Computer Graphics: Modern OpenGL, GLSL and GPU programming

Table of contents

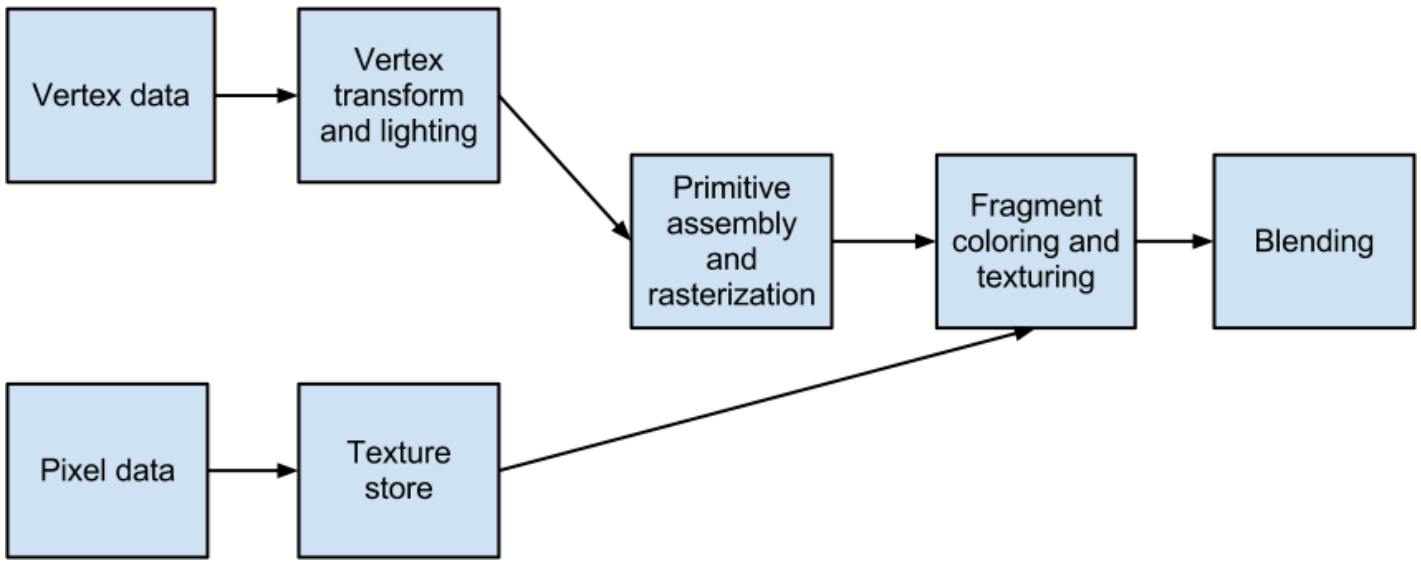
OpenGL

OpenGL is a rendering API:

- The first version is from 1992. OpenGL is derived from IrisGL, which was developped by SGI (Silicon Graphics International) for their graphics workstations,
- It allows for efficient and high-quality rendering of 3D graphics,
- It is a C library. But bindings have been developped for various languages: C++, fortran, Java, Javascript, python ...
- It is platform independent:
  - OpenGL libraries are available for: Windows, Linux, OSX, Unixes ...
  - OpenGL applications can be run inside a browser with WebGL,
  - OpenGL applications can be run on embedded devices with OpenGL ES.
- OpenGL forms the basic for many portable 3D graphics applications.

OpenGL 1.x

- OpenGL 1.0 specification released in 1992
- Fixed-function rendering pipeline



Fixed-function rendering pipeline of OpenGL 1.x

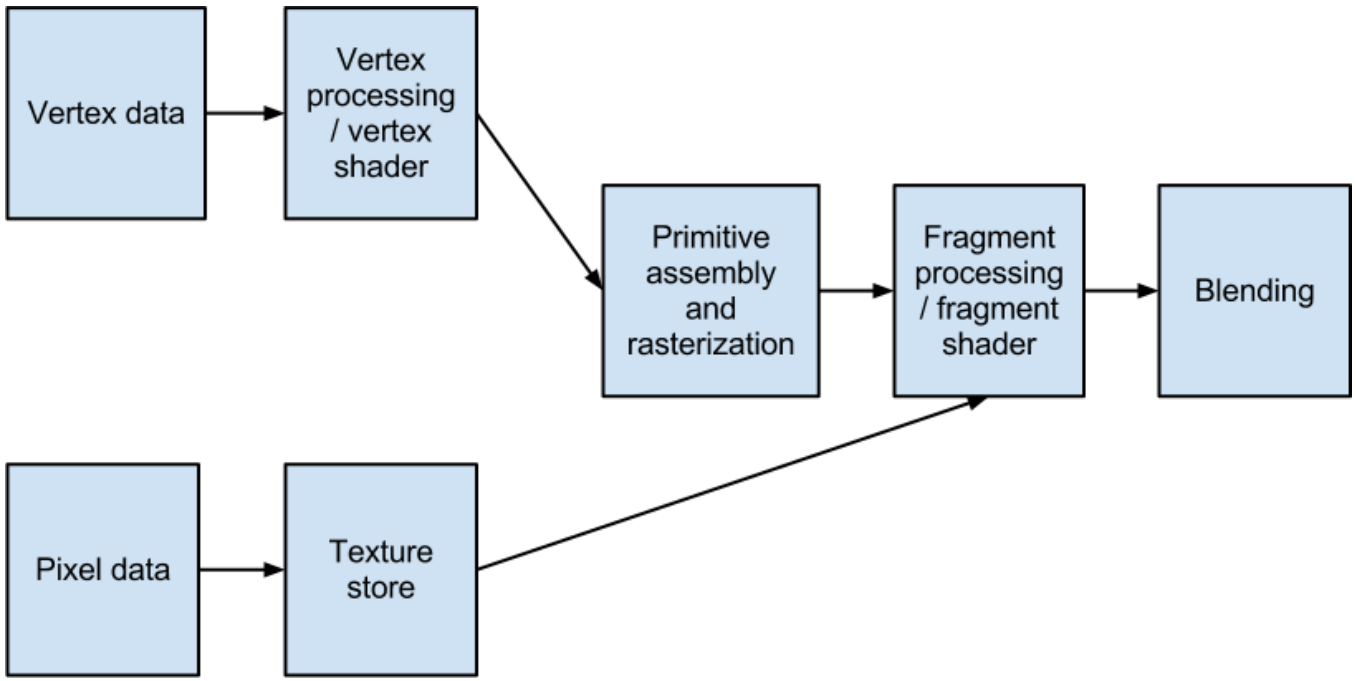
- First there are the per-vertex stages: transformation (geometry and projection), lighting (Blinn-Phong model), texture coordinates generation and transformations, clipping (intersection of the primitive with the clipping volume),
- The primitive is then rasterized to produce fragments,
- Per-fragment operations are then applied: fragments are textured and colored, depth-test, alpha-test, antialiasing, blending . . . are applied depending on the state.

