GLSL Shaders

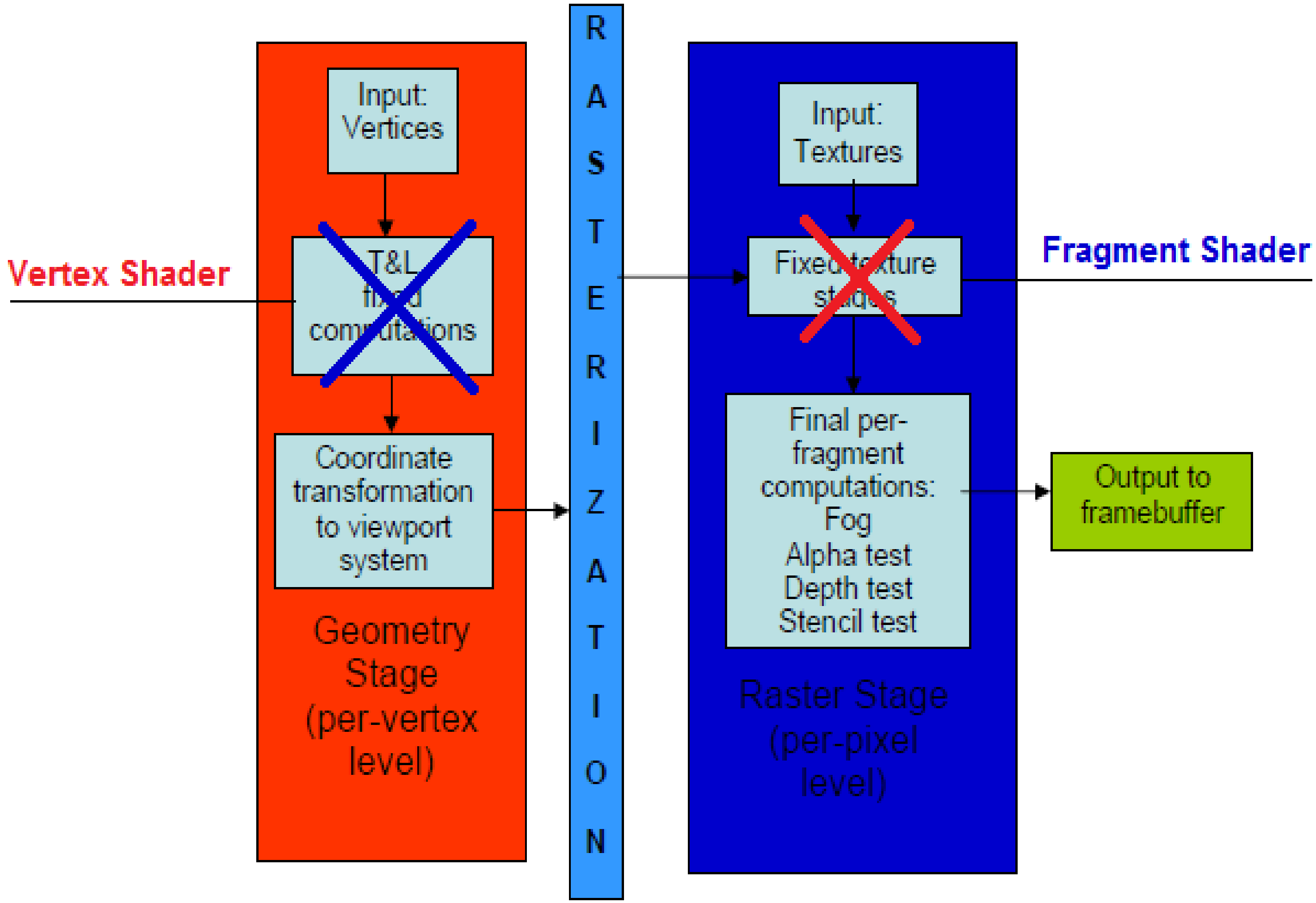
Fondamenti di Computer Graphics L-M

Andrea Bucaletti

Università di Bologna

A.A. 2010/2011

Graphics pipeline



OpenGL Shading Language

- C-like language chosen by OpenGL to manage GPU Shader Units
- Flow control: if, else, while, do, break, return, …
- Functions and function prototypes
- Stuctures
- Arrays
- Built-in functions: math, trigonometry, vectors and matrices operations, ...
- No pointers :)

OpenGL Shading Language

- Data types:
  - Primitive: int, bool, float
  - Float vectors: vec2, vec3, vec4
  - Boolean vectors: bvec2, bvec3, bvec4
  - Integer vectors: ivec2, ivec3, ivec4
  - Float matrices: mat2, mat3, mat4
  - Textures: sampler2D, sampler3D, samplerCube, …
  - And many more…

OpenGL Shading Language

- Global variables types:
  - **Uniform** (VS & FS): read-only values passed from the host OpenGL application to the shader
  - **Attribute** (VS only): these variables can only be used within VS to pass per-vertex values
  - **Varying** (VS & FS): these variables are used to pass data from VS to FS. These values will be ***interpolated*** (*smooth/flat*, *with/without* perspective correction) across the primitive surface.

