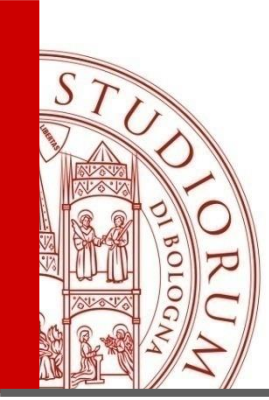


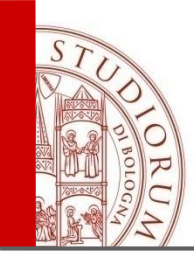
An Introduction to **Programmable Shaders**

Fondamenti di Computer Graphics

A.A. 2018/2019

Ed Angel SIGGRAPH:

“An Introduction to OpenGL Programming”

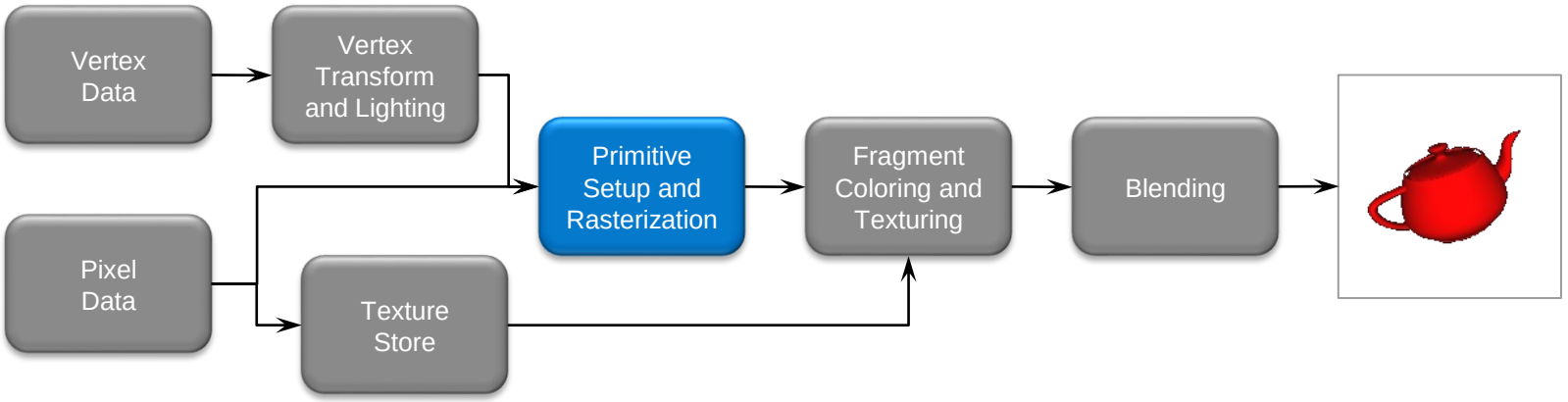


Programmable Shaders

- Introduce programmable pipelines
  - Vertex shaders
  - Fragment shaders
- Introduce shading language: GLSL
- Linking shaders with openGL programs
- Shader Examples and Exercises

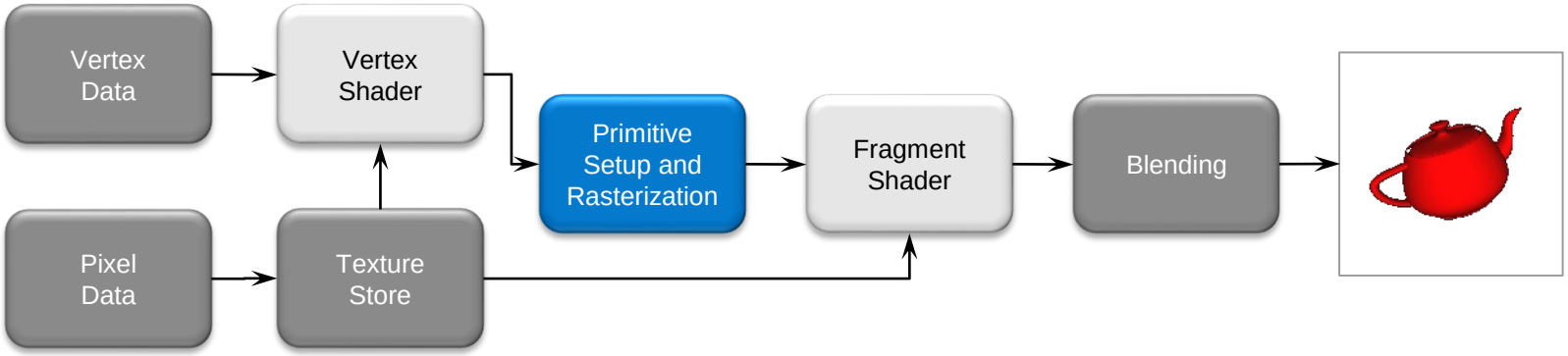
Tratto da: Angel, Interactive Computer Graphics , © Addison-Wesley
Angel, An Introduction to Modern OpenGL Programming,

In the Beginning …

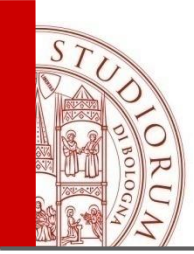


- OpenGL 1.0 was released on July 1st, 1994
- Its pipeline was entirely fixed-function
  - the only operations available were fixed by the implementation
- The pipeline evolved, but remained fixed-function through OpenGL versions 1.1 through 2.0 (Sept. 2004)

The Start of the Programmable Pipeline



- OpenGL 2.0 (officially) added programmable shaders
  - *vertex shading* augmented the fixed-function transform and lighting stage
  - *fragment shading* augmented the fragment coloring stage
- However, the fixed-function pipeline was still available

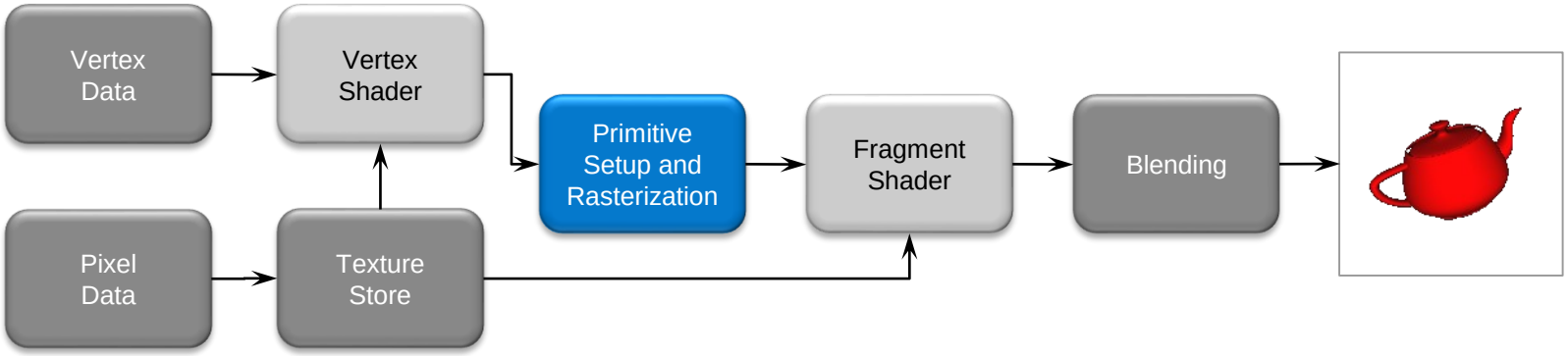


An Evolutionary Change

- OpenGL 3.0 introduced the *deprecation model*
  - the method used to remove features from OpenGL
- The pipeline remained the same until OpenGL 3.1 (released March 24th, 2009)
- Introduced a change in how OpenGL contexts are used

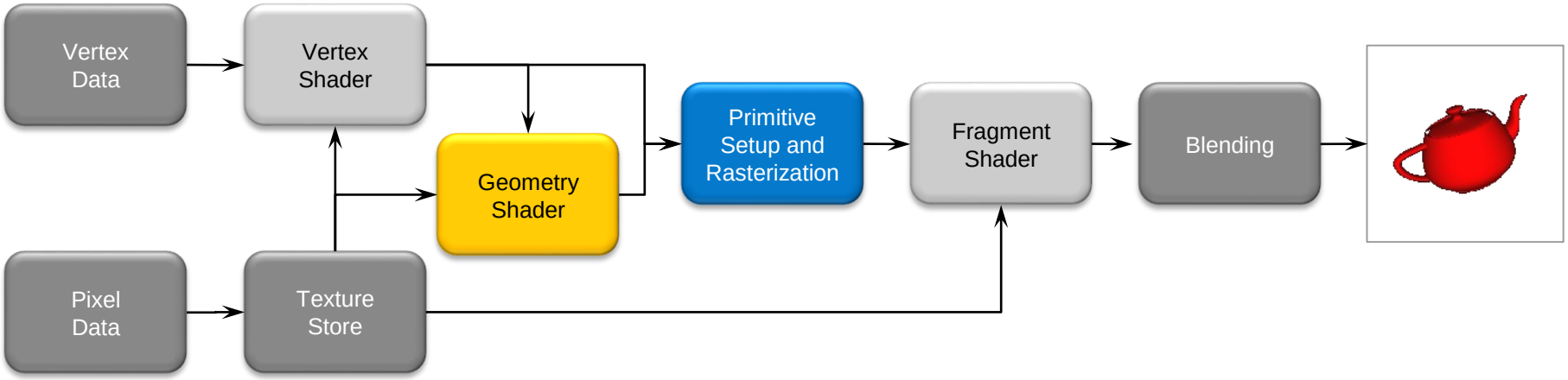
Context Type	Description
Full	Includes all features (including those marked deprecated) available in the current version of OpenGL
Forward Compatible	Includes all non-deprecated features (i.e., creates a context that would be similar to the next version of OpenGL)

The Exclusively Programmable Pipeline

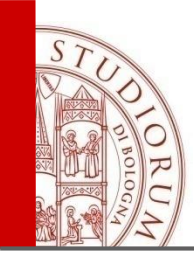


- OpenGL 3.1 removed the fixed-function pipeline
  - programs were required to use only shaders
- Additionally, almost all data is GPU-resident
  - all vertex data sent using buffer objects

More Programmability



- OpenGL 3.2 (released August 3rd, 2009) added an additional shading stage – geometry shaders
- The geometry shader, as opposed to the vertex shader, has full knowledge of the primitive it is working on.

