OPENGL BASICS: PIPELINE, TEXTURING

Pere-Pau Vázquez

MOVING Group – UPC

Contents

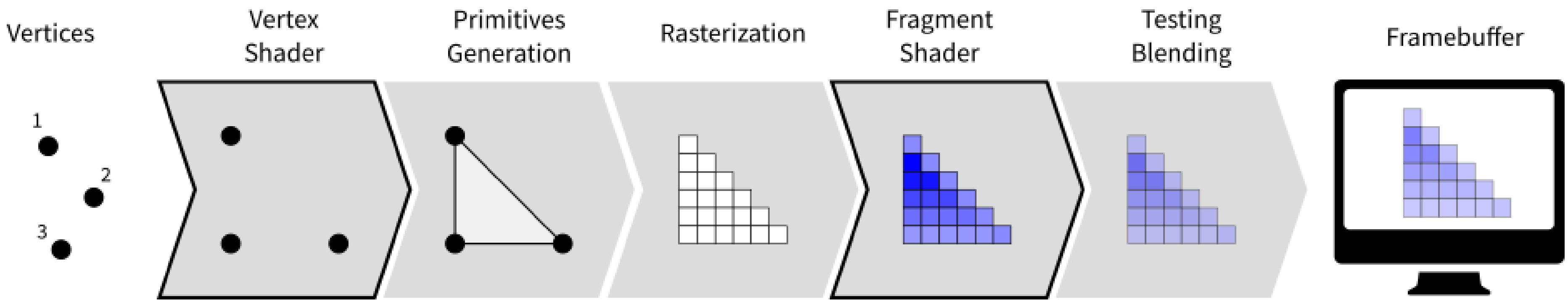
- Basics
- OpenGL pipeline
- Simple texturing
- Advanced texturing
- Environment mapping

Contents

- **Basics**
- OpenGL pipeline
- Simple texturing
- Advanced texturing
- Environment mapping

Basics

- Pipeline overview



Basics

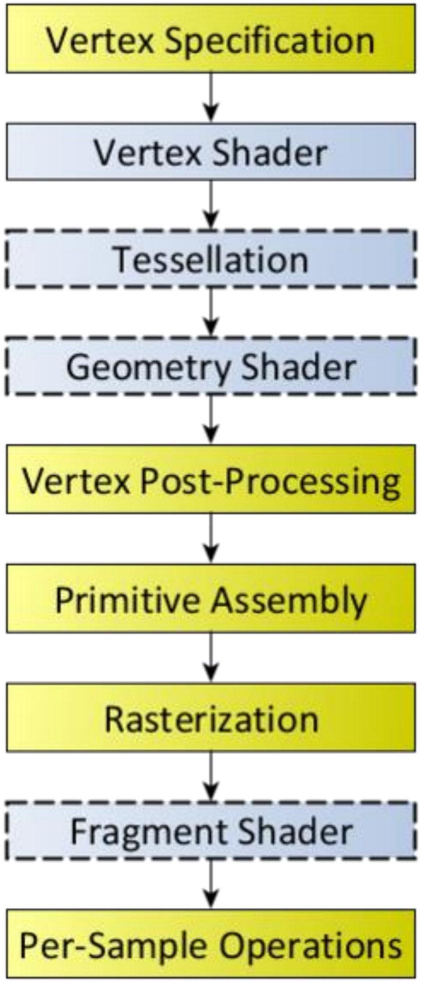
- Vertex shader must output (at least) the vertex transformed position

```glsl
void main()
{
    gl_Position = vec4(0.0,0.0,0.0,1.0);
}
```

- Fragment shader must output (at least) the fragment color

```glsl
void main()
{
    gl_FragColor = vec4(0.0,0.0,0.0,1.0);
}
```

Basics



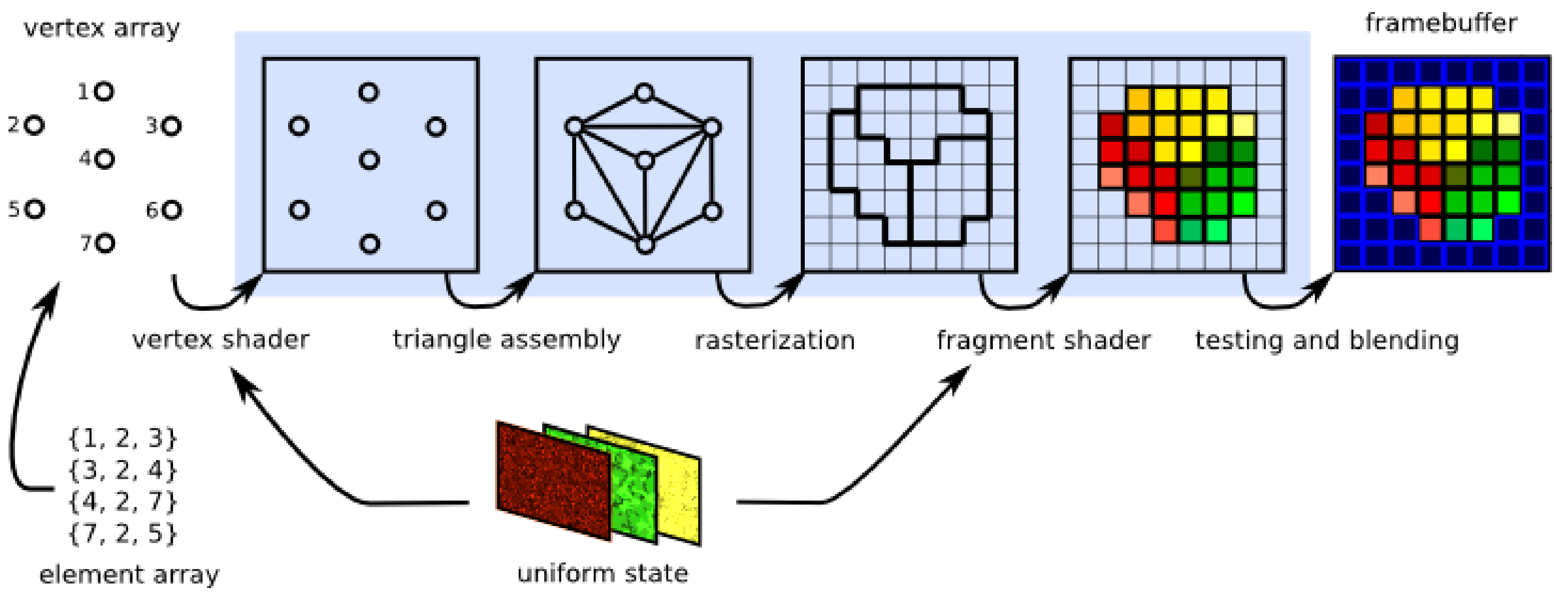
Vertex Specification
Vertex Shader
Tessellation
Geometry Shader
Vertex Post-Processing
Primitive Assembly
Rasterization
Fragment Shader
Per-Sample Operations

Contents

- *Basics*
- **OpenGL pipeline**
- Simple texturing
- Advanced texturing
- Environment mapping

