored in several file formats, each with their own structure and ordering of data
- **OpenGL does not read the image formats such as JPEG, PNG, GIF, PPM, ...**

- stb_image.h : is an image-loading library that supports several popular formats.
  - Download the single header file,
  - add it to your project as stb_image.h
  - include stb_image.h in your program ad compile.

```
#define STB_IMAGE_IMPLEMENTATION
#include "stb_image.h"
......
int width, height, nrChannels;
unsigned char *data = stbi_load("pippo.jpg",
                        &width, &height,&nrChannels, 0);
```

Texture mapping

- **Specify a texture map** and how its coordinates relate to those of the objects in your scene
- Control how a texture image is **filtered** as it's applied to a fragment
- Specify **how the color values in the image combine** with those of the fragment to which it's being applied
- Use **automatic texture coordinate generation** to produce effects like contour maps and environment maps

Apply texture: 4 steps

1. Load & Generate the texture
   - Load an image from file or generate it procedurally
   - Generate texture id
   - Load image in texture memory
2. Mapping the texture to the object
   - Specify s,t coordinates for polygon vertices
3. Specify Texture Parameters
   - texture parameters, filtering,wrapping
4. Visualize textured object

1.
Load & Generate the Texture

Load

- Define an array of *texels* (texture elements) in CPU memory

```
Glubyte my_texels[512][512][3];
```

- Load texture image
  - Image file
  - Generate procedurally

  Final result is always an array

  OpenGL supports 1-4 dimensional texture maps

Generate one or more texture objects

Each texture object stores an image texture

(`init()`)

- Generate n texture ids and return their values in *texIds*

```
glGenTextures( n, &texIds );
```

- Create texture object for a texture image

```
glBindTexture(GL_TEXTURE_2D, texId);
```

- Specify Texture Parameters
  - texture parameters, filtering,wrapping

```
glTexParameteri() see later!
```

Load texture image from CPU to texture memory (GPU)

Specify the texture array as a 2D texture

```
glTexImage2D( target, level, components,
              w, h, border, format, type, texels );
```

target:	type of texture
level:	used for mipmapping (0 =>not used)
components:	numbers components (1-4) color (RGBA);
w, h:	width and height of **texels** in pixels
border:	used for smoothing
format, type:	describe the texels
texels:	pointer to the texel array

```
Es: glTexImage2D(GL_TEXTURE_2D, 0, 3, 512, 512, 0,
                 GL_RGB, GL_UNSIGNED_BYTE, my_texels);
```

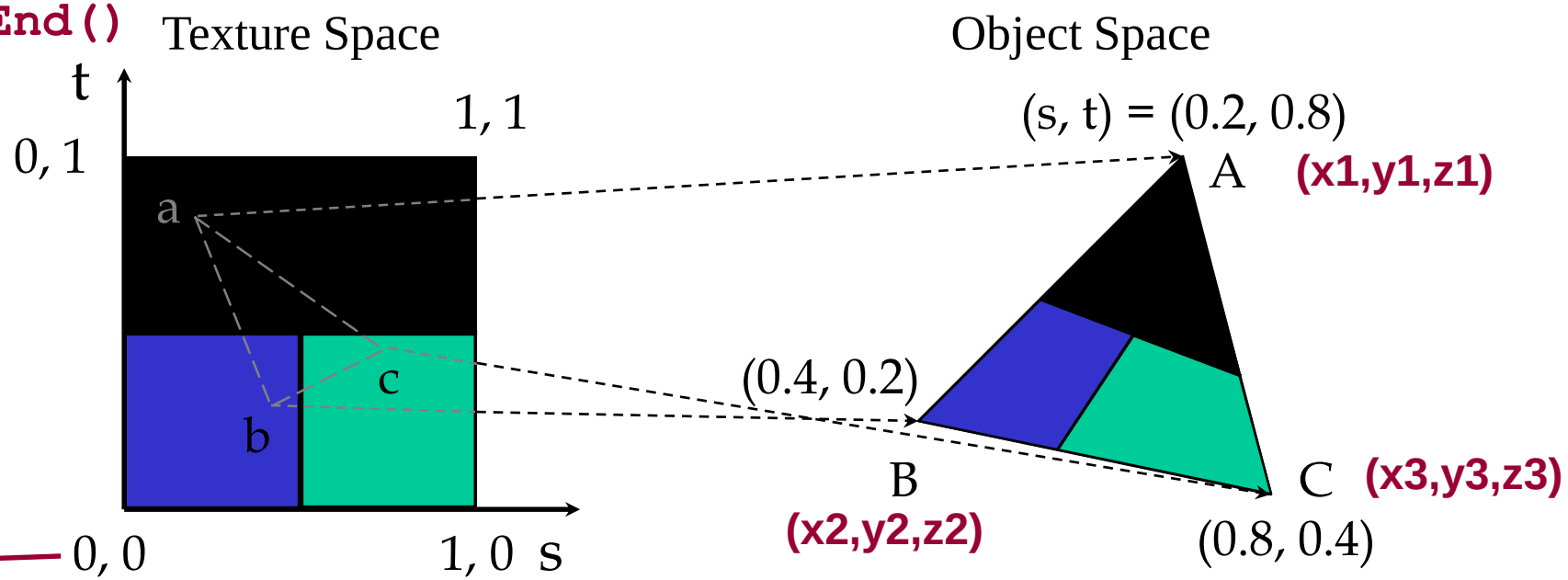
2.
Mapping the texture to the object
(Immediate mode)

Specify how the texture must be applied to the object:

`glTexCoord2f(s,t)`

/* associate a texture point (s,t) to each vertex (x,y,z) of the object */

```
glBegin()
        glTexCoord2f(0.2,0.9);glvertex3f(0.2,0.8,0.0);
        ..............................
glEnd()
```



