Vertex and fragment/pixel shaders

Part 1

– a CAGD approach based on OpenGL and C++ –

Ágoston Róth

Department of Mathematics and Computer Science, Babeş-Bolyai University, Cluj-Napoca, Romania

(agoston.roth@gmail.com)

Lecture 7 – April 11, 2022

A very simplified diagram of the fixed pipeline stages and the data travels among them

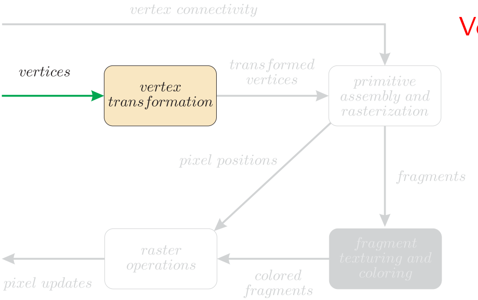

Fig. 1: Simplified overview of the fixed pipeline.

Vertex transformations

- In this stage a vertex is a set of attributes such as its position in space, as well as its color, normal vector, texture coordinates, etc.
- Inputs for this stage are the individual vertex attributes.
- Some of the operations performed by the fixed functionality at this stage are:
  - vertex position transformation;
  - lighting computations per vertex;
  - generation and transformation of texture coordinates.

A very simplified diagram of the fixed pipeline stages and the data travels among them

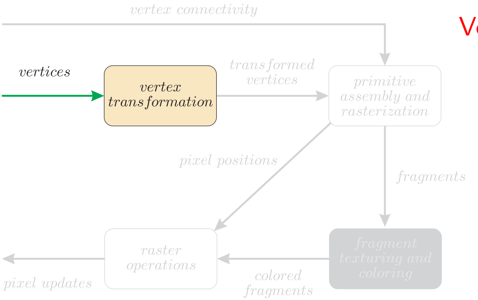

Fig. 1: Simplified overview of the fixed pipeline.

Vertex transformations

- In this stage a vertex is a set of attributes such as its position in space, as well as its color, normal vector, texture coordinates, etc.
- Inputs for this stage are the individual vertex attributes.
- Some of the operations performed by the fixed functionality at this stage are:
  - vertex position transformation;
  - lighting computations per vertex;
  - generation and transformation of texture coordinates.

A very simplified diagram of the fixed pipeline stages and the data travels among them

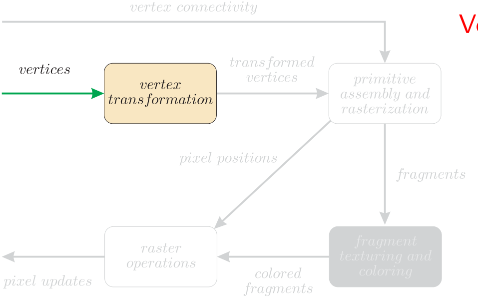

Fig. 1: Simplified overview of the fixed pipeline.

Vertex transformations

- In this stage a vertex is a set of attributes such as its position in space, as well as its color, normal vector, texture coordinates, etc.
- Inputs for this stage are the individual vertex attributes.
- Some of the operations performed by the fixed functionality at this stage are:
  - vertex position transformation;
  - lighting computations per vertex;
  - generation and transformation of texture coordinates.

A very simplified diagram of the fixed pipeline stages and the data travels among them

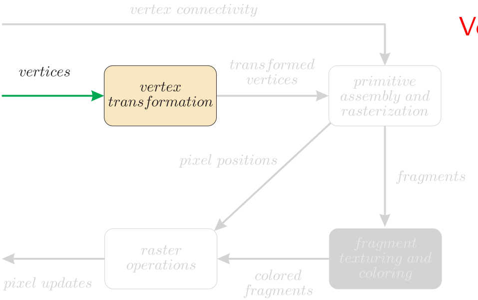

Fig. 1: Simplified overview of the fixed pipeline.

Vertex transformations

- In this stage a vertex is a set of attributes such as its position in space, as well as its color, normal vector, texture coordinates, etc.
- Inputs for this stage are the individual vertex attributes.
- Some of the operations performed by the fixed functionality at this stage are:
  - vertex position transformation;
  - lighting computations per vertex;
  - generation and transformation of texture coordinates.

A very simplified diagram of the fixed pipeline stages and the data travels among them

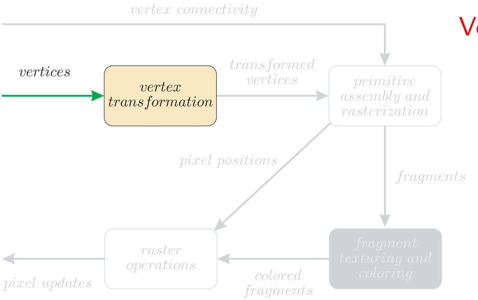

Fig. 1: Simplified overview of the fixed pipeline.

Vertex transformations

- In this stage a vertex is a set of attributes such as its position in space, as well as its color, normal vector, texture coordinates, etc.
- Inputs for this stage are the individual vertex attributes.
- Some of the operations performed by the fixed functionality at this stage are:
  - vertex position transformation;
  - lighting computations per vertex;
  - generation and transformation of texture coordinates.

A very simplified diagram of the fixed pipeline stages and the data travels among them

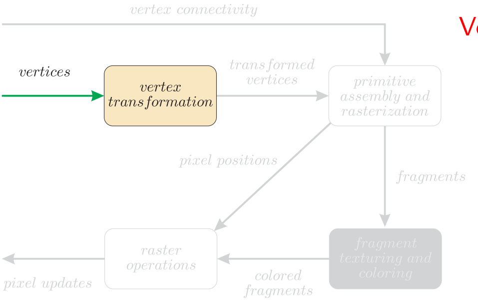

Fig. 1: Simplified overview of the fixed pipeline.

Vertex transformations

- In this stage a vertex is a set of attributes such as its position in space, as well as its color, normal vector, texture coordinates, etc.
- Inputs for this stage are the individual vertex attributes.
- Some of the operations performed by the fixed functionality at this stage are:
  - vertex position transformation;
  - lighting computations per vertex;
  - generation and transformation of texture coordinates.

A very simplified diagram of the fixed pipeline stages and the data travels among them

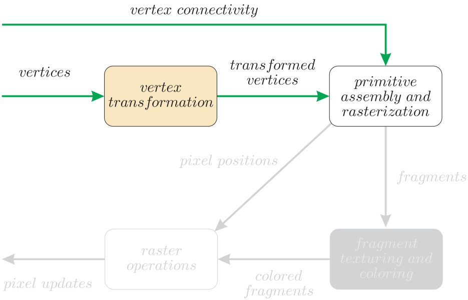

Fig. 1: Simplified overview of the fixed pipeline.

Primitive assembly and rasterization

- Input: transformed vertices and connectivity information.
- This stage is also responsible for clipping operations against the view frustum, and back face culling.
- Rasterization determines the fragments, and pixel coordinates of the primitive.
- A fragment is a piece of data that will be used to update a pixel in the frame buffer at a specific location.
- The output of this stage is twofold:
  - the position of the fragments in the frame buffer;
  - the interpolated values for each fragment of the attributes computed in the vertex transformation stage.

A very simplified diagram of the fixed pipeline stages and the data travels among them

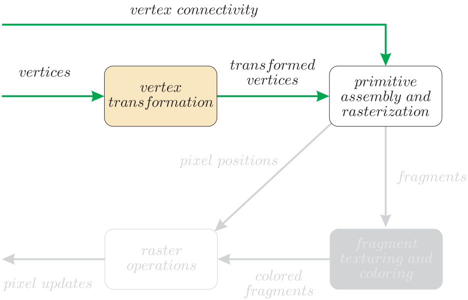

Fig. 1: Simplified overview of the fixed pipeline.

Primitive assembly and rasterization

- Input: transformed vertices and connectivity information.
- This stage is also responsible for clipping operations against the view frustum, and back face culling.
- Rasterization determines the fragments, and pixel coordinates of the primitive.
- A fragment is a piece of data that will be used to update a pixel in the frame buffer at a specific location.
- The output of this stage is twofold:
  - the position of the fragments in the frame buffer;
  - the interpolated values for each fragment of the attributes computed in the vertex transformation stage.

A very simplified diagram of the fixed pipeline stages and the data travels among them

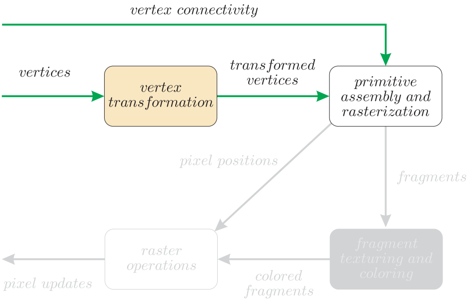

Fig. 1: Simplified overview of the fixed pipeline.

Primitive assembly and rasterization

- Input: transformed vertices and connectivity information.
- This stage is also responsible for clipping operations against the view frustum, and back face culling.
- Rasterization determines the fragments, and pixel coordinates of the primitive.
- A fragment is a piece of data that will be used to update a pixel in the frame buffer at a specific location.
- The output of this stage is twofold:
  - the position of the fragments in the frame buffer;
  - the interpolated values for each fragment of the attributes computed in the vertex transformation stage.

A very simplified diagram of the fixed pipeline stages and the data travels among them

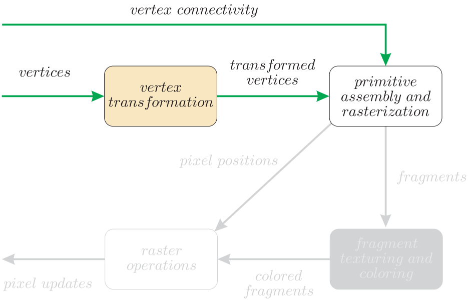

Fig. 1: Simplified overview of the fixed pipeline.

Primitive assembly and rasterization

- Input: transformed vertices and connectivity information.
- This stage is also responsible for clipping operations against the view frustum, and back face culling.
- Rasterization determines the fragments, and pixel coordinates of the primitive.
- A fragment is a piece of data that will be used to update a pixel in the frame buffer at a specific location.
- The output of this stage is twofold:
  - the position of the fragments in the frame buffer;
  - the interpolated values for each fragment of the attributes computed in the vertex transformation stage.

A very simplified diagram of the fixed pipeline stages and the data travels among them

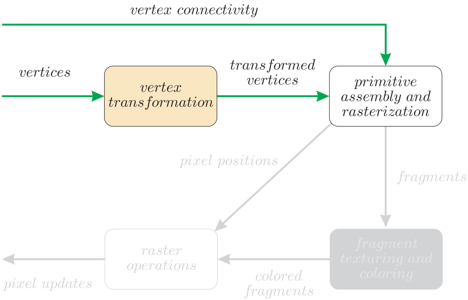

Fig. 1: Simplified overview of the fixed pipeline.

Primitive assembly and rasterization

- Input: transformed vertices and connectivity information.
- This stage is also responsible for clipping operations against the view frustum, and back face culling.
- Rasterization determines the fragments, and pixel coordinates of the primitive.
- A fragment is a piece of data that will be used to update a pixel in the frame buffer at a specific location.
- The output of this stage is twofold:
  - the position of the fragments in the frame buffer;
  - the interpolated values for each fragment of the attributes computed in the vertex transformation stage.

A very simplified diagram of the fixed pipeline stages and the data travels among them

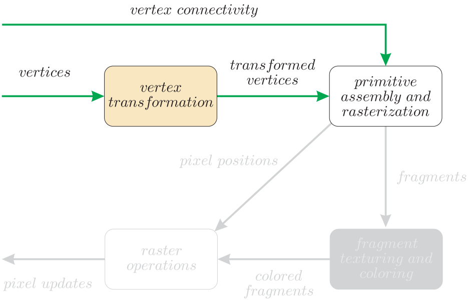

Fig. 1: Simplified overview of the fixed pipeline.

Primitive assembly and rasterization

- Input: transformed vertices and connectivity information.
- This stage is also responsible for clipping operations against the view frustum, and back face culling.
- Rasterization determines the fragments, and pixel coordinates of the primitive.
- A fragment is a piece of data that will be used to update a pixel in the frame buffer at a specific location.
- The output of this stage is twofold:
  - the position of the fragments in the frame buffer;
  - the interpolated values for each fragment of the attributes computed in the vertex transformation stage.

A very simplified diagram of the fixed pipeline stages and the data travels among them

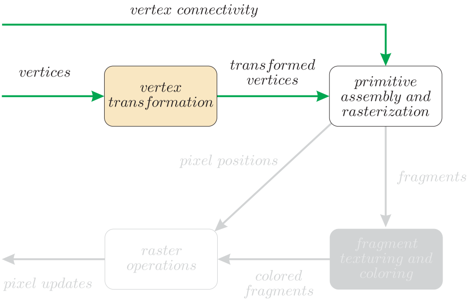

Fig. 1: Simplified overview of the fixed pipeline.

Primitive assembly and rasterization

- Input: transformed vertices and connectivity information.
- This stage is also responsible for clipping operations against the view frustum, and back face culling.
- Rasterization determines the fragments, and pixel coordinates of the primitive.
- A fragment is a piece of data that will be used to update a pixel in the frame buffer at a specific location.
- The output of this stage is twofold:
  - the position of the fragments in the frame buffer;
  - the interpolated values for each fragment of the attributes computed in the vertex transformation stage.

A very simplified diagram of the fixed pipeline stages and the data travels among them

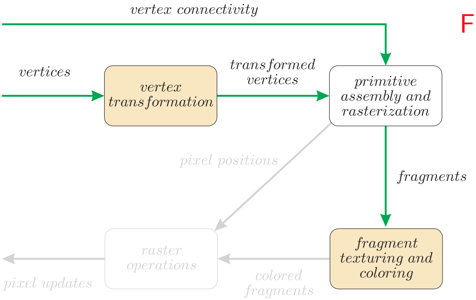

Fig. 1: Simplified overview of the fixed pipeline.

Fragment texturing and coloring

- Interpolated fragment information is the input of this stage.
- A color has already been computed in the previous stage through interpolation, and in here it can be combined with a texel (texture element) for example.
- Texture coordinates have also been interpolated in the previous stage.
- Fog is also applied at this stage.
- The output per fragment of this stage is a color value and a depth for the fragment.

A very simplified diagram of the fixed pipeline stages and the data travels among them

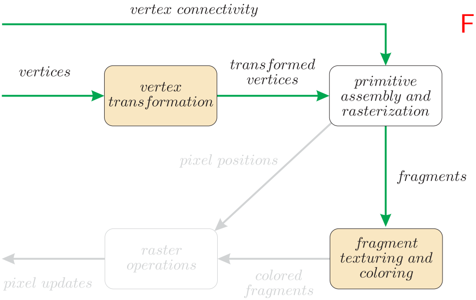

Fig. 1: Simplified overview of the fixed pipeline.

Fragment texturing and coloring

- Interpolated fragment information is the input of this stage.
- A color has already been computed in the previous stage through interpolation, and in here it can be combined with a texel (texture element) for example.
- Texture coordinates have also been interpolated in the previous stage.
- Fog is also applied at this stage.
- The output per fragment of this stage is a color value and a depth for the fragment.

A very simplified diagram of the fixed pipeline stages and the data travels among them

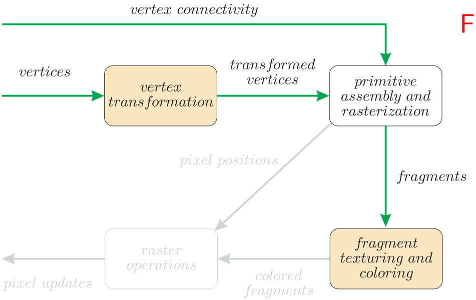

Fig. 1: Simplified overview of the fixed pipeline.

Fragment texturing and coloring

- Interpolated fragment information is the input of this stage.
- A color has already been computed in the previous stage through interpolation, and in here it can be combined with a texel (texture element) for example.
- Texture coordinates have also been interpolated in the previous stage.
- Fog is also applied at this stage.
- The output per fragment of this stage is a color value and a depth for the fragment.

A very simplified diagram of the fixed pipeline stages and the data travels among them

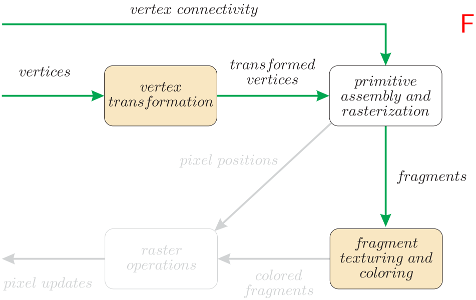

Fig. 1: Simplified overview of the fixed pipeline.

Fragment texturing and coloring

- Interpolated fragment information is the input of this stage.
- A color has already been computed in the previous stage through interpolation, and in here it can be combined with a texel (texture element) for example.
- Texture coordinates have also been interpolated in the previous stage.
- Fog is also applied at this stage.
- The output per fragment of this stage is a color value and a depth for the fragment.