OpenGL 2.x

- OpenGL 2.0 released in 2004
- Introduction of shaders to augment the fixed-function rendering pipeline:
  - Vertex shader: augment the fixed-function for the transformation and lighting stages
  - Fragment shader: augment the fixed-function for the fragment operations

Usage of shaders is optional and the fixed-function pipeline is still there. Shaders source code are written in the GL Shading Language, abbreviated in GLSL.

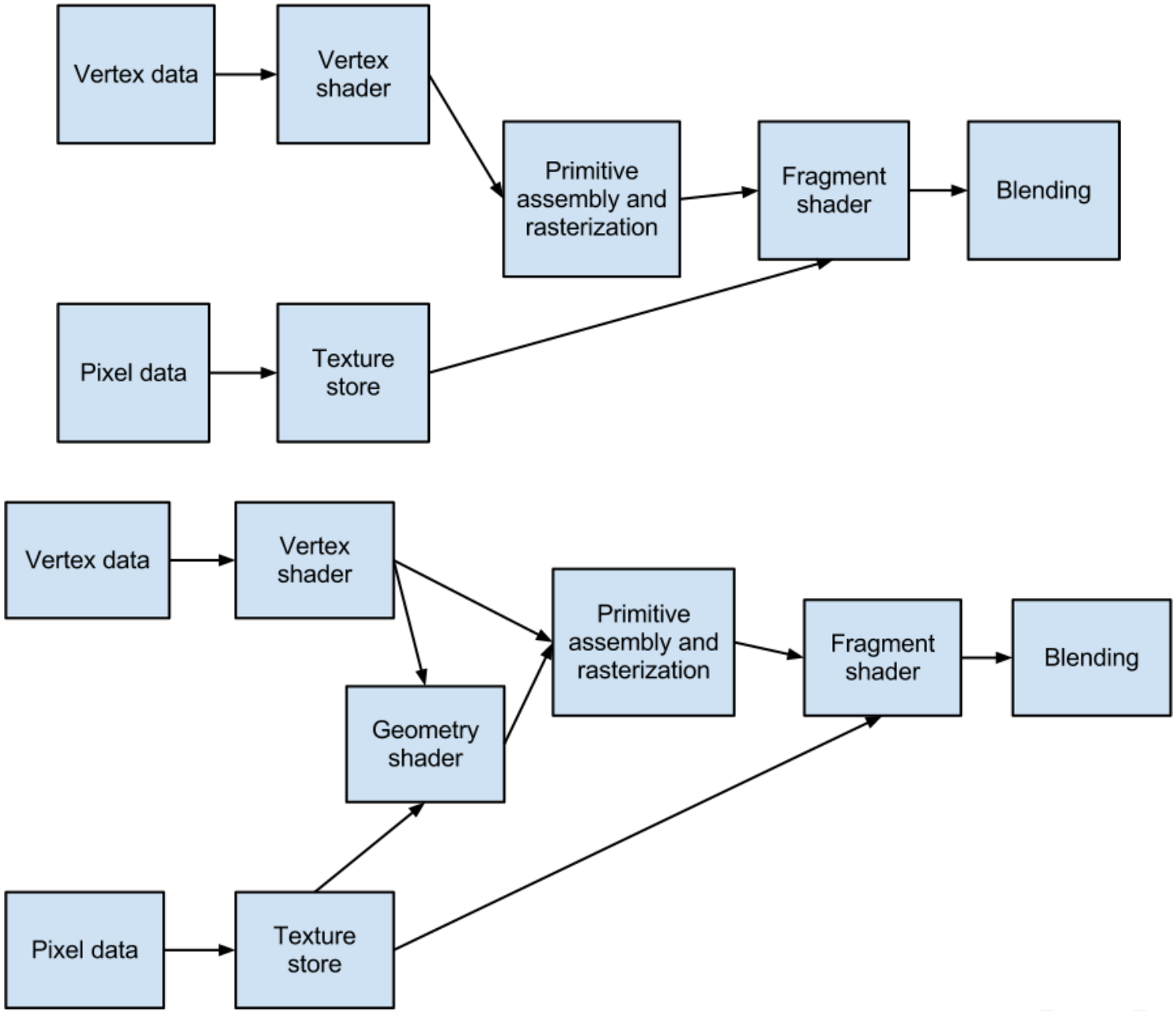
OpenGL 2.x


Vertex data
Vertex processing / vertex shader
Primitive assembly and rasterization
Fragment processing / fragment shader
Blending
Pixel data
Texture store

OpenGL 3.x

- OpenGL 3.0 released in 2008
- Introduction of deprecated features, i.e. features that will disappear in future versions of the API, such as:
  - Fixed-function vertex and fragment processing,
  - Direct mode rendering, i.e. glBegin, glEnd,
  - Display lists
  - Earlier version of GLSL (1.10 and 1.20).
- Deprecated features from 3.0 were removed in 3.1,
- OpenGL 3.2 introduced a new shader: the geometry shader. A geometry shader can be used to emit new primitives (points, lines, triangles) from those primitives that were sent from the beginning of the graphics pipeline.

Graphics pipeline of OpenGL 3.1 and 3.2 and above



OpenGL 4.x

- OpenGL 4.0 released in 2010,
- OpenGL 4.1 introduced a new shader: the tessellation shader.

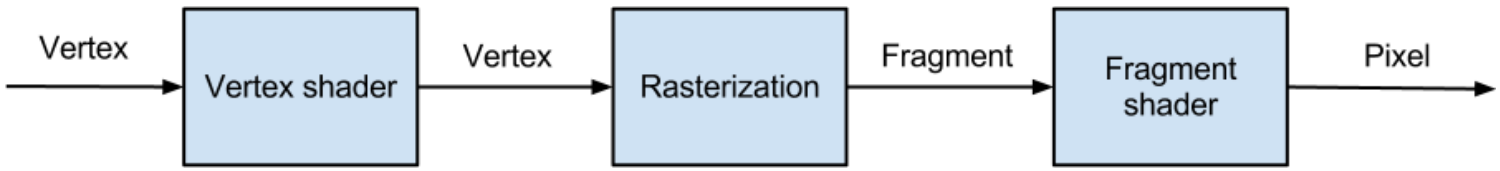
Tessellation shaders are useful for performing efficient subdivision of patches. OpenGL 4.1 adds two programmable stages:

- Tessellation control shader,
- Tessellation evaluation shader.