OpenGL Shading Language

- Built-in variables: these values are computed by OpenGL fixed function

```
//      Current model-view and projection matrices
uniform mat4 gl_ModelViewMatrix;
uniform mat4 gl_ProjectionMatrix;
uniform mat4 gl_ModelViewProjectionMatrix;

// Lights
uniform gl_LightSourceParameters gl_LightSource[gl_MaxLights];

// Vertex position and vertex normal (OCS)
attribute vec4 gl_Vertex;
attribute vec3 gl_Normal;

// Texture coordinates
varying vec4 gl_TexCoords[];

...
```

OpenGL Shading Language

- What you can do:
  - Per-pixel lighting
  - Bump mapping
  - Shadow mapping
  - Special effects: water, sky, vegetation, ...
  - Post-processing
  - Particle systems
  - Toon shading
  - ....

First GLSL Shader

Vertex Shader

```
// sample1.vert

uniform float scale;
varying vec3 normal;

void main()
{
        vec3 position = gl_Vertex.xyz * scale;
        gl_Position = gl_ModelViewProjectionMatrix * vec4(position, 1.0);
        normal = gl_NormalMatrix * gl_Normal;
}
```

- `gl_Vertex` is a vec4, `gl_Vertex.xyz` is a vec3
- `vec4(position, 1.0)` constructs a new vec4 from a vec3 and a float number for the w-coordinate
- `uniform mat3 gl_NormalMatrix` - a 3x3 model-view matrix (**only for vectors!!**)
- `vec4 gl_Position` - the output of a vertex shader. Must be set with the current vertex's coordinates in NCS
- The varying variable `normal` is interpolated across the primitive and passed to the fragment shader

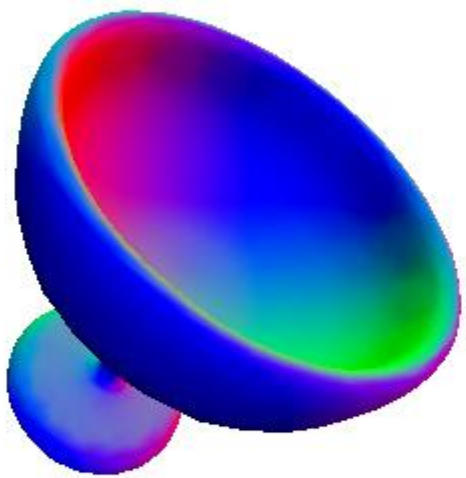
First GLSL Shader

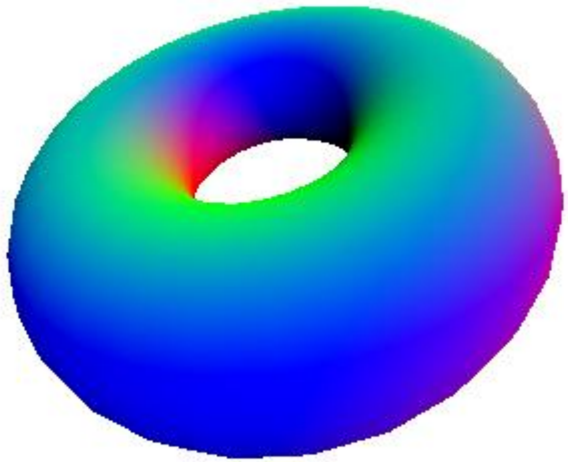
Fragment Shader

```
// sample1.frag

varying vec3 normal;

void main()
{
        gl_FragColor = vec4(normalize(normal), 1.0);
}
```

- `vec4 gl_FragColor` - the output of a fragment shader. Must be set with the color of the current pixel.
- **Colors are treated as vectors and clamped in range [0, 1]**
- the varying variable `normal` contains current pixel's normal
- `normalize(normal)` - the interpolated normal must be re-normalized after interpolation
- The final result is that we can “see” the current pixel's normal as a color