A very simplified diagram of the fixed pipeline stages and the data travels among them

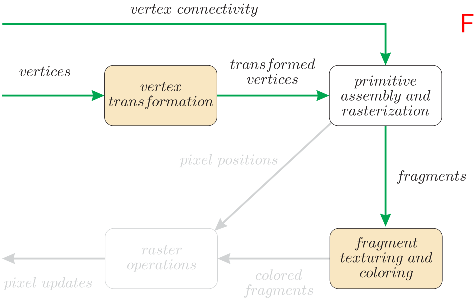

Fig. 1: Simplified overview of the fixed pipeline.

Fragment texturing and coloring

- Interpolated fragment information is the input of this stage.
- A color has already been computed in the previous stage through interpolation, and in here it can be combined with a texel (texture element) for example.
- Texture coordinates have also been interpolated in the previous stage.
- Fog is also applied at this stage.
- The output per fragment of this stage is a color value and a depth for the fragment.

A very simplified diagram of the fixed pipeline stages and the data travels among them

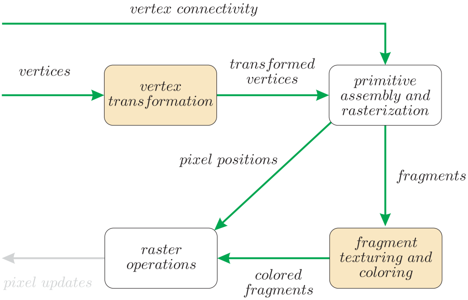

Fig. 1: Simplified overview of the fixed pipeline.

Raster operations

- The inputs of this stage are:
  - the location of pixels;
  - depth and color values of fragments.
- This last stage of the pipeline performs a series of tests on the fragment, namely:
  - scissor test;
  - alpha test;
  - stencil test;
  - depth test.
- A successful fragment information is used to update the value of the pixel according to the current blending mode. (Notice that blending occurs only at this stage because the fragment texturing and coloring stage has no access to the frame buffer. The frame buffer is only accessible at this stage.)

A very simplified diagram of the fixed pipeline stages and the data travels among them

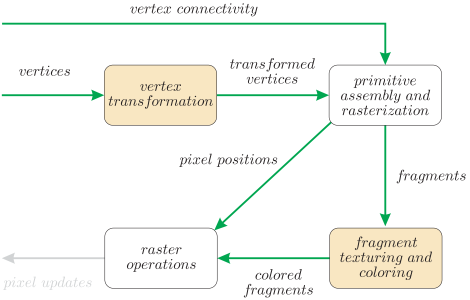

Fig. 1: Simplified overview of the fixed pipeline.

Raster operations

- The inputs of this stage are:
  - the location of pixels;
  - depth and color values of fragments.
- This last stage of the pipeline performs a series of tests on the fragment, namely:
  - scissor test;
  - alpha test;
  - stencil test;
  - depth test.
- A successful fragment information is used to update the value of the pixel according to the current blending mode. (Notice that blending occurs only at this stage because the fragment texturing and coloring stage has no access to the frame buffer. The frame buffer is only accessible at this stage.)

A very simplified diagram of the fixed pipeline stages and the data travels among them

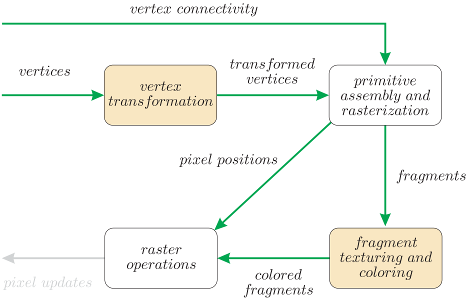

Fig. 1: Simplified overview of the fixed pipeline.

Raster operations

- The inputs of this stage are:
  - the location of pixels;
  - depth and color values of fragments.
- This last stage of the pipeline performs a series of tests on the fragment, namely:
  - scissor test;
  - alpha test;
  - stencil test;
  - depth test.
- A successful fragment information is used to update the value of the pixel according to the current blending mode. (Notice that blending occurs only at this stage because the fragment texturing and coloring stage has no access to the frame buffer. The frame buffer is only accessible at this stage.)

A very simplified diagram of the fixed pipeline stages and the data travels among them

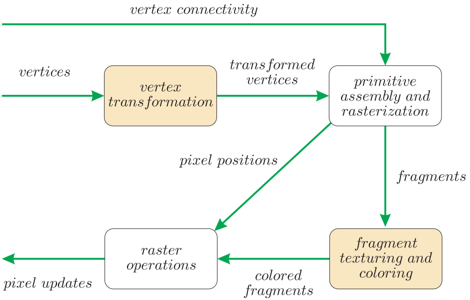

Fig. 1: Simplified overview of the fixed pipeline.

Visual summary

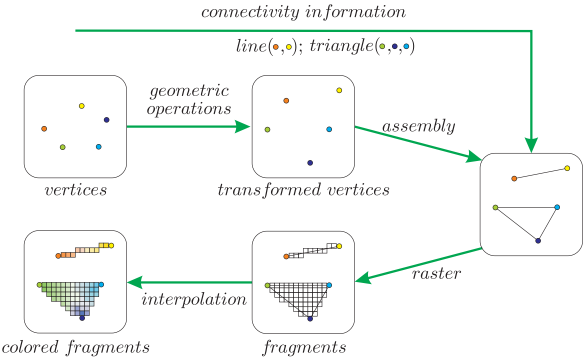

Fig. 2: Visual summary of the fixed pipeline.

Vertex and fragment shaders

Recent graphic cards give the programmer the ability to define the functionality of two of the above described stages:

- vertex shaders may be written for the vertex transformation stage;
- fragment shaders replace the fixed functionality of fragment texturing and coloring stage.

Vertex and fragment shaders

Recent graphic cards give the programmer the ability to define the functionality of two of the above described stages:

- vertex shaders may be written for the vertex transformation stage;
- fragment shaders replace the fixed functionality of fragment texturing and coloring stage.

Vertex and fragment shaders

Recent graphic cards give the programmer the ability to define the functionality of two of the above described stages:

- vertex shaders may be written for the vertex transformation stage;
- fragment shaders replace the fixed functionality of fragment texturing and coloring stage.

Vertex processor

- The vertex processor is responsible for running vertex shaders.
- The input for a vertex shader is the vertex data (e.g. its position, color, normals, etc.) depending on what the OpenGL application sends.
- The OpenGL code

```
glBegin(...);

        glColor3f ( 0.1,  0.5,  0.9);
        glVertex3f(-1.0, -1.0, -1.0);

        glColor3f ( 1.0,  0.8,  0.0);
        glVertex3f( 1.0,  1.0,  1.0);

        glColor3f ( 1.0,  0.0,  0.0);
        glVertex3f(-1.0,  1.0,  1.0);

glEnd();
```

would send to the vertex processor a color and a vertex position for each vertex.

Vertex processor

- The vertex processor is responsible for running vertex shaders.
- The input for a vertex shader is the vertex data (e.g. its position, color, normals, etc.) depending on what the OpenGL application sends.
- The OpenGL code

```
glBegin(...);

        glColor3f ( 0.1,  0.5,  0.9);
        glVertex3f(-1.0, -1.0, -1.0);

        glColor3f ( 1.0,  0.8,  0.0);
        glVertex3f( 1.0,  1.0,  1.0);

        glColor3f ( 1.0,  0.0,  0.0);
        glVertex3f(-1.0,  1.0,  1.0);

glEnd();
```

would send to the vertex processor a color and a vertex position for each vertex.

Vertex processor

- The vertex processor is responsible for running vertex shaders.
- The input for a vertex shader is the vertex data (e.g. its position, color, normals, etc.) depending on what the OpenGL application sends.
- The OpenGL code

```
glBegin(...);

        glColor3f ( 0.1,  0.5,  0.9);
        glVertex3f(-1.0, -1.0, -1.0);

        glColor3f ( 1.0,  0.8,  0.0);
        glVertex3f( 1.0,  1.0,  1.0);

        glColor3f ( 1.0,  0.0,  0.0);
        glVertex3f(-1.0,  1.0,  1.0);

glEnd();
```

would send to the vertex processor a color and a vertex position for each vertex.

Vertex processor – continued

In a vertex shader you can write code for tasks such as:

- vertex position transformation using the modelview and projection matrices;
- normal transformation, and if required its normalization;
- texture coordinate generation and transformation;
- lighting per vertex or computing values for lighting per pixel;
- color computation.

Remark

- There is no requirement to perform all the operations above, your application may not use lighting for instance.
- However, once you write a vertex shader you are replacing the full functionality of the vertex processor, hence you cannot expect the fixed functionality to perform normal transformation and texture coordinate generation. When a vertex shader is used it becomes responsible for replacing all the needed functionality of this stage of the pipeline.

Vertex processor – continued

In a vertex shader you can write code for tasks such as:

- vertex position transformation using the modelview and projection matrices;
- normal transformation, and if required its normalization;
- texture coordinate generation and transformation;
- lighting per vertex or computing values for lighting per pixel;
- color computation.

Remark

- There is no requirement to perform all the operations above, your application may not use lighting for instance.
- However, once you write a vertex shader you are replacing the full functionality of the vertex processor, hence you cannot expect the fixed functionality to perform normal transformation and texture coordinate generation. When a vertex shader is used it becomes responsible for replacing all the needed functionality of this stage of the pipeline.

Vertex processor – continued

In a vertex shader you can write code for tasks such as:

- vertex position transformation using the modelview and projection matrices;
- normal transformation, and if required its normalization;
- texture coordinate generation and transformation;
- lighting per vertex or computing values for lighting per pixel;
- color computation.

Remark

- There is no requirement to perform all the operations above, your application may not use lighting for instance.
- However, once you write a vertex shader you are replacing the full functionality of the vertex processor, hence you cannot expect the fixed functionality to perform normal transformation and texture coordinate generation. When a vertex shader is used it becomes responsible for replacing all the needed functionality of this stage of the pipeline.

Vertex processor – continued

In a vertex shader you can write code for tasks such as:

- vertex position transformation using the modelview and projection matrices;
- normal transformation, and if required its normalization;
- texture coordinate generation and transformation;
- lighting per vertex or computing values for lighting per pixel;
- color computation.

Remark

- There is no requirement to perform all the operations above, your application may not use lighting for instance.
- However, once you write a vertex shader you are replacing the full functionality of the vertex processor, hence you cannot expect the fixed functionality to perform normal transformation and texture coordinate generation. When a vertex shader is used it becomes responsible for replacing all the needed functionality of this stage of the pipeline.

Vertex processor – continued

In a vertex shader you can write code for tasks such as:

- vertex position transformation using the modelview and projection matrices;
- normal transformation, and if required its normalization;
- texture coordinate generation and transformation;
- lighting per vertex or computing values for lighting per pixel;
- color computation.

Remark

- There is no requirement to perform all the operations above, your application may not use lighting for instance.
- However, once you write a vertex shader you are replacing the full functionality of the vertex processor, hence you cannot expect the fixed functionality to perform normal transformation and texture coordinate generation. When a vertex shader is used it becomes responsible for replacing all the needed functionality of this stage of the pipeline.

Vertex processor – continued

In a vertex shader you can write code for tasks such as:

- vertex position transformation using the modelview and projection matrices;
- normal transformation, and if required its normalization;
- texture coordinate generation and transformation;
- lighting per vertex or computing values for lighting per pixel;
- color computation.

Remark

- There is no requirement to perform all the operations above, your application may not use lighting for instance.
- However, once you write a vertex shader you are replacing the full functionality of the vertex processor, hence you cannot expect the fixed functionality to perform normal transformation and texture coordinate generation. When a vertex shader is used it becomes responsible for replacing all the needed functionality of this stage of the pipeline.

Vertex processor – continued

In a vertex shader you can write code for tasks such as:

- vertex position transformation using the modelview and projection matrices;
- normal transformation, and if required its normalization;
- texture coordinate generation and transformation;
- lighting per vertex or computing values for lighting per pixel;
- color computation.

Remark

- There is no requirement to perform all the operations above, your application may not use lighting for instance.
- However, once you write a vertex shader you are replacing the full functionality of the vertex processor, hence you cannot expect the fixed functionality to perform normal transformation and texture coordinate generation. When a vertex shader is used it becomes responsible for replacing all the needed functionality of this stage of the pipeline.

Vertex processor – continued

In a vertex shader you can write code for tasks such as:

- vertex position transformation using the modelview and projection matrices;
- normal transformation, and if required its normalization;
- texture coordinate generation and transformation;
- lighting per vertex or computing values for lighting per pixel;
- color computation.

Remark

- There is no requirement to perform all the operations above, your application may not use lighting for instance.
- However, once you write a vertex shader you are replacing the full functionality of the vertex processor, hence you cannot expect the fixed functionality to perform normal transformation and texture coordinate generation. When a vertex shader is used it becomes responsible for replacing all the needed functionality of this stage of the pipeline.

Vertex processor – continued

- The vertex processor has no information regarding connectivity, hence operations that require topological knowledge cannot be performed in here. For instance it is not possible for a vertex shader to perform back face culling, since it operates on vertices and not on faces. The vertex processor processes vertices individually and has no clue of the remaining vertices.
- A vertex processor has access to OpenGL state, so it can perform operations that involve lighting for instance, and use materials. It can also access textures. However, there is no access to the frame buffer.
- The vertex shader is responsible for at least writing a variable named gl_Position, usually transforming the vertex with the modelview and projection matrices.

Vertex processor – continued

- The vertex processor has no information regarding connectivity, hence operations that require topological knowledge cannot be performed in here. For instance it is not possible for a vertex shader to perform back face culling, since it operates on vertices and not on faces. The vertex processor processes vertices individually and has no clue of the remaining vertices.
- A vertex processor has access to OpenGL state, so it can perform operations that involve lighting for instance, and use materials. It can also access textures. However, there is no access to the frame buffer.
- The vertex shader is responsible for at least writing a variable named gl_Position, usually transforming the vertex with the modelview and projection matrices.

Vertex processor – continued

- The vertex processor has no information regarding connectivity, hence operations that require topological knowledge cannot be performed in here. For instance it is not possible for a vertex shader to perform back face culling, since it operates on vertices and not on faces. The vertex processor processes vertices individually and has no clue of the remaining vertices.
- A vertex processor has access to OpenGL state, so it can perform operations that involve lighting for instance, and use materials. It can also access textures. However, there is no access to the frame buffer.
- The vertex shader is responsible for at least writing a variable named gl_Position, usually transforming the vertex with the modelview and projection matrices.

Fragment processor

- The fragment processor is where the fragment shaders run. This unit is responsible for operations like:
  - computing colors, and texture coordinates per pixel;
  - texture application;
  - fog computation;
  - computing normals if you want lighting per pixel.
- The inputs for this unit are the interpolated values computed in the previous stage of the pipeline such as vertex positions, colors, normals, etc. In the vertex shader these values are computed for each vertex. The fragment shader deals with the fragments inside the primitives, hence the need for the interpolated values.

Remark

- As in the vertex processor, when you write a fragment shader it replaces all the fixed functionality. Therefore it is not possible to have a fragment shader texturing the fragment and leave the fog for the fixed functionality. The programmer must code all effects that the application requires.
- The fragment processor operates on single fragments, i.e. it has no clue about the neighboring fragments. Similar to the vertex shaders, the fragment shader has access to OpenGL state, and therefore it can access e.g. the fog color specified by the OpenGL application.

Fragment processor

- The fragment processor is where the fragment shaders run. This unit is responsible for operations like:
  - computing colors, and texture coordinates per pixel;
  - texture application;
  - fog computation;
  - computing normals if you want lighting per pixel.
- The inputs for this unit are the interpolated values computed in the previous stage of the pipeline such as vertex positions, colors, normals, etc. In the vertex shader these values are computed for each vertex. The fragment shader deals with the fragments inside the primitives, hence the need for the interpolated values.

Remark

- As in the vertex processor, when you write a fragment shader it replaces all the fixed functionality. Therefore it is not possible to have a fragment shader texturing the fragment and leave the fog for the fixed functionality. The programmer must code all effects that the application requires.
- The fragment processor operates on single fragments, i.e. it has no clue about the neighboring fragments. Similar to the vertex shaders, the fragment shader has access to OpenGL state, and therefore it can access e.g. the fog color specified by the OpenGL application.

Fragment processor

- The fragment processor is where the fragment shaders run. This unit is responsible for operations like:
  - computing colors, and texture coordinates per pixel;
  - texture application;
  - fog computation;
  - computing normals if you want lighting per pixel.
- The inputs for this unit are the interpolated values computed in the previous stage of the pipeline such as vertex positions, colors, normals, etc. In the vertex shader these values are computed for each vertex. The fragment shader deals with the fragments inside the primitives, hence the need for the interpolated values.

Remark

- As in the vertex processor, when you write a fragment shader it replaces all the fixed functionality. Therefore it is not possible to have a fragment shader texturing the fragment and leave the fog for the fixed functionality. The programmer must code all effects that the application requires.
- The fragment processor operates on single fragments, i.e. it has no clue about the neighboring fragments. Similar to the vertex shaders, the fragment shader has access to OpenGL state, and therefore it can access e.g. the fog color specified by the OpenGL application.