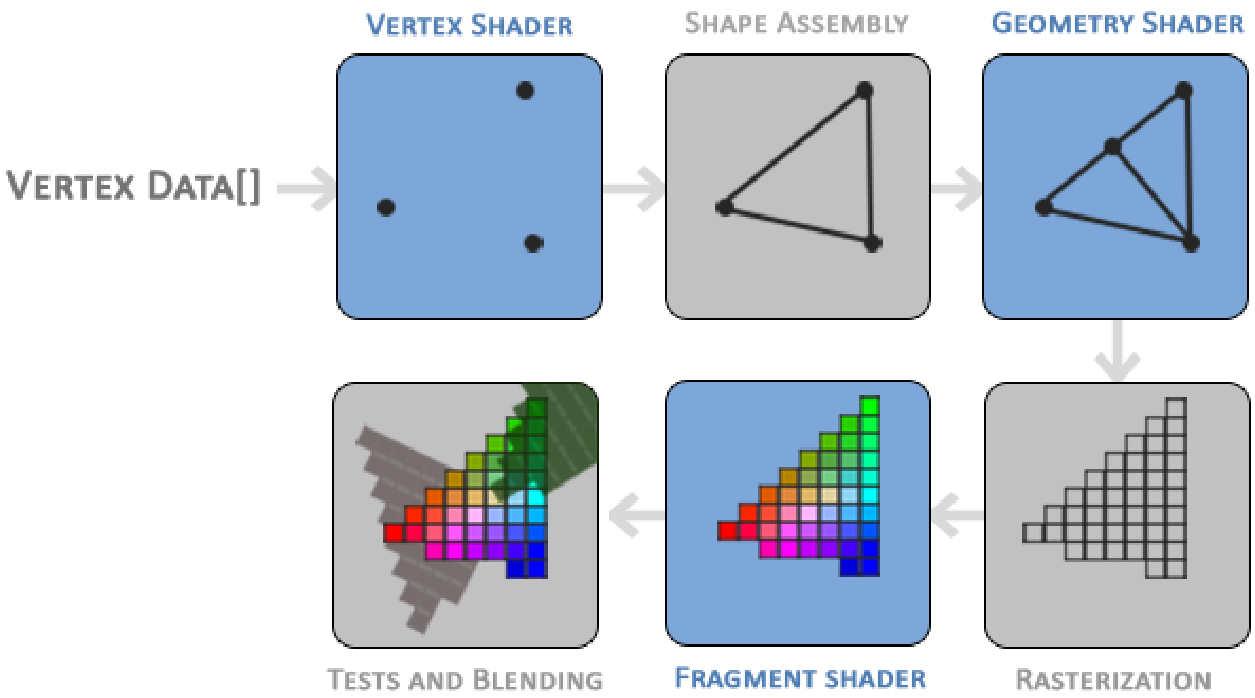
OpenGL pipeline

- OpenGL rendering pipeline

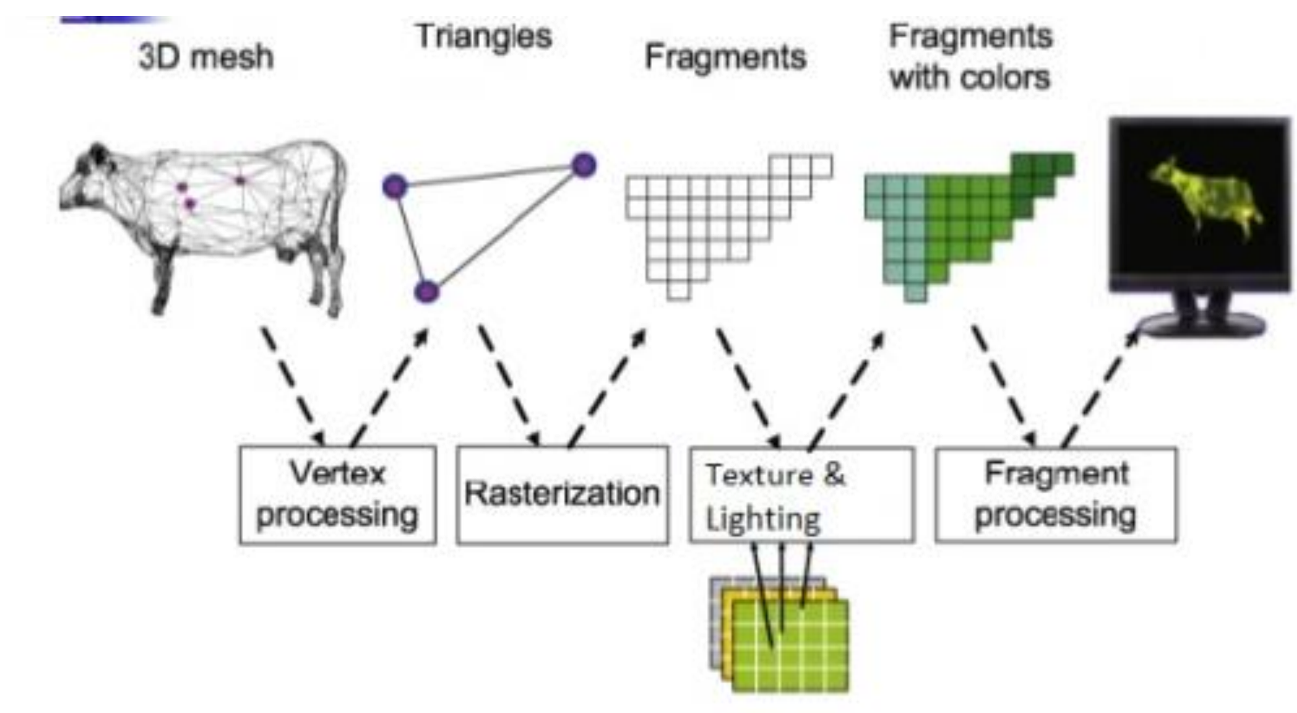


OpenGL pipeline – overview

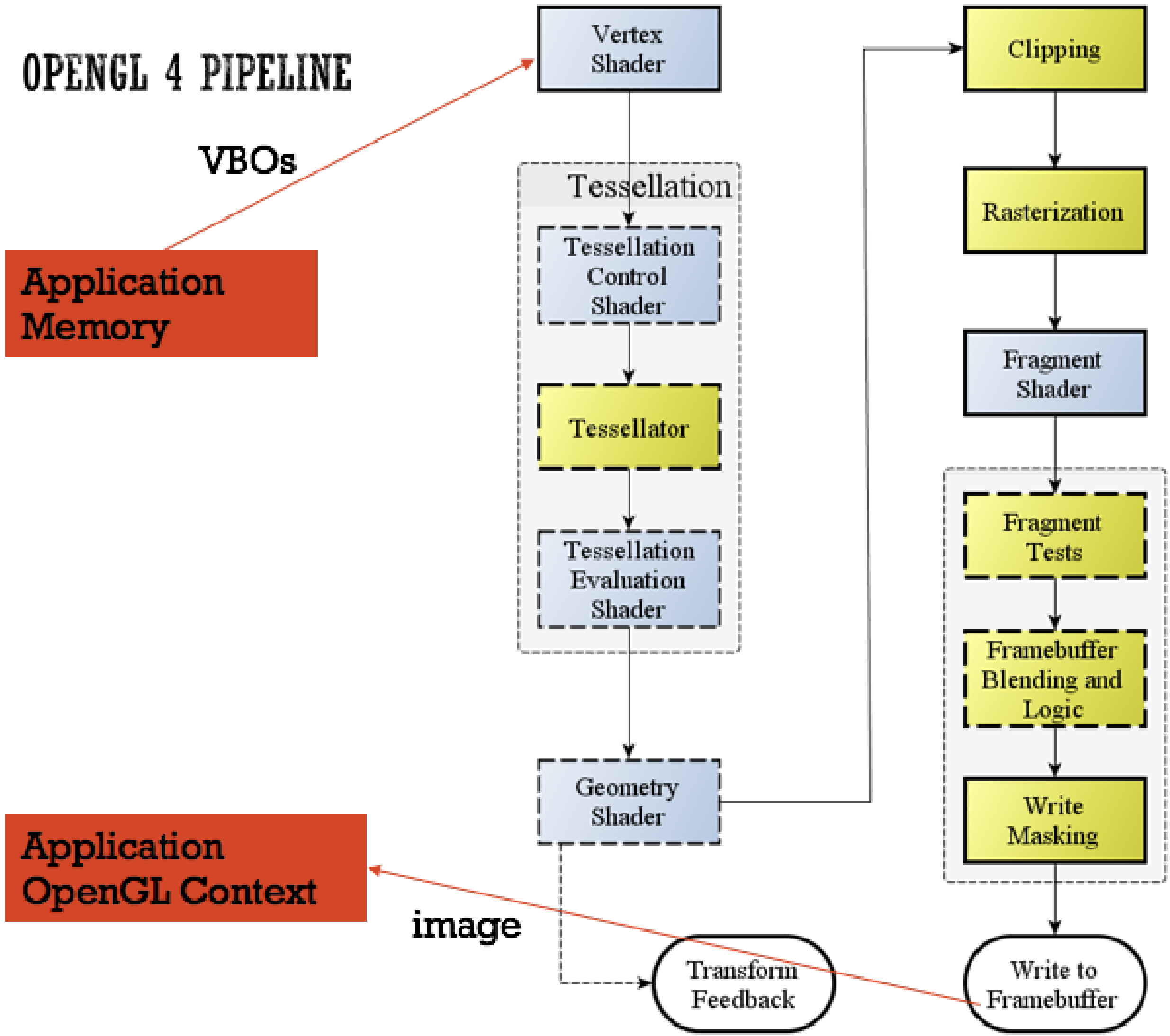
- OpenGL rendering pipeline



OpenGL pipeline – overview

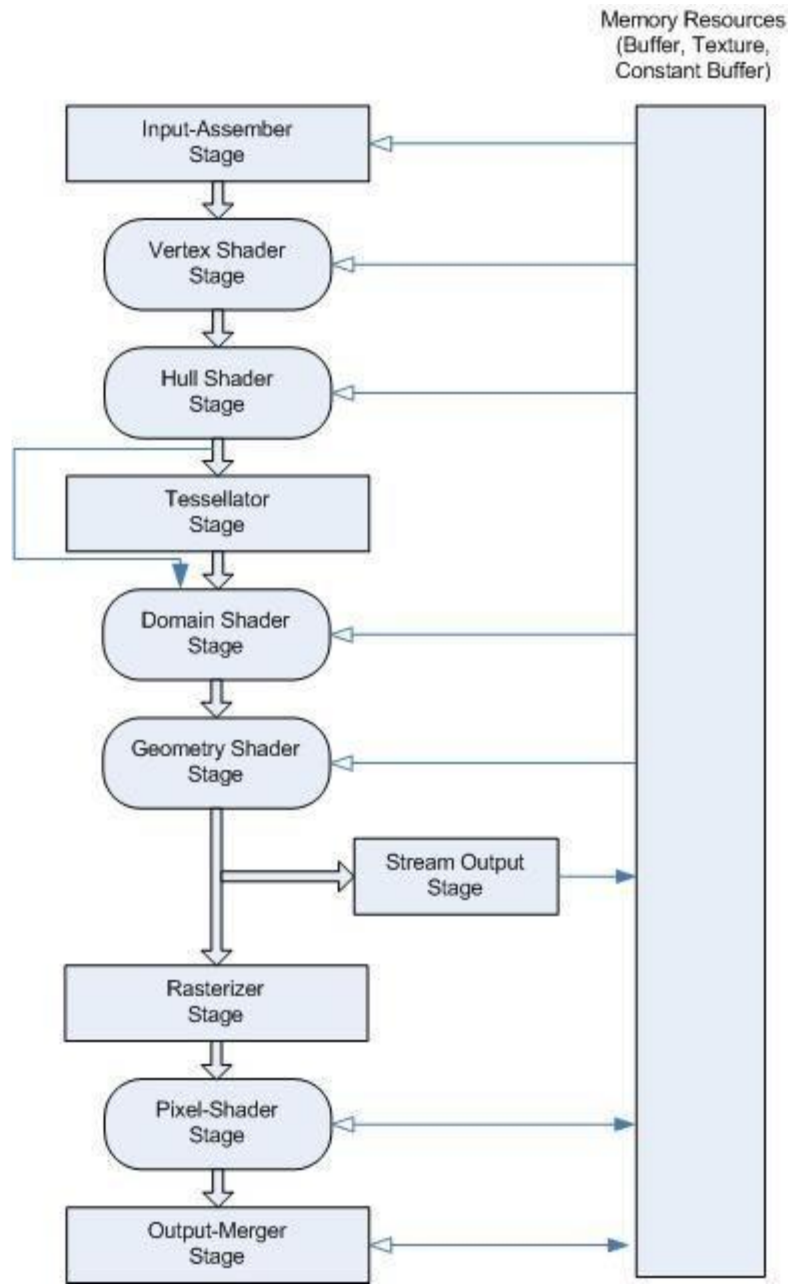


OpenGL pipeline – stages

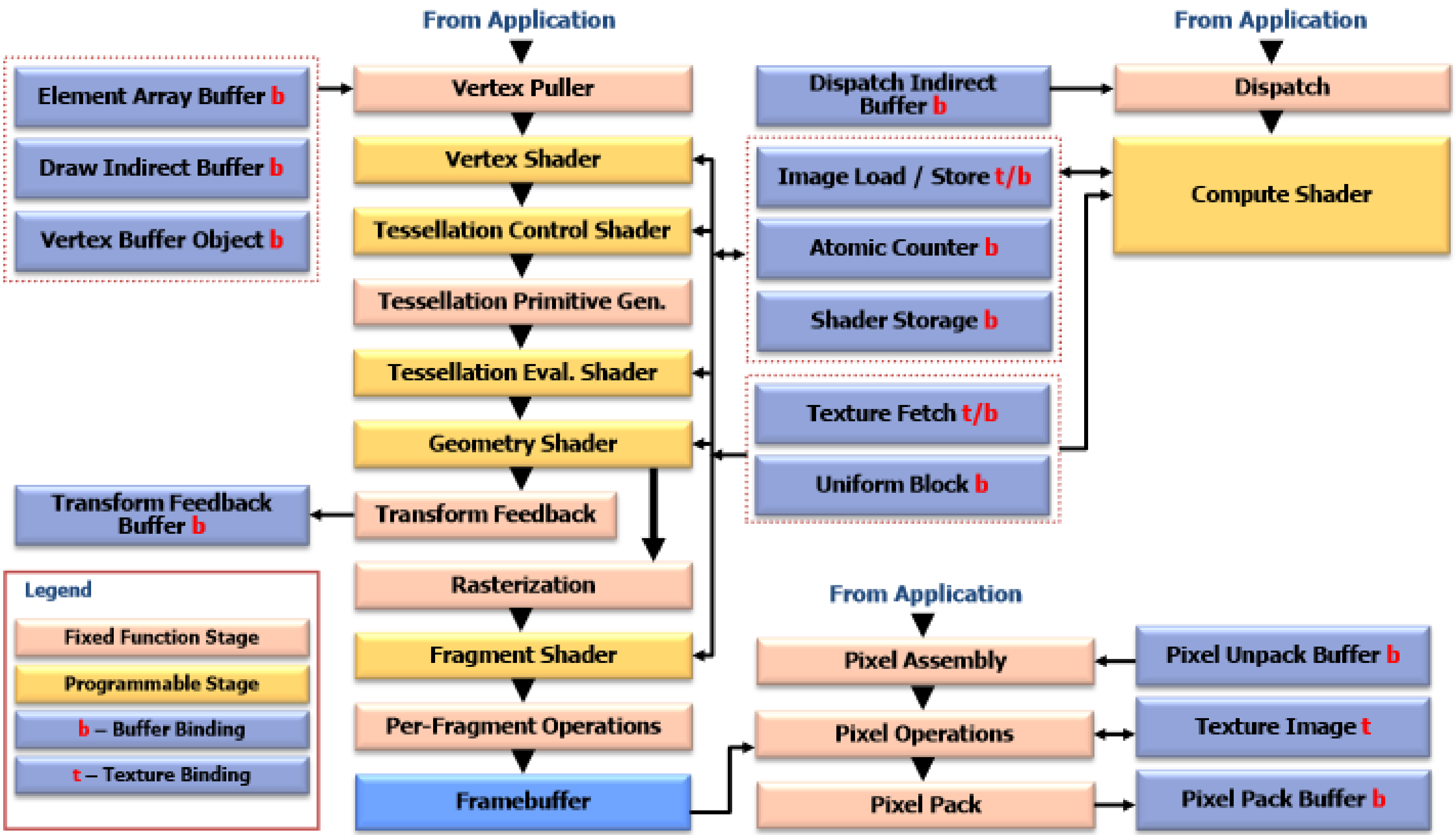


OpenGL pipeline

- Inputs

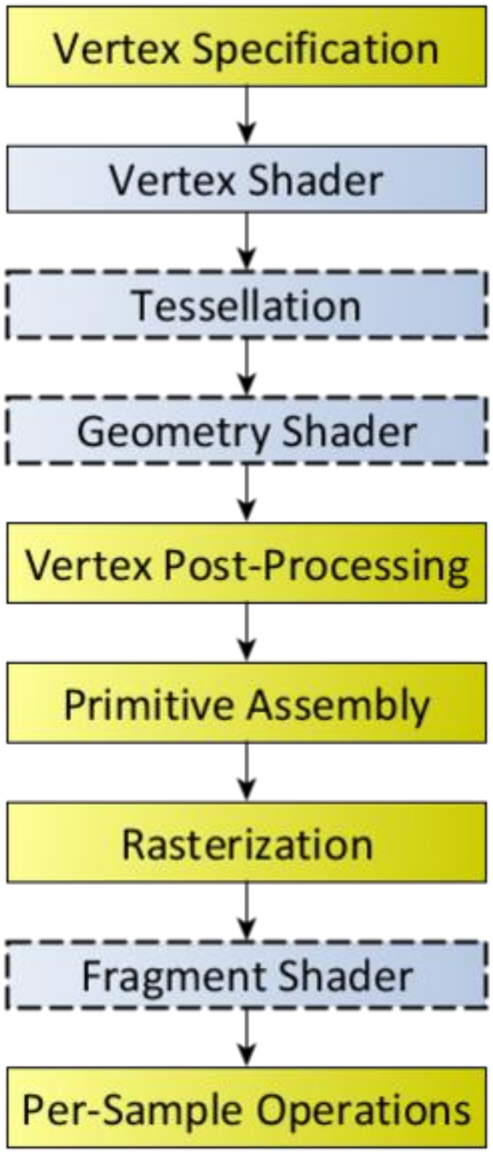


OpenGL pipeline – complete 4.6



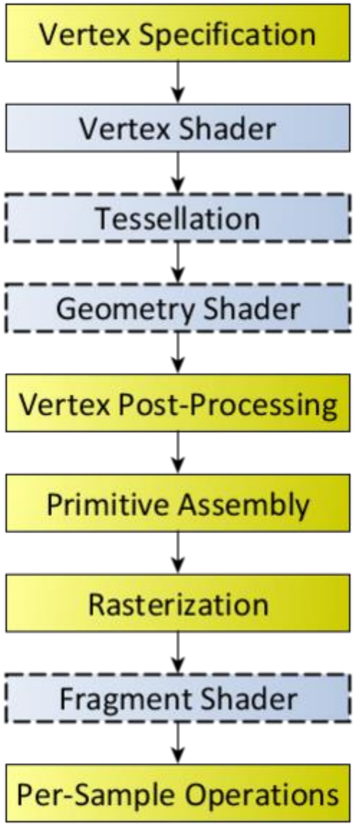
OpenGL pipeline

- Simplified version



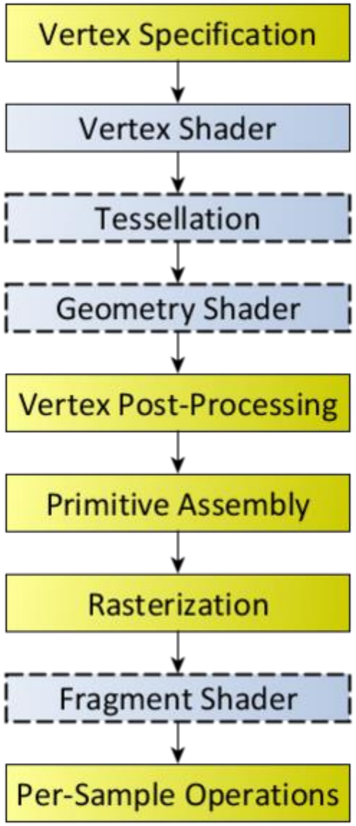
OpenGL pipeline

- Prepare vertex array data and render
- Vertex Processing
- Vertex Post-Processing
  - outputs last stage adjusted/shipped to different locations
- Primitive Assembly
- Scan conversion and primitive parameter interpolation
- A Fragment Shader processes each fragment
  - One or more outputs
- Per-Sample_Processing



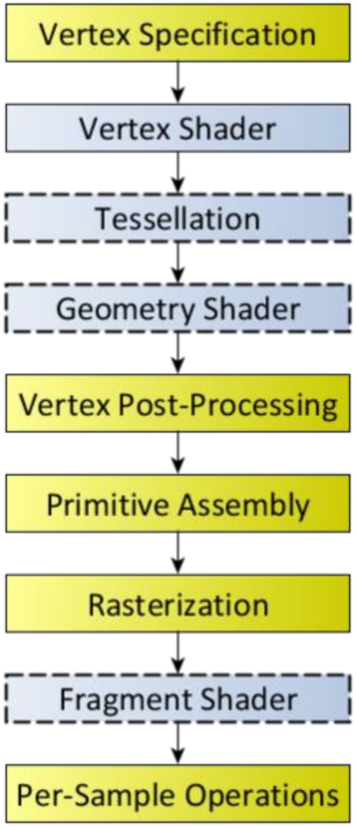
OpenGL pipeline

- Prepare vertex array data and render
  - Generate vertex buffers
    - Geometric data: positions, normals, etc.
    - Optical data: material properties
  - Other stuff (lighting, textures…)
- Render vertex data
  - Send vertex data to the GPU