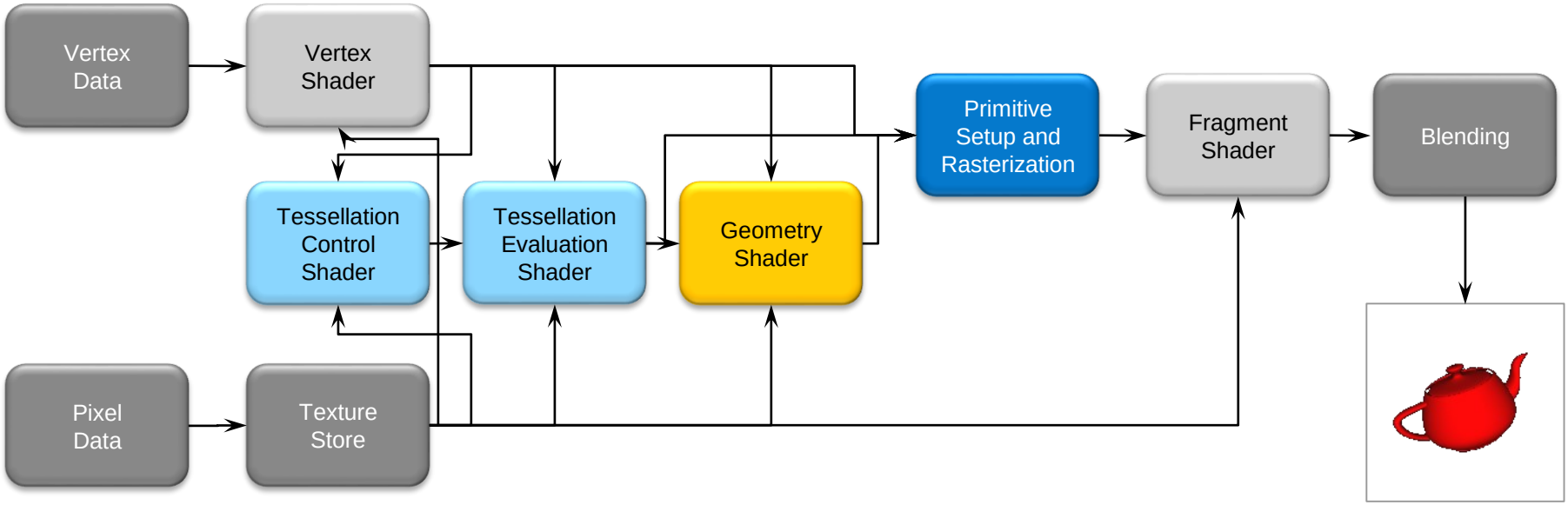


Context Profiles

- OpenGL 3.2 also introduced *context profiles*
  - profiles control which features are exposed
  - currently two types of profiles: *core* and *compatible*

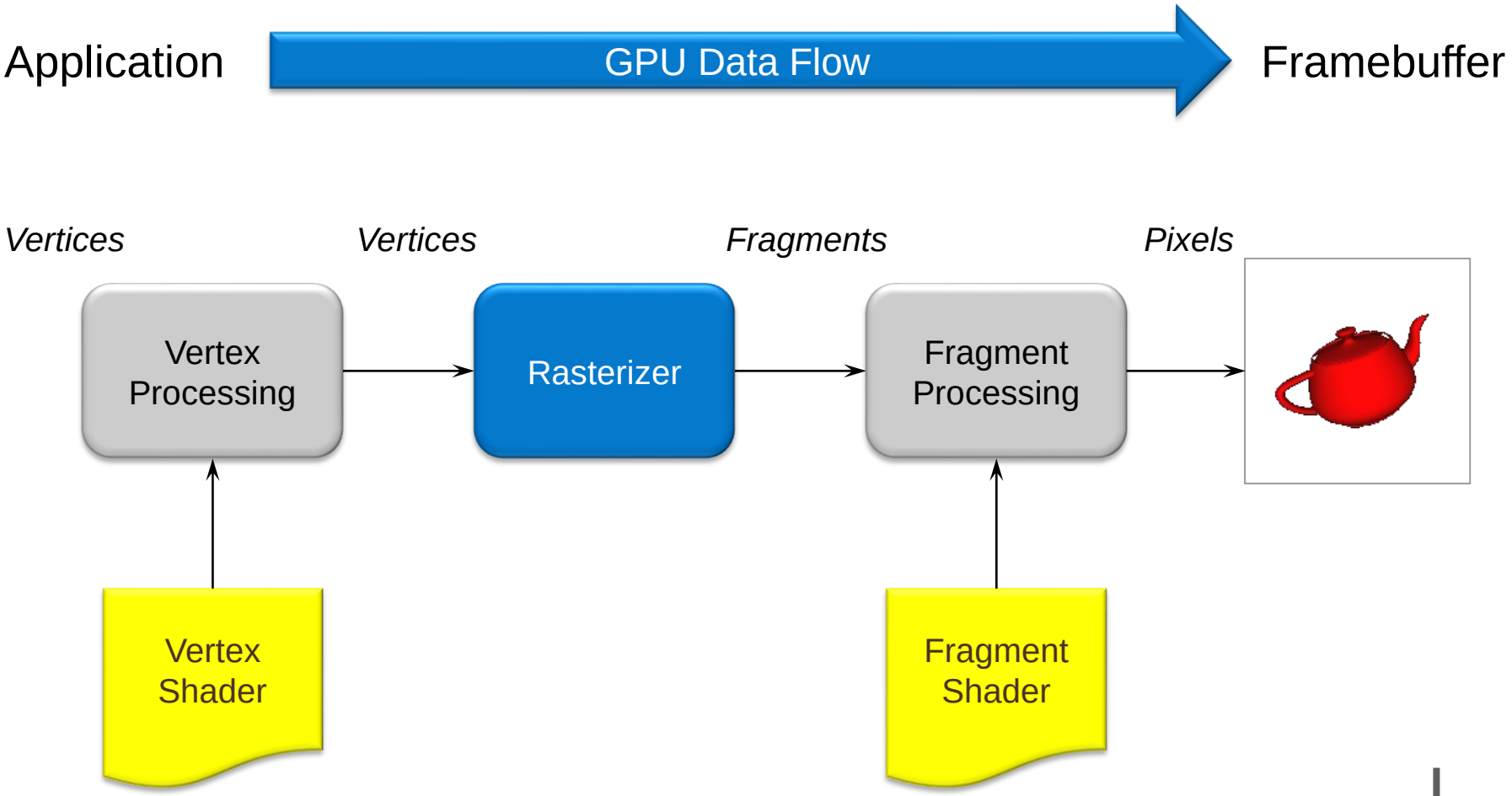
Context Type	Profile	Description
Full	core	All features of the current release
	compatible	All features ever in OpenGL
Forward Compatible	core	All non-deprecated features
	compatible	Not supported

The Latest Pipeline



- OpenGL 4.0 (Released on March 11, 2010) included additional shading stages – tessellation-control and tessellation-evaluation shaders
- Latest: OpenGL 4.5 (Released on August 2014)

A Simplified Pipeline Model



A Simplified Pipeline Model

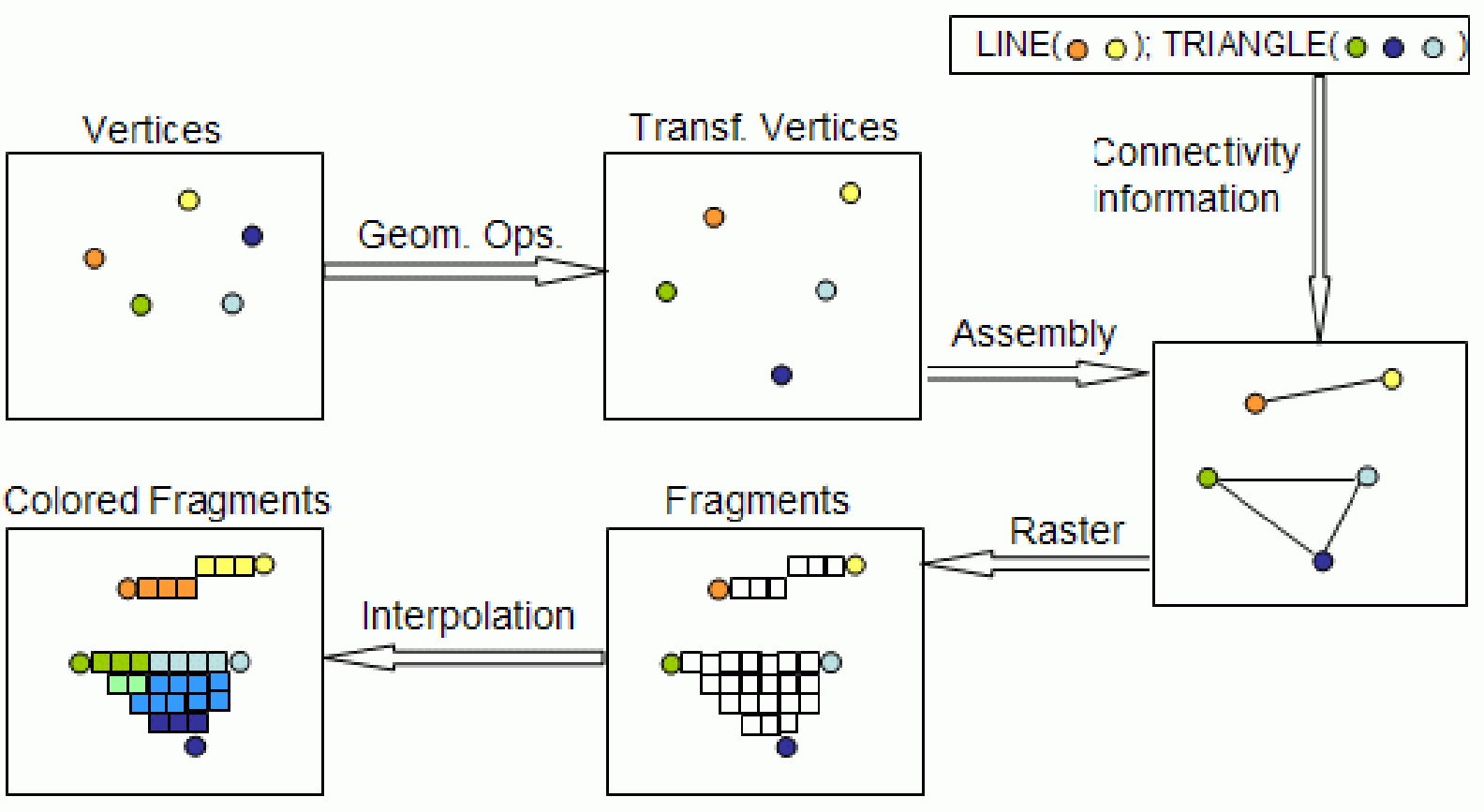


Image credit: lighthouse3d.com

Steps in pipeline

- Application: specifies geometric objects through sets of vertices and attributes which are sent to GPU