Fragment processor

- The fragment processor is where the fragment shaders run. This unit is responsible for operations like:
  - computing colors, and texture coordinates per pixel;
  - texture application;
  - fog computation;
  - computing normals if you want lighting per pixel.
- The inputs for this unit are the interpolated values computed in the previous stage of the pipeline such as vertex positions, colors, normals, etc. In the vertex shader these values are computed for each vertex. The fragment shader deals with the fragments inside the primitives, hence the need for the interpolated values.

Remark

- As in the vertex processor, when you write a fragment shader it replaces all the fixed functionality. Therefore it is not possible to have a fragment shader texturing the fragment and leave the fog for the fixed functionality. The programmer must code all effects that the application requires.
- The fragment processor operates on single fragments, i.e. it has no clue about the neighboring fragments. Similar to the vertex shaders, the fragment shader has access to OpenGL state, and therefore it can access e.g. the fog color specified by the OpenGL application.

Fragment processor

- The fragment processor is where the fragment shaders run. This unit is responsible for operations like:
  - computing colors, and texture coordinates per pixel;
  - texture application;
  - fog computation;
  - computing normals if you want lighting per pixel.
- The inputs for this unit are the interpolated values computed in the previous stage of the pipeline such as vertex positions, colors, normals, etc. In the vertex shader these values are computed for each vertex. The fragment shader deals with the fragments inside the primitives, hence the need for the interpolated values.

Remark

- As in the vertex processor, when you write a fragment shader it replaces all the fixed functionality. Therefore it is not possible to have a fragment shader texturing the fragment and leave the fog for the fixed functionality. The programmer must code all effects that the application requires.
- The fragment processor operates on single fragments, i.e. it has no clue about the neighboring fragments. Similar to the vertex shaders, the fragment shader has access to OpenGL state, and therefore it can access e.g. the fog color specified by the OpenGL application.

Fragment processor

Possible tasks of a fragment shader

Fragment processor

- The fragment processor is where the fragment shaders run. This unit is responsible for operations like:
  - computing colors, and texture coordinates per pixel;
  - texture application;
  - fog computation;
  - computing normals if you want lighting per pixel.
- The inputs for this unit are the interpolated values computed in the previous stage of the pipeline such as vertex positions, colors, normals, etc. In the vertex shader these values are computed for each vertex. The fragment shader deals with the fragments inside the primitives, hence the need for the interpolated values.

Remark

- As in the vertex processor, when you write a fragment shader it replaces all the fixed functionality. Therefore it is not possible to have a fragment shader texturing the fragment and leave the fog for the fixed functionality. The programmer must code all effects that the application requires.
- The fragment processor operates on single fragments, i.e. it has no clue about the neighboring fragments. Similar to the vertex shaders, the fragment shader has access to OpenGL state, and therefore it can access e.g. the fog color specified by the OpenGL application.

Fragment processor

- The fragment processor is where the fragment shaders run. This unit is responsible for operations like:
  - computing colors, and texture coordinates per pixel;
  - texture application;
  - fog computation;
  - computing normals if you want lighting per pixel.
- The inputs for this unit are the interpolated values computed in the previous stage of the pipeline such as vertex positions, colors, normals, etc. In the vertex shader these values are computed for each vertex. The fragment shader deals with the fragments inside the primitives, hence the need for the interpolated values.

Remark

- As in the vertex processor, when you write a fragment shader it replaces all the fixed functionality. Therefore it is not possible to have a fragment shader texturing the fragment and leave the fog for the fixed functionality. The programmer must code all effects that the application requires.
- The fragment processor operates on single fragments, i.e. it has no clue about the neighboring fragments. Similar to the vertex shaders, the fragment shader has access to OpenGL state, and therefore it can access e.g. the fog color specified by the OpenGL application.

Fragment processor

- The fragment processor is where the fragment shaders run. This unit is responsible for operations like:
  - computing colors, and texture coordinates per pixel;
  - texture application;
  - fog computation;
  - computing normals if you want lighting per pixel.
- The inputs for this unit are the interpolated values computed in the previous stage of the pipeline such as vertex positions, colors, normals, etc. In the vertex shader these values are computed for each vertex. The fragment shader deals with the fragments inside the primitives, hence the need for the interpolated values.

Remark

- As in the vertex processor, when you write a fragment shader it replaces all the fixed functionality. Therefore it is not possible to have a fragment shader texturing the fragment and leave the fog for the fixed functionality. The programmer must code all effects that the application requires.
- The fragment processor operates on single fragments, i.e. it has no clue about the neighboring fragments. Similar to the vertex shaders, the fragment shader has access to OpenGL state, and therefore it can access e.g. the fog color specified by the OpenGL application.

Fragment processor – continued

- One important point is that a fragment shader cannot change the pixel coordinate, as computed previously in the pipeline. Recall that in the vertex processor the modelview and projection matrices can be used to transform the vertex. The viewport comes into play after that but before the fragment processor.
- The fragment shader has access to the pixels location on screen but it can't change it.
- A fragment shader has two output options:
  - to discard the fragment, hence outputting nothing;
  - to compute either gl_FragColor (the final color of the fragment), or gl_FragData when rendering to multiple targets.
- Depth can also be written although it is not required since the previous stage already has computed it.
- Notice that the fragment shader has no access to the frame buffer. This implies that operations such as blending occur only after the fragment shader has run.

Fragment processor – continued

- One important point is that a fragment shader cannot change the pixel coordinate, as computed previously in the pipeline. Recall that in the vertex processor the modelview and projection matrices can be used to transform the vertex. The viewport comes into play after that but before the fragment processor.
- The fragment shader has access to the pixels location on screen but it can't change it.
- A fragment shader has two output options:
  - to discard the fragment, hence outputting nothing;
  - to compute either gl_FragColor (the final color of the fragment), or gl_FragData when rendering to multiple targets.
- Depth can also be written although it is not required since the previous stage already has computed it.
- Notice that the fragment shader has no access to the frame buffer. This implies that operations such as blending occur only after the fragment shader has run.

Fragment processor – continued

- One important point is that a fragment shader cannot change the pixel coordinate, as computed previously in the pipeline. Recall that in the vertex processor the modelview and projection matrices can be used to transform the vertex. The viewport comes into play after that but before the fragment processor.
- The fragment shader has access to the pixels location on screen but it can't change it.
- A fragment shader has two output options:
  - to discard the fragment, hence outputting nothing;
  - to compute either gl_FragColor (the final color of the fragment), or gl_FragData when rendering to multiple targets.
- Depth can also be written although it is not required since the previous stage already has computed it.
- Notice that the fragment shader has no access to the frame buffer. This implies that operations such as blending occur only after the fragment shader has run.

Fragment processor – continued

- One important point is that a fragment shader cannot change the pixel coordinate, as computed previously in the pipeline. Recall that in the vertex processor the modelview and projection matrices can be used to transform the vertex. The viewport comes into play after that but before the fragment processor.
- The fragment shader has access to the pixels location on screen but it can't change it.
- A fragment shader has two output options:
  - to discard the fragment, hence outputting nothing;
  - to compute either gl_FragColor (the final color of the fragment), or gl_FragData when rendering to multiple targets.
- Depth can also be written although it is not required since the previous stage already has computed it.
- Notice that the fragment shader has no access to the frame buffer. This implies that operations such as blending occur only after the fragment shader has run.

Fragment processor – continued

- One important point is that a fragment shader cannot change the pixel coordinate, as computed previously in the pipeline. Recall that in the vertex processor the modelview and projection matrices can be used to transform the vertex. The viewport comes into play after that but before the fragment processor.
- The fragment shader has access to the pixels location on screen but it can't change it.
- A fragment shader has two output options:
  - to discard the fragment, hence outputting nothing;
  - to compute either gl_FragColor (the final color of the fragment), or gl_FragData when rendering to multiple targets.
- Depth can also be written although it is not required since the previous stage already has computed it.
- Notice that the fragment shader has no access to the frame buffer. This implies that operations such as blending occur only after the fragment shader has run.

Fragment processor – continued

- One important point is that a fragment shader cannot change the pixel coordinate, as computed previously in the pipeline. Recall that in the vertex processor the modelview and projection matrices can be used to transform the vertex. The viewport comes into play after that but before the fragment processor.
- The fragment shader has access to the pixels location on screen but it can't change it.
- A fragment shader has two output options:
  - to discard the fragment, hence outputting nothing;
  - to compute either gl_FragColor (the final color of the fragment), or gl_FragData when rendering to multiple targets.
- Depth can also be written although it is not required since the previous stage already has computed it.
- Notice that the fragment shader has no access to the frame buffer. This implies that operations such as blending occur only after the fragment shader has run.

Fragment processor – continued

- One important point is that a fragment shader cannot change the pixel coordinate, as computed previously in the pipeline. Recall that in the vertex processor the modelview and projection matrices can be used to transform the vertex. The viewport comes into play after that but before the fragment processor.
- The fragment shader has access to the pixels location on screen but it can't change it.
- A fragment shader has two output options:
  - to discard the fragment, hence outputting nothing;
  - to compute either gl_FragColor (the final color of the fragment), or gl_FragData when rendering to multiple targets.
- Depth can also be written although it is not required since the previous stage already has computed it.
- Notice that the fragment shader has no access to the frame buffer. This implies that operations such as blending occur only after the fragment shader has run.

To check for OpenGL 2.0 availability you could try something like this

```
#include "GLWidget.h"
#include "Exceptions.h"

using namespace cagd;

...

void GLWidget::initializeGL()
{
    ...

    glewInit();

    if ( glewIsSupported("GL_VERSION_2_0") )
    {
        // ready for OpenGL 2.0
    }
    else
        throw Exception("OpenGL 2.0 is not supported!");

    ...
}
```

If relying on extensions, because you have no support for OpenGL 2.0 yet, then two extensions are required

```
#include "GLWidget.h"
#include "Exceptions.h"

using namespace cagd;

...

void GLWidget::initializeGL()
{
    ...

    glewInit();

    if ( GLEW_ARB_vertex_shader && GLEW_ARB_fragment_shader )
    {
        // ready for GLSL
    }
    else
        throw Exception("GLSL is not supported!");

    ...
}
```

OpenGL setup for GLSL

Creating, compiling, attaching, linking, using shaders

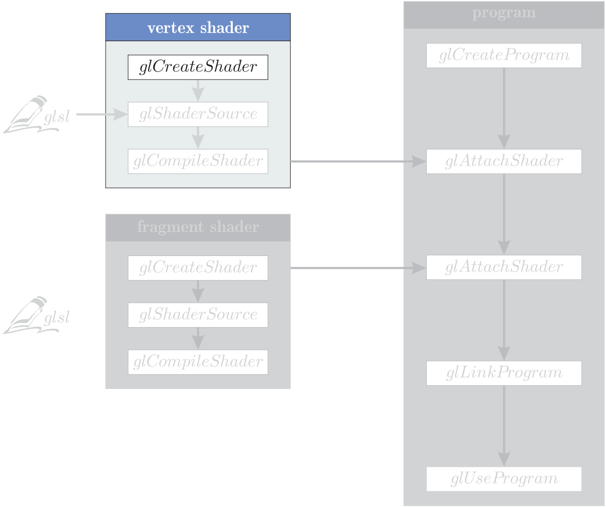

Fig. 3: OpenGL syntax to create, compile, attach, link and use shaders.

OpenGL setup for GLSL

Creating, compiling, attaching, linking, using shaders

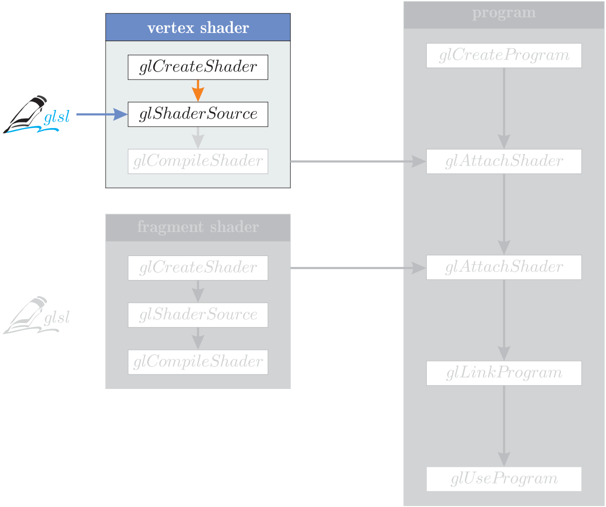

Fig. 3: OpenGL syntax to create, compile, attach, link and use shaders.

OpenGL setup for GLSL

Creating, compiling, attaching, linking, using shaders

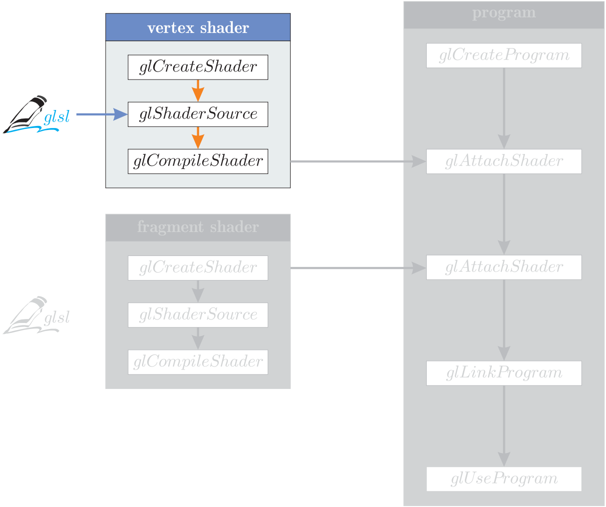

Fig. 3: OpenGL syntax to create, compile, attach, link and use shaders.

OpenGL setup for GLSL

Creating, compiling, attaching, linking, using shaders

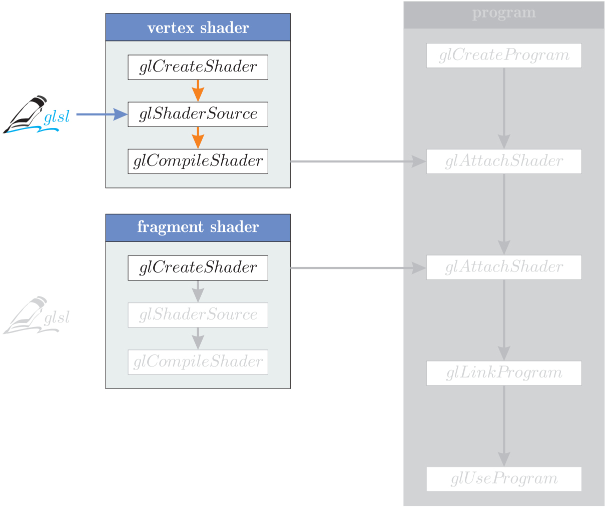

Fig. 3: OpenGL syntax to create, compile, attach, link and use shaders.

OpenGL setup for GLSL

Creating, compiling, attaching, linking, using shaders

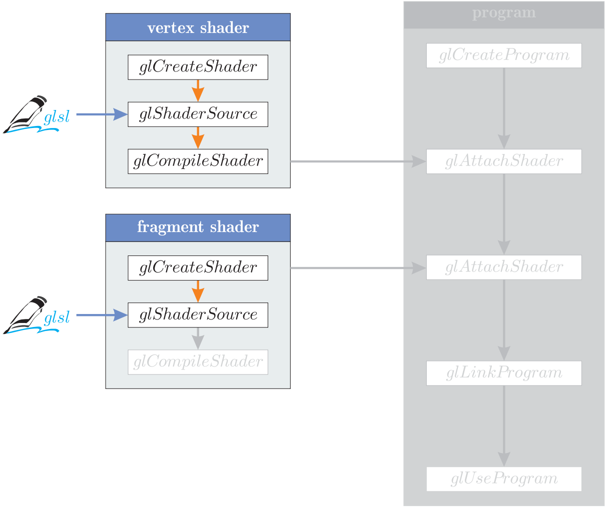

Fig. 3: OpenGL syntax to create, compile, attach, link and use shaders.

OpenGL setup for GLSL

Creating, compiling, attaching, linking, using shaders

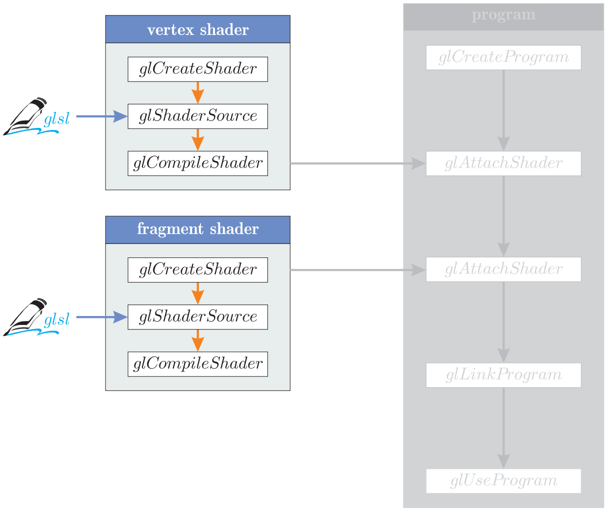

Fig. 3: OpenGL syntax to create, compile, attach, link and use shaders.