First GLSL Shader





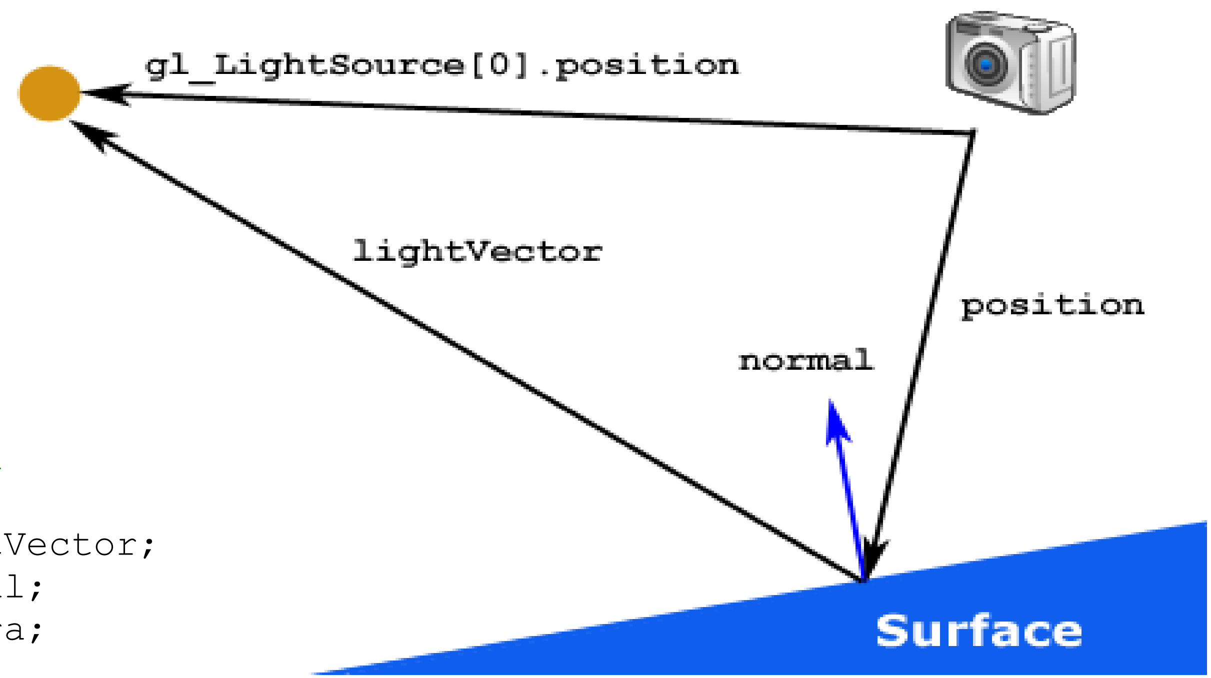
Interpolation is the key to implement per-pixel lighting.

All the lighting computation can be done in the fragment shader using the interpolated values.

Of course, per-pixel lighting is much more expensive than per-vertex lighting, which is what OpenGL fixed function does.

Simple per-pixel lighting shader

Vertex Shader




```
// pplighting.vert

varying vec3 lightVector;
varying vec3 normal;
varying vec3 camera;

void main()
{
        vec3 position = (gl_ModelViewMatrix * gl_Vertex).xyz;

        lightVector = gl_LightSource[0].position.xyz - position;
        normal = gl_NormalMatrix * gl_Normal;
        camera = -position;

        gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

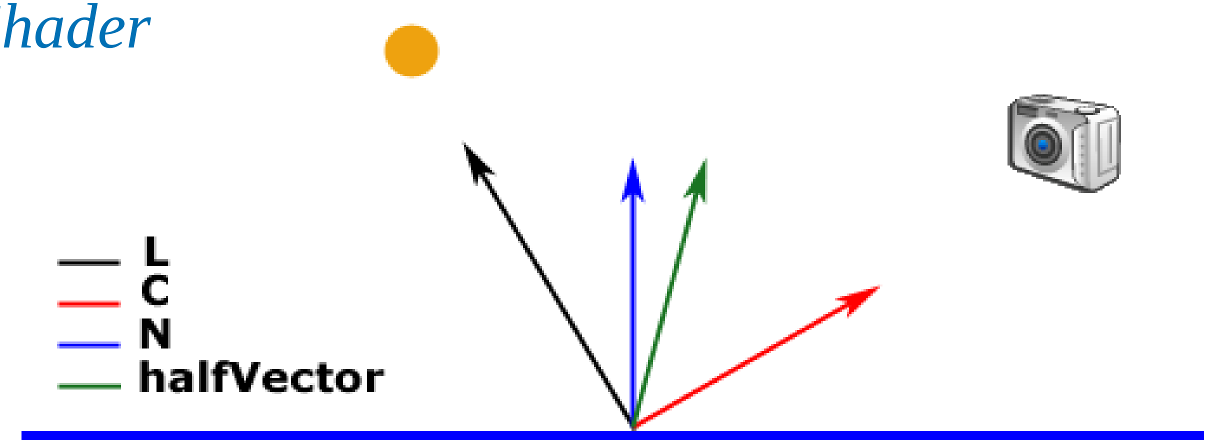
Simple per-pixel lighting shader

Fragment Shader

```
// pplighting.frag

varying vec3 lightVector;
varying vec3 normal;
varying vec3 camera;

void main()
{
        vec4 ambient = vec4(0.0); // final ambient color
        vec4 diffuse = vec4(0.0); // final diffuse color
        vec4 specular = vec4(0.0); // final specular color

        // distance from the light
        // can be used to compute attenuation
        float dist = length(lightVector);

        // normalized light vector, surface normal and eye vector
        vec3 L = normalize(lightVector);
        vec3 N = normalize(normal);
        vec3 C = normalize(camera);