- Graphics processing Unit (GPU) must produce set of pixels in the Frame Buffer
  - Geometric (Vertex) processing
  - **Rasterization**
  - Fragment Processing

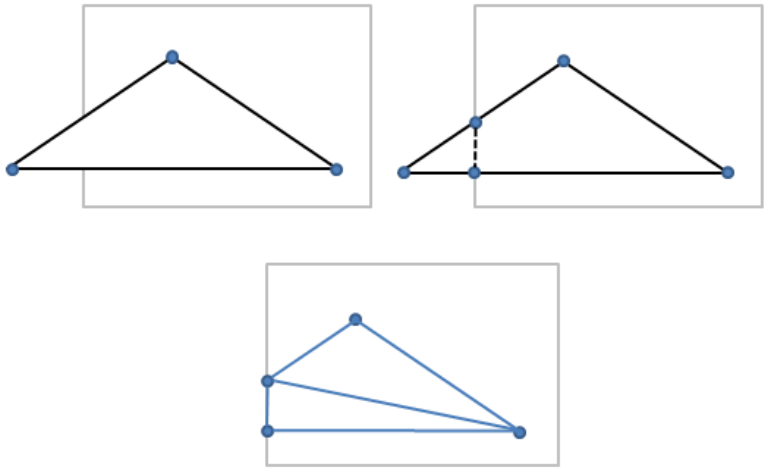
Vertex Processor

- Input: (Vertex) Position, Color, Normal, Texture…
- Operations:
  - Vertex position are transformed by the **Model-View matrix** into eye coordinates
  - normal transformation
  - **Lighting** computations per vertex
  - Generation and transformation of texture coordinates
- Output
  - Vertex Position after projection into 2D (clip coordinates), before viewport transform
  - Vertex color

- **NB: the vertex processor has no information regarding connectivity**

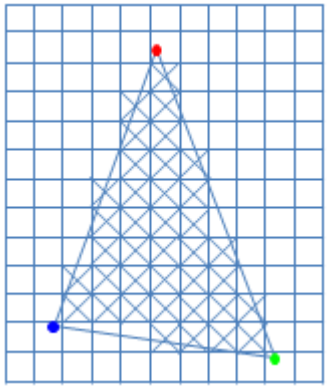
Rasterization: Primitive Assembly

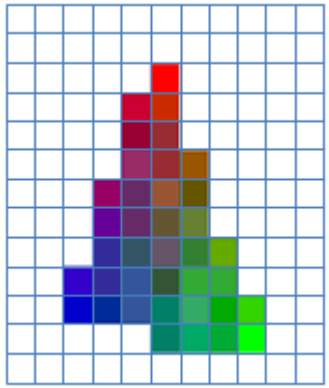
- Vertices are next assembled into objects
  - Polygons
  - Line Segments
  - Points
- Transformation by **Projection matrix**
- **Clipping**
  - Back face culling
  - View Frustum
- Perspective Division
- **Viewport mapping**



Rasterization: fragment generation

- Geometric entities are rasterized into fragments
- Each fragment corresponds to a point on an integer grid: a displayed pixel
- Hence each fragment is a potential pixel
- Each fragment has
  - A color
  - Possibly a depth value
  - Texture coordinates

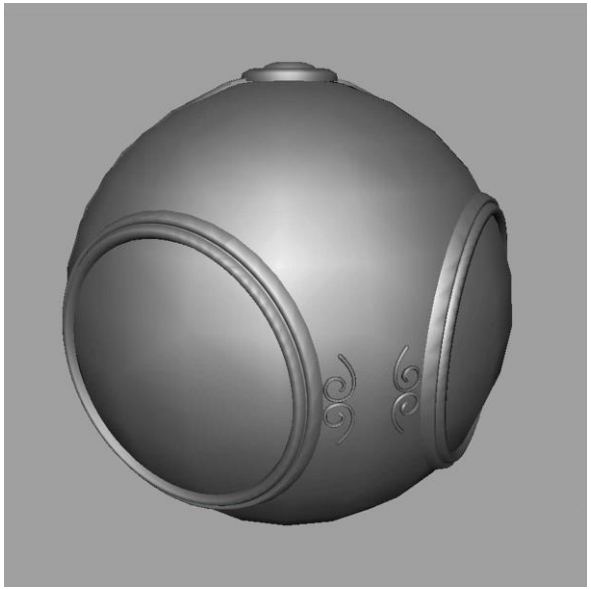




Fragment Processor

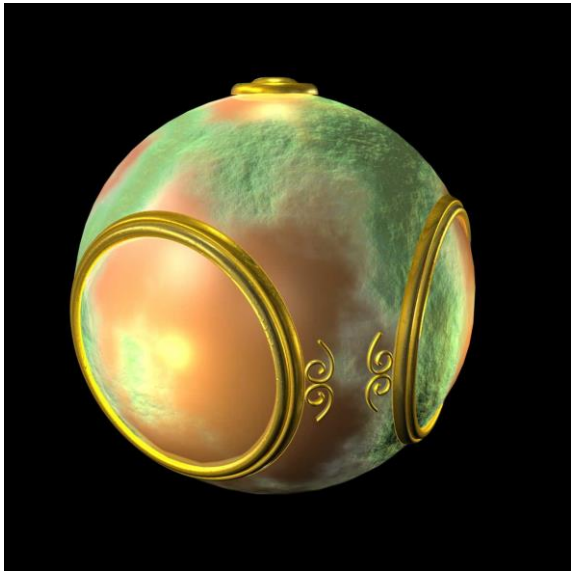
- Takes **in** output of rasterizer (fragments)
  - Vertex values have been interpolated over primitive by rasterizer
- **Outputs** a fragment (Color,Texture)
- Common tasks in the fragment shader
  - Compute Color per-fragment
  - Texture coordinate per-pixel , apply texture
  - Normal per-pixel
  - Lighting per-pixel
  - Compute Fog or other global effects
- Fragments still go through fragment tests
  - **Visibility test: Hidden-surface removal**
  - alpha