OpenGL ES

- OpenGL ES is a subset of OpenGL targetting embedded devices: cell-phones, portable game video systems, car navigation systems, ...
- OpenGL ES 2.0 corresponds to a subset of OpenGL 3.1 in terms of functionalities
- OpenGL ES 3.0: to a subset of OpenGL 4.1
- WebGL is a javascript bindings to OpenGL ES that allows to embed an OpenGL application in a web-page

A simplified view of the graphics rendering pipeline

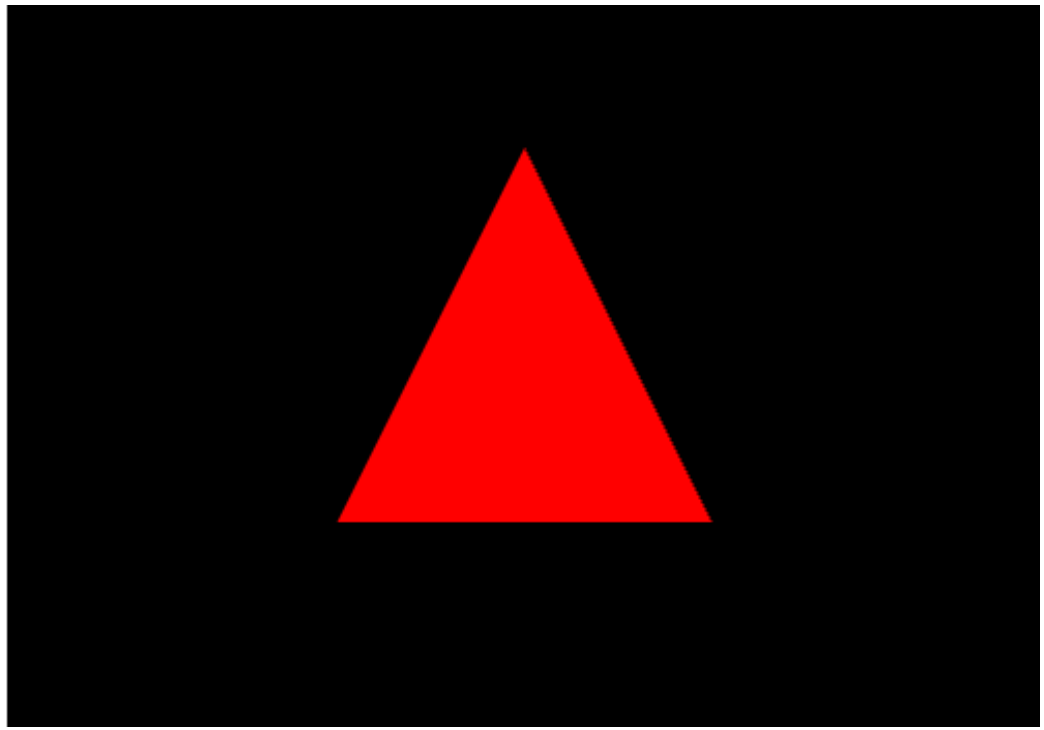


The main steps of a modern OpenGL application

- Create shader programs. Shader programs are made of shader objects (at least a vertex shader and a fragment shader);
- Create buffers and load data to them. The data to be loaded in buffers consist in vertex attributes (position, color, texture coordinates, ...);
- Relate shader variables to data locations, and to user-defined uniform variables;
- Draw.

Example: a triangle

Let us start with the shaders corresponding to a simple triangle.



Initialization of the attributes data

Some setup should be done to prepare the data to be rendered. This is done on the CPU by the main application (for example a C program).
Let us assume that a list of three vertices (each vertex is defined by its coordinates) was prepared and sent to OpenGL for rendering. This data (vertex position) is first sent to the vertex shader for processing.

Simple vertex shader

The code below illustrates the source code for a simple vertex shader:

```
// Input attributes of the shader
in vec4 vPosition;

// main function, i.e. entry point to an executable
// shader
void main() {
  // gl_Position is a built-in output variable
  gl_Position = vPosition;
}
```

All it does is set the position of each vertex in normalized device coordinates (NDC) to the coordinates it received from the CPU.

Simple fragment shader

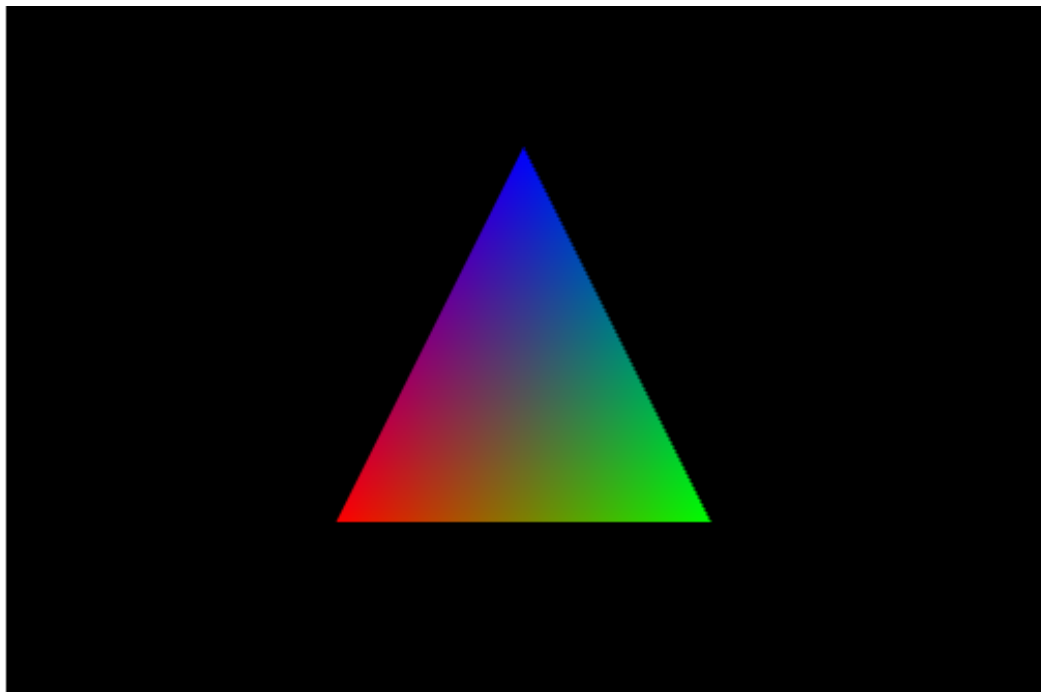
The code below illustrates the source code for a simple fragment shader:

```
1 out vec4 FragColor;
2
3 // main function, i.e. entry point to an executable
4 // shader
5 void main() {
6   // Set each fragment color to red
7   FragColor = vec4(1.0, 0.0, 0.0, 0.0);
8 }
```

This fragment shader sets the color of each processed fragment to red, and emits this fragment color.
Note: for GLSL 1.1, 1.2, use the builtin *gl_FragColor* instead (the declaration of FragColor is not needed).