        .....
        .....
```

Simple per-pixel lighting shader

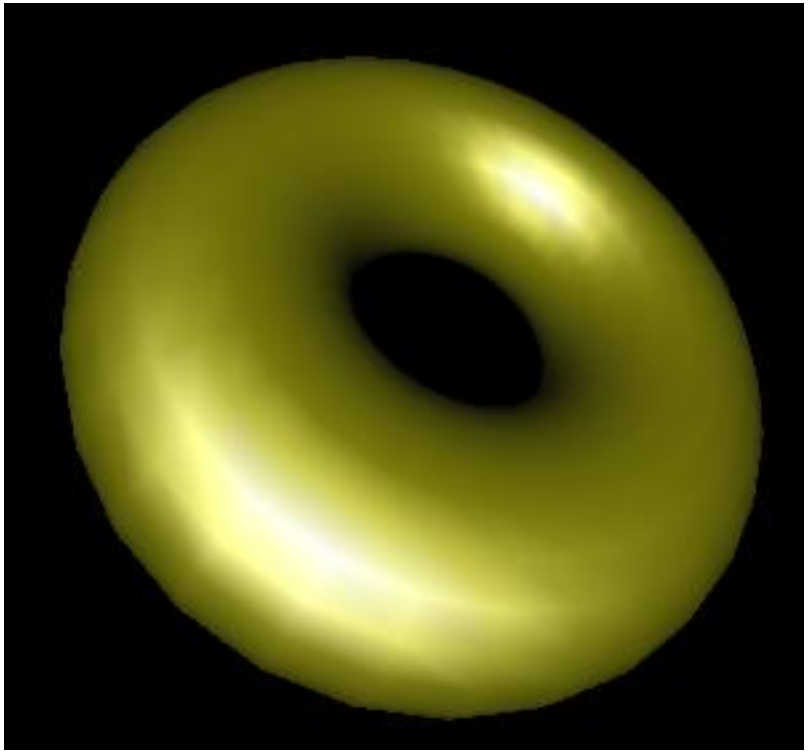
Fragment Shader



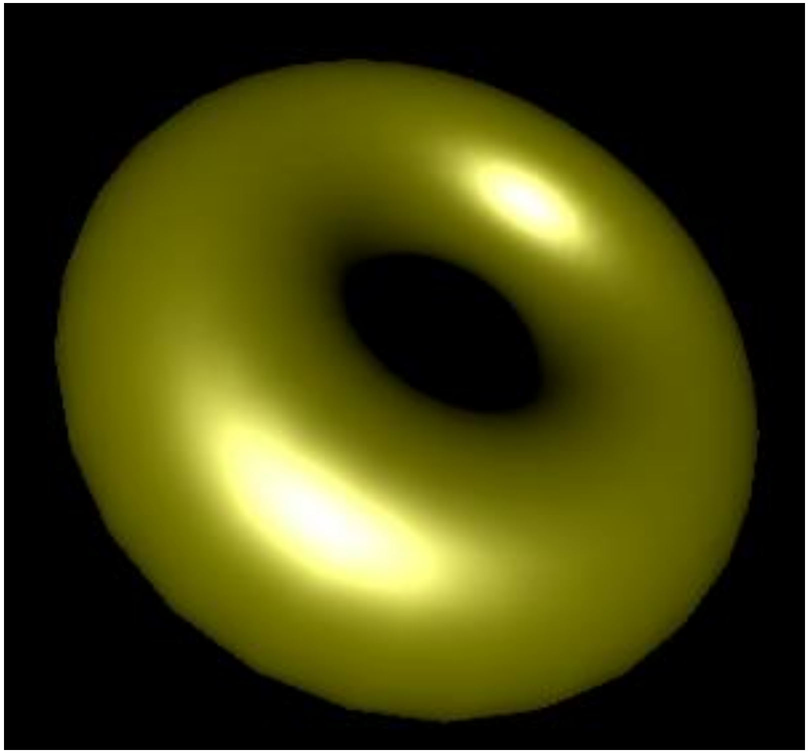

```
        float diffuseFactor = max(0.0, dot(N, L));
        float specularFactor = 0.0;

        if(diffuseFactor != 0.0) {
                vec3 halfVector = normalize(C + L);
                specularFactor = pow(dot(halfVector, N),
gl_FrontMaterial.shininess);
        }

        ambient = gl_FrontMaterial.ambient * gl_LightSource[0].ambient;

        diffuse = gl_FrontMaterial.diffuse * gl_LightSource[0].diffuse
                        * diffuseFactor;