    - May render multiple instances of same geometry
  - Use indexed primitives
    - Facilitates T&L caching



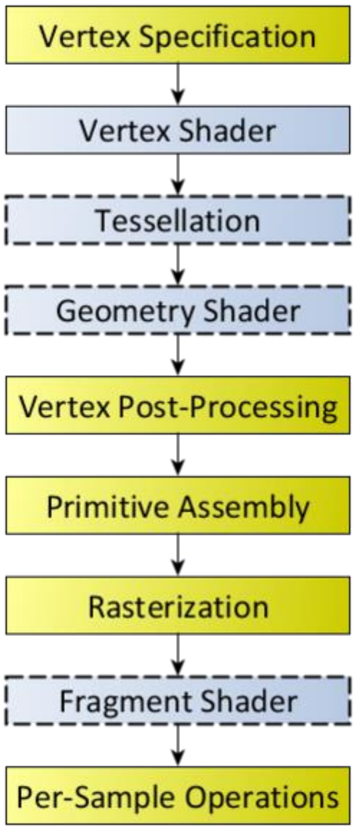
OpenGL pipeline

- Vertex Processing
  - Each vertex is acted upon by a vertex shader
    - Main work: transform vertices
    - Optional work: pass other stuff (texture data, lighting…)
  - Optional tessellation
    - May further subdivide original triangle mesh
  - Optional geometry shader
    - Acts upon full triangles
      - Transforms, culls…
      - Might subdivide and/or generate extra information



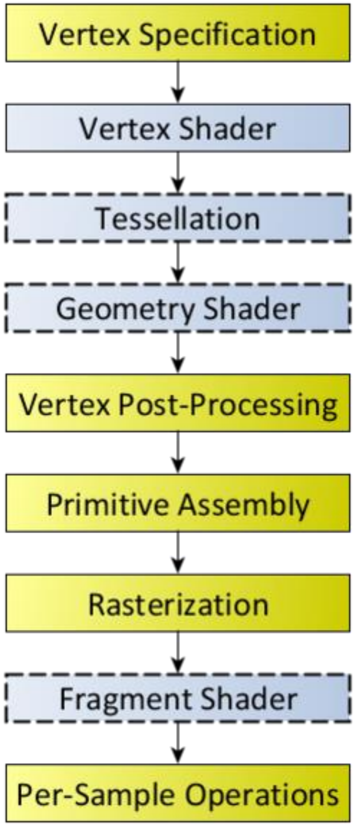
OpenGL pipeline

- Vertex Post-Processing
  - Adapts output for further operations (e.g. primitive assembly and rasterization)
  - May use transform feedback (more on this later)
  - Clipping is performed here
  - Depth clampling, perspective divide, viewport transform…



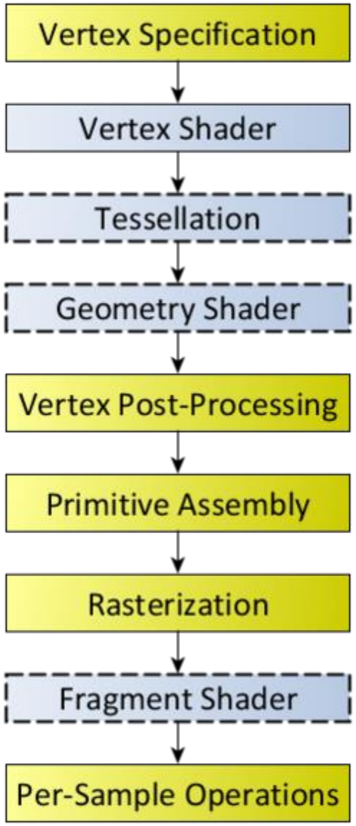
OpenGL pipeline

- Primitive Assembly
  - Transforms a vertex stream into a sequence of primitives (e.g. triangles)
  - Order depends on the vertex rendering sorting
- Scan conversion:
  - Generates the corresponding fragments
    - With depth and other properties
    - Parameters requiring it are interpolated