Fragment Shader Example Applications



smooth shading

environment mapping



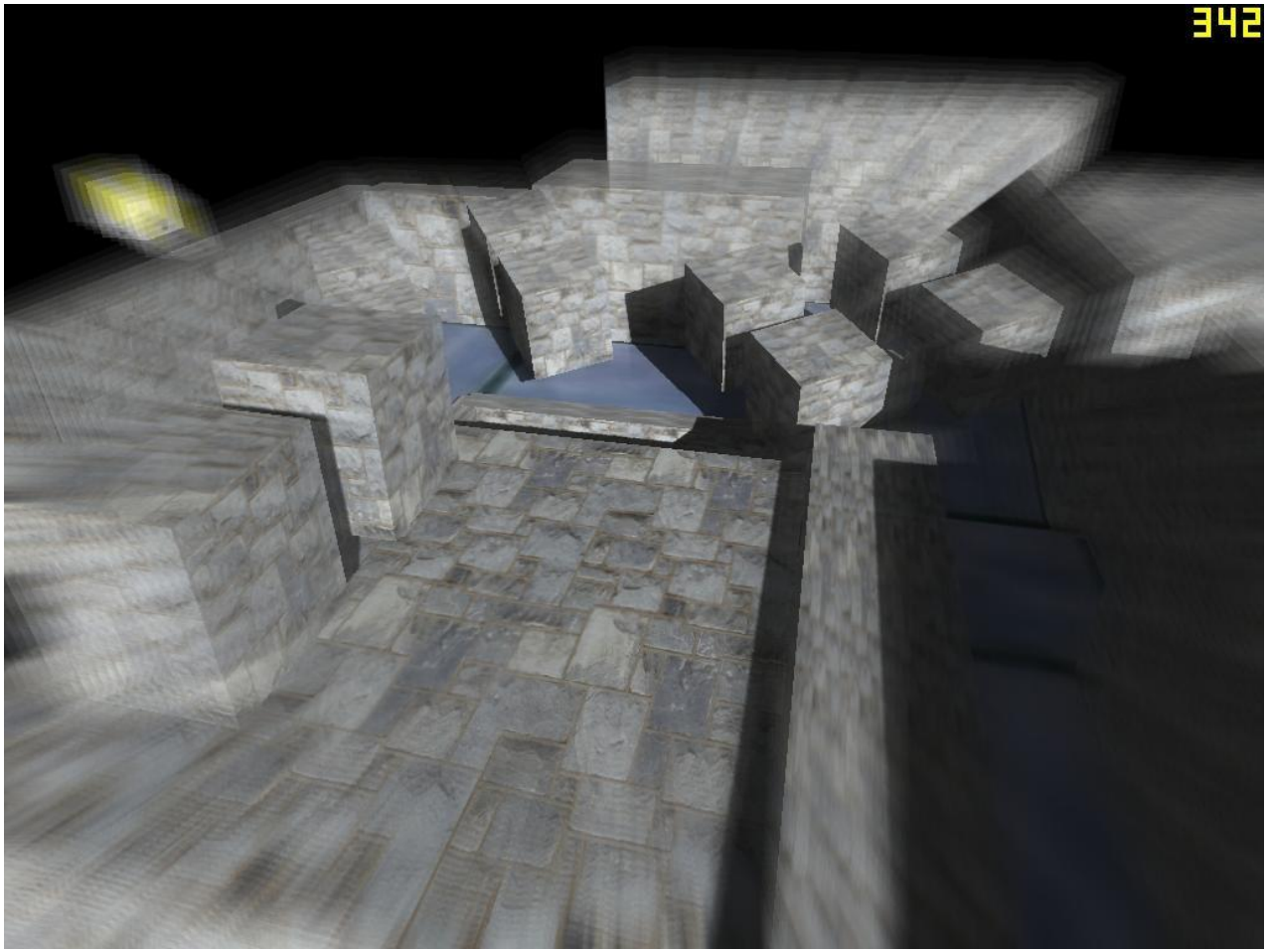
bump mapping

Toon shading

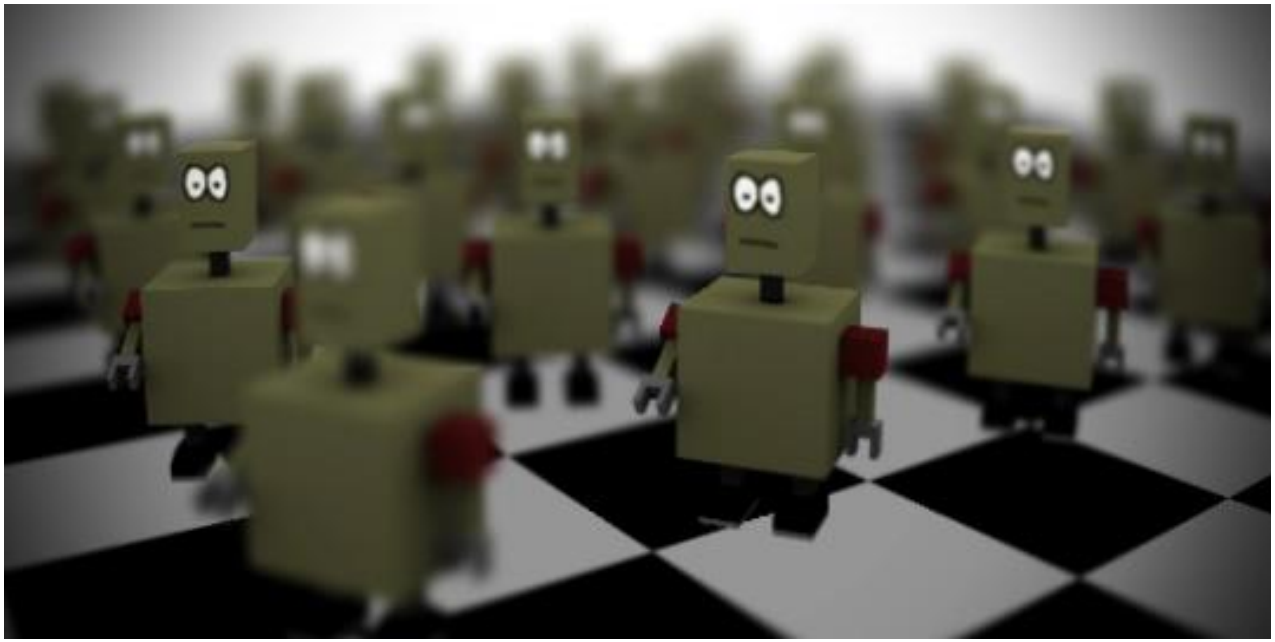
Advanced lighting

Refractions

Motion blur



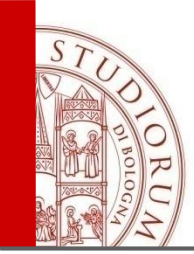
Depth of field



Advanced effects

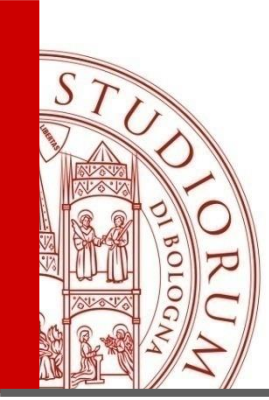
Programmable Shaders

- Replace fixed function vertex and fragment processing by programmable processors called shaders
- Can replace either or both
- If we use a programmable shader we must do all required functions of the fixed function processor



Writing Shaders

- First programmable shaders were programmed in an assembly-like manner
- OpenGL extensions added for vertex and fragment shaders
- **Cg** (C for graphics) C-like language for programming shaders
  - Works with both OpenGL and DirectX
  - Author: nvidia
- OpenGL Shading Language (**GLSL**) is part of OpenGL 2.0 and is based on C
  - Author: the Khronos Group, a self-sponsored group of industry affiliates (ATI, 3DLabs, etc)

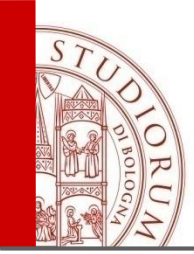


GLSL OpenGL Shading Language

High level C-like language

New data types: Matrices, Vectors, Samplers

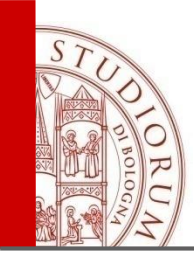
OpenGL state available through built-in variables



GLSL Syntax Overview

- GLSL is like C without
  - Pointers
  - Recursion
  - Dynamic memory allocation
- GLSL is like C with
  - Built-in vector, matrix and sampler types
  - Constructors
  - A great math library
  - Input and output qualifiers

Allow us to write concise, efficient shaders.



GLSL Data Types

Scalar types: float, int, bool

Vector types: vec2, vec3, vec4
ivec2, ivec3, ivec4
bvec2, bvec3, bvec4

Matrix types: mat2, mat3, mat4

Texture sampling: sampler1D, sampler2D,
sampler3D,
samplerCube

GLSL Syntax: Data Types

- **Scalar** types: `float`, `int`, `uint`, `bool`
- **Vectors** are also built-in types:
  - `vec2`, `vec3`, `vec4`
  - `ivec*`, `uvec*`, `bvec*`

Access components three ways:

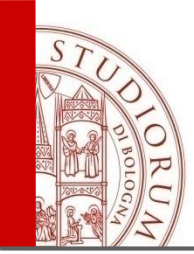
– .x, .y, .z, .w position or direction

– .r, .g, .b, .a color

– .s, .t, .p, .q texture coordinate

```
myColor.xy
    z
```

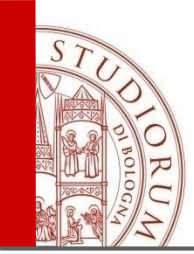
```
myColor.xg
    b
```



GLSL Syntax: Vectors

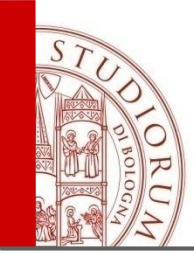
Constructors

```
vec3 xyz = vec3(1.0, 2.0, 3.0);

vec3 xyz = vec3(1.0); // [1.0, 1.0, 1.0]

vec3 xyz = vec3(vec2(1.0, 2.0), 3.0);

vec2 v = vec2(1.0, 2.0);   // compose from values

v = vec2(3.0, 4.0);        // doesn't have to be in
                           // an initialization
vec4   u = vec4(0.0); // initialized all elements to 0

vec2   t = vec2(u);  // take the first two components

vec3   vc = vec3(v, 1.0); // compose vec2 and float
```



Swizzling and Selection

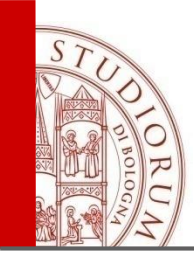
- Access to vector components using `[]` or `.` operators:
  - `x, y, z, w`
  - `r, g, b, a`
  - `s, t, p, q`
  - `a[2] == a.b == a.z == a.p`
- Swizzling operator lets us select multiple components from the vector types:
  - `vec4 a;`
  - `a.yz = vec2(1.0, 2.0);`
  - `a.xy = a.yx;  /* swap elements */`



GLSL Syntax: Vectors

- Swizzle: select or rearrange components

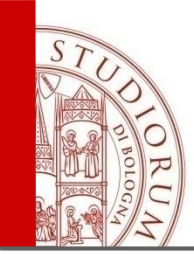
```
vec4 c = vec4(0.5, 1.0, 0.8, 1.0);

vec3 rgb = c.rgb;  // [0.5, 1.0, 0.8]

vec3 bgr = c.bgr;  // [0.8, 1.0, 0.5]

vec3 rrr = c.rrr;  // [0.5, 0.5, 0.5]

c.a = 0.5;         // [0.5, 1.0, 0.8, 0.5]
c.rb = 0.0;        // [0.0, 1.0, 0.0, 0.5]

float g = rgb[1];  // 0.5, indexing, not swizzling
```



GLSL Syntax: Matrices

- **Matrices** are built-in types:
  - Square: `mat2`, `mat3`, `mat4`
  - Rectangular: `mat`*`m`*`x`*`n`* `m` columns, `n` rows

- Stored column major



GLSL Syntax: Matrices

- Constructors

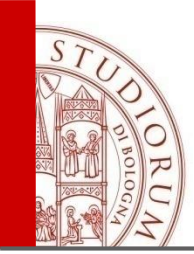
```
mat3 i = mat3(1.0); // 3x3 identity matrix

mat2 m = mat2(1.0, 2.0,  // [1.0 3.0]
              3.0, 4.0); // [2.0 4.0]
```

- Accessing elements

```
float f = m[column][row];

float x = m[0].x; // x component of first column

vec2 yz = m[1].yz; // yz components of second column
```

Treat matrix as array of column vectors

Can swizzle too!



GLSL Syntax: Vectors and Matrices

- Matrix and vector operations are easy and fast
- Operators overloaded for matrix and vector operations:

```
vec3 xyz = // ...

vec3 v0 = 2.0 * xyz; // scale
vec3 v1 = v0 + xyz;  // component-wise
vec3 v2 = v0 * xyz;  // component-wise

mat3 m = // ...
mat3 v = // ...

mat3 mv = v * m;        // matrix * matrix
mat3 xyz2 = mv * xyz; // matrix * vector
mat3 xyz3 = xyz * mv; // vector * matrix
```

GLSL Syntax: Samplers

- Opaque types for accessing textures

```
uniform sampler2D colorMap; // 2D texture

vec3 color = texture(colorMap, vec2(0.5, 0.5)).rgb;

vec3 colorAbove = textureOffset(colorMap,
  vec2(0.5, 0.5), ivec2(0, 1)).rgb;

vec2 size = textureSize(colorMap, 0);

// Lots of sampler types:  sampler1D,
// sampler3D, sampler2DRect, samplerCube,
// isampler*, usampler*, ...

// Lots of sampler functions: texelFetch,
   textureLod
```

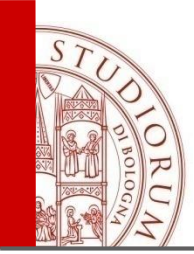
Samplers must be uniforms

`texture ()` returns a `vec4`; extract the components you need

2D texture coordinate

2D texture uses 2D texture coordinates for lookup

2D integer offset



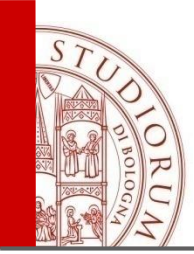
GLSL Built-in Functions

- Selected trigonometry functions

```
float s = sin(theta);
float c = cos(theta);
float t = tan(theta);

float theta = asin(s);
// ...

vec3 angles = vec3(/* ... */);
vec3 vs = sin(angles);
```