        specular = gl_FrontMaterial.specular * gl_LightSource[0].specular
                        * specularFactor;
```

Simple per-pixel lighting shader

OpenGL fixed function

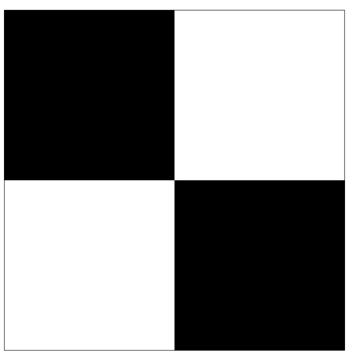


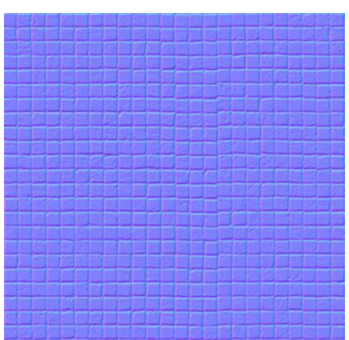
Per-pixel lighting

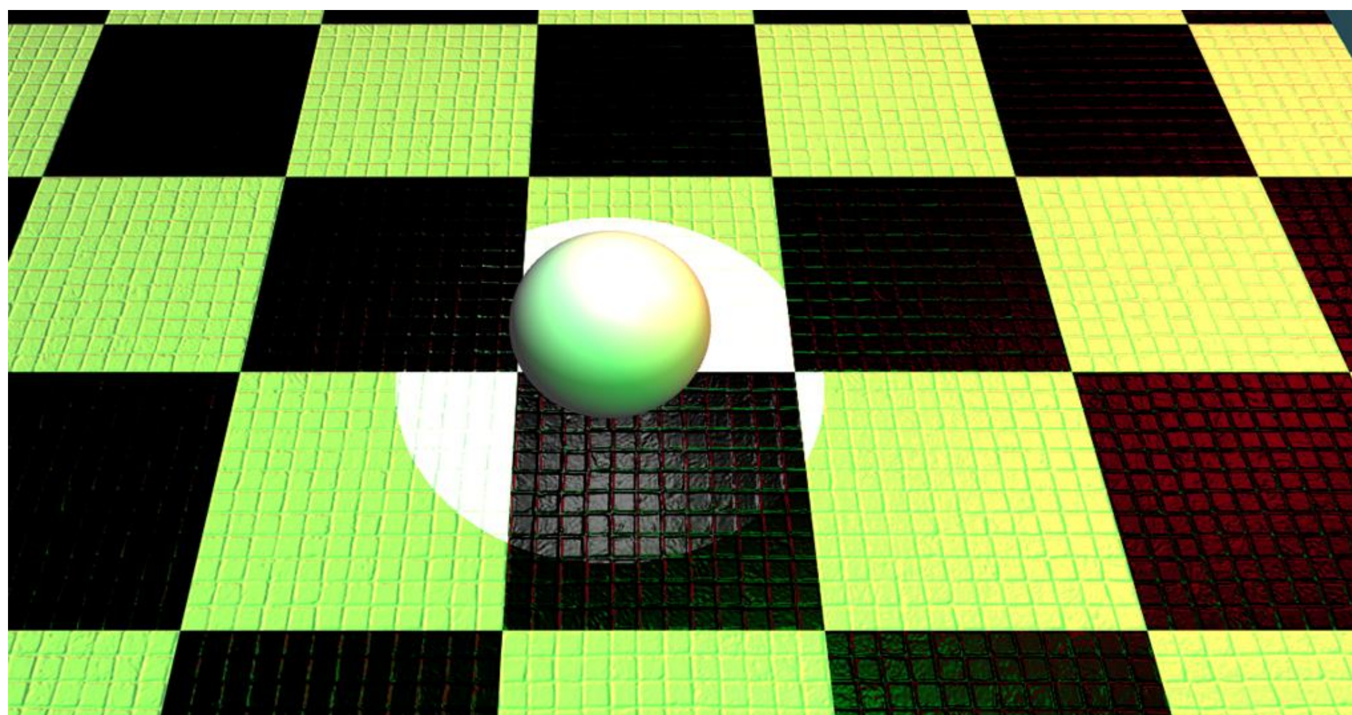


Bump mapping

- Normals are now encoded in a normal map
- We use a texture for the material's diffuse and ambient color







Bump mapping

- In order to use this normals, we need the axes of the surface's tangent space:
  - We already have the normal vector (`gl_Normal`)
  - We need to pass the tangent vector as an attribute/uniform
  - We compute binormal taking the cross product of the other two vectors
- Once we have the tangent space, we can build a rotation matrix (TBN) that transforms a vector in VCS to a vector in tangent space.
- We use TBN to port all vectors we need (light, camera, …) in tangent space and then do the lighting computations.

Bump mapping

Vertex Shader

```
// bumpmapping.vert

#version 120 // to use the transpose function

uniform vec3 tangent;
varying vec3 lightVector;
varying vec3 camera;

void main()
{
        vec3 position = (gl_ModelViewMatrix * gl_Vertex).xyz;

        // Tangent space
        vec3 N = gl_NormalMatrix * gl_Normal;      // normal
        vec3 T = gl_NormalMatrix * tangent;                // tangent
        vec3 B = cross(N, T);
        // binormal
        mat3 TBN = transpose(mat3(T, B, N));               // TBN matrix

        // Light vector in tangent space
        vec3 temp = gl_LightSource[0].position.xyz - position;
        lightVector  = TBN * temp;