OpenGL setup for GLSL

Creating, compiling, attaching, linking, using shaders

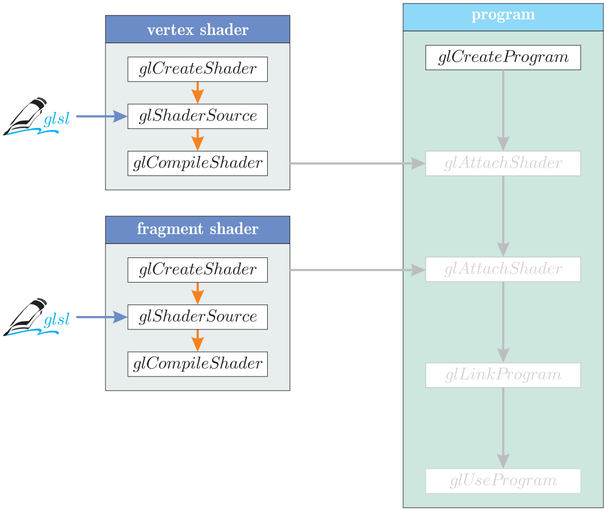

Fig. 3: OpenGL syntax to create, compile, attach, link and use shaders.

OpenGL setup for GLSL

Creating, compiling, attaching, linking, using shaders

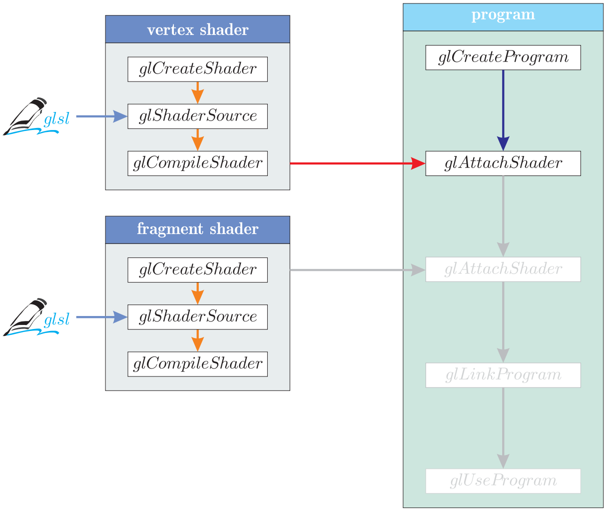

Fig. 3: OpenGL syntax to create, compile, attach, link and use shaders.

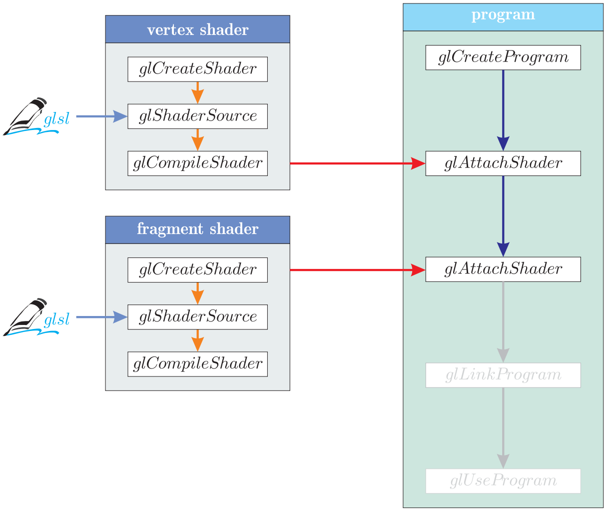

Fig. 3: OpenGL syntax to create, compile, attach, link and use shaders.

OpenGL setup for GLSL

Creating, compiling, attaching, linking, using shaders

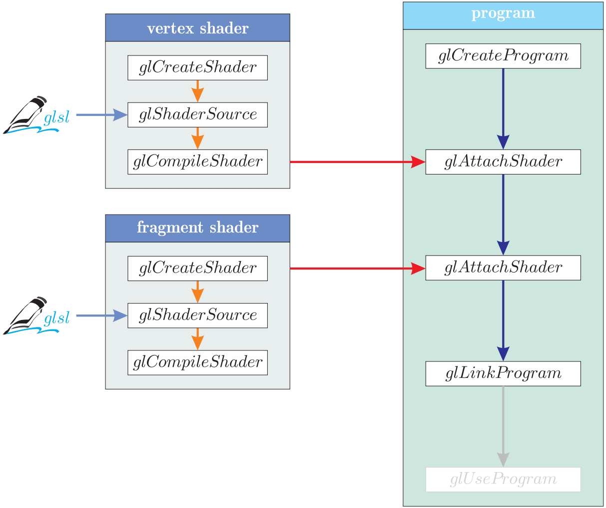

Fig. 3: OpenGL syntax to create, compile, attach, link and use shaders.

OpenGL setup for GLSL

Creating, compiling, attaching, linking, using shaders

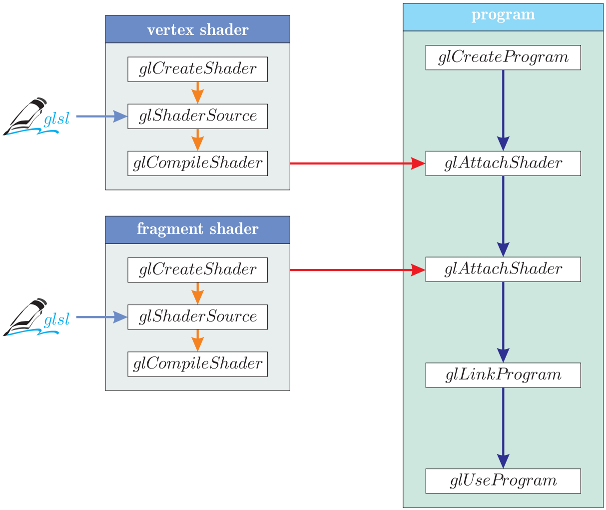

Fig. 3: OpenGL syntax to create, compile, attach, link and use shaders.

OpenGL syntax to create, compile, attach, link, use and delete shaders

Description of command glCreateShader

```
GLuint glCreateShader(GLenum shader_type);

/*
    glCreateShader creates an empty shader object and returns a non-zero value by which it can be
    referenced.

    A shader object is used to maintain the source code strings that define a shader.

    shader_type indicates the type of shader to be created. Two types of shaders are supported.
    A shader of type GL_VERTEX_SHADER is a shader that is intended to run on the programmable
    vertex processor and replace the fixed functionality vertex processing in OpenGL. A shader
    of type GL_FRAGMENT_SHADER is a shader that is intended to run on the programmable fragment
    processor and replace the fixed functionality fragment processing in OpenGL.

    When created, a shader object's GL_SHADER_TYPE parameter is set to either GL_VERTEX_SHADER or
    GL_FRAGMENT_SHADER, depending on the value of shader_type.

    This function returns 0 if an error occurs creating the shader object.

    GL_INVALID_ENUM is generated if shader_type is not an accepted value.

    GL_INVALID_OPERATION is generated if glCreateShader is executed between the execution of glBegin
    and the corresponding execution of glEnd.
*/
```

OpenGL syntax to create, compile, attach, link, use and delete shaders

Description of command glDeleteShader

```
GLvoid glDeleteShader(GLuint shader);

/*
    glDeleteShader frees the memory and invalidates the name associated with the
    shader object specified by shader. This command effectively undoes the effects
    of a call to glCreateShader.

    If a shader object to be deleted is attached to a program object, it will be
    flagged for deletion, but it will not be deleted until it is no longer attached
    to any program object, for any rendering context (i.e. it must be detached from
    wherever it was attached before it will be deleted). A value of 0 for shader will
    be silently ignored.

    To determine whether an object has been flagged for deletion, call glGetShader with
    arguments shader and GL_DELETE_STATUS.

    GL_INVALID_VALUE is generated if shader is not a value generated by OpenGL.

    GL_INVALID_OPERATION is generated if glDeleteShader is executed between the execution of
    glBegin and the corresponding execution of glEnd.
*/
```

OpenGL syntax to create, compile, attach, link, use and delete shaders


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

/*
   glShaderSource sets the source code in shader to the source code in the array of strings
   specified by string.

   Any source code previously stored in the shader object is completely replaced.

   The number of strings in the array is specified by count.

   If length is NULL, each string is assumed to be null terminated. If length is a value other
   than NULL, it points to an array containing a string length for each of the corresponding
   elements of string. Each element in the length array may contain the length of the corresponding
   string (the null character is not counted as part of the string length) or a value less than 0
   to indicate that the string is null terminated.

   The source code strings are not scanned or parsed at this time; they are simply copied into
   the specified shader object.

   GL_INVALID_VALUE is generated if shader is not a value generated by OpenGL.

   GL_INVALID_OPERATION is generated if shader is not a shader object.

   GL_INVALID_VALUE is generated if count is less than 0.

   GL_INVALID_OPERATION is generated if glShaderSource is executed between the execution of glBegin
   and the corresponding execution of glEnd.
*/
```

OpenGL syntax to create, compile, attach, link, use and delete shaders

Description of command glCompileShader

```
GLvoid glCompileShader(GLuint shader);

/*
   glCompileShader compiles the source code strings that have been stored in the shader object
   specified by shader.

   The compilation status will be stored as part of the shader object's state. This value will be
   set to GL_TRUE if the shader was compiled without errors and is ready for use, and GL_FALSE
   otherwise.

   It can be queried by calling glGetShader with arguments shader and GL_COMPILE_STATUS.

   Compilation of a shader can fail for a number of reasons as specified by the OpenGL Shading
   Language Specification. Whether or not the compilation was successful, information about the
   compilation can be obtained from the shader object's information log by calling glGetShaderInfoLog.

   GL_INVALID_VALUE is generated if shader is not a value generated by OpenGL.

   GL_INVALID_OPERATION is generated if shader is not a shader object.

   GL_INVALID_OPERATION is generated if glCompileShader is executed between the execution of glBegin
   and the corresponding execution of glEnd.
*/
```

OpenGL syntax to create, compile, attach, link, use and delete shaders

Description of command glGetShaderInfoLog

```
GLvoid glGetShaderInfoLog(GLuint shader, GLsizei max_length, GLsizei *length, GLchar *info_log);

/*
   glGetShaderInfoLog returns the information log for the specified shader object.
   The information log for a shader object is modified when the shader is compiled.
   The string that is returned will be null terminated.

   glGetShaderInfoLog returns in info_log as much of the information log as it can,
   up to a maximum of max_length characters.

   The number of characters actually returned, excluding the null termination character,
   is specified by length. If the length of the returned string is not required, a value
   of NULL can be passed in the length argument. The size of the buffer required to store
   the returned information log can be obtained by calling glGetShader with the value
   GL_INFO_LOG_LENGTH.

   The information log for a shader object is a string that may contain diagnostic messages,
   warning messages, and other information about the last compile operation. When a shader
   object is created, its information log will be a string of length 0.

   GL_INVALID_VALUE is generated if shader is not a value generated by OpenGL.

   GL_INVALID_OPERATION is generated if shader is not a shader object.

   GL_INVALID_VALUE is generated if max_length is less than 0.

   GL_INVALID_OPERATION is generated if glGetShaderInfoLog is executed between the execution
   of glBegin and the corresponding execution of glEnd.
*/
```

OpenGL syntax to create, compile, attach, link, use and delete shaders

Description of command glGetShaderiv

```
GLvoid glGetShaderiv(GLuint shader, GLenum parameter_name, GLint *parameters);

/*
   glGetShader returns in parameters the value of a parameter for a specific shader object. The
   following parameters are defined:

   GL_SHADER_TYPE

      parameters returns GL_VERTEX_SHADER if shader is a vertex shader object, and GL_FRAGMENT_SHADER
      if shader is a fragmentshader object.

   GL_DELETE_STATUS

      parameters returns GL_TRUE if shader is currently flagged for deletion, and GL_FALSE otherwise.

   GL_COMPILE_STATUS

      parameters returns GL_TRUE if the last compile operation on shader was successful, and GL_FALSE
      otherwise.

   GL_INFO_LOG_LENGTH

      parameters returns the number of characters in the information log for shader including the
      null termination character (i.e. the size of the character buffer required to store the
      information log). If shader has no information log, a value of 0 is returned.

   GL_SHADER_SOURCE_LENGTH

      parameters returns the length of the concatenation of the source strings that make up the
      shader source for the shader, including the null termination character (i.e. the size of the
      character buffer required to store the shader source). If no source code exists, 0 is returned.
*/
```

OpenGL syntax to create, compile, attach, link, use and delete shaders

Description of command glCreateProgram

```
GLuint glCreateProgram(GLvoid);

/*
    glCreateProgram creates an empty program object and returns a non-zero value
    by which it can be referenced.

    A program object is an object to which shader objects can be attached.
    This provides a mechanism to specify the shader objects that will be linked to
    create a program. It also provides a means for checking the compatibility of the
    shaders that will be used to create a program (for instance, checking the compatibility
    between a vertex shader and a fragment shader). When no longer needed as part of a
    program object, shader objects can be detached.

    One or more executables are created in a program object by successfully attaching shader
    objects to it with glAttachShader, successfully compiling the shader objects with
    glCompileShader, and successfully linking the program object with glLinkProgram.

    These executables are made part of current state when glUseProgram is called. Program
    objects can be deleted by calling glDeleteProgram.

    The memory associated with the program object will be deleted when it is no longer part
    of current rendering state for any context.

    This function returns 0 if an error occurs creating the program object.

    GL_INVALID_OPERATION is generated if glCreateProgram is executed between the execution of
    glBegin and the corresponding execution of glEnd.
*/
```

OpenGL syntax to create, compile, attach, link, use and delete shaders

Description of command glDeleteProgram

```
GLvoid glDeleteProgram(GLuint program);

/*
   glDeleteProgram frees the memory and invalidates the name associated with
   the program object specified by program. This command effectively undoes the
   effects of a call to glCreateProgram.

   If a program object is in use as part of current rendering state, it will be
   flagged for deletion, but it will not be deleted until it is no longer part of
   current state for any rendering context.

   If a program object to be deleted has shader objects attached to it, those shader
   objects will be automatically detached but not deleted unless they have already
   been flagged for deletion by a previous call to glDeleteShader.

   A value of 0 for program will be silently ignored.

   To determine whether a program object has been flagged for deletion, call
   glGetProgram with arguments program and GL_DELETE_STATUS.

   GL_INVALID_VALUE is generated if program is not a value generated by OpenGL.

   GL_INVALID_OPERATION is generated if glDeleteProgram is executed between the execution of
   glBegin and the corresponding execution of glEnd.
*/
```

OpenGL syntax to create, compile, attach, link, use and delete shaders

Description of command glAttachShader

```
GLvoid glAttachShader(GLuint program, GLuint shader);

/*
    In order to create an executable, there must be a way to specify the list of things
    that will be linked together. Program objects provide this mechanism.

    Shaders that are to be linked together in a program object must first be attached to
    that program object.

    glAttachShader attaches the shader object specified by shader to the program object
    specified by program. This indicates that shader will be included in link operations
    that will be performed on program.

    All operations that can be performed on a shader object are valid whether or not the
    shader object is attached to a program object.

    It is permissible to attach a shader object to a program object before source code
    has been loaded into the shader object or before the shader object has been compiled.

    It is permissible to attach multiple shader objects of the same type because each may
    contain a portion of the complete shader. It is also permissible to attach a shader
    object to more than one program object. If a shader object is deleted while it is
    attached to a program object, it will be flagged for deletion, and deletion will not
    occur until glDetachShader is called to detach it from all program objects to which
    it is attached.

    GL_INVALID_VALUE is generated if either program or shader is not a value generated by OpenGL.

    GL_INVALID_OPERATION is generated if program is not a program object.
    GL_INVALID_OPERATION is generated if shader is not a shader object.
    GL_INVALID_OPERATION is generated if shader is already attached to program.
    GL_INVALID_OPERATION is generated if glAttachShader is executed between the execution of
    glBegin and the corresponding execution of glEnd.
*/
```

OpenGL syntax to create, compile, attach, link, use and delete shaders

Description of command glDetachShader

```
GLvoid glDetachShader(GLuint program, GLuint shader);

/*
   glDetachShader detaches the shader object specified by shader from the program
   object specified by program. This command can be used to undo the effect of the
   command glAttachShader.

   If shader has already been flagged for deletion by a call to glDeleteShader
   and it is not attached to any other program object, it will be deleted after it has
   been detached.

   GL_INVALID_VALUE is generated if either program or shader is a value that was not
   generated by OpenGL.

   GL_INVALID_OPERATION is generated if program is not a program object.

   GL_INVALID_OPERATION is generated if shader is not a shader object.

   GL_INVALID_OPERATION is generated if shader is not attached to program.

   GL_INVALID_OPERATION is generated if glDetachShader is executed between the execution of
   glBegin and the corresponding execution of glEnd.
*/
```

OpenGL syntax to create, compile, attach, link, use and delete shaders, part I


```
GLvoid glLinkProgram(GLuint program);

/*
   glLinkProgram links the program object specified by program.