`my_texel[0][0]`

Example

The texture (below) is a 256 x 256 image that has been mapped to a rectangular polygon which is viewed in perspective.

```
glBindTexture(GL_TEXTURE_2D,texName);
glBegin(GL_QUAD)
   glTexCoord2f(0.0,0.0);glvertex3f(x1,y1,z1);
   glTexCoord2f(1.0,0.0);glvertex3f(x2,y2,z2);
   glTexCoord2f(0.0,1.0);glvertex3f(x3,y3,z3);
   glTexCoord2f(1.0,1.0);glvertex3f(x4,y4,z4);
glEnd()
```

OpenGL uses bilinear interpolation to determine the texture coords of the interior points.
To apply perspective correction:

```
glHint(GL_PERSPECTIVE_CORRECTION,GL_NICEST)
```

2.
Mapping the texture to the object
(with shaders)

Applying Texture to Triangle in Application

```
// add texture coordinate attribute to triangle function

triangle( int a, int b, int c )
{
     vColors[Index] = colors[a];
     vPositions[Index] = positions[a];
     vTexCoords[Index] = vec2( 0.2, 0.8 );
     Index++;

     vColors[Index] = colors[b];
     vPositions[Index] = positions[b];
     vTexCoords[Index] = vec2( 0.4, 0.2 );
     Index++;

     vColors[Index] = colors[c];
     vPositions[Index] = positions[c];
     vTexCoords[Index] = vec2( 0.8, 0.4 );
     Index++;
}
```

Vertex Shader

```
in vec4 vPosition;
in vec4 vColor;
in vec2 vTexCoord;

// Values that stay constant for the whole mesh.
uniform mat4 P;
uniform mat4 V;
uniform mat4 M; // position*rotation*scaling

out vec4 color;
out vec2 texCoord;

void main()
{
    color       = vColor;
    texCoord    = vTexCoord;
    gl_Position = P * V * M * vPosition;
}
```

Fragment Shader

```
in vec4 color;
in vec2 texCoord;
out vec4 fColor;

uniform sampler2D textureBuffer;

void main()
{
//The output is the (filtered) color of the texture at the
(interpolated) texture coordinate.
fColor = texture( textureBuffer, texCoord);
//Alternatively: the output is the mix between the texture
color and the vertex color.
fColor = color * texture( textureBuffer, texCoord);
}
```

In the application..
Initialize Uniform Variable Values

glGetUniformLocation returns an integer that represents the location of a specific uniform variable within a program object

```
P_Matrix_pointer =    glGetUniformLocation(shader_prg_ID,"P");
V_Matrix_pointer =    glGetUniformLocation(shader_prg_ID,"V");
M_Matrix_pointer =    glGetUniformLocation(shader_prg_ID,"M");
```

```
glUniformMatrix4fv(P_Matrix_pointer, 1, GL_FALSE, value_ptr(P));
glUniformMatrix4fv(V_Matrix_pointer, 1, GL_FALSE, value_ptr(V));
glUniformMatrix4fv(V_Matrix_pointer, 1, GL_FALSE, value_ptr(M));
```

number of matrices

NOTE: **glUniform1i** and **glUniform1iv** are the only two functions that may be used to load uniform variables defined as sampler types. Loading samplers with any other function will result in a GL_INVALID_OPERATION error.

Texture Unit 0

```
glUniform1i(glGetUniformLocation(shader_prg_ID,"textureBuffer"), 0);
```

3.
Specify Texture Parameters

Texture Parameters

OpenGL has a variety of parameters that determine how texture is applied

Wrapping parameters determine what happens if s and t are outside the (0,1) range

Filter modes allow us to use area averaging instead of point samples

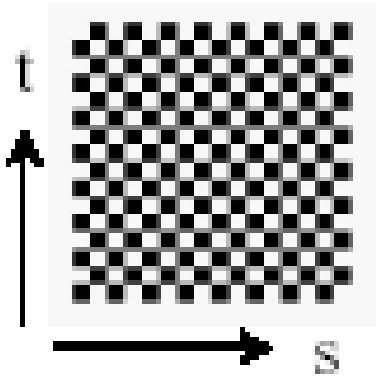
Mipmapping allows us to use textures at multiple resolutions

Blending parameters determine how texture mapping interacts with shading

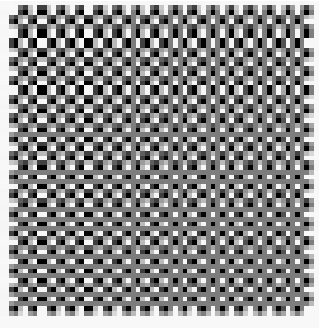
Setting Wrap

Tell OpenGL what to do when the s,t values are not within [0,1]x[0,1] range.

```
glTexParameteri(
   GL_TEXTURE_2D,GL_TEXTURE_WRAP_S,GL_CLAMP )
glTexParameteri(
   GL_TEXTURE_2D,GL_TEXTURE_WRAP_T, GL_REPEAT )
```

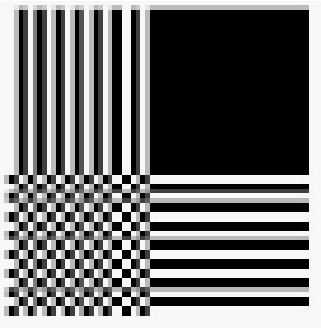


texture



GL_REPEAT

>[0,1]
Uses (s,t mod 1)

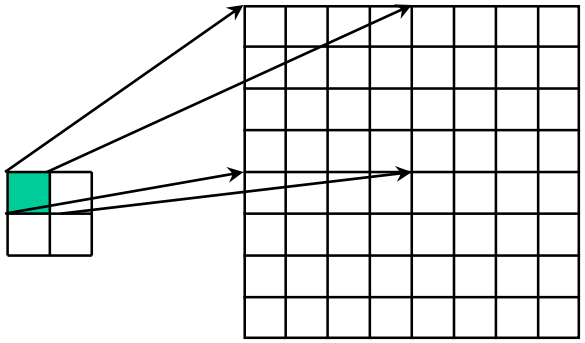


GL_CLAMP

Values >1.0 set 1.0
Values < 0.0 set 0.0

Set the filters for minification/magnification

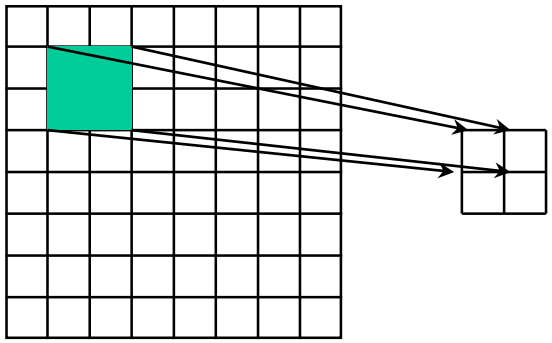
```
glTexParameteri(GL_TEXTURE_2D, type,mode)
```



Texture Polygon

Magnification

More pixels for a texel
Solution: interpolation



Texture Polygon

Minification

More texel in a pixel
Solution: average (procedural image)
Mipmapping (texture image)

Type : GL_TEXTURE_MIN_FILTER or GL_TEXTURE_MAG_FILTER.