Example: a colored triangle

This time we will also specify a color at each vertex and pass it to the vertex shader.



Colored triangle: vertex shader

```
1  // Input attributes of the shader
2  in vec4 vPosition;
3  in vec4 vColor;
4
5  // Interpolated value will be passed to the fragment
       shader
6  out vec4 Color;
7
8  // main function, i.e. entry point to an executable
9  // shader
10 void main() {
11   Color = vColor;
12   // gl_Position is a built-in output variable
13   gl_Position = vPosition;
14 }
```

This vertex shader does two things: set the position of each vertex in NDC, and pass the color at each vertex to the rasterizer.

Colored triangle: fragment shader

```
1 // Input from the rasterized
2 in vec4 Color;
3
4 // Emitted fragment color
5 out vec4 FragColor;
6
7 void main() {
8   FragColor = Color;
9 }
```

This fragment shader outputs as a fragment color, the interpolated color value it received from the rasterizer.

GLSL shader structure

A GLSL shader has the following structure:

```
1  // Input attributes of the shader
2  in vec4 vPosition;
3  in vec4 vColor;
4
5  // Output of the shader
6  out vec4 Color;
7
8  // User-defined uniform variables
9  uniform mat4 MVMatrix; // Model-View matrix
10
11 // The main function, the entry point to an executable
       shader
12 void main() {
13   // Shader code
14   // ...
15 }
```

Shader external variables

Shaders (non-local) variables can have different type of storage qualifier:

- *in*: linkage into a shader from a previous stage, the variable is copied in,
- *out*: linkage out of a shader to a subsequent stage, the variable is copied out,
- *uniform*: a value that does not change across the primitives being processed, e.g. a matrix transformation, light properties, . . . ; uniform forms the linkage between a shader, OpenGL and the application.

Note that in older versions of GLSL: 1.10, 1.20 (corresponding to OpenGL API 2.0 and 2.1), *attribute* is used for input variable in the vertex shader and *varying* is used for variables passed from the vertex shader to the fragment shader. The storage qualifiers *in* and *out* are not available.

Variables data types

Primitive data types:

- float, double, bool, int, uint

Vector data types:

- vec2, vec3, vec4: 2, 3 or 4 components single precision floating point vector
- bvec2, bvec3, bvec4: 2, 3 or 4 components bool vector
- ivec2, ivec3, ivec4: 2, 3 or 4 components integer vector
- uvec2, uvec3, uvec4: 2, 3, or 4 components unsigned int vector

Matrices:

- mat2: 2×2 single precision floating point matrix,
- mat3: 3×3 matrix,
- mat4: 4×4 matrix.

Variables declaration and initialization

Declaration and initialization of variables is similar to C++:

```
int a = 2;
bool b = true;
float c = 2.0;
float d = float(a); // cast int to float
vec3 e;
vec3 f = vec3(1.0,1.0,1.0); // constructor like syntax
mat3 g = mat3(1.0); // initialize matrix with 1.0 on
    the diagonal
// components are in column major order
mat2 h = mat2(1.0,0.0,1.0,0.0);
```

Matrix initialization

Matrices components are written in column major order. So the matrix:

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

corresponds to:

$$\begin{bmatrix} 1.0 & 3.0 \\ 2.0 & 4.0 \end{bmatrix}$$

Vector variables access

Vector and matrix variables can be accessed in different equivalent ways:

```
vec4 pos = vec4(1.0,2.0,3.0,1.0);
pos.x; // or pos.r: access the first element
pos.w; // or pos.a: access the last element
pos.xy; // or pos.rg: access the first two elements
pos.xyz; // or pos.rgb: access the first three elements

vec3 ex = pos.zyx; // ex contains (3.0,2.0,1.0)
vec4 dup = pos.xxyy; // dup contains (1.0,1.0,2.0,2.0)

mat4 m;
m[1] = vec4(1.0,1.0,1.0,1.0); // set the second column
    to all 1.0
m[0][1] = 1.0; // set the first column, second row
    element to 1.0
```

Structures

GLSL provides user-defined structures similar to C and C++.
Their initialization is similar to constructor calls in C++.

```
1 // Structure representing a ray
2 struct Ray {
3   vec3 origin;
4   vec3 direction;
5 };
6
7 // Initialization
8 Ray r = Ray(vec3(0.0,0.0,0.0), vec3(1.0,0.0,0.0));
```

Arrays

Arrays are defined like in C or C++ with the square bracket notation:

```
vec4 points[10];
```

creates an array of ten vec4 variables.
Arrays are 0-indexed.
There are no pointers and the only way to declare an array is with the square brackets notation.

Built-in functions

- Trigonometric functions: cos(), sin(), tan(), atan(), radians(), degrees()
- Power: pow(), exp(), log(), sqrt() ...
- Vector manipulation: dot(), cross(), length(), normalize() ...
- Matrix manipulation: transpose(), determinant(), inverse()
- Utility: min(), max(), floor(), abs() ...