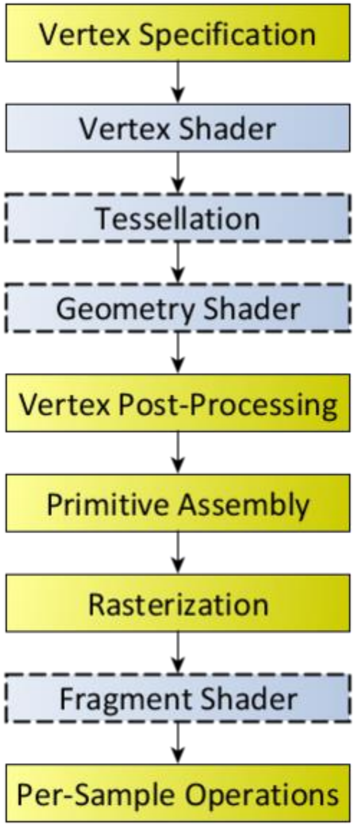
OpenGL pipeline

- Fragment Shader
  - Executed for all rasterized fragments
  - Can only write in its own position
  - Must write color (gl_FragColor -> out data defined in layout)
  - Can generate more than one output (gl_FragData -> out data defined in layout)
  - Can discard data
  - May generate depth



OpenGL pipeline

- Per-Sample Processing
  - Scissor test
  - Stencil test
  - Depth test
  - Blending
  - Logical operation
  - Write Mask



OpenGL pipeline. Fragment shaders

- Fragment shader output
  - Depth value
  - Possible stencil value (unmodified by the fragment shader),
  - **Zero or more color values** to be potentially written to the buffers in the current framebuffers
    - **Opens the possibility of multiple pass rendering**
- Fragment shaders take a single fragment as input and produce a single fragment as output

Contents

- *Basics*
- *OpenGL pipeline*
- **Simple texturing**
- Advanced texturing
- Environment mapping

Simple Texturing

- Complex domain
  - It has infinite possibilities
  - Specially when using shaders
- Basic texturing
  - Get a color and apply it to a surface
- Other uses of textures
  - Store environment information
  - Store functions
  - Store geometric information
  - …

Simple Texturing

- Main stages:
  - Set up texture from the application
  - Access to texture in a shader
    - Do whatever you want with it:
      - The results of one texture access define the parameters of another texture access
      - Textures store intermediate rendering results
      - Textures serve as lookup tables for complex functions
      - Textures store normals, normal perturbation factors, gloss values, visibility information…

Simple Texturing

- Basic method to map a texture onto an object
- One texture unit is required in the pixel shader
- The access to the texture's texels is done using the **texture()** function
  - Takes as parameters the texture we wish to access (a **sampler2D**) and the texture coordinates

Simple Texturing

```
#version 300 es
#define FRAG_COLOR_LOCATION 0

precision highp float;
precision highp int;

struct Material
{
    sampler2D diffuse[2];
};

uniform Material material;

in vec2 v_st;

layout(location = FRAG_COLOR_LOCATION) out vec4 color;
```

Simple Texturing

```
void main()
{

        color = texture(material.diffuse[1], v_st) * 0.77;

}
```

Contents

- *Basics*
- *OpenGL pipeline*
- *Simple texturing*
- **Advanced texturing**
- Environment mapping

Advanced Texturing

- Multiple textures

```
varying vec2 vUv;
```

```
uniform sampler2D image1;
uniform sampler2D image2;
```

```
uniform float time;

void main() {

    // Mix the two images together based on time.

    float percent = 0.5 + 0.5 * sin( time );

     gl_FragColor = mix(
         texture2D( image1, vUv ),
         texture2D( image2, vUv ),
         percent
     );

}
```

Advanced Texturing

- Multiple textures

```
varying vec2 vUv;
```

```
uniform sampler2D image1;
uniform sampler2D image2;
```

```
uniform float time;

void main() {

    // Mix the two images together based on time.

    float percent = 0.5 + 0.5 * sin( time );

     gl_FragColor = mix(
         texture2D( image1, vUv ),
         texture2D( image2, vUv ),
         percent
     );

}
```

Advanced Texturing

- Multiple textures

```
varying vec2 vUv;
```

```
uniform sampler2D image1;
uniform sampler2D image2;
```

```
uniform float time;

void main() {

    // Mix the two images together based on time.

    float percent = 0.5 + 0.5 * sin( time );

     gl_FragColor = mix(
        texture2D( image1, vUv ),
        texture2D( image2, vUv ),
        percent
     );

}
```

Advanced Texturing

- Multiple textures

```
varying vec2 vUv;
```

```
uniform sampler2D image1;
uniform sampler2D image2;
```

```
uniform float time;

void main() {

    // Mix the two images together based on time.

    float percent = 0.5 + 0.5 * sin( time );

     gl_FragColor = mix(
         texture2D( image1, vUv ),
         texture2D( image2, vUv ),
         percent
     );

}
```

Advanced Texturing

- Multiple textures

```
varying vec2 vUv;
```

```
uniform sampler2D image1;
uniform sampler2D image2;
```

```
uniform float time;

void main() {

    // Mix the two images together based on time.

    float percent = 0.5 + 0.5 * sin( time );

     gl_FragColor = mix(
         texture2D( image1, vUv ),
         texture2D( image2, vUv ),
         percent
     );

}
```

Advanced Texturing

- Multiple textures. Result:

Advanced Texturing

- Modifying texture coordinates over time (ripples + blending)

```
precision highp float;
precision highp int;

varying vec2 vUv;

uniform vec2 uvScale;
uniform sampler2D image1;
uniform sampler2D image2;
uniform float speed;
uniform float frequency;
uniform float amplitude;
uniform float time;
```

Advanced Texturing

- Modifying texture coordinates over time (ripples + blending)

```
void main() {

    vec2 ripple = vec2(
        // - 0.5 to center the ripple in the middle of image
        sin(  (length( vUv - 0.5 ) * frequency ) + ( time * speed ) ),
        cos( ( length( vUv - 0.5 ) * frequency ) + ( time * speed ) )
    // Scale amplitude to make input more convenient for users
    ) * ( amplitude / 1000.0 );
```

Advanced Texturing

- Modifying texture coordinates over time (ripples + blending)

```
float percent = 0.5 + 0.5 * sin( time );
gl_FragColor = mix(
    texture2D( image1, vUv + ripple * percent ),
    texture2D( image2, vUv + ripple * ( 1.0 - percent ) ),
    percent
);
```

Advanced Texturing

- Ripples + blending. Result

Advanced Texturing

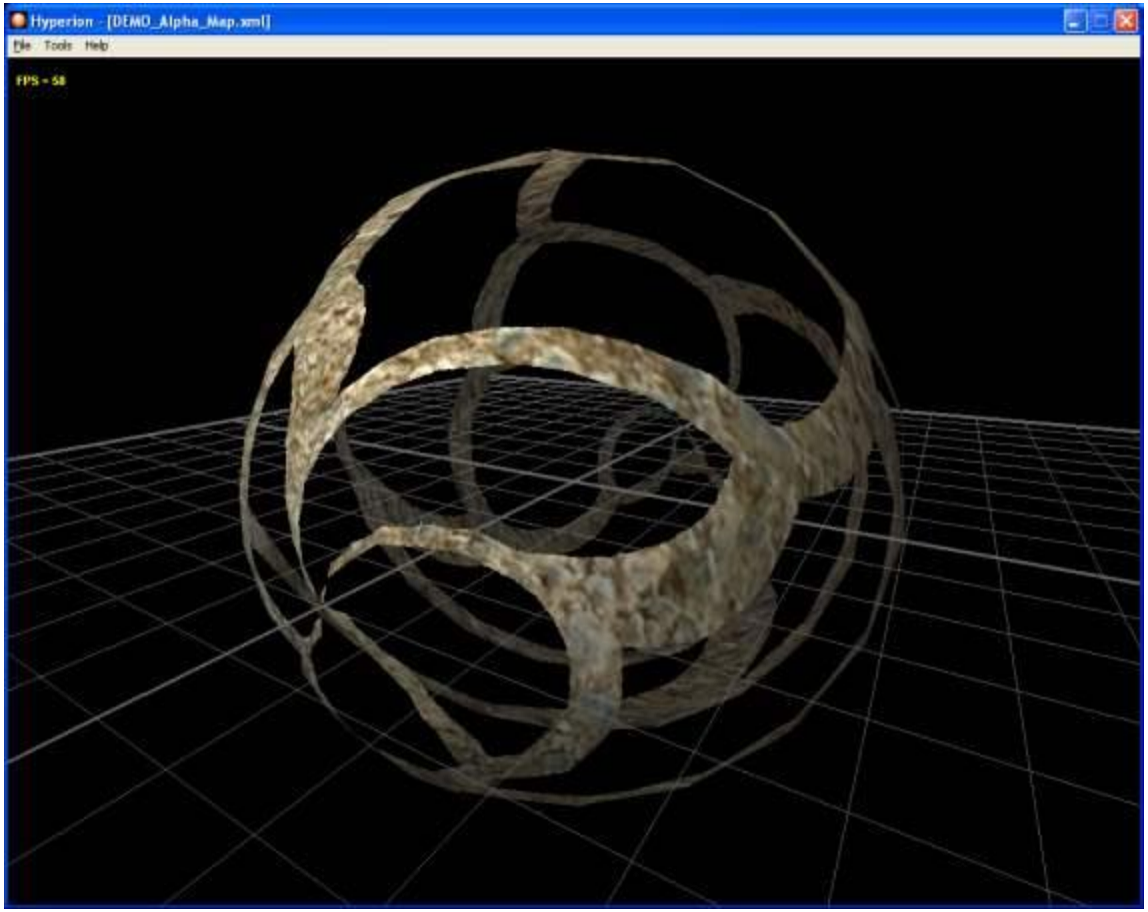
- Alpha Maps
  - An alpha map is a binary texture in the meaning it contains information that allows all or nothing based choices
  - Alpha maps are usually used for tests at the pixel level (alpha-testing) in order to know if a pixel is either fully opaque or fully transparent
    - In this last case, the pixel is not sent to the framebuffer

Advanced Texturing

- Alpha Maps
  - *discard* is a keyword in GLSL to prevent a fragment to update the framebuffer
    - According to the OpenGL implementations, the instructions that follow the discard keyword can or can not be executed but in all cases the framebuffer will not be updated
    - *discard* is available in the pixel shader only

Advanced Texturing

- Alpha Maps



Advanced Texturing

- Alpha Maps. Fragment shader:

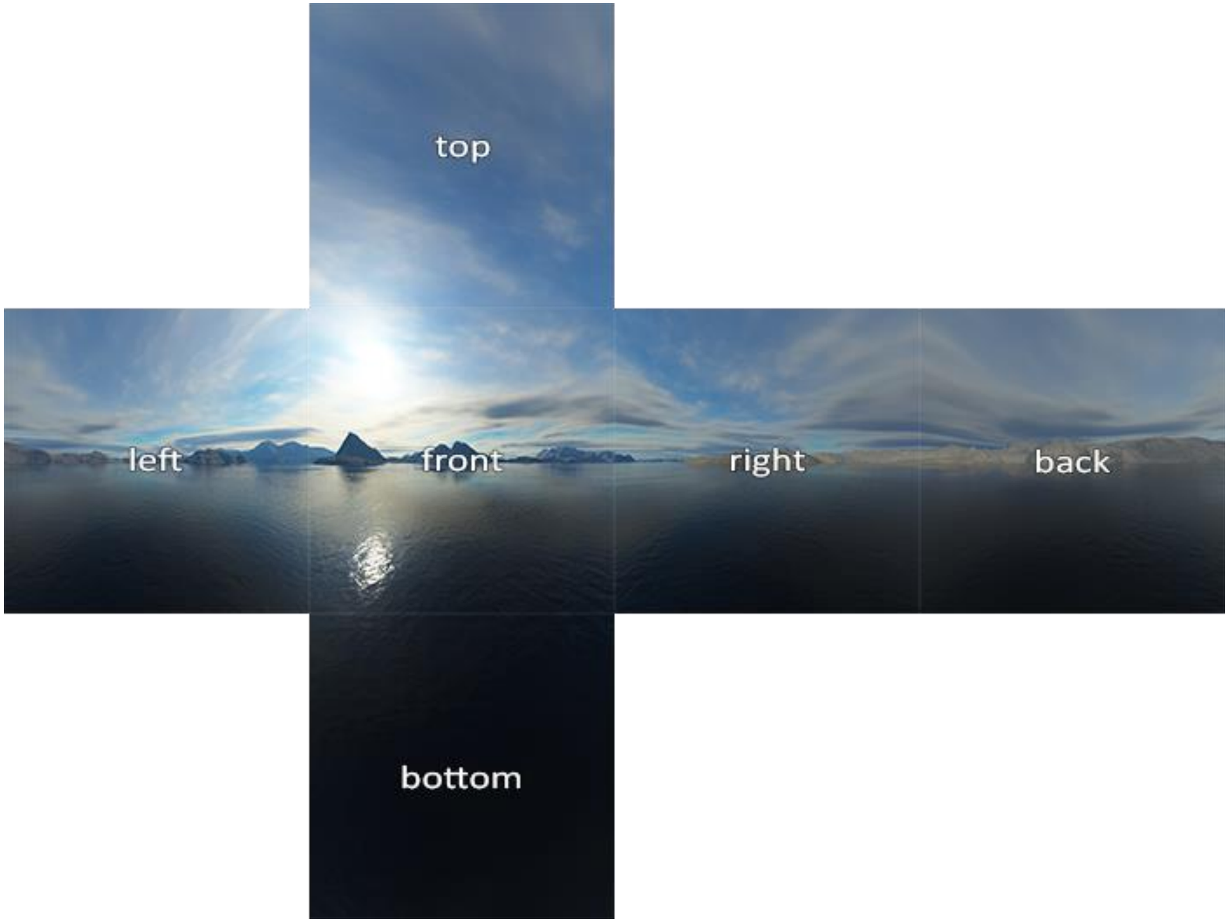
```
varying vec2 vUv;

uniform sampler2D colorMap;
uniform sampler2D alphaMap;

void main (void)
{
  vec4 alpha_color = texture2D(alphaMap, vUv);

  if(alpha_color.r<0.1)
  {
    discard;
  }

  gl_FragColor  = texture2D(colorMap, vUv);
}
```

Advanced Texturing

- Skybox: Large cube that encompasses the entire scene and contains 6 images of a surrounding environment
  - Giving the player the illusion that the environment he's in is actually much larger than it actually is, e.g.:

Advanced Texturing

- Skybox
  - Built using a cubic texture, such as:



Advanced Texturing

- Skybox. Load is performed as any cubic texture

```
unsigned int loadCubemap(vector<std::string> faces)
{
    unsigned int textureID;
    glGenTextures(1, &textureID);
    glBindTexture(GL_TEXTURE_CUBE_MAP, textureID);

    int width, height, nrChannels;
    for (unsigned int i = 0; i < faces.size(); i++)
    {
        unsigned char *data = stbi_load(faces[i].c_str(), &width, &height, &nrChannels, 0);
        if (data)
        {
            glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X + i,
                         0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, data
            );
            stbi_image_free(data);
        }
        else
        {
            std::cout << "Cubemap texture failed to load at path: " << faces[i] << std::endl;
            stbi_image_free(data);
        }
    }
```

Advanced Texturing

- Skybox. Load is performed as any cubic texture

```
unsigned int loadCubemap(vector<std::string> faces)
{
    unsigned int textureID;
    glGenTextures(1, &textureID);
    glBindTexture(GL_TEXTURE_CUBE_MAP, textureID);

    int width, height, nrChannels;
    for (unsigned int i = 0; i < faces.size(); i++)
    {
        unsigned char *data = stbi_load(faces[i].c_str(), &width, &height, &nrChannels, 0);
        if (data)
        {
            glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X + i,
                         0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, data
            );
            stbi_image_free(data);
        }
        else
        {
            std::cout << "Cubemap texture failed to load at path: " << faces[i] << std::endl;
            stbi_image_free(data);
        }
    }
```

Advanced Texturing

- Skybox. Load is performed as any cubic texture

```
unsigned int loadCubemap(vector<std::string> faces)
{
    unsigned int textureID;
    glGenTextures(1, &textureID);
    glBindTexture(GL_TEXTURE_CUBE_MAP, textureID);

    int width, height, nrChannels;
    for (unsigned int i = 0; i < faces.size(); i++)
    {
        unsigned char *data = stbi_load(faces[i].c_str(), &width, &height, &nrChannels, 0);
        if (data)
        {
            glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X + i,
                         0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, data
            );
            stbi_image_free(data);
        }
        else
        {
            std::cout << "Cubemap texture failed to load at path: " << faces[i] << std::endl;
            stbi_image_free(data);
        }
    }
```

Advanced Texturing

- Skybox. Load is performed as any cubic texture

```
        else
        {
            std::cout << "Cubemap texture failed to load at path: " << faces[i] << std::endl;
            stbi_image_free(data);
        }
    }
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_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);

    return textureID;
}
```

Advanced Texturing

- Skybox. Drawing the box
  - Disable depth writing!

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

Advanced Texturing

- Skybox. Vertex shader

```
#version 330 core
layout (location = 0) in vec3 aPos;

out vec3 TexCoords;

uniform mat4 projection;
uniform mat4 view;

void main()
{
    TexCoords = aPos;
    gl_Position = projection * view * vec4(aPos, 1.0);
}
```

Advanced Texturing

- Skybox. Vertex shader

```
#version 330 core
layout (location = 0) in vec3 aPos;

out vec3 TexCoords;

uniform mat4 projection;
uniform mat4 view;

void main()
{
    TexCoords = aPos;
    gl_Position = projection * view * vec4(aPos, 1.0);
}
```

Advanced Texturing

- Skybox. Fragment shader

```
#version 330 core
out vec4 FragColor;

in vec3 TexCoords;

uniform samplerCube skybox;

void main()
{
    FragColor = texture(skybox, TexCoords);
}
```

Advanced Texturing

- Skybox. Fragment shader

```
#version 330 core
out vec4 FragColor;

in vec3 TexCoords;

uniform samplerCube skybox;

void main()
{
    FragColor = texture(skybox, TexCoords);
}
```

Advanced Texturing

- Skybox. Fragment shader

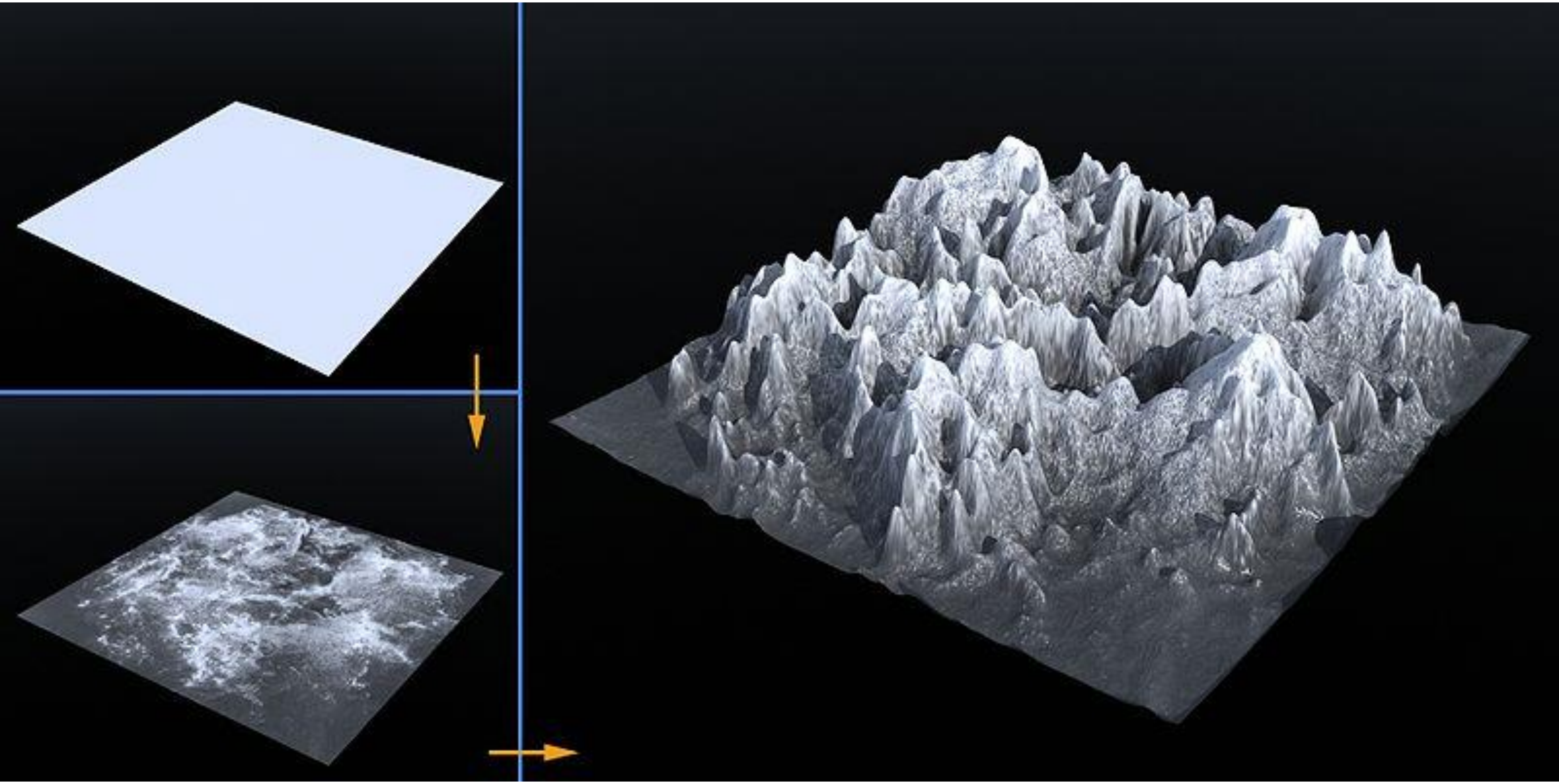
```
#version 330 core
out vec4 FragColor;

in vec3 TexCoords;

uniform samplerCube skybox;

void main()
{
    FragColor = texture(skybox, TexCoords);
}
```