Angles are measured in radius

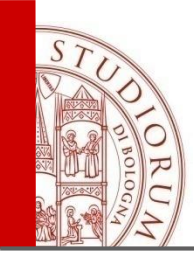
Works on vectors component-wise



GLSL Built-in Functions

- Exponential Functions

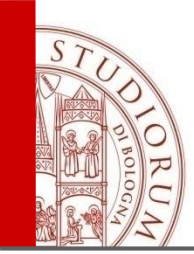
```
float xToTheY = pow(x, y);
float eToTheX = exp(x);
float twoToTheX = exp2(x);

float l = log(x);   // ln
float l2 = log2(x); // log2

float s = sqrt(x);
float is = inversesqrt(x);
```



GLSL Built-in Functions

- Selected common functions

```
float ax = abs(x);  // absolute value
float sx = sign(x); // -1.0, 0.0, 1.0

float m0 = min(x, y); // minimum value
float m1 = max(x, y); // maximum value
float c = clamp(x, 0.0, 1.0);

// many others:  floor(), ceil(),
// step(), smoothstep(), ...
```



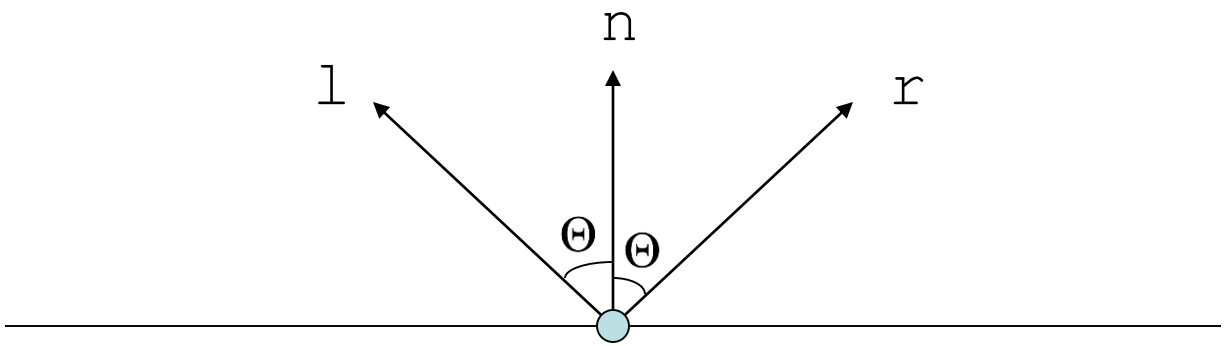
GLSL Built-in Functions

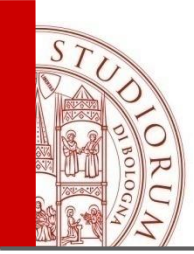
- Selected geometric functions

```
vec3 l = // ...
vec3 n = // ...
vec3 p = // ...
vec3 q = // ...

float f = length(l);        // vector length
float d = distance(p, q); // distance between points

float d2 = dot(l, n);       // dot product
vec3 v2 = cross(l, n);      // cross product
vec3 v3 = normalize(l);     // normalize

vec3 v3 = reflect(l, n);    // reflect

// also:  faceforward() and refract()
```

GLSL Built-in Functions

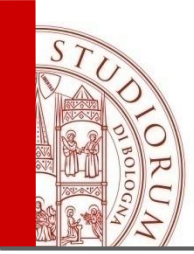
- `reflect(-l, n)`
  - Given l and n, find r. Angle in equals angle out





Flow Control

- if
- if else
- expression ? true-expression : false-expression
- while, do while
- for



OpenGL Programming in a Nutshell

- GLSL programs essentially do the following steps:
  - Create shader programs
  - Create buffer objects and load data into them
  - “Connect” data locations with shader variables
  - Render

Simple Vertex Shader (pass-through)

```
in vec4 vPosition;
in vec4 vColor;
out vec4 color;
uniform mat4 ModelView;
uniform mat4 Projection;

void main()
{
gl_Position = Projection*ModelView*vPosition;
color = vColor;
}
```

in: shader input varies per vertex attribute

out: shader output

uniform: shader input constant across `glDraw`

A trivial vertex shader that transforms every vertex according to the Model View and Projection matrix, mimicking the fixed function. Each execution of glVertex() invokes our shader with a new vertex value (4D vector)

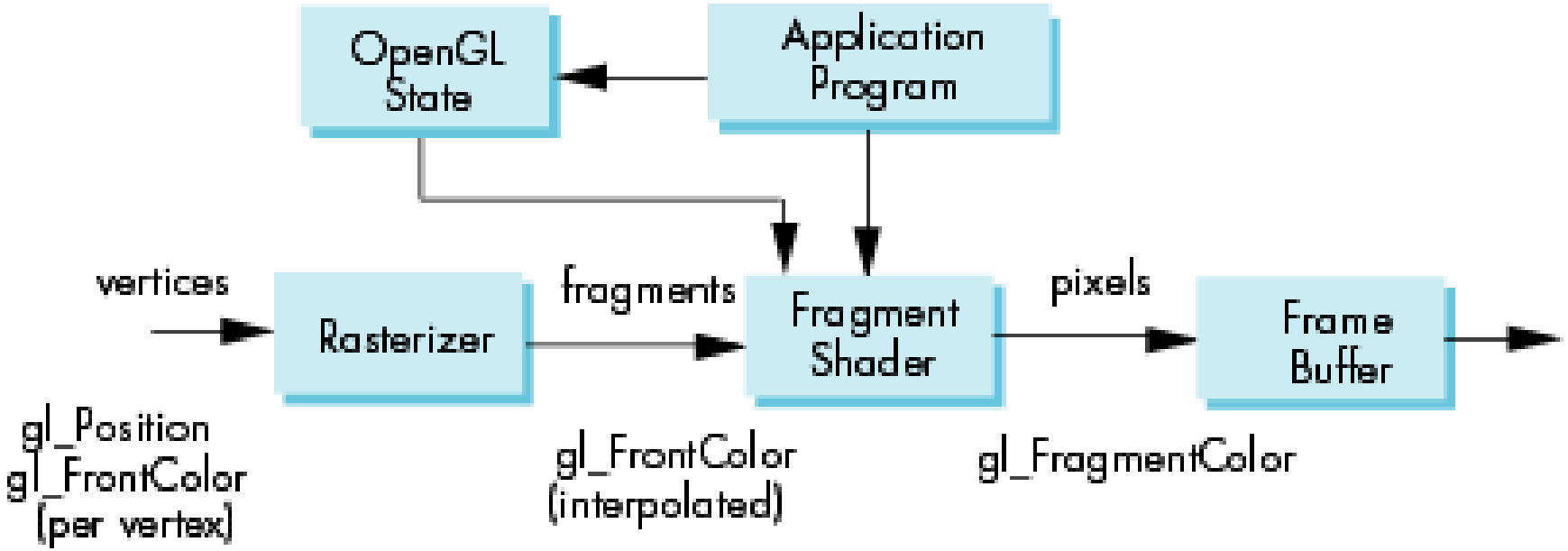
Simple Fragment Shader (pass-through)

```
in vec4 color;
out vec4 FragColor;
void main()
{
    FragColor = color;
    //gl_FragColor = color;
}
```

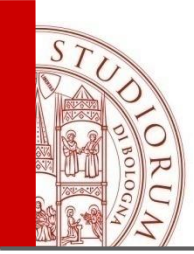
Executed after the rasterizer, and thus operates on each fragment of every displayed primitive

`color` has been produced by interpolating (in the rasterizer) the data **`color`** of the vertices

Fragment shader architecture



The minimal requirement of a fragment shader is to assign a color to the fragment by setting **`gl_FragColor`** or discard it with ‘discard ’



Variable Qualifiers

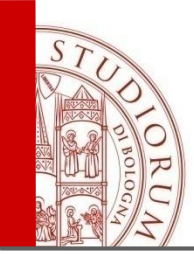
- Qualifiers give a special meaning to the variable.

- `in, out`

  Copy vertex attributes and other variable to/ from shaders

  ```
  in vec2 tex_coord;
  out vec4 color;
  ```

- `uniform` – Global variables that may change per primitive and are passed from the OpenGL application to the shaders.
  - Used for sharing data among an application program, vertex shaders, and fragment shaders.
  - Read-only (cannot be changed in shader)



Initialize Uniform Variable Values in the application

The commands glUniform{1|2|3|4}{f|i|ui} are used to inizialize the value of the uniform variable specified by *location* using the values passed as arguments.

```
GLint location = glGetUniformLocation(ShaderProgram, “name”);

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

Examples:

```
glUniform3f( location, x, y, z );
```

void **glUniform3f**(	GLint *location*,
	GLfloat *v0*,
	GLfloat *v1*,
	GLfloat *v2*);

```
GLfloat  mat[3][4][4] = {…};

glUniformMatrix4fv(
   location,3,transpose, mat);
```

void **glUniformMatrix4fv**(	GLint *location*,
	GLsizei *count*,
	GLboolean *transpose*,
	const GLfloat **value*);

Uniform Variable Example

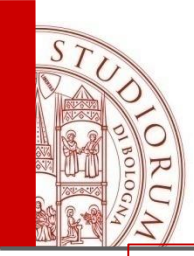
Application

Read-Only and in
the vertex and fragment shader