User-defined functions

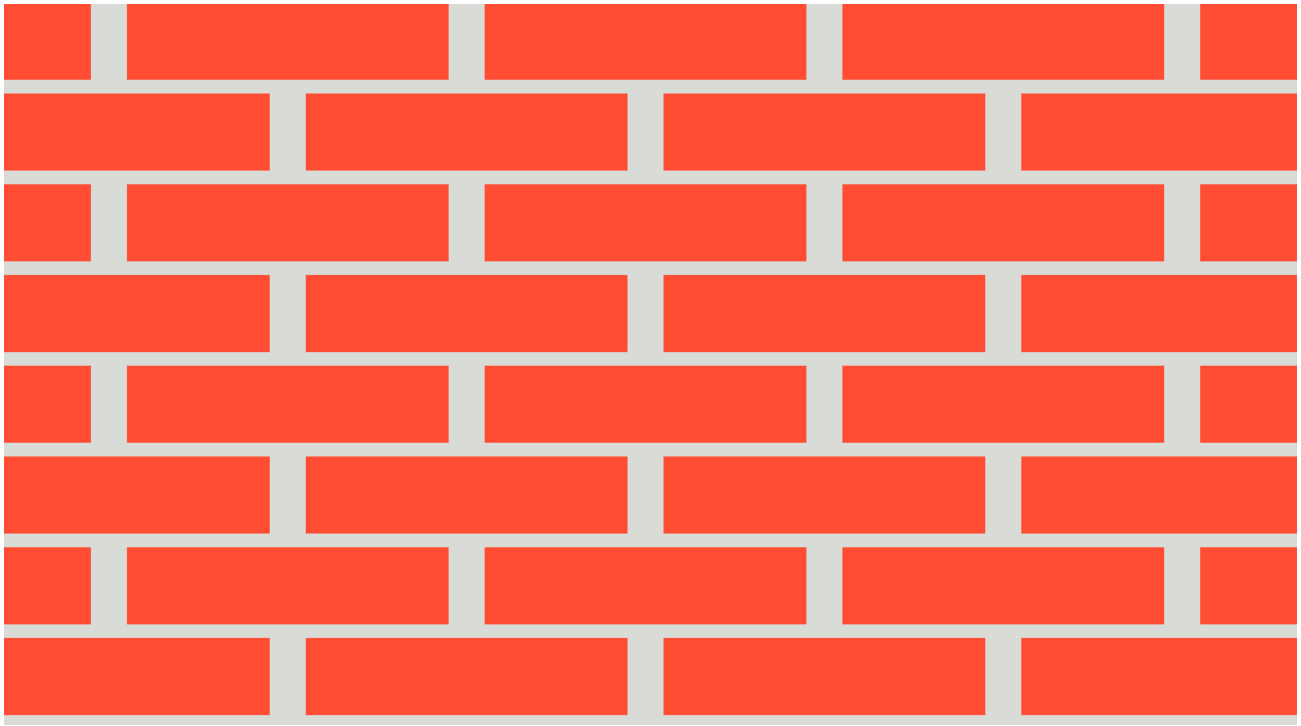
GLSL allows user-defined functions. They use the call by value mechanism. No pointers are allowed. The following qualifiers are available:

- in: copy in but don't copy out; variable is writable inside function
- out: only copy out; undefined at function entry
- inout: copy in and copy out
- const: function not allowed to write in the variable (does not apply to out and inout)

User-defined functions

```
1 void ComputeCoord(in vec3 normal,
2                    vec3 tangent, // same as with in
3                    inout vec3 coord);
4
5 vec3 ComputeCoord(const vec3 normal,
6                    vec3 tangent,
7                    in vec3 coord);
```

Brick shader



Brick shader: vertex shader

```
1 in vec4 vPosition;
2 out vec2 bPosition;
3
4 void main() {
5   bPosition = vPosition.xy;
6   gl_Position = vPosition;
7 }
```

Brick shader: fragment shader

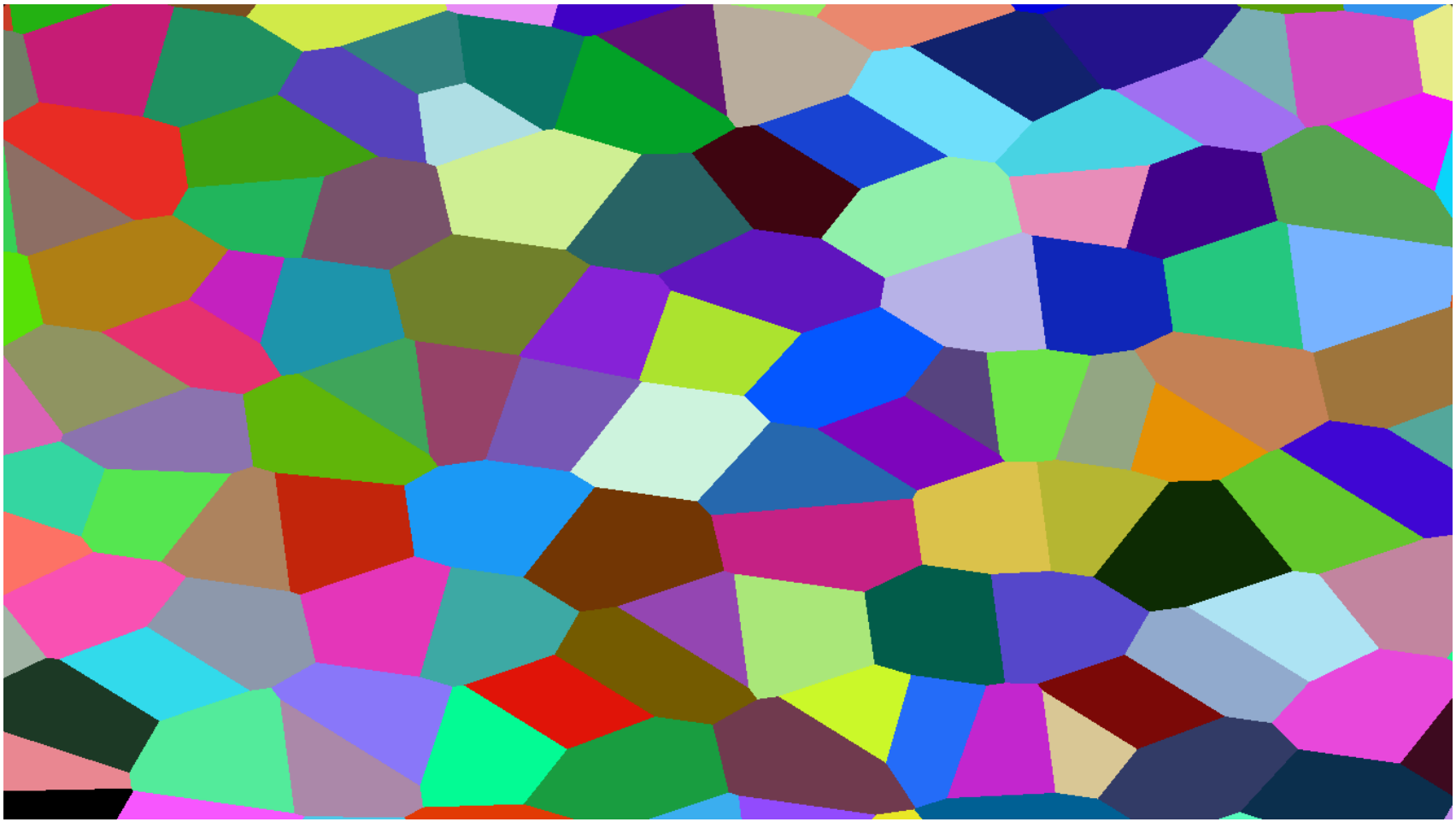
```
in vec2 bPosition;
out vec4 FragColor;

void main() {
  vec3 brickColor = vec3(1.0, 0.3, 0.2);
  vec3 mortarColor = vec3(0.85, 0.86, 0.84);
  vec2 brickSize = vec2(0.3, 0.15);
  vec2 brickPct = vec2(0.9, 0.85);

  vec3 color;
  vec2 position;
  vec2 useBrick;

  // ...
```