Advanced Texturing

- Skybox. Result

Advanced Texturing

- Displacement mapping: Using textures to modify vertices



Advanced Texturing

- Displacement mapping. Vertex shader

```
#version 300 es
#define POSITION_LOCATION 0
#define NORMAL_LOCATION 1
#define TEXCOORD_LOCATION 4

precision highp float;
precision highp int;

uniform mat4 mvMatrix;
uniform mat4 pMatrix;
uniform sampler2D displacementMap;

layout(location = POSITION_LOCATION) in vec3 position;
layout(location = NORMAL_LOCATION) in vec3 normal;
layout(location = TEXCOORD_LOCATION) in vec2 texcoord;

out vec2 v_st;
out vec3 v_position;
```

Advanced Texturing

- Displacement mapping. Vertex shader

```
void main()
{
    v_st = texcoord;
    float height = texture(displacementMap, texcoord).b;
    vec4 displacedPosition = vec4(position, 1.0) + vec4(normal * height, 0.0);
    v_position = vec3(mvMatrix * displacedPosition);
    gl_Position = pMatrix * mvMatrix * displacedPosition;
}
```

Contents

- *Basics*
- *OpenGL pipeline*
- *Simple texturing*
- *Advanced texturing*
- **Environment mapping**

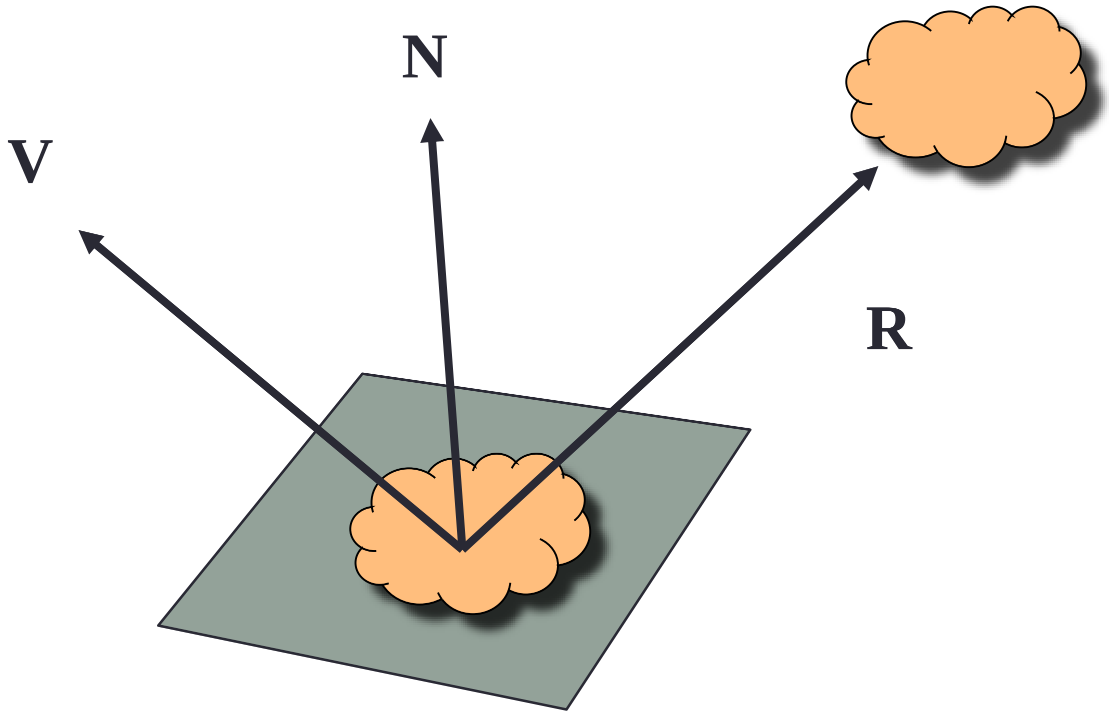
Environment Mapping

- Spherical Environment Mapping
  - The first and simplest of the techniques allowing to simulate the reflection of the environment on an object's surface
  - SEM uses only one texture for the reflection
    - The texture can be whatever you want but usually we use SEM-ready textures

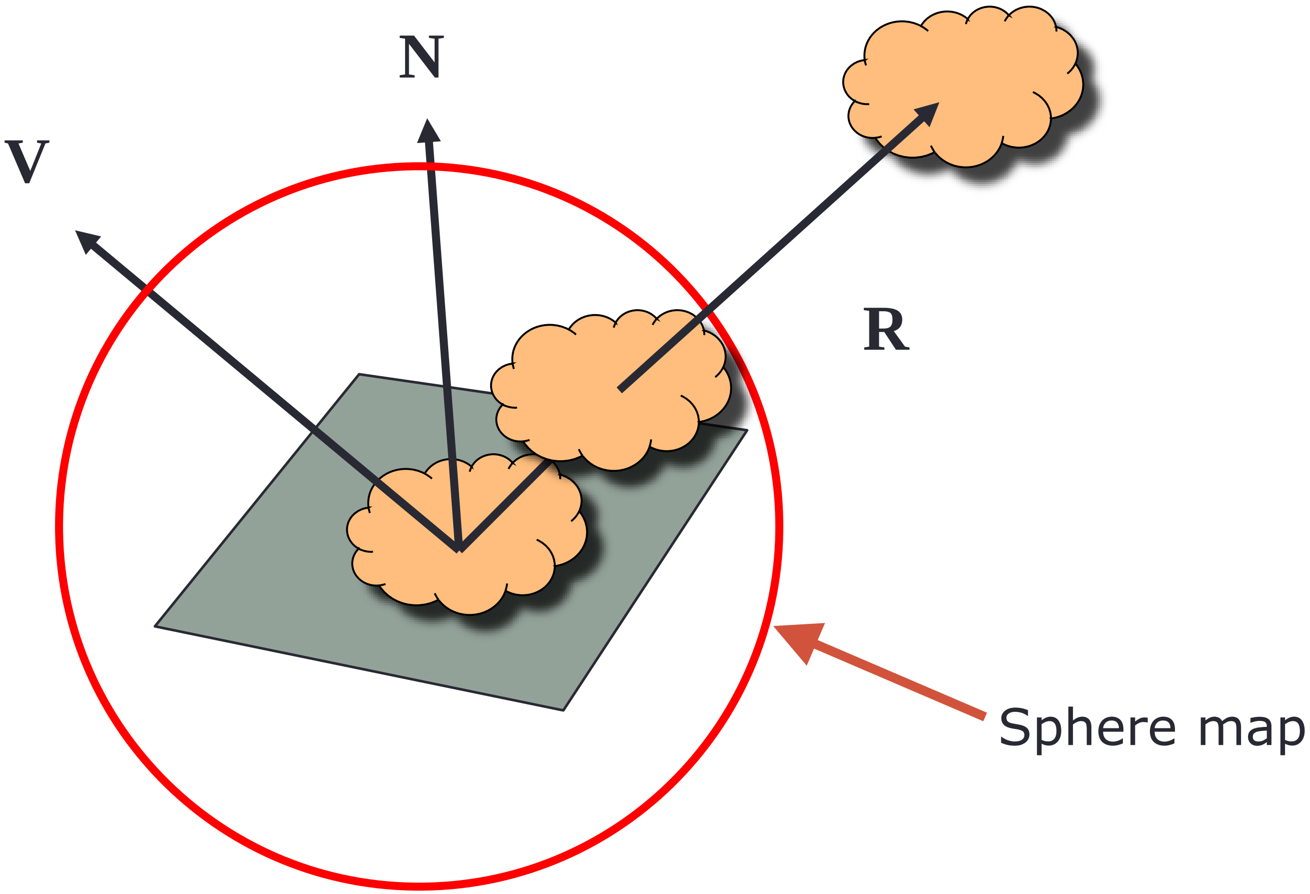
Environment Mapping

- Spherical Environment Mapping

Environment Mapping



Environment Mapping



Environment Mapping

- In order to compute s and t coordinates:
  - **u:** unit vector that goes from the camera to the current vertex:
    - the position of the vertex in eye space and also the view vector
  - **n:** vertex normal in eye space.
  - **r:** is the reflected vision vector against **n**:

    r = reflect(u, n)

    r = 2 * (n dot u) * n - u
  - **m** is an intermediate value:

    $m = \mathrm{sqrt}(r.x^2 + r.y^2 + (r.z + 1.0)^2)$
  - **s** and **t** are the final texture coordinates:

    s = r.x / m + 0.5
    t = r.y / m + 0.5

Environment Mapping

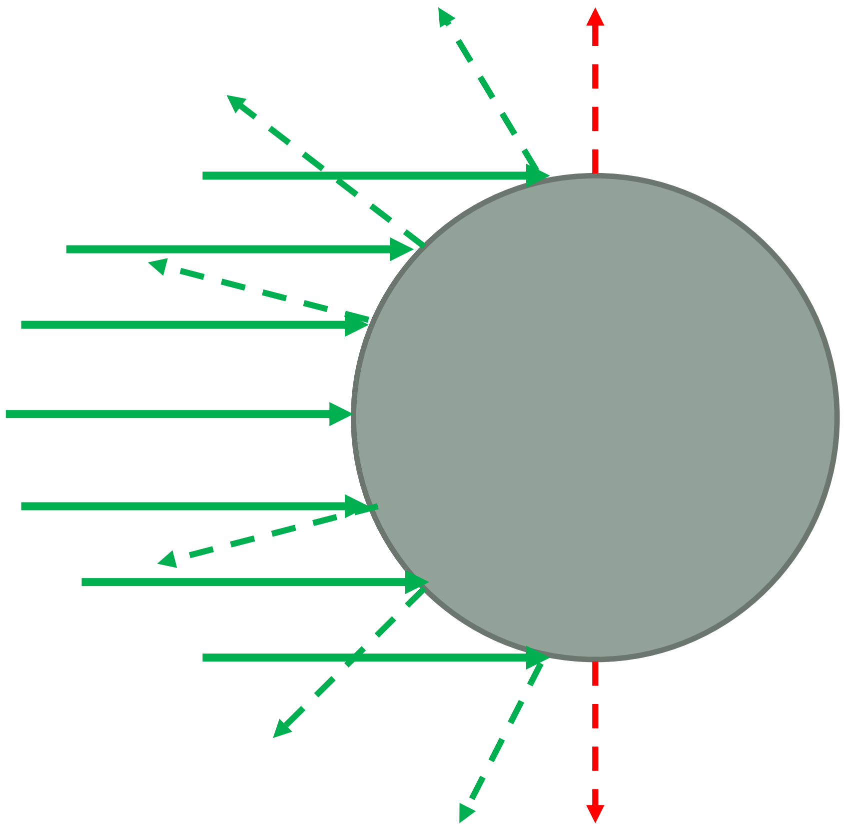
- Construction:
  - Can use a ray tracer
  - Or can use projection-based rendering
    - By modifying the projected coordinates in the vertex shader
  - Typically two spheres can be created

Environment Mapping

- Sampling problems:
  - Highly varying resolution
    - Texels do not represent the same information
      - Subtended solid angle is not the same
  - Minimum accuracy in poles
    - Actually, there is a singularity (no samples for the north & south poles)