   If any shader objects of type GL_VERTEX_SHADER are attached to program,
   they will be used to create an executable that will run on the programmable
   vertex processor.

   If any shader objects of type GL_FRAGMENT_SHADER are attached to program, they
   will be used to create an executable that will run on the programmable fragment
   processor.

   The status of the link operation will be stored as part of the program object's state.
   This value will be set to GL_TRUE if the program object was linked without errors and
   is ready for use, and GL_FALSE otherwise. It can be queried by calling glGetProgram
   with arguments program and GL_LINK_STATUS.

   As a result of a successful link operation, all active user-defined uniform variables
   belonging to program will be initialized to 0, and each of the program object's active
   uniform variables will be assigned a location that can be queried by calling
   glGetUniformLocation.

   Also, any active user-defined attribute variables that have not been bound to a generic
   vertex attribute index will be bound to one at this time.

   Linking of a program object can fail for a number of reasons as specified in the OpenGL
   Shading Language Specification. The following lists some of the conditions that will cause
   a link error.

   * The number of active attribute variables supported by the implementation has been exceeded.
   * The storage limit for uniform variables has been exceeded.
   * The number of active uniform variables supported by the implementation has been exceeded.
```

OpenGL syntax to create, compile, attach, link, use and delete shaders, part II

Description of command glLinkProgram

- The main function is missing for the vertex shader or the fragment shader.
- A varying variable actually used in the fragment shader is not declared in the same way (or is not declared at all) in the vertex shader.
- A reference to a function or variable name is unresolved.
- A shared global is declared with two different types or two different initial values.
- One or more of the attached shader objects has not been successfully compiled.
- Binding a generic attribute matrix caused some rows of the matrix to fall outside the allowed maximum of GL_MAX_VERTEX_ATTRIBS.
- Not enough contiguous vertex attribute slots could be found to bind attribute matrices.

When a program object has been successfully linked, the program object can be made part of current state by calling glUseProgram.

Whether or not the link operation was successful, the program object's information log will be overwritten. The information log can be retrieved by calling glGetProgramInfoLog.

glLinkProgram will also install the generated executables as part of the current rendering state if the link operation was successful and the specified program object is already currently in use as a result of a previous call to glUseProgram.

If the program object currently in use is relinked unsuccessfully, its link status will be set to GL_FALSE, but the executables and associated state will remain part of the current state until a subsequent call to glUseProgram removes it from use. After it is removed from use, it cannot be made part of current state until it has been successfully relinked.

If program contains shader objects of type GL_VERTEX_SHADER but does not contain shader objects of type GL_FRAGMENT_SHADER, the vertex shader will be linked against the implicit interface for fixed functionality fragment processing. Similarly, if program contains shader objects of type GL_FRAGMENT_SHADER but it does not contain shader objects of type GL_VERTEX_SHADER, the fragment shader will be linked against the implicit interface for fixed functionality vertex processing.

The program object's information log is updated and the program is generated at the time

OpenGL syntax to create, compile, attach, link, use and delete shaders, part III

Description of command glLinkProgram

```
    of the link operation. After the link operation, applications are free to modify attached
    shader objects, compile attached shader objects, detach shader objects, delete shader objects,
    and attach additional shader objects. None of these operations affects the information log or
    the program that is part of the program object.

    GL_INVALID_VALUE is generated if program is not a value generated by OpenGL.

    GL_INVALID_OPERATION is generated if program is not a program object.

    GL_INVALID_OPERATION is generated if glLinkProgram is executed between the execution of glBegin
    and the corresponding execution of glEnd.
*/
```

OpenGL syntax to create, compile, attach, link, use and delete shaders, part I

Description of command glUseProgram

```
GLvoid glUseProgram(GLuint  program);

/*
   glUseProgram installs the program object specified by program as part of current
   rendering state. One or more executables are created in a program object by
   successfully attaching shader objects to it with glAttachShader, successfully compiling
   the shader objects with glCompileShader, and successfully linking the program object with
   glLinkProgram.

   A program object will contain an executable that will run on the vertex processor if it
   contains one or more shader objects of type GL_VERTEX_SHADER that have been successfully compiled
   and linked.

   Similarly, a program object will contain an executable that will run on the fragment
   processor if it contains one or more shader objects of type GL_FRAGMENT_SHADER that have been
   successfully compiled and linked.

   Successfully installing an executable on a programmable processor will cause the corresponding
   fixed functionality of OpenGL to be disabled.

   Specifically, if an executable is installed on the vertex processor, the OpenGL fixed
   functionality will be disabled as follows.

   * The modelview matrix is not applied to vertex coordinates.
   * The projection matrix is not applied to vertex coordinates.
   * The texture matrices are not applied to texture coordinates.
   * Normals are not transformed to eye coordinates.
   * Normals are not rescaled or normalized.
   * Normalization of GL_AUTO_NORMAL evaluated normals is not performed.
   * Texture coordinates are not generated automatically.
   * Per-vertex lighting is not performed.
   * Color material computations are not performed.
   * Color index lighting is not performed.
```

OpenGL syntax to create, compile, attach, link, use and delete shaders, part II

Description of command glUseProgram

```
* This list also applies when setting the current raster position.

The executable that is installed on the vertex processor is expected to implement any or all
of the desired functionality from the preceding list.

Similarly, if an executable is installed on the fragment processor, the OpenGL fixed
functionality will be disabled as follows.

* Texture environment and texture functions are not applied.
* Texture application is not applied.
* Color sum is not applied.
* Fog is not applied.

Again, the fragment shader that is installed is expected to implement any or all of the
desired functionality from the preceding list.

While a program object is in use, applications are free to modify attached shader objects,
compile attached shader objects, attach additional shader objects, and detach or delete shader
objects. None of these operations will affect the executables that are part of the current state.
However, relinking the program object that is currently in use will install the program
object as part of the current rendering state if the link operation was successful (see
glLinkProgram).

If the program object currently in use is relinked unsuccessfully, its link status will be
set to GL_FALSE, but the executables and associated state will remain part of the current
state until a subsequent call to glUseProgram removes it from use. After it is removed from
use, it cannot be made part of current state until it has been successfully relinked.

If program contains shader objects of type GL_VERTEX_SHADER but it does not contain shader
objects of type GL_FRAGMENT_SHADER, an executable will be installed on the vertex processor,
but fixed functionality will be used for fragment processing.

Similarly, if program contains shader objects of type GL_FRAGMENT_SHADER but it does not contain
shader objects of type GL_VERTEX_SHADER, an executable will be installed on the fragment processor,
```

```
    but fixed functionality will be used for vertex processing. If program is 0, the programmable
    processors will be disabled, and fixed functionality will be used for both vertex and fragment
    processing.

    GL_INVALID_VALUE is generated if program is neither 0 nor a value generated by OpenGL.

    GL_INVALID_OPERATION is generated if program is not a program object.

    GL_INVALID_OPERATION is generated if program could not be made part of current
    state.

    GL_INVALID_OPERATION is generated if glUseProgram is executed between the
    execution of glBegin and the corresponding execution of glEnd.
*/
```

OpenGL syntax to create, compile, attach, link, use and delete shaders

Description of command glValidateProgram

```
GLvoid glValidateProgram(GLuint program);

/*
   glValidateProgram checks to see whether the executables contained in program can execute
   given the current OpenGL state. The information generated by the validation process will
   be stored in program's information log.

   The validation information may consist of an empty string, or it may be a string containing
   information about how the current program object interacts with the rest of current OpenGL
   state. This provides a way for OpenGL implementers to convey more information about why the
   current program is inefficient, suboptimal, failing to execute, and so on.

   The status of the validation operation will be stored as part of the program object's state.
   This value will be set to GL_TRUE if the validation succeeded, and GL_FALSE otherwise. It can
   be queried by calling glGetProgram with arguments program and GL_VALIDATE_STATUS.

   If validation is successful, program is guaranteed to execute given the current state. Otherwise,
   program is guaranteed to not execute.

   This function is typically useful only during application development.
   The informational string stored in the information log is completely implementation
   dependent; therefore, an application should not expect different OpenGL implementations
   to produce identical information strings.

   GL_INVALID_VALUE is generated if program is not a value generated by OpenGL.

   GL_INVALID_OPERATION is generated if program is not a program object.

   GL_INVALID_OPERATION is generated if glValidateProgram is executed between the execution of
   glBegin and the corresponding execution of glEnd.
*/
```

OpenGL syntax to create, compile, attach, link, use and delete shaders, part I

Description of command glGetProgramiv

```
GLvoid glGetProgramiv(GLuint program, GLenum  parameter_name, GLint *parameters);

/*
   glGetProgram returns in params the value of a parameter for a specific program object.
   The following parameters are defined:

   GL_DELETE_STATUS

       parameters returns GL_TRUE if program is currently flagged for deletion, and GL_FALSE
       otherwise.

   GL_LINK_STATUS

       parameters returns GL_TRUE if the last link operation on program was successful, and
       GL_FALSE otherwise.

   GL_VALIDATE_STATUS

       parameters returns GL_TRUE or if the last validation operation on program was successful,
       and GL_FALSE otherwise.

   GL_INFO_LOG_LENGTH

       parameters returns the number of characters in the information log for program including
       the  null termination character (i.e., the size of the character buffer required to store
       the information log). If program has no information log, a value of 0 is returned.

   GL_ATTACHED_SHADERS

       parameters returns the number of shader objects attached to program.
```

OpenGL syntax to create, compile, attach, link, use and delete shaders, part II

Description of command glGetProgramiv

```
GL_ACTIVE_ATTRIBUTES

    parameters returns the number of active attribute variables for program.

GL_ACTIVE_ATTRIBUTE_MAX_LENGTH

    parameters returns the length of the longest active attribute name for program, including the
    null termination character (i.e. the size of the character buffer required to store the longest
    attribute name). If no active attributes exist, 0 is returned.

GL_ACTIVE_UNIFORMS

    parameters returns the number of active uniform variables for program.

GL_ACTIVE_UNIFORM_MAX_LENGTH

    parameters returns the length of the longest active uniform variable name for program, including
    the null termination character (i.e., the size of the character buffer required to store the
    longest uniform variable name). If no active uniform variables exist, 0 is returned.

GL_INVALID_VALUE is generated if program is not a value generated by OpenGL.

GL_INVALID_OPERATION is generated if program does not refer to a program object.

GL_INVALID_ENUM is generated if parameter_name is not an accepted value.

GL_INVALID_OPERATION is generated if glGetProgram is executed between the execution of
glBegin and the corresponding execution of glEnd.
*/
```

OpenGL syntax to create, compile, attach, link, use and delete shaders

Description of command glGetProgramInfoLog

```
GLvoid glGetProgramInfoLog(GLuint program, GLsizei max_length, GLsizei *length, GLchar *info_log);

/*
   glGetProgramInfoLog returns the information log for the specified program object.
   The information log for a program object is modified when the program object is linked or validated.

   The string that is returned will be null terminated.

   glGetProgramInfoLog returns in info_log as much of the information log as it can, up to a maximum
   of max_length characters.

   The number of characters actually returned, excluding the null termination character, is specified
   by length. If the length of the returned string is not required, a value of NULL can be passed in
   the length argument.

   The size of the buffer required to store the returned information log can be obtained by calling
   glGetProgram with the value GL_INFO_LOG_LENGTH.

   The information log for a program object is either an empty string, or a string containing
   information about the last link operation, or a string containing information about the last
   validation operation.

   It may contain diagnostic messages, warning messages, and other information. When a program object
   is created, its information log will be a string of length 0.
*/
```

Implementation details – header file, part I

Class Shader, ShaderPrograms.h

```
#pragma once

#include <GL/glew.h>
#include <iostream>
#include <vector>
#include <string>

namespace cagd
{
    class ShaderProgram
    {
    protected:
        // handles of objects
        GLuint      _vertex_shader;
        GLuint      _fragment_shader;
        GLuint      _program;

        // file names of sources
        std::string _vertex_shader_file_name;
        std::string _fragment_shader_file_name;

        // sources
        std::string _vertex_shader_source;
        std::string _fragment_shader_source;

        // status values
        GLint       _vertex_shader_compiled;
        GLint       _fragment_shader_compiled;
        GLint       _linked;

        // log
        GLboolean   _ListOpenGLErrors(
                const char *file_name, GLint line,
                std::ostream& output = std::cout) const;  // returns GL_TRUE if an OpenGL error occurred
```

Implementation details – header file, part II

Class Shader, ShaderPrograms.h

```
        GLvoid      _ListVertexShaderInfoLog(std::ostream& output = std::cout) const;
        GLvoid      _ListFragmentShaderInfoLog(std::ostream& output = std::cout) const;
        GLvoid      _ListProgramInfoLog(std::ostream& output = std::cout) const;
        GLvoid      _ListValidateInfoLog(std::ostream& output = std::cout) const;

    public:
        // default constructor
        ShaderProgram();

        // copy constructor
        ShaderProgram(const ShaderProgram &shader);

        // assignment operator
        ShaderProgram& operator =(const ShaderProgram &rhs);

        // install shaders
        GLboolean InstallShaders(
                const std::string &vertex_shader_file_name,
                const std::string &fragment_shader_file_name,
                GLboolean logging_is_enabled = GL_FALSE, std::ostream &output = std::cout);

        // set uniform variables
        GLboolean SetUniformVariable1i(const GLchar *name, GLint parameter) const;

        GLboolean SetUniformVariable1f(const GLchar *name, GLfloat parameter) const;

        GLboolean SetUniformVariable2f(
                const GLchar *name,
                GLfloat parameter_1, GLfloat parameter_2) const;

        GLboolean SetUniformVariable3f(
                const GLchar *name,
                GLfloat parameter_1, GLfloat parameter_2, GLfloat parameter_3) const;
        // ...

        // get location of uniform variables
```

Implementation details – header file, part III

Class Shader, ShaderPrograms.h

```
        GLint GetUniformVariableLocation(
                const GLchar *name,
                GLboolean logging_is_enabled = GL_FALSE, std::ostream& output = std::cout) const;

        // disable/enable shader
        GLvoid Disable() const;
        GLvoid Enable(GLboolean logging_is_enabled = GL_FALSE, std::ostream& output = std::cout) const;

        // destructor
        virtual ~ShaderProgram();
    };
}
```

Implementation details – source file, part I

Class Shader, ShaderPrograms.cpp

```cpp
#include "Exceptions.h"
#include <fstream>
#include "ShaderPrograms.h"

using namespace cagd;
using namespace std;

ShaderProgram::ShaderProgram():
        _vertex_shader(0), _fragment_shader(0), _program(0),
        _vertex_shader_file_name(""), _fragment_shader_file_name(""),
        _vertex_shader_source(""), _fragment_shader_source(""),
        _vertex_shader_compiled(0), _fragment_shader_compiled(0), _linked(0)
{
}

// returns GL_TRUE if an OpenGL error occurred, GL_FALSE otherwise.
GLboolean ShaderProgram::_ListOpenGLErrors(const char *file_name, GLint line, ostream& output) const
{
    GLenum gl_error;
    GLboolean result = GL_FALSE;

    gl_error = glGetError();
        output << "\t\\begin{OpenGL␣Errors}" << endl;

    while (gl_error != GL_NO_ERROR)
    {
        output << "\t\tError␣in␣file␣" << file_name
               << "␣at␣line␣" << line
               << ":␣" << endl
               << gluErrorString(gl_error) << endl;

        result = GL_TRUE;
        gl_error = glGetError();
    }
```

Implementation details – source file, part II

Class Shader, ShaderPrograms.cpp

```cpp
    output << "\t\\end{OpenGL_Errors}" << endl << endl;
    return result;
}

GLvoid ShaderProgram::_ListVertexShaderInfoLog(ostream& output) const
{
    GLint info_log_length = 0;
    GLint chars_written  = 0;
    GLchar *info_log = 0;

    // check for OpenGL errors
    _ListOpenGLErrors(__FILE__, __LINE__, output);

    glGetShaderiv(_vertex_shader, GL_INFO_LOG_LENGTH, &info_log_length);
    if (info_log_length > 0)
    {
        info_log = new GLchar[info_log_length];
        if (!info_log)
            throw Exception(
                "ShaderProgram::_ListVertexShaderInfoLog - Could not allocate information log buffer!");

        glGetShaderInfoLog(_vertex_shader, info_log_length, &chars_written, info_log);

        output << "\t\\begin{Vertex_Shader_Information_Log}" << endl
               << "\t\tid = " << _vertex_shader << ", name = " << _vertex_shader_file_name << endl;
        output <<  "\t\t" << info_log << endl;
        output << "\t\\end{Vertex_Shader_Information_Log}" << endl << endl;

        delete[] info_log;
    }

    // check for OpenGL errors
    _ListOpenGLErrors(__FILE__, __LINE__, output);
}

GLvoid ShaderProgram::_ListFragmentShaderInfoLog(ostream& output) const
```

Implementation details – source file, part III

Class Shader, ShaderPrograms.cpp

```cpp
{
    GLint info_log_length = 0;
    GLint chars_written   = 0;
    GLchar *info_log = 0;