        // Camera vector in tangent space
        camera = TBN * (-position);

        gl_Position    = gl_ModelViewProjectionMatrix * gl_Vertex;
```

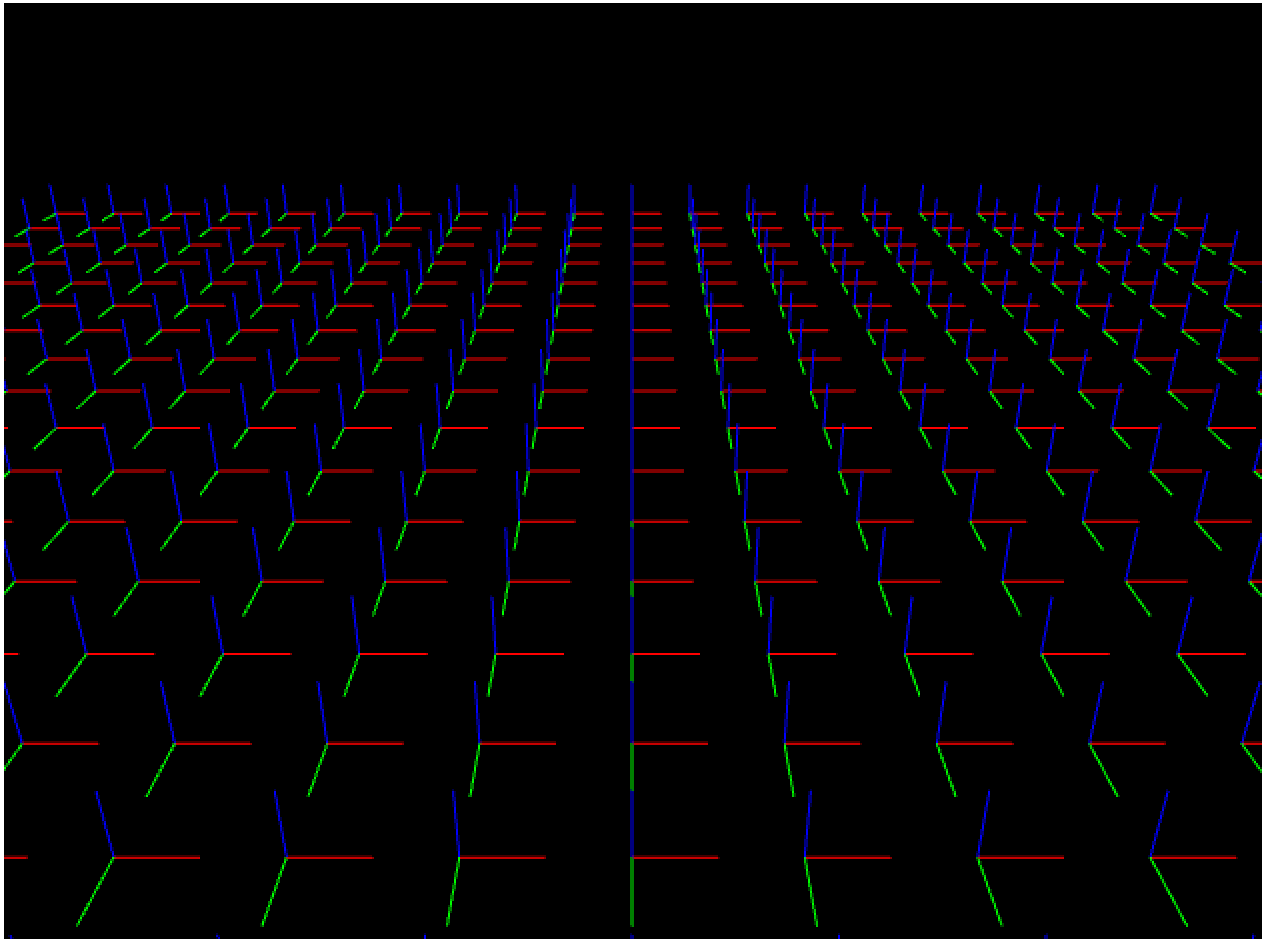
Bump mapping

Fragment Shader

```
// bumpmapping.frag

varying vec3 lightVector;
varying vec3 camera;

uniform sampler2D diffuse; // material ambient and diffuse texture
uniform sampler2D normalMap; // normal map texture

void main()
{

        // Sample fragment ambient and diffuse color
        vec4 matDiffuse = texture2D(diffuse, gl_TexCoord[0].st);

        // Sample surface normal from the normal map
        vec3 normal = texture2D(normalMap, gl_TexCoord[0].st).xyz * 2.0 -
1.0;

        ::: Do the lighting computations with Phong model :::

        ambient = matDiffuse * gl_LightSource[0].ambient;
        diffuse = matDiffuse * gl_LightSource[0].diffuse * diffuseFactor;
        specular = gl_LightSource[0].specular * specularFactor;

        gl_FragColor = ambient + diffuse + specular;