Brick shader: fragment shader

```
1   // ...
2   position = bPosition / brickSize;
3   // fract(x) returns x*floor(x)
4   if (fract(position.y * 0.5) > 0.5)
5     position.x += 0.5;
6
7   position = fract(position);
8   // step(edge,x) returns 0 if x<edge; otherwise 1
9   useBrick = step(position, brickPct);
10
11  color = mix(mortarColor, brickColor, useBrick.x*
      useBrick.y);
12  FragColor = vec4(color, 1.0);
13 }
```

Remark: we could have passed the parameters for the brick layout from the C program as uniform variables.

2D Voronoi diagram



2D Voronoi diagram - fragment shader

```
in vec2 fragCoord;
out vec4 fragColor;
void main()
{
    // Normalized pixel coordinates (from 0 to 1)
    vec2 uv = 0.5*(1.0+fragCoord);

    vec2 pos = uv * 100.0;

    // Time varying pixel color
    vec3 col = vor2d(pos);

    // Output to screen
    fragColor = vec4(col,1.0);
}
```

The vertex shader is a simple pass-through that passes the x, y coordinates of each vertex to the variable fragCoord.

2D Voronoi diagram - 1

```
vec3 vor2d(vec2 pos)
{
    float step = 10.0;

    int xi = int(floor(pos.x/step));
    int yj = int(floor(pos.y/step));

    ivec2 nearest;
    float min_dist = 1e5;

    for (int i = xi-1; i <= xi+1; ++i) {
        for (int j = yj-1; j <= yj+1; ++j) {
            vec2 disp = dispHash(i,j);
            vec2 seed = vec2(
                (float(i)+disp.x)*step,
                (float(j)+disp.y)*step);

      // ...
```

2D Voronoi diagram - 2

```
vec3 vor2d(vec2 pos)
{
    // ...

    for (int i = xi-1; i <= xi+1; ++i) {
        for (int j = yj-1; j <= yj+1; ++j) {
      // ...

            float dist = length(pos-seed);
            if (dist<min_dist) {
                min_dist = dist;
                nearest = ivec2(i, j);
            }
        }
    }
    // color of the cell
    vec3 col = colHash(nearest.x, nearest.y);
    return col;
}
```

The hash functions - 1

```
1  // map a pair of int (i,j) to a displacement vector in
       [0,1]x[0,1]
2  vec2 dispHash(int i, int j)
3  {
4      float x = float(i);
5      float y = float(j);
6
7      for (int i=0; i<2; ++i) {
8        x = mod(87.0*x + 23.0*y, 257.0);
9        y = mod(87.0*x + 23.0*y, 1009.0);
10     }
11
12     return vec2(x/257.0, y/1009.0);
13 }
```

The hash functions - 2

```
// map a pair of int (i,j) to a color in [0,1]x[0,1]x
    [0,1]
vec3 colHash(int i, int j)
{
    float r = float(i);
    float g = float(j);
    float b = float(i+j);

    for (int i=0; i<2; ++i) {
      r = mod(87.0*r + 23.0*g + 125.0*b, 257.0);
      g = mod(87.0*r + 23.0*g + 125.0*b, 1009.0);
      b = mod(87.0*r + 23.0*g + 125.0*b, 21001.0);
    }

    return vec3(r/257.0,g/1009.0,b/21001.0);
}
```

3D transforms

In this vertex shader, we are passing information about the model view matrix and the projection matrix from the application to the vertex shader through uniform variables:

```
1  in vec4 vPosition;
2  in vec4 vColor;
3  out vec4 Color;
4
5  uniform mat4 MVMatrix;
6  uniform mat4 PMatrix;
7
8  void main() {
9    gl_Position = PMatrix * MVMatrix * vPosition;
10   Color = vColor;
11 }
```

A simple fragment shader that outputs the color received from the rasterizer can be used.

Per-vertex lighting with the Blinn-Phong model

This vertex shader computes a per-vertex lighting assuming the Blinn-Phong illumination model. There is only one light, with white color. A simple fragment shader outputting the fragment color from the rasterizer can be used together with this vertex shader. It corresponds to Gouraud shading.

```
1  in vec4 vPosition;
2  in vec4 vNormal;
3
4  out vec4 Color;
5
6  uniform mat4 MVMatrix; // Model View
7  uniform mat4 NormalMatrix;
8  uniform mat4 PMatrix; // Projection
9  uniform vec4 lightPosition;
10 uniform vec4 materialColor;
```

Per-vertex lighting with the Blinn-Phong model

```
1  void main() {
2    // Apply the modelview and projection transforms to
       the vertex
3    gl_Position = PMatrix * MVMatrix * vPosition;
4
5    // Diffuse component:
6    // Transformed normal vector (at the vertex)
7    vec4 N = NormalMatrix * vNormal;
8    N.w = 0.0;
9    N = normalize(N);
10
11   // Vector from vertex to the light position in camera
       space
12   vec4 V = MVMatrix * vPosition;
13   vec4 L = normalize(lightPosition - V);
14
15   // ...
16 }
```

Per-vertex lighting with the Blinn-Phong model

```
1 void main() {
2   // ...
3   // N.L for the diffuse component:
4   float NdotL = max(0.0, dot(N, L));
5   vec4 diffuse = materialColor * NdotL;
6
7   // Specular Component:
8   // ...
9 }
```

Per-vertex lighting with the Blinn-Phong model

```
1  void main() {
2    // ...
3    // Reflected light vector:
4    vec4 R = reflect(-L,N);
5
6    // Unit vector in the viewer direction
7    vec4 View = normalize(-V);
8    // (View.R)^shiny for the specular component:
9    float spec = max(dot(View, R), 0.0);
10   float shiny = 64.0;
11   spec = pow(spec, shiny);
12   vec4 specular = vec4(1,1,1,0) * spec;
13
14   // Ignore ambient component
15   Color = diffuse + specular;
16 }
```