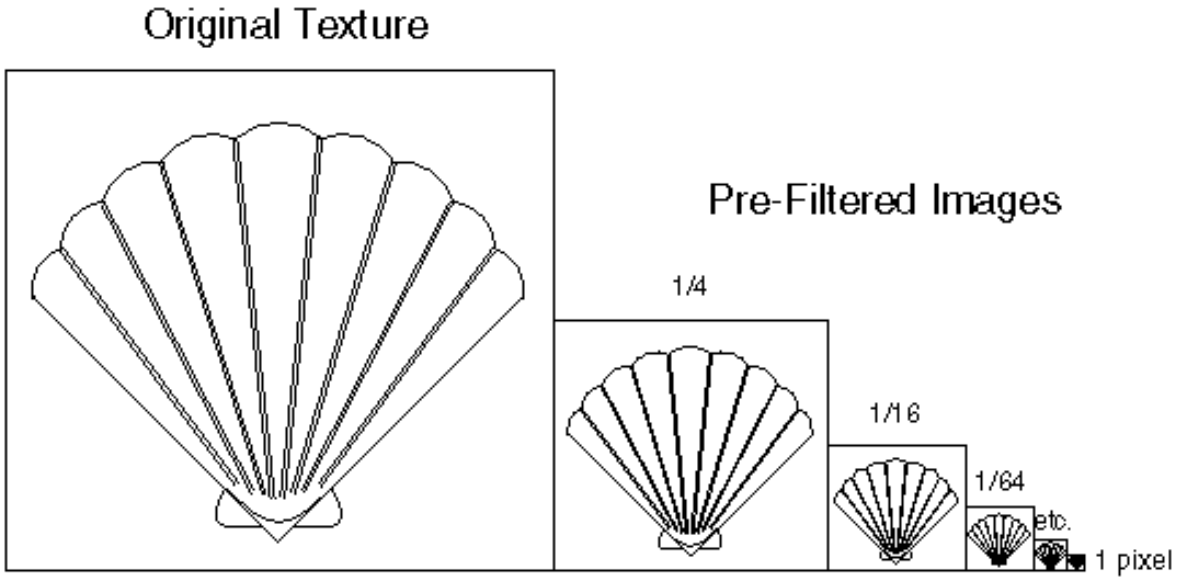
Mode: GL_NEAREST, GL_LINEAR (filter2x2), or other mipmap options (GL_LINEAR_MIPMAP_LINEAR)

Setting Mip-Maps

Pre-filtered textures at different levels

- Generates mipmaps for the texture attached to **target** of the active texture: GL_TEXTURE_1D, GL_TEXTURE_2D, GL_TEXTURE_3D, GL_TEXTURE_1D_ARRAY, GL_TEXTURE_2D_ARRAY, GL_TEXTURE_CUBE_MAP, or GL_TEXTURE_CUBE_MAP_ARRAY.
- Create all the texture levels from the original image

`glGenerateMipmap(`GLenum *target*`)`



Combine texture and shading (**only in fixed-function pipeline**)

glTexEnv must be replaced by code in the fragment shader

```
glTexEnv{fi}[v](GL_TEXTURE_ENV,
                GL_TEXTURE_ENV_MODE, param )
```

param:

- **GL_REPLACE:** use only texture color
- **GL_MODULATE (default) :**
  modulates with computed shade
  COMBINE LIGHTING WITH TEXTURING

- **GL_BLEND:** blends with an environmental color:

```
glTexEnv{fi}[v](GL_TEXTURE_ENV,
          GL_TEXTURE_ENV_COLOR,<RGBA vector>)
```

- **GL_DECAL:** if alpha<1 the fragment's color is blended with the texture color in a ratio determined by the texture alpha