Environment Mapping

- Sampling problems



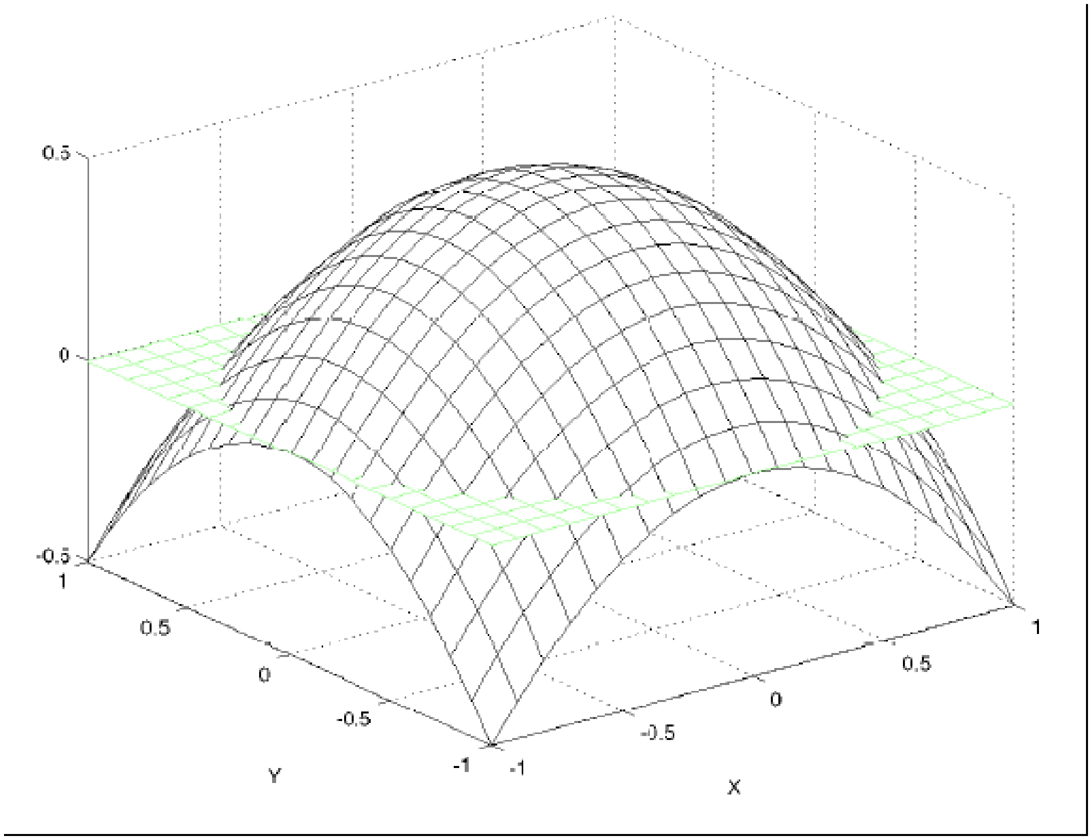
Environment Mapping

- Sampling problems:
  - Can be alleviated using dual-paraboloids mapping
    - No singularity
    - Better sampling ratio (≈ 4 to 1)
  - Dual paraboloid mathematical definition:

$$f(x,y)=\frac{1}{2}-\frac{1}{2}(x^2+y^2) \ \text{ for } \ x^2+y^2\leq 1$$

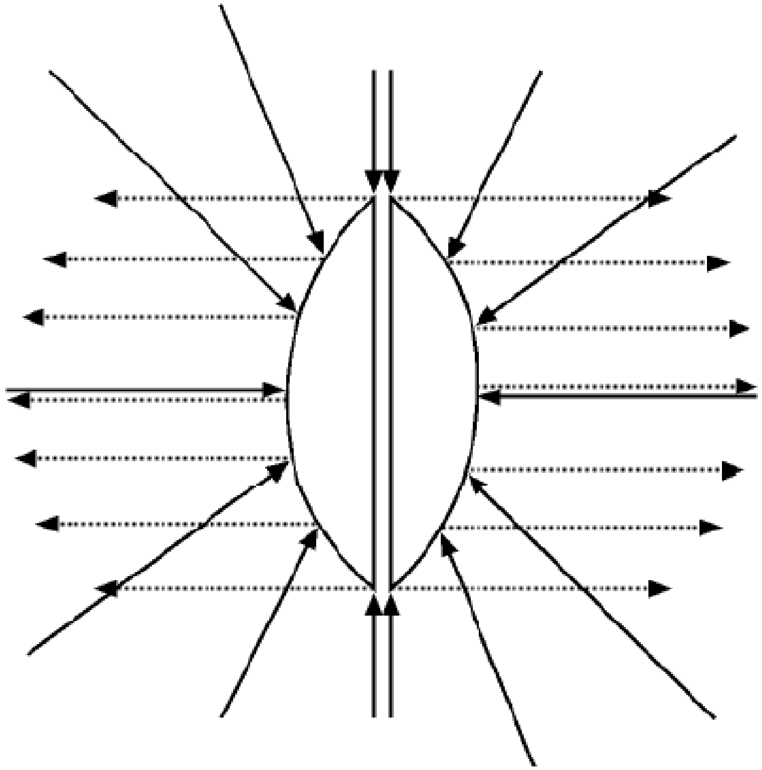
Environment Mapping

- Paraboloid:



Environment Mapping

- Dual Paraboloid:



Environment Mapping

- Dual paraboloid. Texture coordinates. Vertex shader:

```
void main()
{
    ...

    // find world space position.
    vec4 WorldPos = ModelWorld4x4 * gl_Vertex;
    // find world space normal.
    vec3 N = normalize( ModelWorld3x3 * gl_Normal );
    // find world space eye vector.
    vec3 E = normalize( WorldPos.xyz - CameraPos.xyz );
    // calculate the reflection vector in world space.
    R = reflect( E, N );
}
```

Environment Mapping

- Dual paraboloid. Texture coordinates. Fragment shader:

```
uniform sampler2D frontMap;
uniform sampler2D backMap;
varying vec3 R;

void main (void)
{
   vec4 output_color; vec3 vR = normalize(R);
   // Select front or back env map according to vR.z sign
   if(vR.z>0.0)
   {
      // calculate the forward paraboloid map
      // texture coordinates
      vec2 frontUV  = (vR.xy / (2.0*(1.0 + vR.z))) + 0.5;
      output_color = texture2D( frontMap, frontUV );
   }
```

Environment Mapping

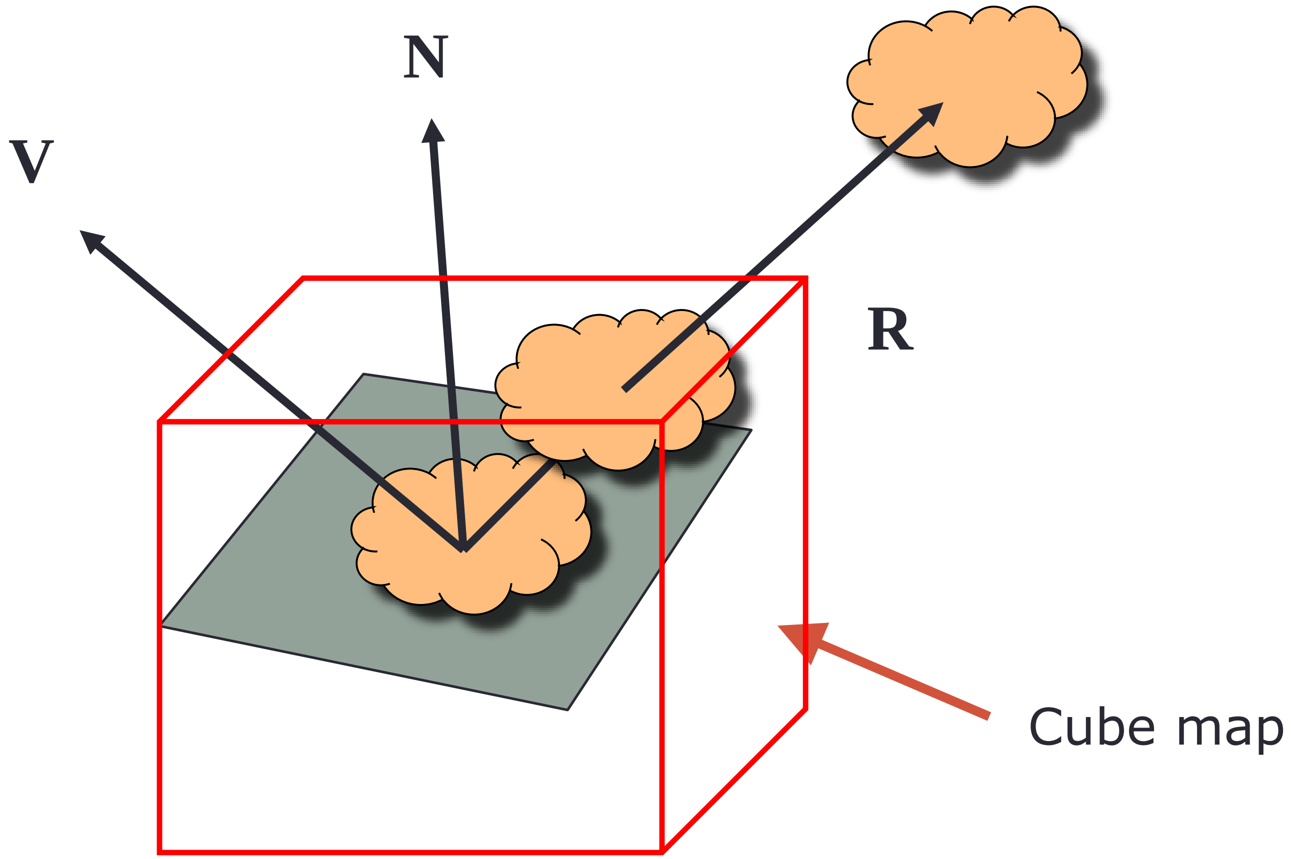
- Dual paraboloid. Texture coordinates. Fragment shader:

```
else
   {
      // calculate the backward paraboloid map
      // texture coordinates
      vec2 backUV;
      backUV = (vR.xy / (2.0*(1.0 - vR.z))) + 0.5;
      output_color = texture2D(backMap, backUV);
   }
   gl_FragColor = output_color;
}
```

Environment Mapping

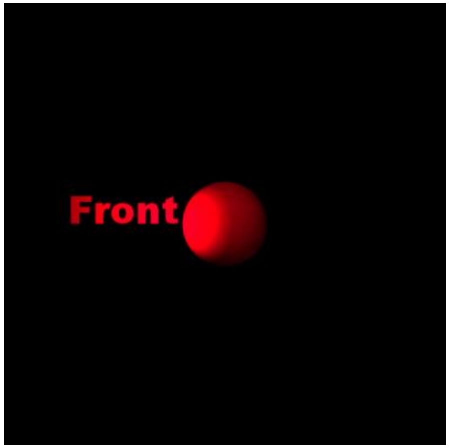
- OpenGL supports spherical and cube maps
  - Cube maps share the same idea
  - But a better sampling rate