    // check for OpenGL errors
    _ListOpenGLErrors(__FILE__, __LINE__, output);

    glGetShaderiv(_fragment_shader, GL_INFO_LOG_LENGTH, &info_log_length);

    if (info_log_length > 0)
    {
        info_log = new GLchar[info_log_length];
        if (!info_log)
            throw Exception(
            "ShaderProgram::_ListFragmentShaderInfoLog - Could not allocate information log buffer!");

        glGetShaderInfoLog(_fragment_shader, info_log_length, &chars_written, info_log);

        output << "\t\\begin{Fragment_Shader_InfoLog}" << endl
               << "\t\tid = " << _fragment_shader << ", name = "  << _fragment_shader_file_name << endl;
        output <<  "\t\t" << info_log << endl;
        output << "\t\\end{Fragment_Shader_InfoLog}" << endl << endl;

        delete[] info_log;
    }

    // check for OpenGL errors
    _ListOpenGLErrors(__FILE__, __LINE__, output);
}

GLvoid ShaderProgram::_ListProgramInfoLog(ostream& output) const
{
    GLint info_log_length = 0;
    GLint chars_written   = 0;
    GLchar *info_log = 0;
```

Implementation details – source file, part IV

Class Shader, ShaderPrograms.cpp

```
    // check for OpenGL errors
    _ListOpenGLErrors(__FILE__, __LINE__, output);

    glGetProgramiv(_program, GL_INFO_LOG_LENGTH, &info_log_length);

    if (info_log_length > 0)
    {
        info_log = new GLchar[info_log_length];
        if (!info_log)
            throw Exception(
                "ShaderProgram::_ListProgramInfoLog - Could not allocate information log buffer!");

        glGetProgramInfoLog(_program, info_log_length, &chars_written, info_log);

        output << "\t\\begin{Program_InfoLog}" << endl << "\t\tid = " << _program << endl;
        output <<  "\t\t" << info_log << endl;
        output << "\t\\end{Program_InfoLog}" << endl << endl;

        delete[] info_log;
    }

    // check for OpenGL errors
    _ListOpenGLErrors(__FILE__, __LINE__, output);
}

GLvoid ShaderProgram::_ListValidateInfoLog(ostream& output) const
{
    GLint status = GL_FALSE;

    // check for OpenGL errors
    _ListOpenGLErrors(__FILE__, __LINE__, output);

    glGetProgramiv(_program, GL_VALIDATE_STATUS, &status);

    output << "\t\\begin{Program_Validate_InfoLog}" << endl<< "\t\tid = " << _program << endl;
```

Implementation details – source file, part V

Class Shader, ShaderPrograms.cpp

```
    output <<  (status? "\t\tValidated." : "\t\tNot validated.") << endl;
    output << "\t\\end{Program_Validate_InfoLog}" << endl << endl;

    // check for OpenGL errors
    _ListOpenGLErrors(__FILE__, __LINE__, output);
}

GLint ShaderProgram::GetUniformVariableLocation(
        const GLchar *name,
        GLboolean logging_is_enabled, ostream& output) const
{
    GLint loc = glGetUniformLocation(_program, name);

    if (loc == -1)
    {
        string reason = "\t\tNo such uniform named: ";
        reason += name;
        output << reason << endl;

        // check for OpenGL errors
        if (logging_is_enabled)
            _ListOpenGLErrors(__FILE__, __LINE__, output);
    }
    return loc;
}

GLboolean ShaderProgram::InstallShaders(
        const string &vertex_shader_file_name,
        const string &fragment_shader_file_name,
        GLboolean logging_is_enabled, std::ostream &output)
{

    // loading source codes into shader objects
    _vertex_shader_file_name = vertex_shader_file_name;
    _fragment_shader_file_name = fragment_shader_file_name;
```

Implementation details – source file, part VI

Class Shader, ShaderPrograms.cpp

```
fstream vertex_shader_file(vertex_shader_file_name.c_str(), ios_base::in);

_vertex_shader_source = "";
string aux;

if (logging_is_enabled)
{
    output << "Source of vertex shader" << endl;
    output << "--------------------------" << endl;
}

while (!vertex_shader_file.eof())
{
    getline(vertex_shader_file, aux, '\n');
    _vertex_shader_source += aux + '\n';

    if (logging_is_enabled)
        output << "\t" << aux << endl;
}

vertex_shader_file.close();

if (logging_is_enabled)
    output << endl;

fstream fragment_shader_file(fragment_shader_file_name.c_str(), ios_base::in);
_fragment_shader_source = "";

if (logging_is_enabled)
{
    output << "Source of fragment shader" << endl;
    output << "----------------------------" << endl;
}

while (!fragment_shader_file.eof())
{
```

Implementation details – source file, part VII

Class Shader, ShaderPrograms.cpp

```
            getline(fragment_shader_file, aux, '\n');
            _fragment_shader_source += aux + '\n';

            if (logging_is_enabled)
                output << "\t" << aux << endl;
        }

        fragment_shader_file.close();

        if (logging_is_enabled)
            output << endl;

        // 1) creating two empty shader objects
        {
            if (logging_is_enabled)
            {
                output << "Creating empty vertex and fragment shader objects..." << endl;
                output << "----------------------------------------------------" << endl;
            }

            _vertex_shader = glCreateShader(GL_VERTEX_SHADER);
            _fragment_shader = glCreateShader(GL_FRAGMENT_SHADER);

            if (logging_is_enabled)
                output << "Done." << endl << endl;
        }

        // 2) setting the source codes for the shaders
        {
            if (logging_is_enabled)
            {
                output << "Setting the source codes for the shaders..." << endl;
                output << "-------------------------------------------" << endl;
            }

            const GLchar *pointer_to_vertex_shader_source = &_vertex_shader_source[0];
```

Implementation details – source file, part VIII

Class Shader, ShaderPrograms.cpp

```
        glShaderSource(_vertex_shader, 1, &pointer_to_vertex_shader_source, NULL);

        const GLchar *pointer_to_fragment_shader_source = &_fragment_shader_source[0];
        glShaderSource(_fragment_shader, 1, &pointer_to_fragment_shader_source, NULL);

        if (logging_is_enabled)
        {
            // check for OpenGL errors
            _ListOpenGLErrors(__FILE__, __LINE__, output);

            output << "Done." << endl << endl;
        }
    }

    // 3) compiling the vertex shader
    {
        if (logging_is_enabled)
        {
            output << "Compiling the vertex shader..." << endl;
            output << "----------------------------------------" << endl;
        }

        glCompileShader(_vertex_shader);
        glGetShaderiv(_vertex_shader, GL_COMPILE_STATUS, &_vertex_shader_compiled);

        if (logging_is_enabled)
        {
            _ListVertexShaderInfoLog(output);
            output << (_vertex_shader_compiled ? "\tSuccessful." : "\tUnsuccessful.") << endl
                   << "Done." << endl << endl;
        }

        if (!_vertex_shader_compiled)
        {
            glDeleteShader(_vertex_shader);
            return GL_FALSE;
```

Implementation details – source file, part IX

Class Shader, ShaderPrograms.cpp

```
        }
    }

    // 4) compiling the fragment shader
    {
        if (logging_is_enabled)
        {
            output << "Compiling the fragment shader..." << endl;
            output << "--------------------------------" << endl;
        }

        glCompileShader(_fragment_shader);
        glGetShaderiv(_fragment_shader, GL_COMPILE_STATUS, &_fragment_shader_compiled);

        if (logging_is_enabled)
        {
            _ListFragmentShaderInfoLog(output);
            output << (_fragment_shader_compiled ? "\tSuccessful." : "\tUnsuccessful.") << endl
                   << "Done." << endl << endl;
        }

        if (!_fragment_shader_compiled)
        {
            glDeleteShader(_vertex_shader);
            glDeleteShader(_fragment_shader);
            return GL_FALSE;
        }
    }

    // 5) creating the program object
    {
        if (logging_is_enabled)
        {
            output << "Creating the program object..." << endl;
            output << "------------------------------" << endl;
        }
```

Implementation details – source file, part X

Class Shader, ShaderPrograms.cpp

```
        _program = glCreateProgram();

        if (logging_is_enabled)
            output << "Done." << endl << endl;

        // attaching the vertex and fragment shaders to the program object
        if (logging_is_enabled)
            output << "\tAttaching vertex and fragment shaders to the program object..." << endl;

        glAttachShader(_program, _vertex_shader);
        glAttachShader(_program, _fragment_shader);

        // check for OpenGL errors
        if (logging_is_enabled)
        {
            _ListOpenGLErrors(__FILE__, __LINE__, output);
            output << "Done." << endl << endl;
        }

        // linking the program
        if (logging_is_enabled)
            output << "\tLinking the program..." << endl;

        glLinkProgram(_program);
        glGetProgramiv(_program, GL_LINK_STATUS, &_linked);


        if (logging_is_enabled)
        {
            _ListProgramInfoLog(output);
            output << (_linked ? "\tSuccessful." : "\tUnsuccessful.") << endl << "Done." << endl << endl;
        }

        if (!_linked)
        {
```

Implementation details – source file, part XI

Class Shader, ShaderPrograms.cpp

```
                // flag shaders for deletion
                glDeleteShader(_vertex_shader);
                glDeleteShader(_fragment_shader);
                // all the attached shader objects will be automatically detached, and, because they are
                // flagged for deletion, they will be automatically deleted at that time as well
                glDeleteProgram(_program);
                return GL_FALSE;
            }
        }

        // 6) flag shaders for deletion
        {
            if (logging_is_enabled)
            {
                output << "Flag shaders for deletion..." << endl;
                output << "----------------------------------" << endl;
            }

            glDeleteShader(_vertex_shader);
            glDeleteShader(_fragment_shader);

            output << "Done." << endl << endl;
        }

        return GL_TRUE;
}

GLboolean ShaderProgram::SetUniformVariable1i(const GLchar *name, GLint parameter) const
{
        if (!_program)
            return GL_FALSE;

        GLint location = GetUniformVariableLocation(name);
        if (location == -1)
            return GL_FALSE;
```

Implementation details – source file, part XII

Class Shader, ShaderPrograms.cpp

```
    glUniform1i(location, parameter);

    return GL_TRUE;
}

GLboolean ShaderProgram::SetUniformVariable2f(
        const GLchar *name,
        GLfloat parameter_1, GLfloat parameter_2) const
{
    if (!_program)
        return GL_FALSE;

    GLint location = GetUniformVariableLocation(name);
    if (location == -1)
            return GL_FALSE;

    glUniform2f(location, parameter_1, parameter_2);

    return GL_TRUE;
}
GLboolean ShaderProgram::SetUniformVariable3f(
        const GLchar *name, GLfloat parameter_1, GLfloat parameter_2, GLfloat parameter_3) const
{
    if (!_program)
        return GL_FALSE;

    GLint location = GetUniformVariableLocation(name);
    if (location == -1)
            return GL_FALSE;

    glUniform3f(location, parameter_1, parameter_2, parameter_3);

    return GL_TRUE;
}

GLvoid ShaderProgram::Disable() const
```

Implementation details – source file, part XIII

Class Shader, ShaderPrograms.cpp

```
{
    glUseProgram(0);
}

GLvoid ShaderProgram::Enable(GLboolean logging_is_enabled, ostream& output) const
{
    if (_vertex_shader_compiled && _fragment_shader_compiled && _linked)
    {
        glUseProgram(_program);
        glValidateProgram(_program);

        if (logging_is_enabled)
            _ListValidateInfoLog(output);
    }
}

ShaderProgram::~Shader()
{
    // all the attached shader objects will be automatically detached, and, because they are
    // flagged for deletion, they will be automatically deleted at that time as well
    if (_program)
        glDeleteProgram(_program);
}
```

Implementation details – testing shaders, part I

Files GLwidget.h and GLWidget.cpp

```
#include "Exceptions.h"
#include "Core/ShaderPrograms.h"
#include "Core/TriangulatedMeshes3.h"

class GLWidget: public QOpenGLWidget
{
private:

    ...

    cagd::ShaderProgram       _shader;
    cagd::TriangulatedMesh3 _mesh;

    ...
};


void GLWidget::initializeGL()
{
    ...

    if (!_mesh.LoadFromOFF("Models/*.off", GL_TRUE))
    {
        // error
    }

    if (!_mesh.UpdateVertexBufferObjects())
    {
        // error
    }
```

Implementation details – testing shaders, part II

Files GLwidget.h and GLWidget.cpp

```
    try
    {
        // install vertex and fragment shaders; enbale/disable information log output
        if (!_shader.InstallShaders("Shaders/*.vert", "Shaders/*.frag", GL_TRUE))
        {
            // error
        }
    }
    catch (Exception &e)
    {
        cerr << e << endl;
    }

    ...

}


void GLWidget::paintGL()
{
    ...
    glPushMatrix();

        ...
        _shader.Enable(GL_TRUE);
            MatFBBrass.Apply();
            _mesh.Render();
        _shader.Disable();
        ...