4.
Visualize textured object

Select texture image before drawing

(**`display()`**)
makes **`texId`** the current texture object

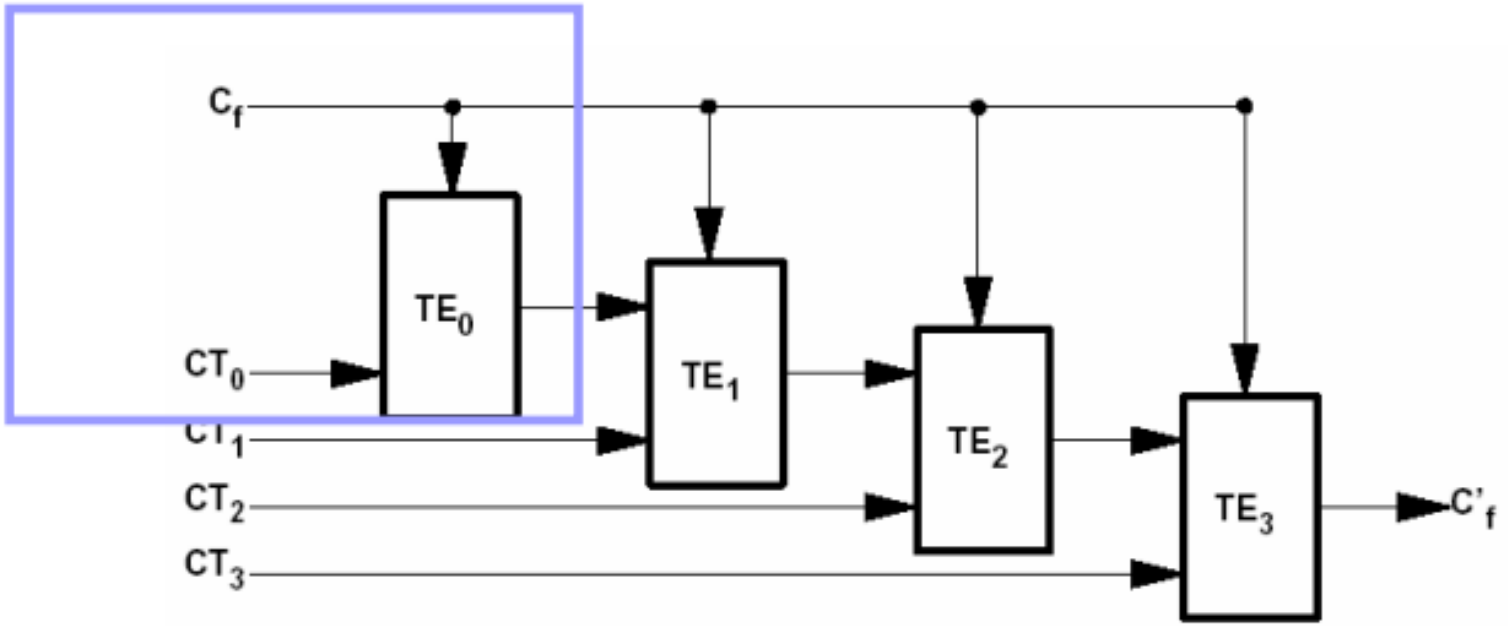
```
glBindTexture(GL_TEXTURE_2D, texId);
```

It is called every time one renders an object with associated texture.

```
glDrawElements(..)
```

Multitexturing

Multitexturing



C_f = fragment primary color input to texturing

C'_f = fragment color output from texturing

CT_i = texture color from texture lookup *i*

TE_i = texture environment *i*

Texture Units

- The main purpose of texture units is to allow us to use more than 1 texture in the shaders

- Activate the texture unit first before binding texture

```
glActiveTexture(GL_TEXTUREi) i=0,..
glBindTexture(GL_TEXTURE_2D, texture);
```

  All the openGL commands will use texture "i" as the texture unit (default unit is 0)

- GLSL's built-in mix function takes two values x,y as input and linearly interpolates between them based on its third argument (a)

```
mix(x,y,a)
```

Example 1: Fragment shader

- Blending between 2 texture based on a alpha value.
- Texture0 is your 1st texture.
- Texture1 is your 2nd texture.
- Texture2 is your alpha mask source.

```
uniform sampler2D Texture0;
uniform sampler2D Texture1;
uniform sampler2D Texture2; //Mask
uniform  vec2 TexCoord0;

void main() {
vec4 texel0, texel1, texel2, resultColor;

texel0 = texture(Texture0, TexCoord0);
texel1 = texture(Texture1, TexCoord0);
texel2 = texture(Texture2, TexCoord0);
resultColor = mix(texel0, texel1, texel2.a);
gl_FragColor = resultColor;
}
```

Example 2: Fragment shader

- Blending between 2 texture based on a alpha value.
- Texture0 is your 1st texture. This is also the source of the alpha mask.
- Texture1 is your 2nd texture.

```
uniform sampler2D Texture0; //Mask
uniform sampler2D Texture1;
uniform  vec2 TexCoord0;

void main() {
vec4 texel0, texel1, resultColor;

texel0 = texture(Texture0, TexCoord0);
texel1 = texture(Texture1, TexCoord0);

resultColor = mix(texel0, texel1, texel0.a);
gl_FragColor = resultColor;
}
```

In the application..generate more texture objects

(`init()`)

1. Generate n texture ids and return their values in *texIds*

```
glGenTextures( n, &texIds );
```

2. For each texture id:

   Create texture object for a texture image

```
glBindTexture(GL_TEXTURE_2D, texId);
```

   Load the image in the texture

```
glTexImage2D(GL_TEXTURE_2D, texId);
glTexParameteri(GL_TEXTURE_2D,..);
```

3. Tell OpenGL to which texture unit each shader sampler belongs to

```
// activate the shader before setting uniforms!

glUniform1i(glGetUniformLocation(ourShader.ID,"Texture0"),0;
glUniform1i(glGetUniformLocation(ourShader.ID,"Texture1"),1;
```

In the application..generate more texture objects

4. select texture image before drawing

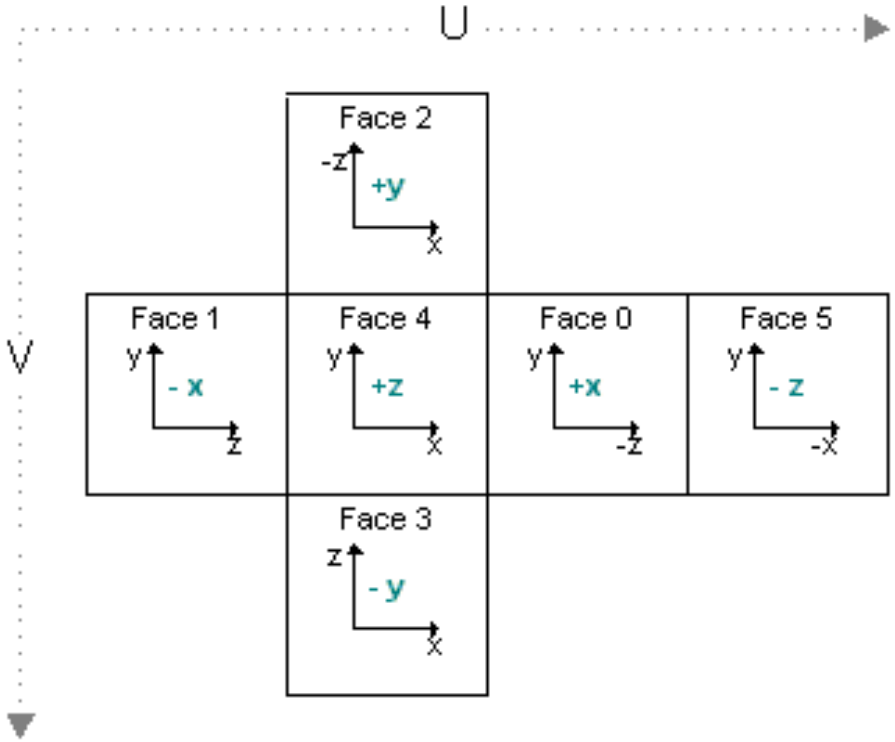
```
(display())
// makes texIds the current texture objects
   glActiveTexture(GL_TEXTURE0);
   glBindTexture(GL_TEXTURE_2D, texId1);
   glActiveTexture(GL_TEXTURE1);
   glBindTexture(GL_TEXTURE_2D, texId2);

   glBindVertexArray(VAO);
   glDrawElements(…)
```