```
GLuint timeParam;

void init() {
prog = initShader("v.glsl", "f.glsl");
/* Otteniamo i puntatori alle variabili uniform per poterle utilizzare in seguito */
timeParam = glGetUniformLocation(prog,"time");
}

void draw(void){
float time_s = 2.0;
/* Communicate the variable value to the shader */
glUniform1f(timeParam, time_s);
...
}
```

Number of Components

Type of Components



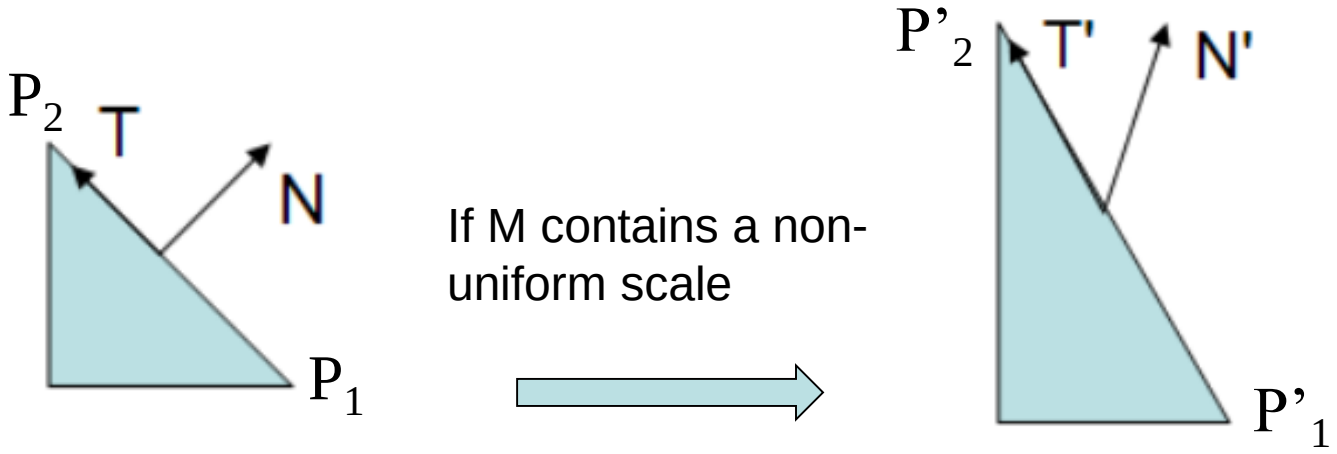
Uniform Variable Example

Vertex SHADER

```
uniform float time;
uniform float vx, vy, vz;
uniform mat4 ModelView;
uniform mat4 Projection;
out     vec4 Color;

void main()
{
    const float   a = -0.000001;
    vec4          t = gl_Vertex;
    t.y = gl_Vertex.y + vy*time+0.5*a*time*time;
    t.x = gl_Vertex.x + vx*time;
    t.z = gl_Vertex.z + vz*time;
    gl_Position = Projection * ModelView * t;
    Color =  gl_Color;
}
```

The Normal Matrix

- Vertex locations are transformed by the model-view matrix M into VCS (eye coordinates)



$$T = P_2 - P_1$$

$$MT = M(P_2 - P_1) = MP_2 - MP_1$$

T' $\quad$ P_2' $\quad$ P_1'

- Preserves the tangents. The transformed normal is no longer perpendicular to the surface!

The Normal Matrix

- Consider a matrix *G*, and lets see how this matrix could be computed to properly transform the normal vectors:

$$N \cdot T = 0 \quad \text{and it must remain} \quad N' \cdot T' = 0$$

$$GN \cdot MT = 0$$

$$(GN)^T (MT) = 0$$

$$N^T G^T MT = 0 \quad \rightarrow If \quad G^T M = I \rightarrow N \cdot T = 0$$

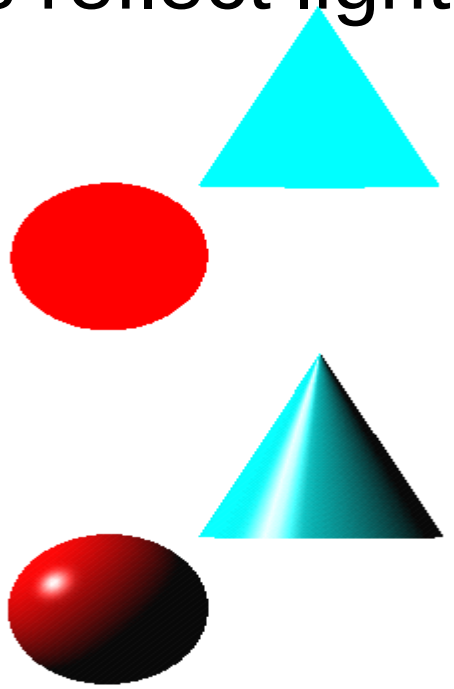
$$G^T M = I \quad \Leftrightarrow \quad G = (M^{-1})^T$$

- Normals must be transformed with the inverse transpose of the model-view matrix M.
- Precalculate G on the CPU and pass it to the shader as a uniform:

```
mat4 normalMatrix = transpose(inverse(modelView));
```

Lighting Principles

- Lighting simulates how objects reflect light
  - material composition of object
  - light's color and position
  - global lighting parameters
- Usually implemented in
  - vertex shader for faster speed
    - Gouraud shading
  - fragment shader for nicer shading
    - Phong shading



Gouraud shading

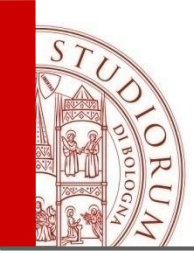
- Computes a color for each vertex using
  - Surface normals
  - Diffuse and specular reflections
  - Viewer's position and viewing direction
  - Ambient light
  - Emission
- Vertex colors are interpolated across polygons by the rasterizer
  - *Phong shading* does the same computation per pixel, interpolating the normal across the polygon
    - more accurate results

Material Properties

- Define the surface properties of a primitive

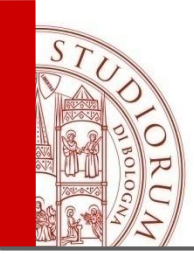
Property	Description
Diffuse	Base object color
Specular	Highlight color
Ambient	Low-light color
Emission	Glow color
Shininess	Surface smoothness

  - you can have separate materials for front and back

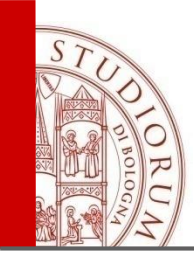


Vertex shader for Gouraud shading

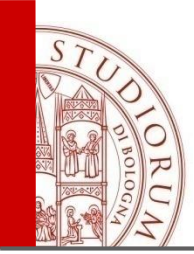
```
// vertex shader

in vec4 vPosition;
in vec3 vNormal;
out vec4 color;

struct Material {
    vec4 ambient;
    vec4 diffuse;
    vec4 specular;
    float shininess;
};
uniform Material material;
uniform mat4 M;  //Model
uniform mat4 V;   //View
uniform mat4 P;   //Projection

uniform vec4 LightPosition;
uniform vec4 LightPower
```



```
void main()
{
   gl_Position = P * V * M * vPosition;

   // Transform vertex position into VCS
   vec3 eyePosition = vec3( V * M * vPosition);
    // Transform LightPos into VCS
   vec3 eyeLightPos = vec3( V * LightPosition.xyz);

    // Transform vertex normal into VCS
   vec3 N =
       normalize(mat3(transpose(inverse(V * M)))* vNormal);
   vec3 L = normalize(eyeLightPos - eyePosition);
   vec3 E = normalize(- eyePosition);
   vec3 R = reflect(-L, N);
```



```
    // Compute terms in the illumination equation
    vec4 ambient = LightPower * material.ambient;

    // diffuse
    float Kd = max( dot(L, N), 0.0 );
    vec4  diffuse = Kd * LightPower * material.diffuse;

    // specular
    float Ks = pow(max(dot(E, R),0.0),material.shininess);
    vec4  specular = Ks * LightPower * material.specular;

    color = ambient + diffuse + specular;
    color.a = 1.0;
}
```



COMMUNICATION WITH THE HOST OPENGL PROGRAM

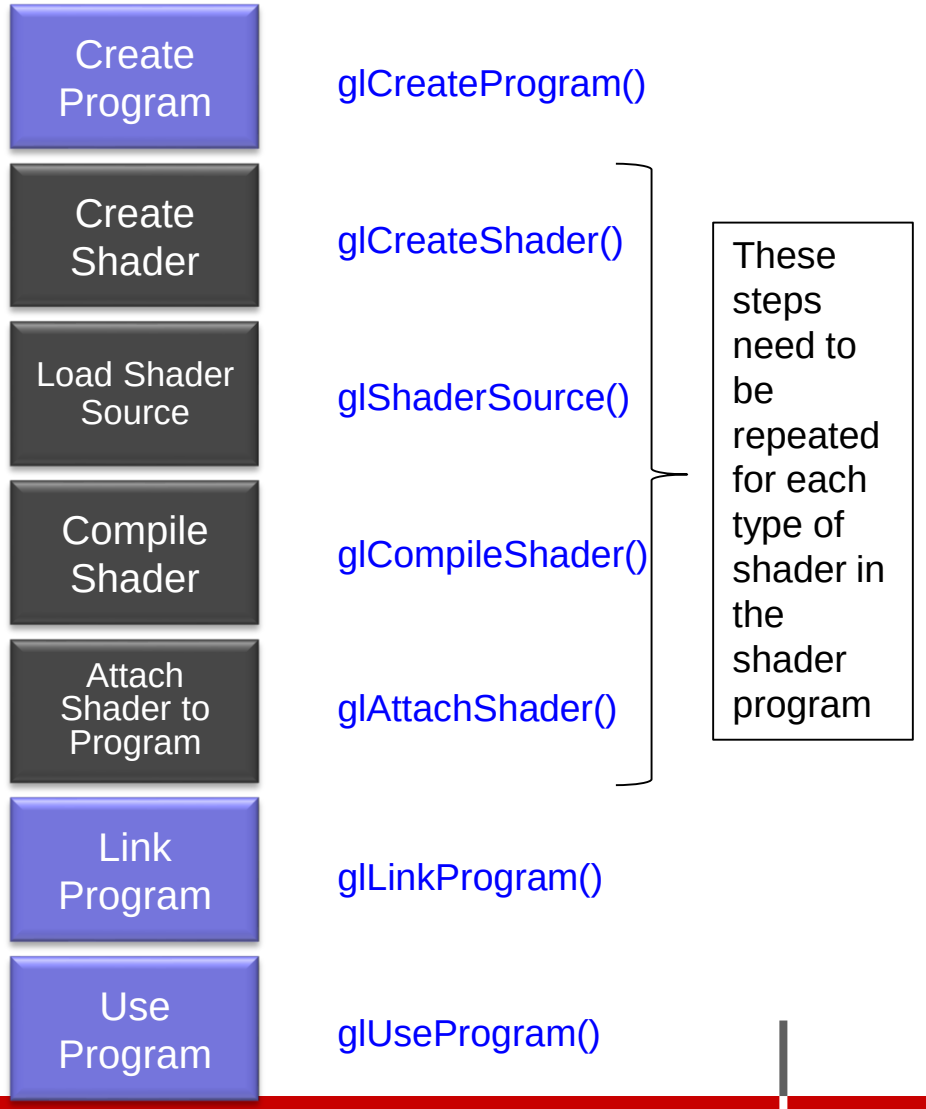
Getting Your Shaders into OpenGL

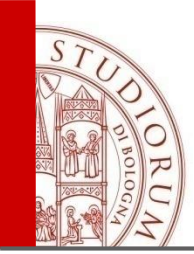
Shaders need to be compiled and linked to form an executable shader program

OpenGL provides the compiler and linker

A program must contain

- vertex and fragment shaders
- other shaders are optional





A Simpler Way

- We've created a routine for this course to make it easier to load your shaders
  - See `common/common.h`

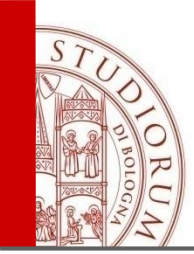
```
IdS = initShaders(const char* vFile, const char* fFile);
```

- InitShaders takes two filenames
  - `vFile` for the vertex shader
  - `fFile` for the fragment shader

- Fails if shaders don't compile, or program doesn't link

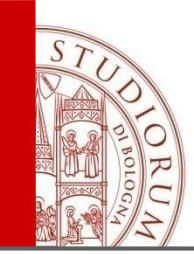
- The value returned from `InitShaders()` will be a valid GLSL program id that you can pass into `glUseProgram()`