    glPopMatrix();
    ...
}
```

Communication with shaders

Variable qualifiers

- The shader has access to part of the OpenGL state, therefore when an application alters this subset of the OpenGL state it is effectively communicating with the shader. E.g. if an application wants to pass a light color to the shader it can simply alter the OpenGL state as it is normally done with the fixed functionality.
- However, using the OpenGL state is not always the most intuitive way of setting values for the shaders to act upon.
- Fortunately, GLSL allows the definition of user defined variables for an OpenGL application to communicate with a shader.
- Qualifiers give a special meaning to the variable. The following qualifiers are available:
  - const – the declaration is of a compile time constant;
  - attribute – global variables that may change per vertex, that are passed from the OpenGL application to vertex shaders (this qualifier can only be used in vertex shaders; for the shader this is a read-only variable).
  - uniform – global variables that may change per primitive (may not be set inside glBegin/glEnd), that are passed from the OpenGL application to the shaders (this qualifier can be used in both vertex and fragment shaders; for the shaders this is a read-only variable).
  - varying – used for interpolated data between a vertex shader and a fragment shader (available for writing in the vertex shader, and read-only in a fragment shader).

- The shader has access to part of the OpenGL state, therefore when an application alters this subset of the OpenGL state it is effectively communicating with the shader. E.g. if an application wants to pass a light color to the shader it can simply alter the OpenGL state as it is normally done with the fixed functionality.
- However, using the OpenGL state is not always the most intuitive way of setting values for the shaders to act upon.
- Fortunately, GLSL allows the definition of user defined variables for an OpenGL application to communicate with a shader.
- Qualifiers give a special meaning to the variable. The following qualifiers are available:
  - const – the declaration is of a compile time constant;
  - attribute – global variables that may change per vertex, that are passed from the OpenGL application to vertex shaders (this qualifier can only be used in vertex shaders; for the shader this is a read-only variable).
  - uniform – global variables that may change per primitive (may not be set inside glBegin/glEnd), that are passed from the OpenGL application to the shaders (this qualifier can be used in both vertex and fragment shaders; for the shaders this is a read-only variable).
  - varying – used for interpolated data between a vertex shader and a fragment shader (available for writing in the vertex shader, and read-only in a fragment shader).

- The shader has access to part of the OpenGL state, therefore when an application alters this subset of the OpenGL state it is effectively communicating with the shader. E.g. if an application wants to pass a light color to the shader it can simply alter the OpenGL state as it is normally done with the fixed functionality.
- However, using the OpenGL state is not always the most intuitive way of setting values for the shaders to act upon.
- Fortunately, GLSL allows the definition of user defined variables for an OpenGL application to communicate with a shader.
- Qualifiers give a special meaning to the variable. The following qualifiers are available:
  - const – the declaration is of a compile time constant;
  - attribute – global variables that may change per vertex, that are passed from the OpenGL application to vertex shaders (this qualifier can only be used in vertex shaders; for the shader this is a read-only variable).
  - uniform – global variables that may change per primitive (may not be set inside glBegin/glEnd), that are passed from the OpenGL application to the shaders (this qualifier can be used in both vertex and fragment shaders; for the shaders this is a read-only variable).
  - varying – used for interpolated data between a vertex shader and a fragment shader (available for writing in the vertex shader, and read-only in a fragment shader).

- The shader has access to part of the OpenGL state, therefore when an application alters this subset of the OpenGL state it is effectively communicating with the shader. E.g. if an application wants to pass a light color to the shader it can simply alter the OpenGL state as it is normally done with the fixed functionality.
- However, using the OpenGL state is not always the most intuitive way of setting values for the shaders to act upon.
- Fortunately, GLSL allows the definition of user defined variables for an OpenGL application to communicate with a shader.
- Qualifiers give a special meaning to the variable. The following qualifiers are available:
  - const – the declaration is of a compile time constant;
  - attribute – global variables that may change per vertex, that are passed from the OpenGL application to vertex shaders (this qualifier can only be used in vertex shaders; for the shader this is a read-only variable).
  - uniform – global variables that may change per primitive (may not be set inside glBegin/glEnd), that are passed from the OpenGL application to the shaders (this qualifier can be used in both vertex and fragment shaders; for the shaders this is a read-only variable).
  - varying – used for interpolated data between a vertex shader and a fragment shader (available for writing in the vertex shader, and read-only in a fragment shader).

- The shader has access to part of the OpenGL state, therefore when an application alters this subset of the OpenGL state it is effectively communicating with the shader. E.g. if an application wants to pass a light color to the shader it can simply alter the OpenGL state as it is normally done with the fixed functionality.
- However, using the OpenGL state is not always the most intuitive way of setting values for the shaders to act upon.
- Fortunately, GLSL allows the definition of user defined variables for an OpenGL application to communicate with a shader.
- Qualifiers give a special meaning to the variable. The following qualifiers are available:
  - const – the declaration is of a compile time constant;
  - attribute – global variables that may change per vertex, that are passed from the OpenGL application to vertex shaders (this qualifier can only be used in vertex shaders; for the shader this is a read-only variable).
  - uniform – global variables that may change per primitive (may not be set inside glBegin/glEnd), that are passed from the OpenGL application to the shaders (this qualifier can be used in both vertex and fragment shaders; for the shaders this is a read-only variable).
  - varying – used for interpolated data between a vertex shader and a fragment shader (available for writing in the vertex shader, and read-only in a fragment shader).

- The shader has access to part of the OpenGL state, therefore when an application alters this subset of the OpenGL state it is effectively communicating with the shader. E.g. if an application wants to pass a light color to the shader it can simply alter the OpenGL state as it is normally done with the fixed functionality.
- However, using the OpenGL state is not always the most intuitive way of setting values for the shaders to act upon.
- Fortunately, GLSL allows the definition of user defined variables for an OpenGL application to communicate with a shader.
- Qualifiers give a special meaning to the variable. The following qualifiers are available:
  - const – the declaration is of a compile time constant;
  - attribute – global variables that may change per vertex, that are passed from the OpenGL application to vertex shaders (this qualifier can only be used in vertex shaders; for the shader this is a read-only variable).
  - uniform – global variables that may change per primitive (may not be set inside glBegin/glEnd), that are passed from the OpenGL application to the shaders (this qualifier can be used in both vertex and fragment shaders; for the shaders this is a read-only variable).
  - varying – used for interpolated data between a vertex shader and a fragment shader (available for writing in the vertex shader, and read-only in a fragment shader).

- The shader has access to part of the OpenGL state, therefore when an application alters this subset of the OpenGL state it is effectively communicating with the shader. E.g. if an application wants to pass a light color to the shader it can simply alter the OpenGL state as it is normally done with the fixed functionality.
- However, using the OpenGL state is not always the most intuitive way of setting values for the shaders to act upon.
- Fortunately, GLSL allows the definition of user defined variables for an OpenGL application to communicate with a shader.
- Qualifiers give a special meaning to the variable. The following qualifiers are available:
    - const – the declaration is of a compile time constant;
    - attribute – global variables that may change per vertex, that are passed from the OpenGL application to vertex shaders (this qualifier can only be used in vertex shaders; for the shader this is a read-only variable).
    - uniform – global variables that may change per primitive (may not be set inside glBegin/glEnd), that are passed from the OpenGL application to the shaders (this qualifier can be used in both vertex and fragment shaders; for the shaders this is a read-only variable).
    - varying – used for interpolated data between a vertex shader and a fragment shader (available for writing in the vertex shader, and read-only in a fragment shader).

- The shader has access to part of the OpenGL state, therefore when an application alters this subset of the OpenGL state it is effectively communicating with the shader. E.g. if an application wants to pass a light color to the shader it can simply alter the OpenGL state as it is normally done with the fixed functionality.
- However, using the OpenGL state is not always the most intuitive way of setting values for the shaders to act upon.
- Fortunately, GLSL allows the definition of user defined variables for an OpenGL application to communicate with a shader.
- Qualifiers give a special meaning to the variable. The following qualifiers are available:
  - const – the declaration is of a compile time constant;
  - attribute – global variables that may change per vertex, that are passed from the OpenGL application to vertex shaders (this qualifier can only be used in vertex shaders; for the shader this is a read-only variable).
  - uniform – global variables that may change per primitive (may not be set inside glBegin/glEnd), that are passed from the OpenGL application to the shaders (this qualifier can be used in both vertex and fragment shaders; for the shaders this is a read-only variable).
  - varying – used for interpolated data between a vertex shader and a fragment shader (available for writing in the vertex shader, and read-only in a fragment shader).

Uniform variables

- A uniform variable can have its value changed by primitive only, i.e. its value cannot be changed between a glBegin/glEnd pair. This implies that it cannot be used for vertex attributes.
- Uniform variables are suitable for values that remain constant along a primitive, frame, or even the whole scene.
- Uniform variables can be read (but not written) in both vertex and fragment shaders.
- To set a uniform variable, the first thing you have to do is to get the memory location of the variable.
  - Note that this information is only available after you link the program.
  - Moreover, you may be required to use the program, i.e. you will have to call glUseProgram or glUseProgramObjectARB before attempting to get the location.

Uniform variables

- A uniform variable can have its value changed by primitive only, i.e. its value cannot be changed between a glBegin/glEnd pair. This implies that it cannot be used for vertex attributes.
- Uniform variables are suitable for values that remain constant along a primitive, frame, or even the whole scene.
- Uniform variables can be read (but not written) in both vertex and fragment shaders.
- To set a uniform variable, the first thing you have to do is to get the memory location of the variable.
  - Note that this information is only available after you link the program.
  - Moreover, you may be required to use the program, i.e. you will have to call glUseProgram or glUseProgramObjectARB before attempting to get the location.

Uniform variables

- A uniform variable can have its value changed by primitive only, i.e. its value cannot be changed between a glBegin/glEnd pair. This implies that it cannot be used for vertex attributes.
- Uniform variables are suitable for values that remain constant along a primitive, frame, or even the whole scene.
- Uniform variables can be read (but not written) in both vertex and fragment shaders.
- To set a uniform variable, the first thing you have to do is to get the memory location of the variable.
  - Note that this information is only available after you link the program.
  - Moreover, you may be required to use the program, i.e. you will have to call glUseProgram or glUseProgramObjectARB before attempting to get the location.

Uniform variables

- A uniform variable can have its value changed by primitive only, i.e. its value cannot be changed between a glBegin/glEnd pair. This implies that it cannot be used for vertex attributes.
- Uniform variables are suitable for values that remain constant along a primitive, frame, or even the whole scene.
- Uniform variables can be read (but not written) in both vertex and fragment shaders.
- To set a uniform variable, the first thing you have to do is to get the memory location of the variable.
  - Note that this information is only available after you link the program.
  - Moreover, you may be required to use the program, i.e. you will have to call glUseProgram or glUseProgramObjectARB before attempting to get the location.

Uniform variables

- A uniform variable can have its value changed by primitive only, i.e. its value cannot be changed between a glBegin/glEnd pair. This implies that it cannot be used for vertex attributes.
- Uniform variables are suitable for values that remain constant along a primitive, frame, or even the whole scene.
- Uniform variables can be read (but not written) in both vertex and fragment shaders.
- To set a uniform variable, the first thing you have to do is to get the memory location of the variable.
  - Note that this information is only available after you link the program.
  - Moreover, you may be required to use the program, i.e. you will have to call glUseProgram or glUseProgramObjectARB before attempting to get the location.

Uniform variables

- A uniform variable can have its value changed by primitive only, i.e. its value cannot be changed between a glBegin/glEnd pair. This implies that it cannot be used for vertex attributes.
- Uniform variables are suitable for values that remain constant along a primitive, frame, or even the whole scene.
- Uniform variables can be read (but not written) in both vertex and fragment shaders.
- To set a uniform variable, the first thing you have to do is to get the memory location of the variable.
  - Note that this information is only available after you link the program.
  - Moreover, you may be required to use the program, i.e. you will have to call glUseProgram or glUseProgramObjectARB before attempting to get the location.

Communication with shaders

Uniform variables. Description of command glGetUniformLocation

```
GLint glGetUniformLocation(GLuint program, const GLchar *name);

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

   name must be a null terminated string that contains no white space.

   name must be an active uniform variable name in program that is not a structure,
   an array of structures, or a subcomponent of a vector or a matrix.

   This function returns -1 if name does not correspond to an active uniform variable in
   program or if name starts with the reserved prefix "gl_".

   Uniform variables that are structures or arrays of structures may be queried by calling
   glGetUniformLocation for each field within the structure. The array element operator "[]"
   and the structure field operator "." may be used in name in order to select elements within
   an array or fields within a structure.

   The result of using these operators is not allowed to be another structure, an array of
   structures, or a subcomponent of a vector or a matrix. Except if the last part of name
   indicates a uniform variable array, the location of the first element of an array
   can be retrieved by using the name of the array, or by using the name appended by "[0]".

   The actual locations assigned to uniform variables are not known until the program object
   is linked successfully. After linking has occurred, the command glGetUniformLocation can be
   used to obtain the location of a uniform variable.

   This location value can then be passed to glUniform to set the value of the uniform variable
   or to glGetUniform in order to query the current value of the uniform variable.

   After a program object has been linked successfully, the index values for uniform variables
   remain fixed until the next link command occurs. Uniform variable locations and values can only
   be queried after a link if the link was successful.
*/
```

Communication with shaders, part I

Uniform variables. Description of command glUniform

```
void glUniform1f(GLint location, GLfloat v0);
void glUniform2f(GLint location, GLfloat v0, GLfloat v1);
void glUniform3f(GLint location, GLfloat v0, GLfloat v1, GLfloat v2);
void glUniform4f(GLint location, GLfloat v0, GLfloat v1, GLfloat v2, GLfloat v3);

void glUniform1i(GLint location, GLint v0);
void glUniform2i(GLint location, GLint v0, GLint v1);
void glUniform3i(GLint location, GLint v0, GLint v1, GLint v2);
void glUniform4i(GLint location, GLint v0, GLint v1, GLint v2, GLint v3);

void glUniform1fv(GLint location, GLsizei count, const GLfloat *value);
void glUniform2fv(GLint location, GLsizei count, const GLfloat *value);
void glUniform3fv(GLint location, GLsizei count, const GLfloat *value);
void glUniform4fv(GLint location, GLsizei count, const GLfloat *value);

void glUniform1iv(GLint location, GLsizei count, const GLint *value);
void glUniform2iv(GLint location, GLsizei count, const GLint *value);
void glUniform3iv(GLint location, GLsizei count, const GLint *value);
void glUniform4iv(GLint location, GLsizei count, const GLint *value);

void glUniformMatrix2fv  (GLint location, GLsizei count, GLboolean transpose, const GLfloat *value);
void glUniformMatrix3fv  (GLint location, GLsizei count, GLboolean transpose, const GLfloat *value);
void glUniformMatrix4fv  (GLint location, GLsizei count, GLboolean transpose, const GLfloat *value);
void glUniformMatrix2x3fv(GLint location, GLsizei count, GLboolean transpose, const GLfloat *value);
void glUniformMatrix3x2fv(GLint location, GLsizei count, GLboolean transpose, const GLfloat *value);
void glUniformMatrix2x4fv(GLint location, GLsizei count, GLboolean transpose, const GLfloat *value);
void glUniformMatrix4x2fv(GLint location, GLsizei count, GLboolean transpose, const GLfloat *value);
void glUniformMatrix3x4fv(GLint location, GLsizei count, GLboolean transpose, const GLfloat *value);
void glUniformMatrix4x3fv(GLint location, GLsizei count, GLboolean transpose, const GLfloat *value);

/*
    glUniform modifies the value of a uniform variable or a uniform variable array. The location of
    the uniform variable to be modified is specified by location, which should be a value returned
    by glGetUniformLocation. glUniform operates on the program object that was made part of current
    state by calling glUseProgram.
```

Communication with shaders, part II

Uniform variables. Description of command glUniform

The commands glUniform{1|2|3|4}{f|i} are used to change the value of the uniform variable specified by location using the values passed as arguments. The number specified in the command should match the number of components in the data type of the specified uniform variable (e.g. 1 for float, int, bool; 2 for vec2, ivec2, bvec2, etc.). The suffix f indicates that floating−point values are being passed; the suffix i indicates that integer values are being passed, and this type should also match the data type of the specified uniform variable. The i variants of this function should be used to provide values for uniform variables defined as int, ivec2, ivec3, ivec4, or arrays of these. The f variants should be used to provide values for uniform variables of type float, vec2, vec3, vec4, or arrays of these. Either the i or the f variants may be used to provide values for uniform variables of type bool, bvec2, bvec3, bvec4, or arrays of these. The uniform variable will be set to false if the input value is 0 or 0.0f, and it will be set to true otherwise.

All active uniform variables defined in a program object are initialized to 0 when the program object is linked successfully. They retain the values assigned to them by a call to glUniform until the next successful link operation occurs on the program object, when they are once again initialized to 0.

The commands glUniform{1|2|3|4}{f|i}v can be used to modify a single uniform variable or a uniform variable array. These commands pass a count and a pointer to the values to be loaded into a uniform variable or a uniform variable array. A count of 1 should be used if modifying the value of a single uniform variable, and a count of 1 or greater can be used to modify an entire array or part of an array. When loading n elements starting at an arbitrary position m in a uniform variable array, elements $m + n - 1$ in the array will be replaced with the new values. If $m + n - 1$ is larger than the size of the uniform variable array, values for all array elements beyond the end of the array will be ignored. The number specified in the name of the command indicates the number of components for each element in value, and it should match the number of components in the data type of the specified uniform variable (e.g. 1 for float, int, bool; 2 for vec2, ivec2, bvec2, etc.). The data type specified in the name of the command must match the data type for the specified uniform variable as described previously for glUniform{1|2|3|4}{f|i}.

For uniform variable arrays, each element of the array is considered to be of the type indicated in the name of the command (e.g. glUniform3f or glUniform3fv can be used to load a uniform variable array of type vec3). The number of elements of the uniform variable array to be modified

```
    is specified by count.

    The commands glUniformMatrix{2|3|4|2x3|3x2|2x4|4x2|3x4|4x3}fv are used to modify a matrix or an
    array of matrices. The numbers in the command name are interpreted as the dimensionality of the
    matrix. The number 2 indicates a 2x2 matrix (i.e. 4 values), the number 3 indicates a 3 x 3
    matrix (i.e. 9 values), and the number 4 indicates a 4 x 4 matrix (i.e. 16 values). Non-square
    matrix dimensionality is explicit, with the first number representing the number of columns and
    the second number representing the number of rows.  For example, 2x4 indicates a 2 x 4 matrix
    with 2 columns and 4 rows (i.e. 8 values). If transpose is GL_FALSE, each matrix is assumed to be
    supplied in column major order. If transpose is GL_TRUE, each matrix is assumed to be supplied
    in row major order. The count argument indicates the number of matrices to be passed. A count of
    1 should be used if modifying the value of a single matrix, and a count greater than 1 can be
    used to modify an array of matrices.
*/
```

Communication with shaders, part I

Uniform variables. Example: using commands glGetUniformLocation, and glUniform

```
// assume that a shader with the following variables is being used
uniform float   specular_intensity;
uniform vec4    specular_color;
uniform float   threshold[2];
uniform vec4    colors[3];


// in an OpenGL 2.0 application, the code for setting these variables could be
GLuint program;

...