}
```

Simple water effect

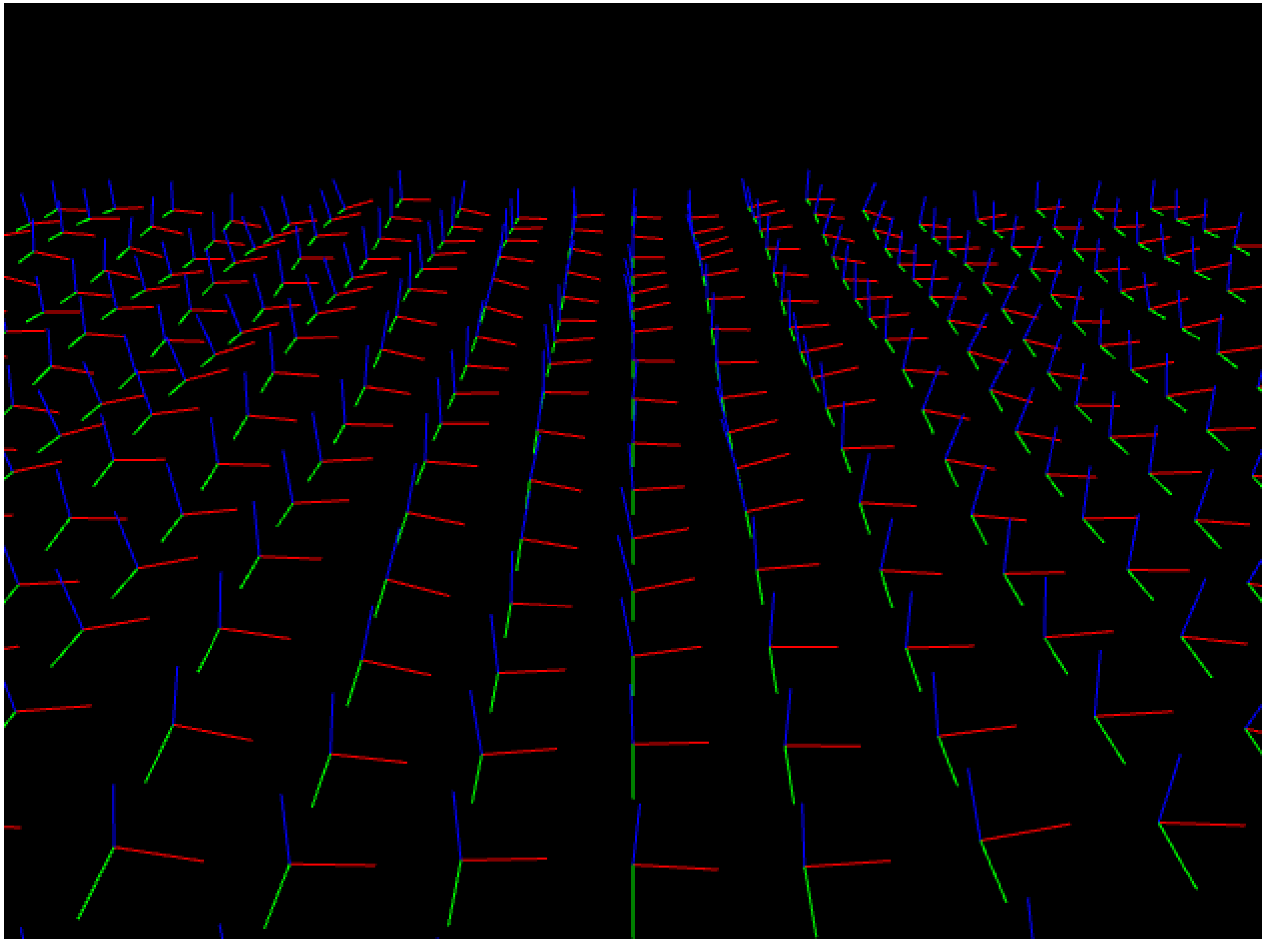
We use a grid of quads laying on the xz-plane to model water surface



Simple water effect

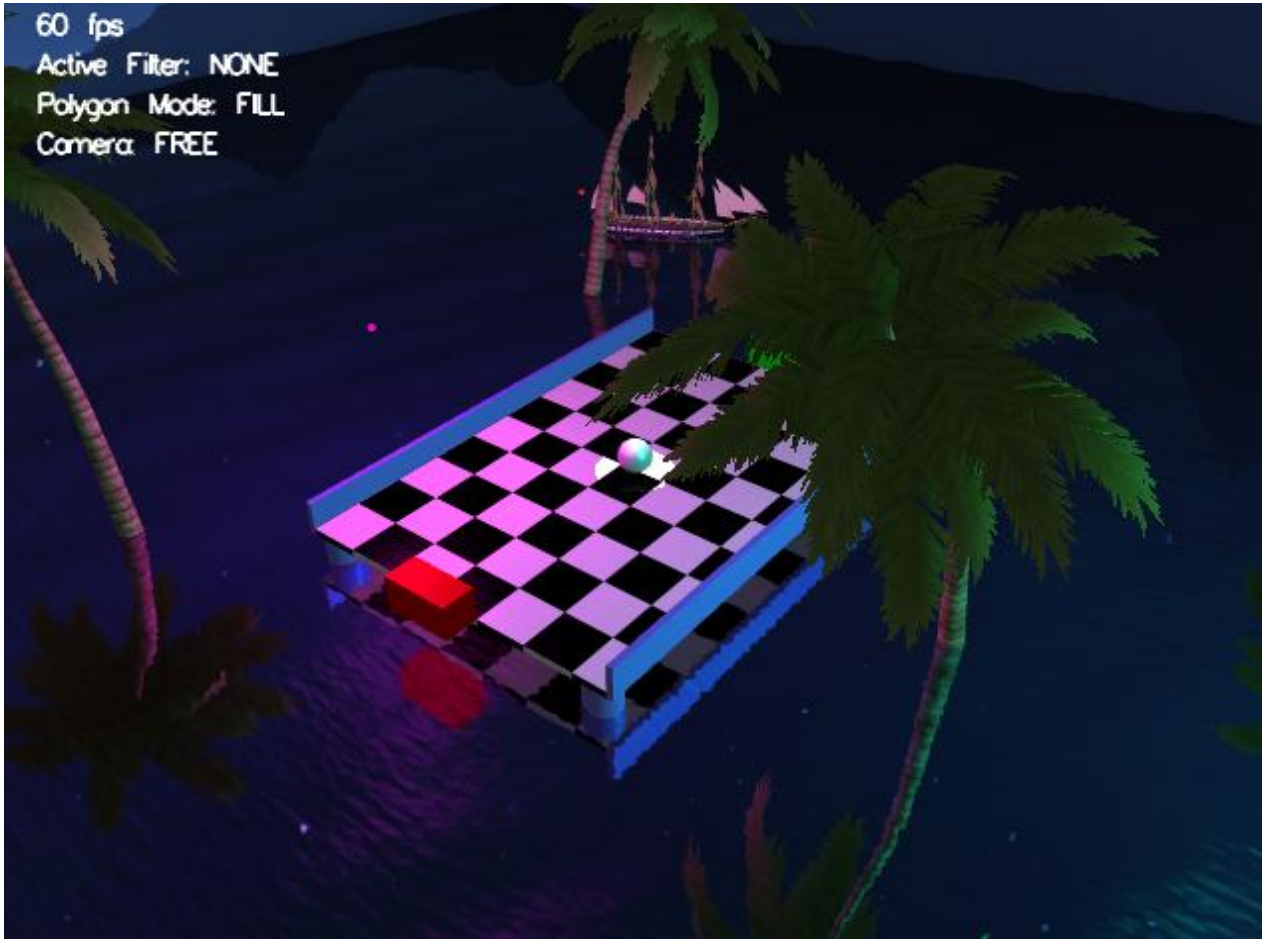
Our vertex shader will apply a function to each vertex:

```
v(x, y, z) = (x, f(x, z, time), z)
```



Simple water effect

Use a dynamic bump map (surface details) and a reflection texture



Using shaders in C/C++ applications

- Creating a shader:

```
Gluint glCreateShader(GLenum shaderType)
```

creates a shader and returns an integer ID

- Compile a shader:

```
void glShaderSource(GLuint shader, GLsizei count,
     const GLchar ** string, const GLint * length)
```

sets up the shader source code

```
void glCompileShader(GLuint shader)
```

compiles the shader

Using shaders in C/C++ applications

- Once VS and FS are created, they must be attached to a program and linked:

```
GLuint glCreateProgram(void)
void glAttachShader(GLuint program, Gluint shader)
void glLinkProgram(GLuint program)
```

- A program is ready to be used after linking:

```
....
glUseProgram(myProgram); // use the program myProgram
// draw stuff
glUseProgram(0); // back to OpenGL fixed function
.....
```

Using shaders in C/C++ applications

- Setting up uniform variables:

```
GLint glGetUniformLocation(GLuint program, const GLchar * name)
void glUniform{1|2|3|4}{f|i} (GLint location, ...)
void glUniform{1|2|3|4}{f|i}v (GLint location, GLsizei count, ...)
```

- Passing textures to the shader:

```
....
glActiveTexture(GL_TEXTURE_0 + textureUnit);
glBindTexture(GL_TEXTURE_2D, myTexture);
glUniform1i(location, textureUnit);
....
```

Reference

- TyphoonLabs' GLSL tutorials
- http://www.opengl.org/sdk/docs/tutorials/TyphoonLabs/
- Clockworkcoders tutorials
- http://www.opengl.org/sdk/docs/tutorials/ClockworkCoders/
- OpenGL SDK Documentation
- http://www.opengl.org/sdk/docs/
- Game Rendering
- http://www.gamerendering.com/
- GLSL wiki
- http://en.wikipedia.org/wiki/GLSL