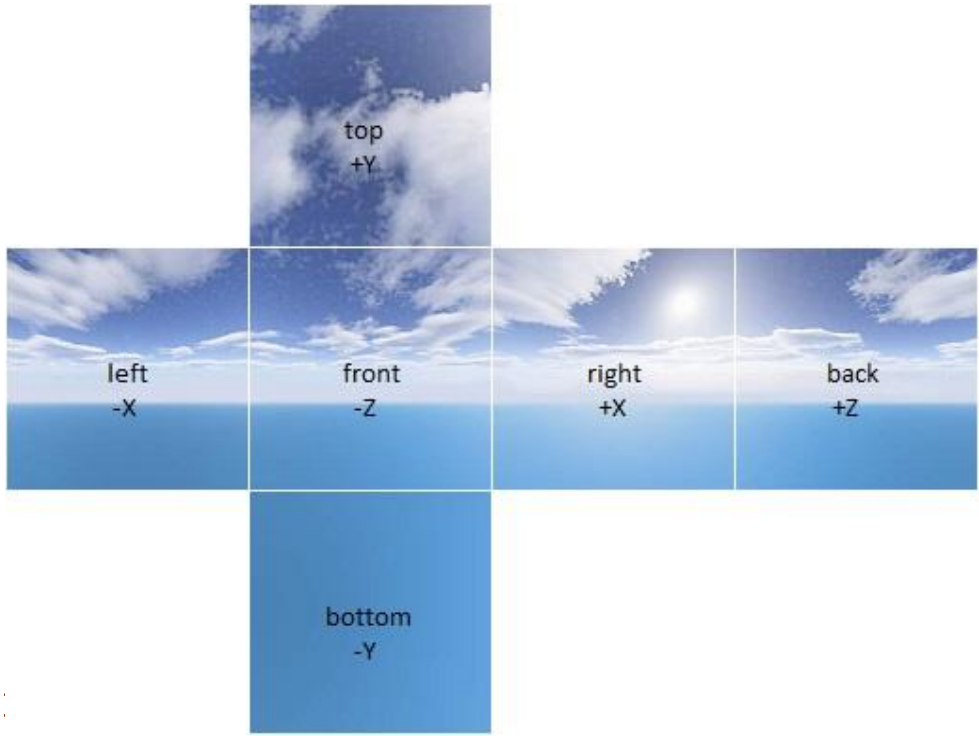
Serena Morigi Dipartimento di Matematica Università di Bologna

Environment Cube Mapping

Create a Cube-Map Texture:

`create_cube_map()` function generates the opengl textures for the 6 separate textures and apply some texture filtering to the texture





1) Creating a cube texture map

Call glTexImage2D once for each face of the cubemap

```
glTexImage2D( GL_TEXTURE_CUBE_MAP_POSITIVE_X, 0, GL_RGBA, w, h,
              0, GL_RGB, GL_UNSIGNED_BYTE, bitmap_data );
```

where texture targets are

Texture target	Orientation
GL_TEXTURE_CUBE_MAP_POSITIVE_X	Right
GL_TEXTURE_CUBE_MAP_NEGATIVE_X	Left
GL_TEXTURE_CUBE_MAP_POSITIVE_Y	Top
GL_TEXTURE_CUBE_MAP_NEGATIVE_Y	Bottom
GL_TEXTURE_CUBE_MAP_POSITIVE_Z	Back
GL_TEXTURE_CUBE_MAP_NEGATIVE_Z	Front

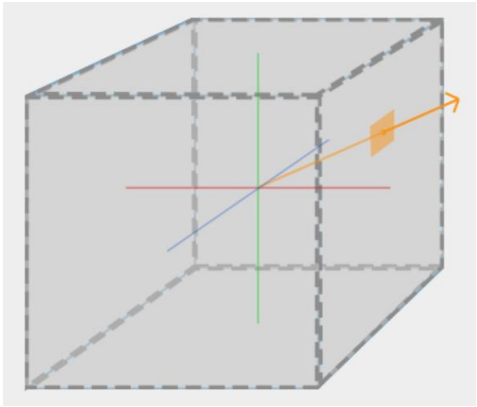
2) Apply filtering

```
glTexParameteri(GL_TEXTURE_CUBE_MAP,GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_CUBE_MAP,GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_CUBE_MAP,GL_TEXTURE_WRAP_S,GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_CUBE_MAP,GL_TEXTURE_WRAP_T,GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_CUBE_MAP,GL_TEXTURE_WRAP_R,GL_CLAMP_TO_EDGE);
```

3) Create texture object

```
static GLuint tex_cubo;
glGenTextures(1, &tex_cubo);
glBindTexture(GL_TEXTURE_CUBE_MAP, tex_cubo);
```

4) Displaying the environment cubemap on a textured cube:



Vertex and Fragment shaders

Because a texture image is drawn on a cube we'll need another VAO, VBO and a set of vertices like any other object.

we set the incoming position vectors as the outcoming texture coordinates for the fragment shader.