Phong shading

Obtained by applying the illumination model per fragment. The vertex shader becomes much simpler and only outputs the transformed normal and vertex (by the Model-View transform).

```
1 in vec4 vPosition;
2 in vec4 vNormal;
3 out vec4 oNormal; // normal after MV transformation
4 out vec4 oPosition; // vertex after MV transformation
5
6 uniform mat4 MVMatrix; // Model View
7 uniform mat4 NormalMatrix;
8 uniform mat4 PMatrix; // Projection
```

Phong shading: vertex shader

```
void main() {
  gl_Position = PMatrix * MVMatrix * vPosition;
  vec4 N = NormalMatrix * vNormal;
  N.w = 0.0;
  oNormal = normalize(N);
  oPosition = MVMatrix * vPosition;
}
```

Phong shading: fragment shader

The fragment shader for the Blinn-Phong illumination model and Phong shading is obtained by repeating the computations from the per-vertex lighting vertex shader but in the fragment shader using oPosition and oNormal (the transformed vertex position and normal).

```
1 in vec4 oNormal;
2 in vec4 oPosition;
3 out vec4 Color;
4
5 uniform vec4 lightPosition;
6 uniform vec4 materialColor;
```

Phong shading: fragment shader

```
1 void main() {
2   // Compute the diffuse and specular components using
3   // lightPosition, oPosition, oNormal and
      materialColor
4   // ...
5
6   // Ignore ambient color
7   Color = diffuse + specular;
8 }
```

Textured object: vertex shader

```
in vec4 vPosition;
in vec2 vUV;
out vec2 UV;

uniform mat4 MVPMatrix;

void main() {
  gl_Position = MVPMatrix * vPosition;
  UV = vUV;
}
```

Textured object: fragment shader

```
in vec2 UV;
out vec4 FragColor;

// Texture sampler
uniform sampler2D texSampler;

void main() {
  // Use the texel color for the output color
  FragColor = texture(texSampler, UV);
}
```

Note: this corresponds to the model GL_REPLACE in legacy OpenGL.

Initialization of the attributes data

In order to use these shaders, some setup should be done to prepare the data to be rendered. This is done on the CPU by the main application (for example a C program). Let us start from a simple example: a triangle.

- A triangle is made of three vertices;
- Each vertex has one attribute: its position;
- The triangle will be modelled in two-dimension. So each vertex will have two coordinates only (x and y).

The data (vertex position) is then sent to the vertex shader for processing.

Initialization of the triangle data

```
GLfloat vertex_attributes[] = {
  // vert 1 position
 -0.5f, -0.5f,
 // and color (yellow)
 1, 1, 0,
 // vert 2 position
 +0.0f, +0.5f,
 // and color (magenta)
 1, 0, 1,
 // vert 3 position
 +0.5f, -0.5f,
 // and color (cyan)
 0, 1, 1
};
```

Vertex Array Objects (VAO)

Instead of sending data to the graphics card with several function calls (e.g. in direct mode), we will pack the data and send it once. Vertex Array Objects (VAO) and Vertex Buffer Objects (VBO) are used for this purpose. A VAO is an OpenGL object that stores all of the state needed to supply vertex data (the format of the vertex data and the buffer objects).
Steps for using a VAO:

- Create VAO names with glGenVertexArrays(),
- Bind a specific VAO with glBindVertexArray() for initialization,
- Bind and update a VBO associated with this VAO. VBO (Vertex Buffer Object) are buffers used to hold data,
- Bind a specific VAO for use in rendering.

Creation and initialization of a VAO

```
1 GLuint vao;
2 // Generate one VAO name
3 glGenVertexArrays(1, &vao);
4 // Bind this VAO for manipulation
5 glBindVertexArray(vao);
```

Storing vertex attributes

Vertex attributes (position, color, texture coordinates) need to be stored in a VBO associated with a VAO. The steps for using a VBO are:

- Create VBO names with glGenBuffers(),
- Bind a specific VBO for initialization with: glBindBuffer(GL_ARRAY_BUFFER, ...),
- Load data in the buffer object: glBufferData(GL_ARRAY_BUFFER, ...),
- Bind the associated VAO for rendering.

Creation and initialization of a VBO

```
1 GLuint vbo;
2 glGenBuffers(1, &vbo);
3 glBindBuffer(GL_ARRAY_BUFFER, vbo);
4 glBufferData(GL_ARRAY_BUFFER, sizeof(vertex_attributes)
      ,
5   vertex_attributes, GL_STATIC_DRAW);
```

Connecting data to a shader input

Attributes data need to be sent to the shader in order to be used. Each attribute has its own index within the shader that can be set by:

- the OpenGL application before linking the shader program with glBindAttribLocation(program handle, attribute index, attribute name in the shader)
- otherwise it is assigned automatically by the shader program linker and can be obtained with: glGetAttribLocation(program handle, attribute name)

Connecting data to a shader input - Method 1

```
1 enum {POSITION_LOCATION, COLOR_LOCATION};
2 //  ...
3
4 // When building the shader program, before linking
5 glBindAttribLocation(program_handle, POSITION_LOCATION,
      "vPosition");
6 glBindAttribLocation(program_handle, COLOR_LOCATION, "
     vColor");
7
8 //  ...
```

Connecting data to a shader input - Method 1 (continued)

```
1 // ...
2 // When building the geometry
3 glEnableVertexAttribArray(POSITION_LOCATION);
4 glVertexAttribPointer(POSITION_LOCATION, 2, GL_FLOAT,
      GL_FALSE, stride, 0);
5
6 glEnableVertexAttribArray(COLOR_LOCATION);
7 glVertexAttribPointer(COLOR_LOCATION, 3, GL_FLOAT,
      GL_FALSE, stride, color_offset);
```

Connecting data to a shader input - Method 2

```
1  GLuint position_location;
2  GLuint color_location;
3
4  // Assume that locations were automatically created by
       the linker
5  position_location = glGetAttribLocation(program_handle,
       "vPosition");
6  glEnableVertexAttribArray(position_location);
7  glVertexAttribPointer(position_location, 2, GL_FLOAT,
       GL_FALSE, stride, 0);
8
9  color_location = glGetAttribLocation(program_handle, "
       vColor");
10 glEnableVertexAttribArray(color_location);
11 glVertexAttribPointer(color_location, 3, GL_FLOAT,
       GL_FALSE, stride, color_offset);
```