```
glUseProgram(IdS);
```



Shader Examples

Shader examples in GLSL

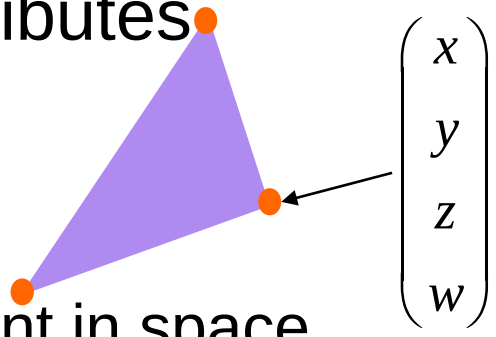


Example 1: CUBE!

- We'll render a cube with colors at each vertex
- Our example demonstrates:
  - initializing vertex data
  - organizing data for rendering
  - simple object modeling
    - building up 3D objects from geometric primitives
    - building geometric primitives from vertices

Vertex Buffer Object (VBO)

- A vertex is a collection of generic attributes
  - positional coordinates
  - colors
  - texture coordinates
  - any other data associated with that point in space



- Position stored in 4 dimensional homogeneous coordinates
- Vertex data must be stored in

  vertex buffer objects (VBOs)
- VBOs must be stored in

  vertex array objects (VAOs)

Initializing the Cube's Data

- We'll build each cube face from individual triangles
- Need to determine how much storage is required
  - (6 faces)(2 triangles/face)(3 vertices/triangle)

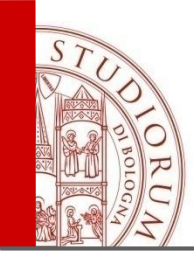
```
const int NumVertices = 36;
```

- To simplify communicating with GLSL, we'll use a `vec4` class (implemented in C++) similar to GLSL's `vec4` type
  - we'll also typedef it to add logical meaning

```
typedef  vec4  point4;
typedef  vec4  color4;
```

- We create two arrays to hold the VBO data

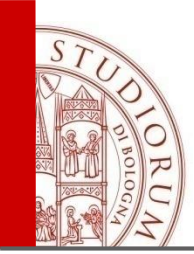
```
point4  vPositions[NumVertices];
color4  vColors[NumVertices];
```



Cube Data

- Vertices of a unit cube centered at origin
  - sides aligned with axes

```
point4 positions[8] = {
    point4( -0.5, -0.5,  0.5, 1.0 ),
    point4( -0.5,  0.5,  0.5, 1.0 ),
    point4(  0.5,  0.5,  0.5, 1.0 ),
    point4(  0.5, -0.5,  0.5, 1.0 ),
    point4( -0.5, -0.5, -0.5, 1.0 ),
    point4( -0.5,  0.5, -0.5, 1.0 ),
    point4(  0.5,  0.5, -0.5, 1.0 ),
    point4(  0.5, -0.5, -0.5, 1.0 )
};
```



Cube Data (cont'd)

- We'll also set up an array of RGBA colors

```
color4 colors[8] = {
    color4( 0.0, 0.0, 0.0, 1.0 ),  // black
    color4( 1.0, 0.0, 0.0, 1.0 ),  // red
    color4( 1.0, 1.0, 0.0, 1.0 ),  // yellow
    color4( 0.0, 1.0, 0.0, 1.0 ),  // green
    color4( 0.0, 0.0, 1.0, 1.0 ),  // blue
    color4( 1.0, 0.0, 1.0, 1.0 ),  // magenta
    color4( 1.0, 1.0, 1.0, 1.0 ),  // white
    color4( 0.0, 1.0, 1.0, 1.0 )   // cyan
};
```

Generating a Cube Face from Vertices

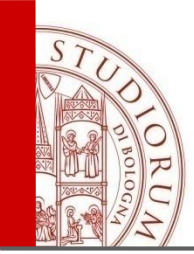
- To simplify generating the geometry, we use a convenience function polygon( )
  - create two triangles for each face and assigns colors to the vertices

```
int Index = 0;  // global variable indexing into VBO arrays

void polygon( int a, int b, int c, int d )
{
    vColors[Index] = colors[a]; vPositions[Index] = positions[a]; Index++;
    vColors[Index] = colors[b]; vPositions[Index] = positions[b]; Index++;
    vColors[Index] = colors[c]; vPositions[Index] = positions[c]; Index++;
    vColors[Index] = colors[a]; vPositions[Index] = positions[a]; Index++;
    vColors[Index] = colors[c]; vPositions[Index] = positions[c]; Index++;
    vColors[Index] = colors[d]; vPositions[Index] = positions[d]; Index++;
}
```

Generating the Cube from Faces

- Generate 12 triangles for the cube
  - 36 vertices with 36 colors

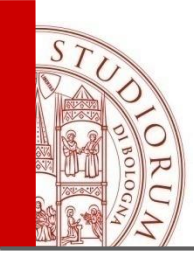
```
void colorcube()
{
 polygon( 1, 0, 3, 2 );
 polygon( 2, 3, 7, 6 );
 polygon( 3, 0, 4, 7 );
 polygon( 6, 5, 1, 2 );
 polygon( 4, 5, 6, 7 );
 polygon( 5, 4, 0, 1 );
}
```



VAO & VBOs in Code

In init()

```
// Create a vertex array object
GLuint vao;
glGenVertexArrays( 1, &vao );
glBindVertexArray( vao );

// 1st attribute VBO : positions
Gluint vpositionID;
glGenBuffers(1, &vpositionID);
glBindBuffer(GL_ARRAY_BUFFER, vpositionID);
glBufferData(GL_ARRAY_BUFFER,  sizeof(vposition),
        vposition, GL_STATIC_DRAW);

// 2nd attribute VBO : colors
Gluint vcolorID;
glGenBuffers(1, &vcolorID);
glBindBuffer(GL_ARRAY_BUFFER, vcolorID);
glBufferData(GL_ARRAY_BUFFER,   sizeof(vcolor),
        vcolor, GL_STATIC_DRAW);
```



VAO & VBOs in Code

```
In display()
void display( void )
{
// 1st attribute buffer : positions
glEnableVertexAttribArray(0);
glBindBuffer(GL_ARRAY_BUFFER, vpositionID);
glVertexAttribPointer(
          0,                  // attribute index in the shader
          3,                  // size
          GL_FLOAT,           // type
          GL_FALSE,           // normalized
          0,                  // stride
          (void*)0            // array buffer offset);

// 2nd attribute buffer : colors
glEnableVertexAttribArray(1);
glBindBuffer(GL_ARRAY_BUFFER, vcolorID);
glVertexAttribPointer(
          1, // attribute index.
          No particular reason for 1, but must match layout in the shader.
          3,                  // size
          GL_FLOAT,           // type
          GL_FALSE,           // normalized?
          0,                  // stride
          (void*)0            // array buffer offset );
```

Drawing Geometric Primitives

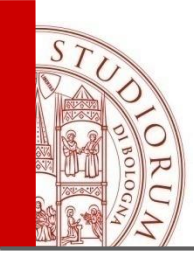
```
glDrawArrays( GL_TRIANGLES, 0, NumVertices );
glDisableVertexAttribArray(0);
glDisableVertexAttribArray(1);
glutSwapBuffers();
}
```

if you call glDrawArrays() specifiying 100 vertices, your vertex shader will be called 100 times, each time with a different vertex.

Vertex Shader for Rotation of Cube

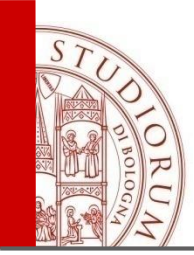
```
layout (location = 0) in vec4 vPosition;
layout (location = 1) in vec4 vcolor;

out vec4 color;
uniform vec3 theta;

void main()
{
    // Compute the sines and cosines of theta for
    // each of the three axes in one computation.
    vec3 angles = radians( theta );
    vec3 c = cos( angles );
    vec3 s = sin( angles );
```

`layout(location = 0)` refers to the buffer we use. Setting the layout to the same value as the first parameter to glVertexAttribPointer.



Vertex Shader for Rotation of Cube

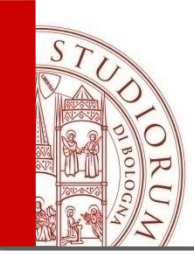
```
// Remember: these matrices are column-major

mat4 rx = mat4( 1.0,  0.0,  0.0, 0.0,
                0.0,  c.x,  s.x, 0.0,
                0.0, -s.x,  c.x, 0.0,
                0.0,  0.0,  0.0, 1.0 );

mat4 ry = mat4( c.y, 0.0, -s.y, 0.0,
                0.0, 1.0,  0.0, 0.0,
                s.y, 0.0,  c.y, 0.0,
                0.0, 0.0,  0.0, 1.0 );
```



Vertex Shader for Rotation of Cube

```
mat4 rz = mat4( c.z, -s.z, 0.0, 0.0,
                s.z,  c.z, 0.0, 0.0,
                0.0,  0.0, 1.0, 0.0,
                0.0,  0.0, 0.0, 1.0 );


    color = vColor;
    gl_Position = rz * ry * rx * vPosition;
}
```

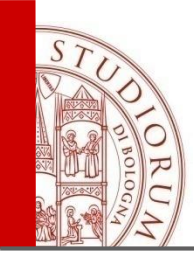
- Send the vertex data to the GPU once.
- Every time we update the rotation matrix in the vertex shader by sending only the rotation angles to the vertex shader and updating the vertex positions on the GPU.



Sending Angles from Application

- Here, we compute our angles (`Theta`) in our mouse callback