Vertex shader

```
layout (location = 0) in vec3 aPos;
out     vec3 TexCoords;
uniform mat4 P;
uniform mat4 V;
void main()
{    TexCoords = aPos;
     gl_Position = P * V * vec4(aPos, 1.0);
}
```

A cubemap used to texture a 3D cube can be sampled using the positions of the cube as the texture coordinates.

When a cube is centered on the origin (0,0,0) each of its position vectors is also a direction vector from the origin. This direction vector is exactly what we need to get the corresponding texture value at that specific cube's position.

We only need to supply position vectors and don't need texture coordinates.

Fragment shader

```
in vec3 textureDir; // direction vector representing a 3D
                                    texture coordinate
uniform samplerCube cubemap; // cubemap texture sampler
void main()
{   FragColor = texture(cubemap, textureDir);
}
```

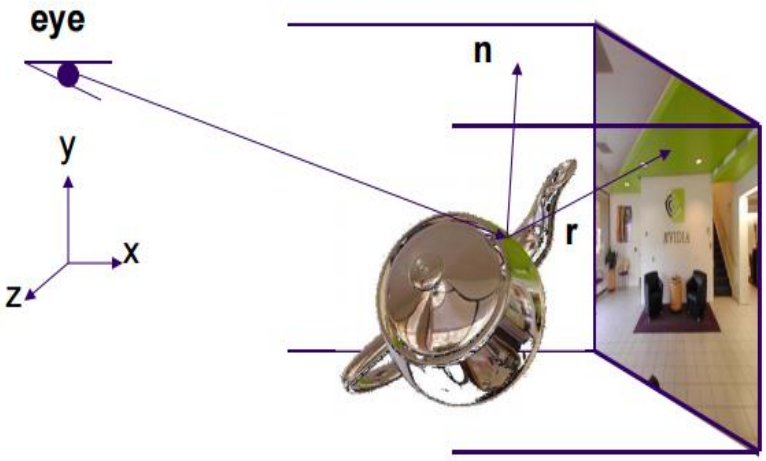
In the application ..
display the environment cubemap

```
glDepthMask(GL_FALSE);
cubemapShader.use();
// ... set view and projection matrix
glBindVertexArray(skyVAO);
glBindTexture(GL_TEXTURE_CUBE_MAP, cubemapTexture);
glDrawArrays(GL_TRIANGLES, 0, 36);
glDepthMask(GL_TRUE);
// ... draw rest of the scene
```

We now have the entire surrounding environment mapped in a single texture object ..

Let's draw the **object** inside with the environment onto it!

Reflection

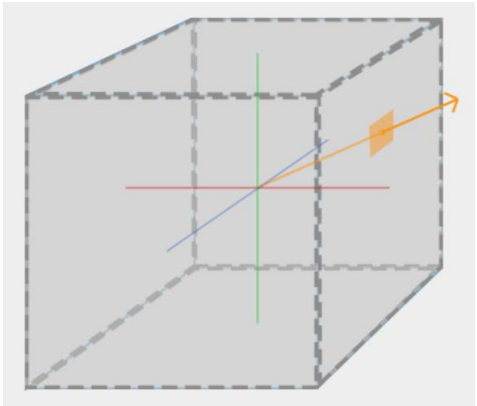


Displaying a reflected object

Calculate a reflection vector R and use that vector to sample from a cubemap:

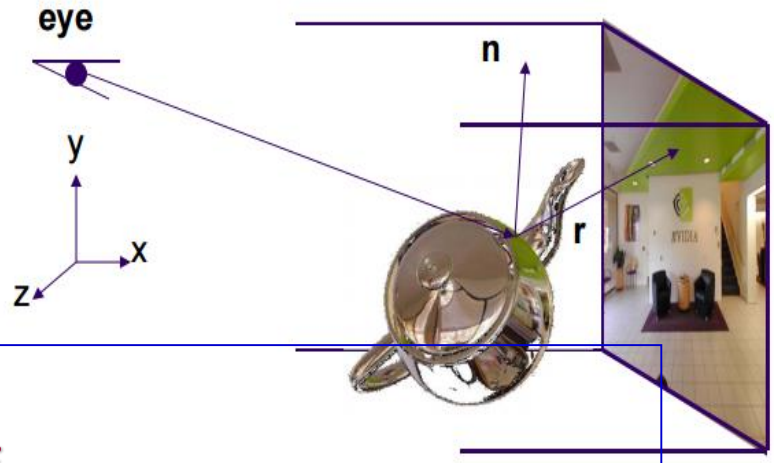
Fragment shader

```
out vec4 FragColor;
in vec3 Normal;
in vec3 Position;
uniform vec3 cameraPos;
uniform samplerCube cubemap;

void main()
{    vec3 I = normalize(Position - cameraPos);
     vec3 R = reflect(I, normalize(Normal));
     FragColor = vec4(texture(cubemap, R).rgb, 1.0);
}
```



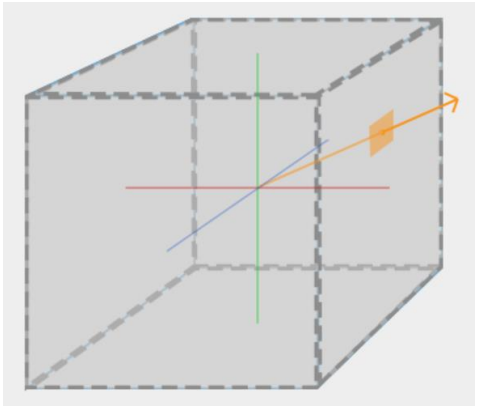
Displaying a reflected object

Calculate a reflection vector and use that vector to sample from a cubemap:



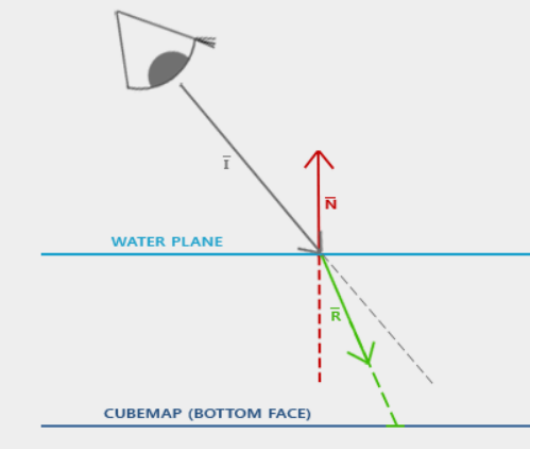
Vertex shader

```
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;
out vec3 Normal;
out vec3 Position;
uniform mat4 M;
uniform mat4 V;
uniform mat4 P;
void main()
{
    Normal = mat3(transpose(inverse(M))) * aNormal;
    Position = vec3(M * vec4(aPos, 1.0)); //in WCS
    gl_Position = P * V * M * vec4(aPos,1.0);
}
```



Refraction

Another form of environment mapping is called refraction and is similar to reflection. The direction of the view vector is slightly bend. This resulting bended vector *R* is then used to sample from the cubemap.