Drawing

In order to render, bind a VAO and call glDrawArrays() or glDrawElements() to draw the geometry. Example:

```
1 glBindVertexArray(vao);
2 glDrawArrays(GL_TRIANGLES, 0, 3);
```

Main steps for working with shaders

- Compile a shader
  - Create a shader object
  - Provide source code (as a string) of the shader
  - Compile the code and verify the compilation status
- Link a shader
  - Create a program object
  - Attach the shader objects to the program
  - Link the program and verify the link status

Create a shader object

```
1 GLuint vertex_shader = glCreateShader(GL_VERTEX_SHADER)
      ;
2 if (vertex_shader == 0) {
3   // Error in creating a vertex shader object
4   exit(1);
5 }
```

To create a shader object, call glCreateShader. The argument can be GL_VERTEX_SHADER or GL_FRAGMENT_SHADER depending on whether a vertex or fragment shader is to be created. A shader object is used to maintain the source code that defines a shader.

Shaders source

Simple vertex shader:

```
in vec4 vPosition;
in vec4 vColor;
out vec4 Color
void main() {
  Color = vColor;
  gl_Position = vPosition;
}
```

Simple fragment shader:

```
in vec4 Color;
out vec4 FragColor;

void main() {
  FragColor = Color;
}
```

Provide a shader source code to a shader object

```
1 // User defined function to load shader from file
2 GLchar* shader_code = GetShaderSource("simple_vs.glsl")
      ;
3 glShaderSource(vertex_shader, 1, &shader_source, 0);
```

Associating a shader source code to a shader object is done by calling glShaderSource(handle, number, array, 0) where:

- handle: an handle to a shader object (obtained with glCreateShader),
- number: number of source code strings to be passed as third argument
- array: array of strings corresponding to the source codes to be compiled
- the fourth argument is an array of GLint that correspond to the length of each source code. When NULL (or 0) is passed, it indicates that each source code is terminated by a NULL character.

Compile a shader and verify the compilation status

```
1 // Compile the source for the shader object
2 glCompileShader(vertex_shader);
3 GLint result;
4 // Get the compile status
5 glGetShaderiv(vertex_shader, GL_COMPILE_STATUS, &result
      );
6 if (result == GL_FALSE) {
7   // An error occur; get its description
8   // ...
9 }
```

Compile a shader and verify the compilation status

```
1  // ...
2  if (result == GL_FALSE) {
3    // An error occur; get its description
4    GLint log_length;
5    glGetShaderiv(vertex_shader, GL_INFO_LOG_LENGTH, &
       log_length);
6    if (log_length > 0) {
7      // Retrieve the error message
8      GLchar* log = (GLchar*)malloc(log_length * sizeof(
       GLchar));
9      GLsizei size;
10     glGetShaderInfoLog(vertex_shader, log_length, &size
       , log);
11     fprintf(stderr, "%s\n", log);
12     free(log);
13   }
14   exit(1);
15 }
```

Linking

Compiled shaders have to be linked together to form a shader program. At least vertex shader and fragment shader need to be compiled and linked. The steps for linking a shader program are:

- Create a program object
- Attach the shaders to the program object
- Link the program and verify the link status

Create a program object

A program object holds code for all the shaders to be used for rendering (at least vertex and fragment shaders).

```
1 GLuint program;
2 program = glCreateProgram();
3 if (program == 0) {
4   // An error occurred
5   exit(1);
6 }
```

Attach shader objects

Attach shader objects to a program with glAttachShader():

```
1 glAttachShader(program, vertex_shader);
2 glAttachShader(program, fragment_shader);
```

Link a program and check the linking status

Call glLinkProgram() to link a program, i.e. create an executable that will run on the vertex processor (if a vertex shader object was attached to the program) and on the fragment processor (if a fragment shader object was attached to the program).

```
1 glLinkProgram(program);
```

Checking the linking status is identical to checking the compilation status with glGetProgramiv() instead of glGetShaderiv() and glGetProgramInfoLog() instead of glGetShaderInfoLog().

Install the program in the rendering pipeline

A program is installed in the rendering pipeline by calling glUseProgram():

```
1 glUseProgram(program);
```

where program is the handle to the program object that will be used for rendering.
It is possible to compile and link several programs and switch between them for rendering by calling glUseProgram.

Connection to application data

- A vertex shader is processing vertex attributes. This data is changing per vertex;
- Attributes are declared in GLSL by using the storage qualifier *in*;
- The application provides data for these attributes with the help of vertex buffer objects and has to make connection between the buffer and the input attributes;
- When rendering, OpenGL reads the input attribute from the buffer for each invocation of the vertex shader.

Uniform variables

Uniform variables are shader variables qualified with the keyword uniform. Uniform variables correspond to:

- Read-only variables within the shaders used as input variables,
- Variables used for data that changes infrequently compared to per-vertex attributes, e.g. projection matrix or modelview matrix,
- Variables whose value is changed by OpenGL

Uniform variables

Example of a vertex shader source code with a uniform variable:

```
1  in vec4 vPosition;
2  in vec4 vColor;
3  out vec4 Color;
4
5  uniform mat4 MVMatrix;
6
7  void main() {
8    // Pass the vertex color to the fragment shader
9    Color = vColor;
10
11   // Compute the position after model-view
       transformation
12   gl_Position = MVMatrix * vPosition;
13 }
```

Connecting data to uniform variables

Example of code that sends the data corresponding to a rotation matrix from the application to the shader:

```
1 // ...
2 GLfloat rotation_vector[] = {1.0,0.0,0.0,0.0};
3 GLfloat angle = 15; // in degrees
4 // Form the rotation matrix about the axis
      rotation_vector
5 // Assume the matrix to be in row major mode
6 GLfloat rotation_matrix[4][4];
7 FormRotationMatrix(rotation_vector, angle,
      rotation_matrix);
8 // ...
```

Connecting data to uniform variables (continued)

```
1  // ...
2  // Get the location of the uniform variable "MVMatrix"
3  GLuint location = glGetUniformLocation(program, "
       MVMatrix");
4
5  // Send the rotation matrix to the uniform variable
6  // GL_TRUE indicates that rotation_matrix is "
       transposed"
7  if (location >= 0) {
8    glUniformMatrix4fv(location, 1, GL_TRUE, &
       rotation_matrix[0][0]);
9  }
10
11 // Draw
12 glBindVertexArray(vao);
13 glDrawArrays(GL_TRIANGLES, 0, 3);
14
15 // ...
```