```
GLuint theta;  // theta uniform location
vec3  Theta;   // Axis angles

void display( void )
{
   glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

   glUniform3fv( theta, 1, Theta );
   glDrawArrays( GL_TRIANGLES, 0, NumVertices );

   glutSwapBuffers();
}
```



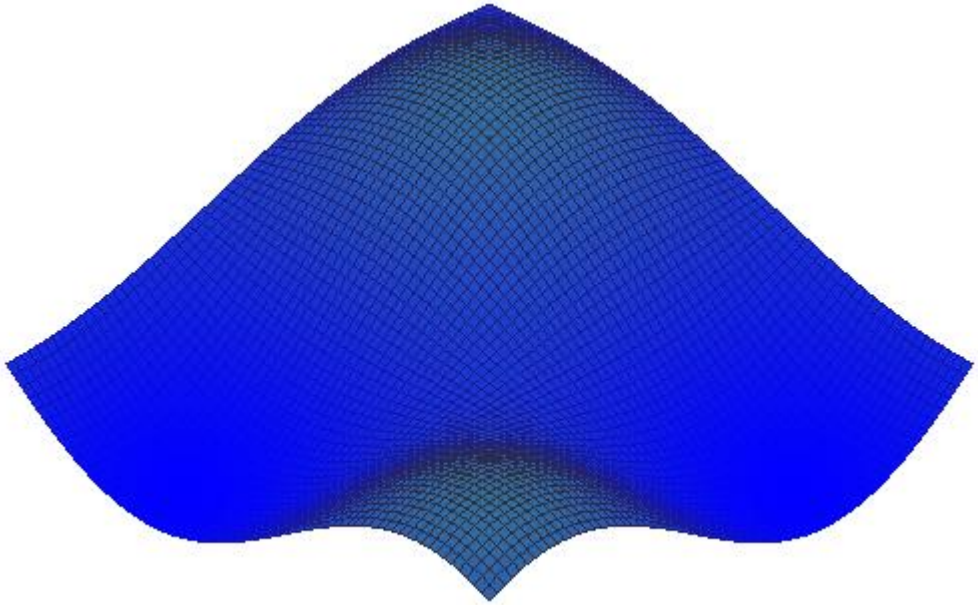
Example 1: Wave Motion

- See:

  `WAVE/wave.c`

  `WAVE/v.glsl`

  `WAVE/f.glsl`



- Exercise 1:

a. when the user clicks the left mouse button, set a different (random) amplitude

b. when the user clicks the right mouse button, set a different (random) oscillation frequency.

Example 2: Particle System

- See:

  `PARTICLE/particle.c`

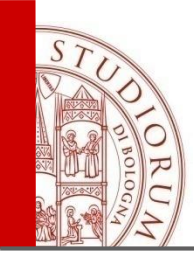
  `PARTICLE/v.glsl`

  `PARTICLE/f.glsl`

- Exercise 2:

a. height dependent point size. The higher, the bigger.

b. make it 3D



Particle System

```
uniform vec3 init_vel;
uniform float g, m, t;
// Values that stay constant for the whole mesh.
uniform mat4 P;
uniform mat4 V;
uniform mat4 M; // position*rotation*scaling

void main()
{
  vec3 object_pos;
  object_pos.x = gl_Vertex.x + vel.x*t;
  object_pos.y = gl_Vertex.y + vel.y*t + g/(2.0*m)*t*t;
  object_pos.z = gl_Vertex.z + vel.z*t;
  gl_Position = P * V * M * vec4(object_pos,1);
}
```

Example 3: Phong Lighting

See:

```
PHONG/phong.c
PHONG/v1.glsl
PHONG/f1.glsl
PHONG/v2.glsl
PHONG/f2.glsl
```

- Per-fragment

- Per-vertex

Exercise 3:

a. Add a third shader that implements the exact light reflection ray **r**

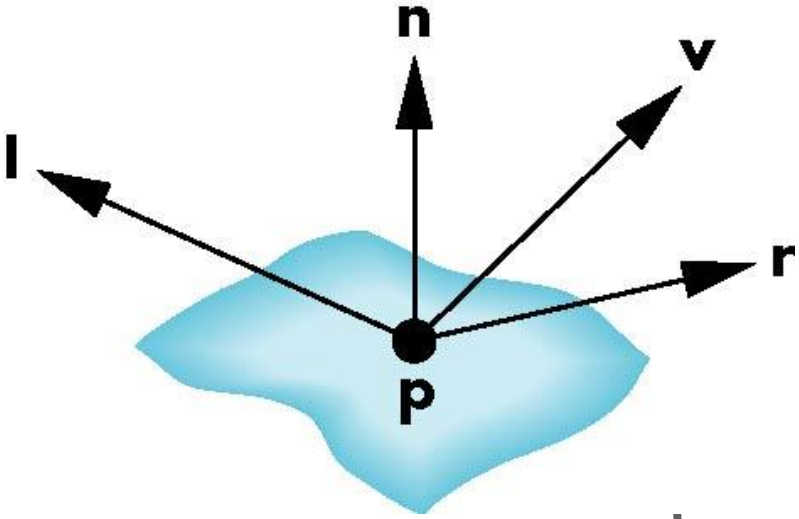
b. Show three teapots in the same scene with the tree different shaders

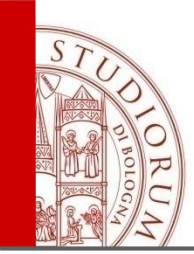
Phong Lighting

For each light source and each color component, the Phong model can be written (without the distance terms) as

$$I = k_d I_d (\mathbf{l} \cdot \mathbf{n}) + k_s I_s (\mathbf{v} \cdot \mathbf{r})^{\alpha} + k_a I_a$$

For each color component we add contributions from all sources





Modified Phong Model

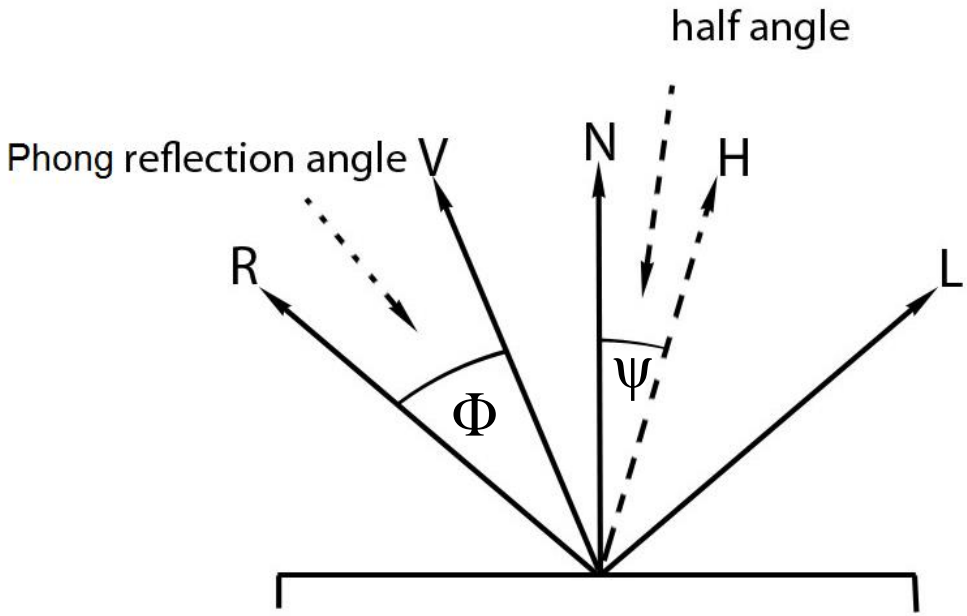
- The specular term in the Phong model is problematic because it requires the calculation of a new reflection vector **r** and view vector **v** for each vertex
- Blinn suggested an approximation using the halfway vector that is more efficient

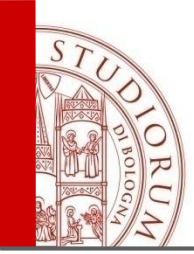
The Halfway Vector

- **h** is normalized vector halfway between **l** and **v**

$$\mathbf{h} = (\mathbf{l} + \mathbf{v}) / |\mathbf{l} + \mathbf{v}|$$

$$2\psi = \Phi$$





Using the halfway vector

- If we replace $(\mathbf{v} \cdot \mathbf{r})^{\alpha}$ by $(\mathbf{n} \cdot \mathbf{h})^{\beta}$ we avoid the calculation of r.
- β is chosen to match shiness
- Note that halfway angle is half of angle between $\mathbf{r}$ and $\mathbf{v}$ if vectors are coplanar
- Resulting model is known as the Modified Phong or Blinn lighting model
  - Default in OpenGL fixed function pipeline

Example 4: Toon Shading

- See:

  `NONPHOTO/nonphoto.c`

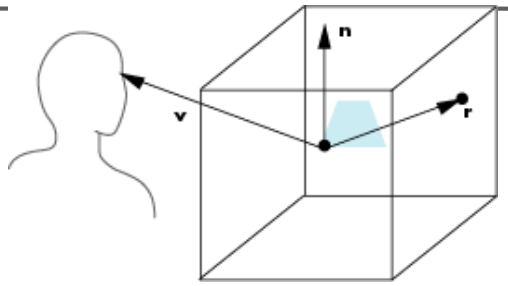
  `NONPHOTO/v.glsl`

  `NONPHOTO/f.glsl`

- Exercise 4:

a. Add an outline/silhouette effect

Example 5: Cube Maps



- Form a cube map texture by defining six 2D texture maps that correspond to the sides of a box
- Supported by OpenGL
- Also supported in GLSL through cubemap sampler

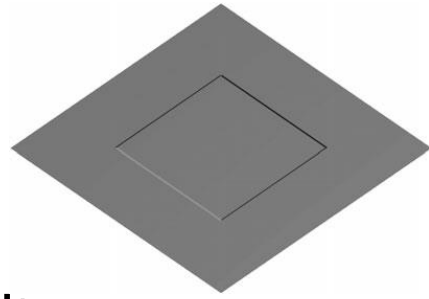
```
vec4 texColor = textureCube(mycube, texcoord);
```

- Use reflection vector to locate texture in cube map
- Exercise 5:

a. Use texture images instead of color for faces

Example 6: Bump Mapping

- Perturb normal for each fragment
- Single square in the plane $y = 0$ with a light source above the plane that rotates in the plane $y = 10.0$. The displacement is a small square in the center of the original square



Store normal map (perturbation) as texture

- The normal map is computing by taking the finite differences to approximate the partial derivatives for two of the components and using 1.0 for the third to form the array normals. Because these values are stored as colors in a texture image, the components are scaled to the interval (0.0, 1.0)