Material	Refractive index
Air	1.00
Water	1.33
Ice	1.309
Glass	1.52
Diamond	2.42

The light/view ray goes from *air* to *glass.*

Apply the Snell's law:

Iif we assume the object is made of glass, so the ratio becomes 1.00/1.52=0.658.

Refraction

Fragment shader

```
out vec4 FragColor;
in vec3 Normal;
in vec3 Position;
uniform vec3 cameraPos;
uniform samplerCube cubemap;

void main()
{    vec3 I = normalize(Position - cameraPos);
     float ratio = 1.00 / 1.52;
     vec3 I = normalize(Position - cameraPos);
     vec3 R = refract(I, normalize(Normal), ratio);
     FragColor = vec4(texture(cubemap, R).rgb, 1.0);
}
```

Vertex shader and cubemap definition in the application as in the reflective object case.

By changing the refractive indices you can create completely different visual results.

BLENDING in Fragment Rendering

Serena Morigi **Dipartimento di Matematica** **Università di Bologna**

Fragment Rendering

- The classic features that take place per fragment include:
  - Test the **zbuffer** to see the pixel should be rendered at all
  - Perform texture lookups (this may involve mipmapping, or bilinear sampling, for example)
  - Multiply the **texture** color with the Gouraud interpolated color (including the alpha component)
  - Blend this result with the **fog** color, based on the pixel z depth
  - Blend this result with the existing color in the framebuffer, based on the **alpha** value

    When drawing a scene with non-transparent and transparent objects the general outline is usually as follows:

    1) Draw all opaque objects first.

    2) Sort all the transparent objects.

    3) Draw all the transparent objects in sorted order.
  - Write this final color, as well as the interpolated z depth into the color framebuffer/zbuffer

- One could certainly do more complex things just using programmable fragment shaders

Fragment output: blending

- The result of the fragment processing stage of the OpenGL Pipeline, whether using Fragment Shaders or not, are color values that have to be written in the color buffers in the current framebuffer.
- **When blending is active**, the input colors from the fragment processor does not merely overwrite the corresponding value currently in the output buffer, but instead the two colors are combined together based on a function.
- Blending happens independently on each fragment shader output value.
- Blending operations take place between a source color and a destination color

  src = fragment color dst = framebuffer color

Fragment output: blending

- Blending state for a buffer has two components:
- The **first** defines the basic equation used for blending

```
glEnable( GL_BLEND )
void glBlendEquation(Glenum mode);
```

mode:

`GL_FUNC_ADD (default)`	$O = src\ \mathrm{s} + dst\ \mathrm{d}$
`GL_FUNC_SUBTRACT`	$O = src\ \mathrm{s} - dst\ \mathrm{d}$
`GL_REVERSE_SUBCTRACT`	$O = dst\ \mathrm{d} - src\ \mathrm{s}$

- it is also possible to set different options for the RGB and alpha channel individually

```
void glBlendEquationSeparate(
        GLenum modeRGB, GLenum modeAlpha );
```

- The **second** defines the blending factors s,d

Fragment output: blending

- Blending factors s,d $O = src\,\mathrm{s} \pm dst\,\mathrm{d}$

O output color, **dst** framebuffer color, **src** fragment color, **s,d** blending factors

```
void glBlendFunc(Glenum s, Glenum d);
```

- it is also possible to set different options for the RGB and alpha channel individually

```
void glBlendFuncSeparate(
        GLenum s_RGB, GLenum s_Alpha,
        GLenum d_RGB, GLenum d_Alpha );
```

Common factors: GL_ZERO, GL_UNO, GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA, GL_DST_ALPHA, GL_ONE_MINUS_DST_ALPHA

Examples

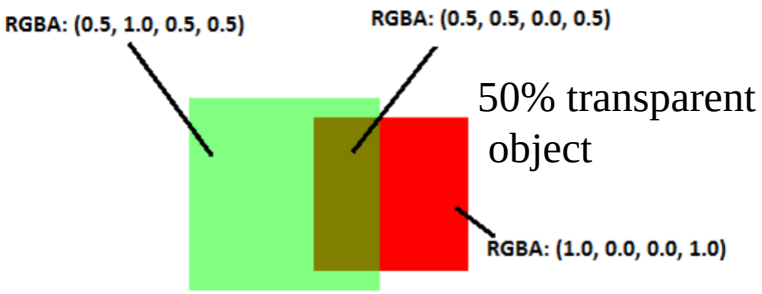
Opaque object: (default) The object is opaque: the fragment color must replace the framebuffer color.

$$O = src\,1 + dst\,0$$

```
glBlendEquation( GL_FUNC_ADD );
glBlendFunc( GL_ONE,GL_ZERO );
```

Semi-Transparent object: the fragment color must be mixed with the framebuffer color. The resulting color is then stored in the color framebuffer, replacing the previous color.