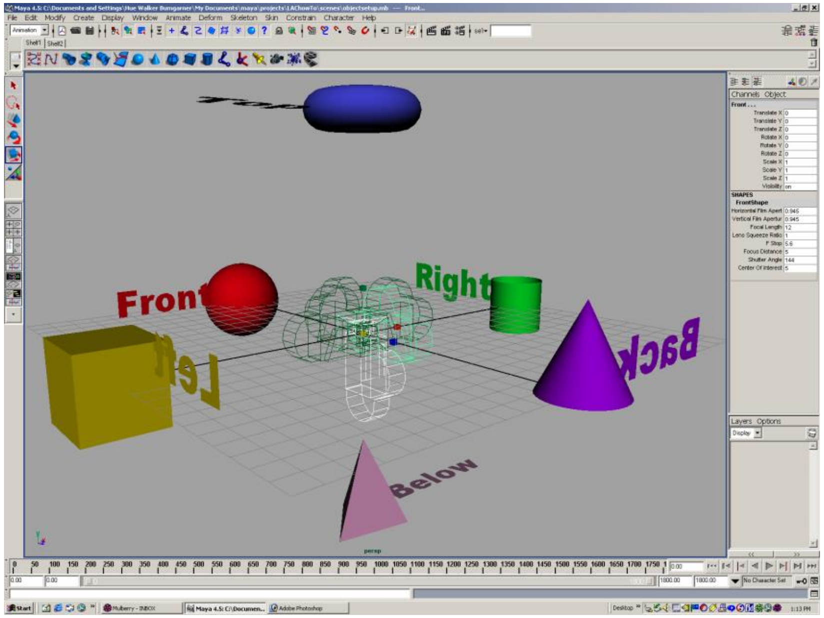
Environment Mapping



Environment Mapping

- Construction:
  - Use six cameras, each with a 90 degree angle of view





Environment Mapping

- OpenGL supports spherical and cube maps
- First must form map
  - Use images from a real camera
  - Form images with OpenGL
- Texture map it to object

Environment Mapping

- GLSL implementation. Vertex Shader:

```
precision highp float;
precision highp int;

uniform mat4 modelMatrix;
uniform mat4 modelViewMatrix;
uniform mat4 projectionMatrix;
uniform mat4 viewMatrix;
uniform mat3 normalMatrix;
uniform vec3 cameraPosition;

attribute vec3 position;
attribute vec3 normal;
attribute vec2 uv;
attribute vec2 uv2;

varying vec3 vReflect;
```

Environment Mapping

- GLSL implementation. Vertex Shader:

```
void main() {
    vec3 worldPosition = ( modelMatrix * vec4( position, 1.0 )).xyz;
    vec3 cameraToVertex = normalize( worldPosition - cameraPosition );
    vec3 worldNormal = normalize(
        mat3( modelMatrix[ 0 ].xyz, modelMatrix[ 1 ].xyz, modelMatrix[ 2 ].xyz ) * normal
    );
    vReflect = reflect( cameraToVertex, worldNormal );
    gl_Position = projectionMatrix * modelViewMatrix * vec4(position, 1.0);
}
```

Environment Mapping

- GLSL implementation. Vertex Shader:

```
void main() {
    vec3 worldPosition = ( modelMatrix * vec4( position, 1.0 )).xyz;
    vec3 cameraToVertex = normalize( worldPosition - cameraPosition );
    vec3 worldNormal = normalize(
        mat3( modelMatrix[ 0 ].xyz, modelMatrix[ 1 ].xyz, modelMatrix[ 2 ].xyz ) * normal
    );
    vReflect = reflect( cameraToVertex, worldNormal );
    gl_Position = projectionMatrix * modelViewMatrix * vec4(position, 1.0);
}
```

Environment Mapping

- GLSL implementation. Vertex Shader:

```
void main() {
    vec3 worldPosition = ( modelMatrix * vec4( position, 1.0 )).xyz;
    vec3 cameraToVertex = normalize( worldPosition - cameraPosition );
    vec3 worldNormal = normalize(
        mat3( modelMatrix[ 0 ].xyz, modelMatrix[ 1 ].xyz, modelMatrix[ 2 ].xyz ) * normal
    );
    vReflect = reflect( cameraToVertex, worldNormal );
    gl_Position = projectionMatrix * modelViewMatrix * vec4(position, 1.0);
}
```

Environment Mapping

- GLSL implementation. Vertex Shader:

```
void main() {
    vec3 worldPosition = ( modelMatrix * vec4( position, 1.0 )).xyz;
    vec3 cameraToVertex = normalize( worldPosition - cameraPosition );
    vec3 worldNormal = normalize(
        mat3( modelMatrix[ 0 ].xyz, modelMatrix[ 1 ].xyz, modelMatrix[ 2 ].xyz ) * normal
    );
    vReflect = reflect( cameraToVertex, worldNormal );
    gl_Position = projectionMatrix * modelViewMatrix * vec4(position, 1.0);
}
```

Environment Mapping

- GLSL implementation. Vertex Shader:

```
void main() {
    vec3 worldPosition = ( modelMatrix * vec4( position, 1.0 )).xyz;
    vec3 cameraToVertex = normalize( worldPosition - cameraPosition );
    vec3 worldNormal = normalize(
        mat3( modelMatrix[ 0 ].xyz, modelMatrix[ 1 ].xyz, modelMatrix[ 2 ].xyz ) * normal
    );
    vReflect = reflect( cameraToVertex, worldNormal );
    gl_Position = projectionMatrix * modelViewMatrix * vec4(position, 1.0);
}
```

Environment Mapping

- GLSL implementation. Fragment Shader:

```
precision highp float;
precision highp int;

varying vec3 vReflect;

uniform float mirrorReflection;
uniform samplerCube reflectionSampler;

void main() {
    vec4 cubeColor = textureCube( reflectionSampler, vec3( mirrorReflection * vReflect.x, vReflect.yz ) );
    cubeColor.w = 1.0;
    gl_FragColor = cubeColor;
}
```

Environment Mapping

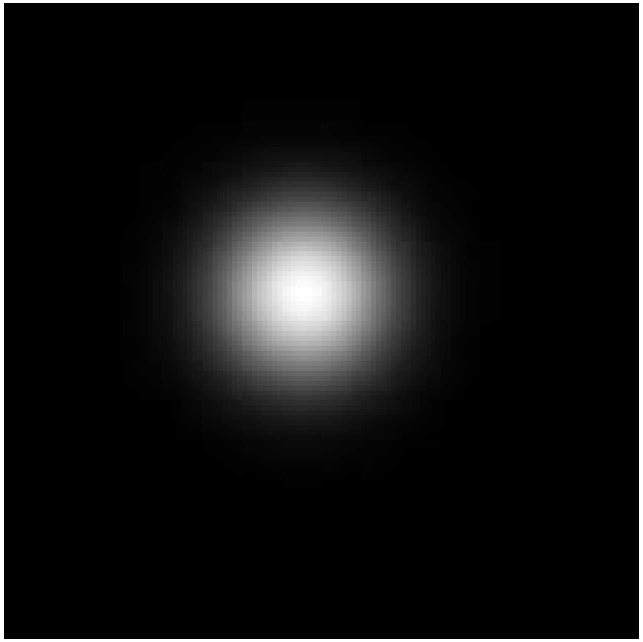
- Result

Environment Mapping

- Spherical mapping. Discussion:
  - Advantage:
    - Simple to use
  - Disadvantages:
    - Must assume environment is very far from object
    - It is view-dependent: the reflection slides on the object and follows the view
      - Need a new map if viewer moves
    - If we map all objects to hemisphere, we cannot tell if they are on the sphere or anywhere else along the reflector
    - Object cannot be concave
    - No reflections between objects
      - Need a reflection map for each object

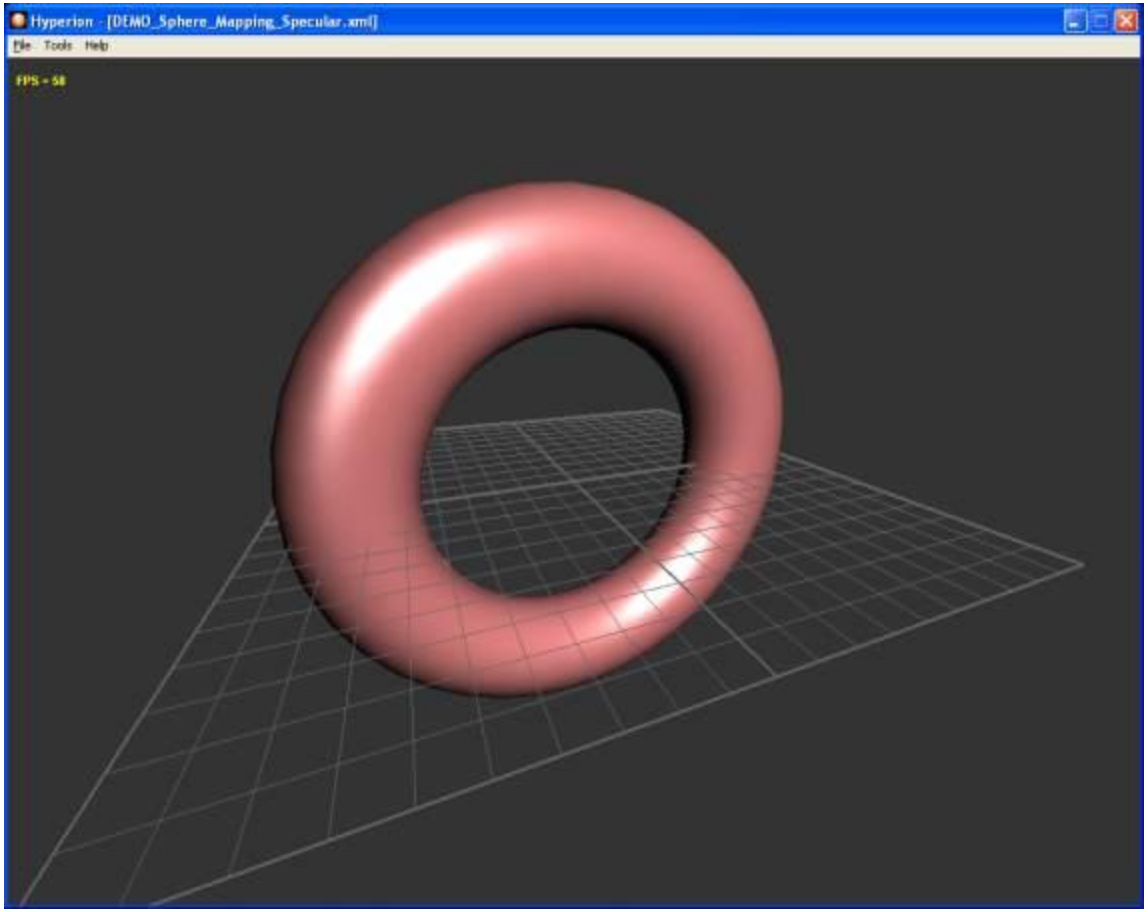
Environment Mapping

- Spherical mapping: Other uses:
  - We can take advantage of the fact that spherical mapping is view-dependent in order to generate of specular highlights
  - Use for instance the following texture:



Environment Mapping

- Result:



OPENGL BASICS: PIPELINE, TEXTURING

Pere-Pau Vázquez

MOVING Group – UPC