$$O = src\,\alpha + dst\,(1-\alpha)$$



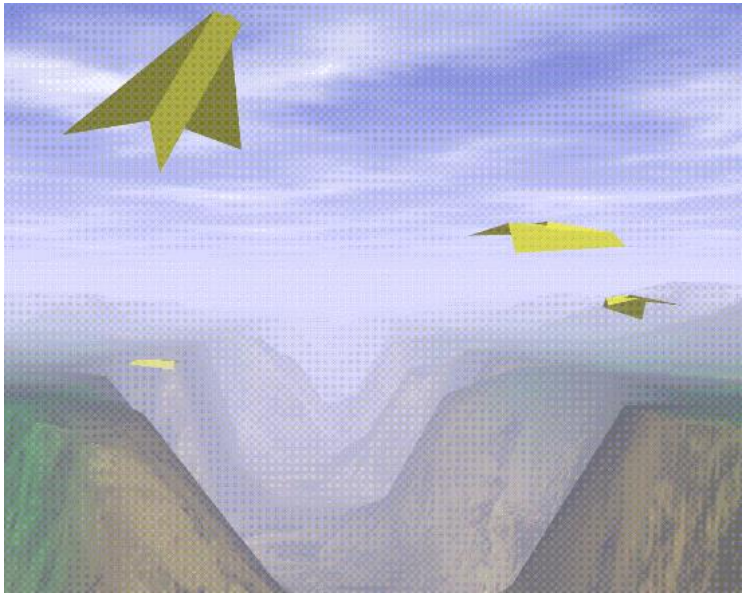

```
glBlendEquation( GL_FUNC_ADD);
glBlendFunc(GL_SRC_ALPHA,GL_ONE_MINUS_SRC_ALPHA );
```

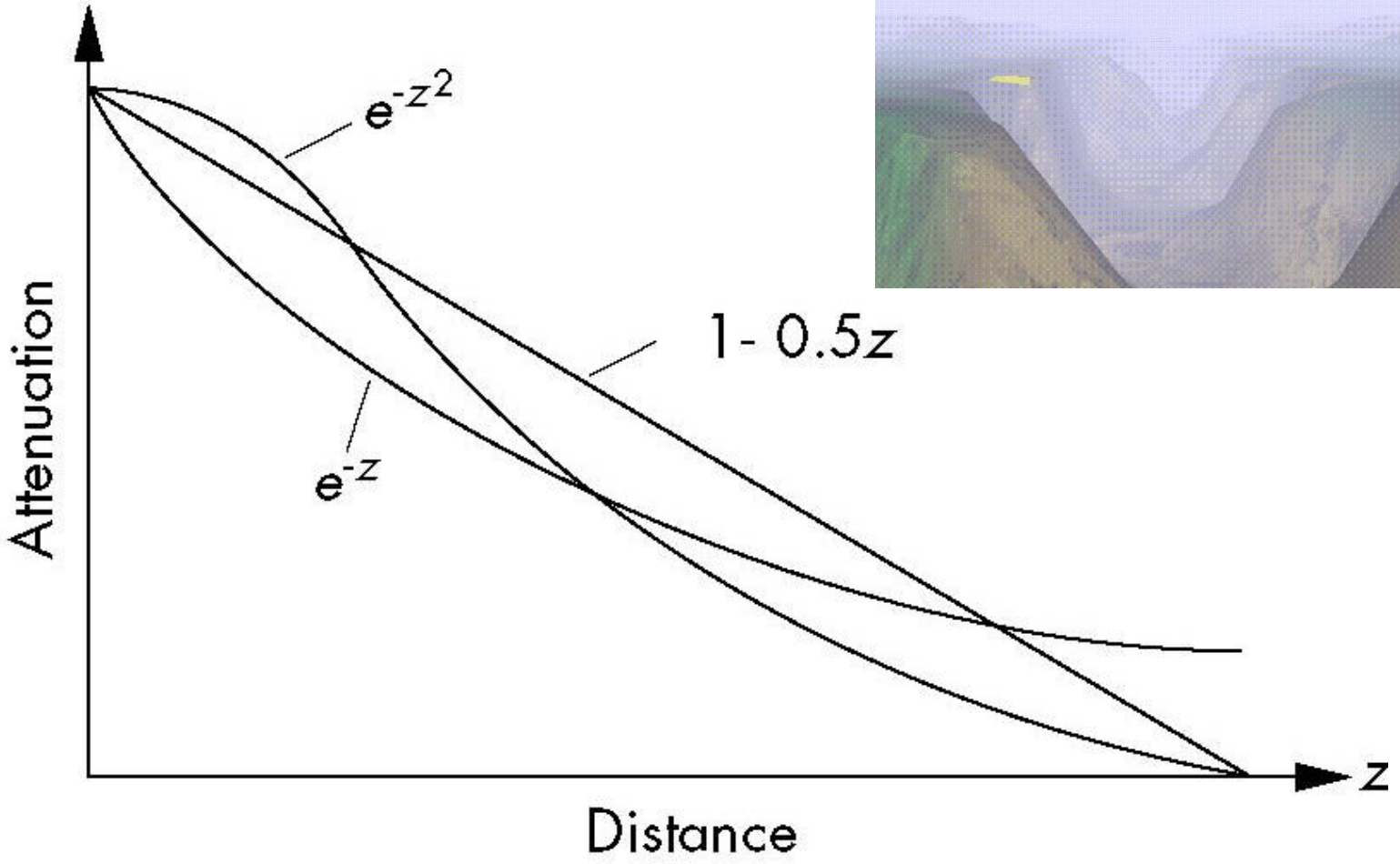
Fog

`glFog{if}(` ***`property, value`*** `)`

- Depth Cueing
  - Specify a range for a linear fog ramp
    - **GL_FOG_LINEAR**
- Environmental effects
  - Simulate more realistic fog
    - **GL_FOG_EXP**
    - **GL_FOG_EXP2**

Fog Functions





Linear Fog mode

Provide OpenGL with a starting and ending distance from the eye, and between those distances, the fog color is blended into the primitive in a linear manner based on distance from the eye.

The fog coefficient is computed as $$f = \frac{z - start}{end - start}$$

```
glFogf(GL_FOG_MODE, GL_FOG_LINEAR);
glFogf(GL_FOG_START, fogStart);
glFogf(GL_FOG_END, fogEnd);
glFogfv(GL_FOG_COLOR, fogColor);
glEnable(GL_FOG);
```