GLint linked = 0;
if (program)
{
    glLinkProgram(program);
    glGetProgramiv(program, GL_LINK_STATUS, &linked);
}

...

if (linked)
{
    float my_specular_intensity    = 0.98;
    float my_specular_color[4]     = {0.8, 0.8, 0.8, 1.0};
    float my_threshold[2]          = {0.5, 0.25};
    float my_colors[12]            = {0.4, 0.4, 0.8, 1.0,
                                      0.2, 0.2, 0.4, 1.0,
                                      0.1, 0.1, 0.1, 1.0};

    GLint location = -1;

    location = glGetUniformLocation(program, "specular_intensity");

    if (location > 0)
```

```
        glUniform1f(location, my_specular_intensity);

    location = glGetUniformLocation(program, "specular_color");
    if (location > 0)
        glUniform4fv(location, 1, my_specular_color);

    location = glGetUniformLocation(program, "threshold");
    if (location > 0)
        glUniform1fv(location, 2, my_threshold);

    location = glGetUniformLocation(program, "colors");
    if (location > 0)
        glUniform4fv(location, 3, my_colors);
}

...
```

Attribute variables

- As mentioned before, uniform variables can only be set by primitive, i.e. they cannot be set inside a glBegin/glEnd block.
- If it is required to set variables per vertex, then attribute variables must be used.
- In fact attribute variables can be updated at any time.
- Attribute variables can only be read (not written) in a vertex shader. This is because they contain vertex data, hence not applicable directly in a fragment shader.
- As for uniform variables, first it is necessary to get the location in memory of the attribute variable.
  - Note that the program must be linked previously and some drivers may require that the program is in use.

Attribute variables

- As mentioned before, uniform variables can only be set by primitive, i.e. they cannot be set inside a glBegin/glEnd block.
- If it is required to set variables per vertex, then attribute variables must be used.
- In fact attribute variables can be updated at any time.
- Attribute variables can only be read (not written) in a vertex shader. This is because they contain vertex data, hence not applicable directly in a fragment shader.
- As for uniform variables, first it is necessary to get the location in memory of the attribute variable.
  - Note that the program must be linked previously and some drivers may require that the program is in use.

Attribute variables

- As mentioned before, uniform variables can only be set by primitive, i.e. they cannot be set inside a glBegin/glEnd block.
- If it is required to set variables per vertex, then attribute variables must be used.
- In fact attribute variables can be updated at any time.
- Attribute variables can only be read (not written) in a vertex shader. This is because they contain vertex data, hence not applicable directly in a fragment shader.
- As for uniform variables, first it is necessary to get the location in memory of the attribute variable.
  - Note that the program must be linked previously and some drivers may require that the program is in use.

Attribute variables

- As mentioned before, uniform variables can only be set by primitive, i.e. they cannot be set inside a glBegin/glEnd block.
- If it is required to set variables per vertex, then attribute variables must be used.
- In fact attribute variables can be updated at any time.
- Attribute variables can only be read (not written) in a vertex shader. This is because they contain vertex data, hence not applicable directly in a fragment shader.
- As for uniform variables, first it is necessary to get the location in memory of the attribute variable.
    - Note that the program must be linked previously and some drivers may require that the program is in use.

Attribute variables

- As mentioned before, uniform variables can only be set by primitive, i.e. they cannot be set inside a glBegin/glEnd block.
- If it is required to set variables per vertex, then attribute variables must be used.
- In fact attribute variables can be updated at any time.
- Attribute variables can only be read (not written) in a vertex shader. This is because they contain vertex data, hence not applicable directly in a fragment shader.
- As for uniform variables, first it is necessary to get the location in memory of the attribute variable.
    - Note that the program must be linked previously and some drivers may require that the program is in use.

Attribute variables

- As mentioned before, uniform variables can only be set by primitive, i.e. they cannot be set inside a glBegin/glEnd block.
- If it is required to set variables per vertex, then attribute variables must be used.
- In fact attribute variables can be updated at any time.
- Attribute variables can only be read (not written) in a vertex shader. This is because they contain vertex data, hence not applicable directly in a fragment shader.
- As for uniform variables, first it is necessary to get the location in memory of the attribute variable.
  - Note that the program must be linked previously and some drivers may require that the program is in use.

Communication with shaders

Attribute variables. Description of command glGetAttribLocation

```
GLint glGetAttribLocation(GLuint program, const GLchar *name);

/*
   glGetAttribLocation queries the previously linked program object specified by
   program for the attribute variable specified by name and returns the index
   of the generic vertex attribute that is bound to that attribute variable.

   If name is a matrix attribute variable, the index of the first column of the
   matrix is returned.

   If the named attribute variable is not an active attribute in the specified
   program object or if name starts with the reserved prefix "gl_", a value of -1
   is returned.

   The association between an attribute variable name and a generic attribute index
   can be specified at any time by calling glBindAttribLocation.

   Attribute bindings do not go into effect until glLinkProgram is called. After a
   program object has been linked successfully, the index values for attribute variables
   remain fixed until the next link command occurs. The attribute values can only be
   queried after a link if the link was successful.

   glGetAttribLocation returns the binding that actually went into effect the last time
   glLinkProgram was called for the specified program object. Attribute bindings
   that have been specified since the last link operation are not returned by
   glGetAttribLocation.
*/
```

Communication with shaders, part I

Attribute variables. Description of command glVertexAttrib

```
GLvoid glVertexAttrib1f  (GLuint index, GLfloat  v0);
GLvoid glVertexAttrib1s  (GLuint index, GLshort  v0);
GLvoid glVertexAttrib1d  (GLuint index, GLdouble v0);

GLvoid glVertexAttrib2f  (GLuint index, GLfloat  v0, GLfloat  v1);
GLvoid glVertexAttrib2s  (GLuint index, GLshort  v0, GLshort  v1);
GLvoid glVertexAttrib2d  (GLuint index, GLdouble v0, GLdouble v1);

GLvoid glVertexAttrib3f  (GLuint index, GLfloat  v0, GLfloat  v1, GLfloat  v2);
GLvoid glVertexAttrib3s  (GLuint index, GLshort  v0, GLshort  v1, GLshort  v2);
GLvoid glVertexAttrib3d  (GLuint index, GLdouble v0, GLdouble v1, GLdouble v2);

GLvoid glVertexAttrib4f  (GLuint index, GLfloat  v0, GLfloat  v1, GLfloat  v2, GLfloat  v3);
GLvoid glVertexAttrib4s  (GLuint index, GLshort  v0, GLshort  v1, GLshort  v2, GLshort  v3);
GLvoid glVertexAttrib4d  (GLuint index, GLdouble v0, GLdouble v1, GLdouble v2, GLdouble v3);
GLvoid glVertexAttrib4Nub(GLuint index, GLubyte  v0, GLubyte  v1, GLubyte  v2, GLubyte  v3);

GLvoid glVertexAttrib1fv  (GLuint index, const GLfloat  *v);
GLvoid glVertexAttrib1sv  (GLuint index, const GLshort  *v);
GLvoid glVertexAttrib1dv  (GLuint index, const GLdouble *v);

GLvoid glVertexAttrib2fv  (GLuint index, const GLfloat  *v);
GLvoid glVertexAttrib2sv  (GLuint index, const GLshort  *v);
GLvoid glVertexAttrib2dv  (GLuint index, const GLdouble *v);

GLvoid glVertexAttrib3fv  (GLuint index, const GLfloat  *v);
GLvoid glVertexAttrib3sv  (GLuint index, const GLshort  *v);
GLvoid glVertexAttrib3dv  (GLuint index, const GLdouble *v);

GLvoid glVertexAttrib4fv  (GLuint index, const GLfloat  *v);
GLvoid glVertexAttrib4sv  (GLuint index, const GLshort  *v);
GLvoid glVertexAttrib4dv  (GLuint index, const GLdouble *v);
GLvoid glVertexAttrib4iv  (GLuint index, const GLint    *v);
GLvoid glVertexAttrib4bv  (GLuint index, const GLbyte   *v);
```

```
GLvoid glVertexAttrib4ubv (GLuint index, const GLubyte  *v);
GLvoid glVertexAttrib4usv (GLuint index, const GLushort *v);
GLvoid glVertexAttrib4uiv (GLuint index, const GLuint   *v);

GLvoid glVertexAttrib4Nbv (GLuint index, const GLbyte   *v);
GLvoid glVertexAttrib4Nsv (GLuint index, const GLshort  *v);
GLvoid glVertexAttrib4Niv (GLuint index, const GLint    *v);

GLvoid glVertexAttrib4Nubv(GLuint index, const GLubyte  *v);
GLvoid glVertexAttrib4Nusv(GLuint index, const GLushort *v);
GLvoid glVertexAttrib4Nuiv(GLuint index, const GLuint   *v);

/*  OpenGL defines a number of standard vertex attributes that applications can modify with
    standard API entry points (color, normal, texture coordinates, etc.). The glVertexAttrib
    family of entry points allows an application to pass generic vertex attributes in numbered
    locations.

    Generic attributes are defined as four-component values that are organized into an array.
    The first entry of this array is numbered 0, and the size of the array is specified by the
    implementation-dependent constant GL_MAX_VERTEX_ATTRIBS. Individual elements of this array
    can be modified with a glVertexAttrib call that specifies the index of the element to be
    modified and a value for that element.

    These commands can be used to specify one, two, three, or all four components of the generic
    vertex attribute specified by index. A 1 in the name of the command indicates that only one
    value is passed, and it will be used to modify the first component of the generic
    vertex attribute. The second and third components will be set to 0, and the fourth component
    will be set to 1. Similarly, a 2 in the name of the command indicates that values are provided
    for the first two components, the third component will be set to 0, and the fourth component
    will be set to 1. A 3 in the name of the command indicates that values are provided for the
    first three components and the fourth component will be set to 1, whereas a 4 in the name
    indicates that values are provided for all four components.

    The letters s, f, i, d, ub, us, and ui indicate whether the arguments are of type short, float,
    int, double, unsigned byte, unsigned short, or unsigned int. When v is appended to the name,
    the commands can take a pointer to an array of such values. The commands containing N indicate
```

```
that the arguments will be passed as fixed-point values that are scaled to a normalized range
according to the component conversion rules defined by the OpenGL specification. Signed values
are understood to represent fixed-point values in the range [-1,1], and unsigned values are
understood to represent fixed-point values in the range [0,1].

OpenGL Shading Language attribute variables are allowed to be of type mat2, mat3, or mat4.
Attributes of these types may be loaded using the glVertexAttrib entry points. Matrices must
be loaded into successive generic attribute slots in column major order, with one column of
the matrix in each generic attribute slot.

A user-defined attribute variable declared in a vertex shader can be bound to a generic
attribute index by calling glBindAttribLocation. This allows an application to use more
descriptive variable names in a vertex shader. A subsequent change to the specified
generic vertex attribute will be immediately reflected as a change to the corresponding attribute
variable in the vertex shader.

The binding between a generic vertex attribute index and a user-defined attribute variable in
a vertex shader is part of the state of a program object, but the current value of the generic
vertex attribute is not. The value of each generic vertex attribute is part of current state,
just like standard vertex attributes, and it is maintained even if a different program object
is used.

An application may freely modify generic vertex attributes that are not bound to a named vertex
shader attribute variable. These values are simply maintained as part of current state and
will not be accessed by the vertex shader. If a generic vertex attribute bound to an attribute
variable in a vertex shader is not updated while the vertex shader is executing, the vertex
shader will repeatedly use the current value for the generic vertex attribute.

The generic vertex attribute with index 0 is the same as the vertex position attribute previously
defined by OpenGL. A glVertex2, glVertex3, or glVertex4 command is completely equivalent to the
corresponding glVertexAttrib command with an index argument of 0. A vertex shader can access
generic vertex attribute 0 by using the built-in attribute variable gl_Vertex.
There are no current values for generic vertex attribute 0. This is the only generic vertex
attribute with this property; calls to set other standard vertex attributes can be freely mixed
with calls to set any of the other generic vertex attributes.*/
```

Communication with shaders, part I

Attribute variables. Example: using commands glGetAttribLocation, and glVertexAttrib

```
// it is assumed that the vertex shader declares a float attribute named parameter
attribute float parameter;

// in an OpenGL 2.0+ application, the code for setting this attribute variable could be
GLuint program;

...

GLint linked = 0;
if (program)
{
    glLinkProgram(program);
    glGetProgramiv(program, GL_LINK_STATUS, &linked);
}

...

if (linked)
{
    GLint location = -1;

    location = glGetAttribLocation(program, "parameter");

    ...

    glBegin(...);

        ...

        glVertexAttrib1f(location, 1.0);
        glVertex3f(-1.0, 1.0, 0.0);

        glVertexAttrib1f(location, 3.0);
        glVertex3f( 1.0, 1.0, 0.0);
```

```
        glVertexAttrib1f(location ,  -2.0);
        glVertex3f(-1.0,  -1.0,   1.0);

        glVertexAttrib1f(location ,  -1.0);
        glVertex3f( 1.0,  -1.0,  -1.0);

        ...

    glEnd();

    ...

}

...
```

Attribute variables and vertex arrays

- Vertex arrays can also be used together with attribute variables. For this you will need to enable/disable attribute arrays.

```
GLvoid glEnableVertexAttribArray (GLuint index);
GLvoid glDisableVertexAttribArray(GLuint index);

/*
    glEnableVertexAttribArray enables the generic vertex attribute array
    specified by index.

    glDisableVertexAttribArray disables the generic vertex attribute array
    specified by index.

    By default, all client-side capabilities are disabled, including all generic vertex
    attribute arrays. If enabled, the values in the generic vertex attribute array will
    be accessed and used for rendering when calls are made to vertex array commands such as
    glDrawArrays, glDrawElements, glDrawRangeElements, glArrayElement, glMultiDrawElements,
    or glMultiDrawArrays.
*/
```

Attribute variables and vertex arrays – continued

- After you have enabled the vertex attributes array, you need to provide the pointer to the array with the data.

```
GLvoid glVertexAttribPointer(GLuint index, GLint size, GLenum type, GLboolean normalized,
                             GLsizei stride, const GLvoid *pointer);

/*
   glVertexAttribPointer specifies the location and data format of the array of generic
   vertex attributes at index index to use when rendering.

   size specifies the number of components per attribute and must be 1, 2, 3, or 4.

   type specifies the data type of each component, and stride specifies the byte stride from
   one attribute to the next, allowing vertices and attributes to be packed into a single array
   or stored in separate arrays.

   If set to GL_TRUE, normalized indicates that values stored in an integer format are to be
   mapped to the range [-1,1] (for signed values) or [0,1] (for unsigned values) when they are
   accessed and converted to floating point. Otherwise, values will be converted to floats
   directly without normalization.

   If a non-zero named buffer object is bound to the GL_ARRAY_BUFFER target
   (see glBindBuffer) while a generic vertex attribute array is specified, pointer is treated as
   a byte offset into the buffer object's data store.

   Also, the buffer object binding (GL_ARRAY_BUFFER_BINDING) is saved as generic vertex
   attribute array client-side state (GL_VERTEX_ATTRIB_ARRAY_BUFFER_BINDING) for index index.

   When a generic vertex attribute array is specified, size, type, normalized, stride, and
   pointer are saved as client-side state, in addition to the current vertex array buffer object
   binding.
*/
```

```
// it is assumed that the vertex shader declares a float attribute named parameter
attribute float parameter;

// in an OpenGL 2.0+ application, the code for setting this attribute variable could be
GLuint program;

...

GLint linked = 0;
if (program)
{
    glLinkProgram(program);
    glGetProgramiv(program, GL_LINK_STATUS, &linked);
}

...

if (linked)
{
    GLint location = -1;

    location = glGetAttribLocation(program, "parameter");

    ...

    float vertices[12] = {-1.0,  1.0,  0.0,
                           1.0,  1.0,  0.0,
                          -1.0, -1.0,  1.0,
                           1.0, -1.0, -1.0};

    float parameters[4] = {1.0, 3.0, -2.0, -1.0};

    ...
```

Communication with shaders, part II

Attribute variables. Example: using commands glEnableVertexAttribArray, and glVertexAttribPointer

```
    glEnableClientState(GL_VERTEX_ARRAY);
    glEnableVertexAttribArray(location);

    ...

    glVertexPointer(3, GL_FLOAT, 0, vertices);
    glVertexAttribPointer(location, 1, GL_FLOAT, 0, 0, parameters);

    ...

}

...
```

Varying variables

- In order to compute values per fragment it is often required to access vertex interpolated data. For instance, when performing lighting computation per fragment, we need to access the normal at the fragment. However in OpenGL, the normals are only specified per vertex. These normals are accessible to the vertex shader, but not to the fragment shader since they come from the OpenGL application as an attribute variable.
- After the vertices, including all the vertex data, are processed they move on to the next stage of the pipeline (which still remains fixed functionality) where connectivity information is available. It is in this stage that the primitives are assembled and fragments computed. For each fragment there is a set of variables that are interpolated automatically and provided to the fragment shader. An example is the color of the fragment. The color that arrives at the fragment shader is the result of the interpolation of the colors of the vertices that make up the primitive.
- This type of variables, where the fragment receives interpolated, data are varying variables. GLSL has some predefined varying variables, such as the above mentioned color. GLSL also allows user defined varying variables. These must be declared in both the vertex and fragment shaders, for instance:

```
varying float intensity;
```

- A varying variable must be written on a vertex shader, where we compute the value of the variable for each vertex. In the fragment shader the variable, whose value results from an interpolation of the vertex values computed previously, can only be read.

Varying variables

- In order to compute values per fragment it is often required to access vertex interpolated data. For instance, when performing lighting computation per fragment, we need to access the normal at the fragment. However in OpenGL, the normals are only specified per vertex. These normals are accessible to the vertex shader, but not to the fragment shader since they come from the OpenGL application as an attribute variable.
- After the vertices, including all the vertex data, are processed they move on to the next stage of the pipeline (which still remains fixed functionality) where connectivity information is available. It is in this stage that the primitives are assembled and fragments computed. For each fragment there is a set of variables that are interpolated automatically and provided to the fragment shader. An example is the color of the fragment. The color that arrives at the fragment shader is the result of the interpolation of the colors of the vertices that make up the primitive.
- This type of variables, where the fragment receives interpolated, data are varying variables. GLSL has some predefined varying variables, such as the above mentioned color. GLSL also allows user defined varying variables. These must be declared in both the vertex and fragment shaders, for instance:

```
varying float intensity;
```

- A varying variable must be written on a vertex shader, where we compute the value of the variable for each vertex. In the fragment shader the variable, whose value results from an interpolation of the vertex values computed previously, can only be read.

Varying variables

- In order to compute values per fragment it is often required to access vertex interpolated data. For instance, when performing lighting computation per fragment, we need to access the normal at the fragment. However in OpenGL, the normals are only specified per vertex. These normals are accessible to the vertex shader, but not to the fragment shader since they come from the OpenGL application as an attribute variable.
- After the vertices, including all the vertex data, are processed they move on to the next stage of the pipeline (which still remains fixed functionality) where connectivity information is available. It is in this stage that the primitives are assembled and fragments computed. For each fragment there is a set of variables that are interpolated automatically and provided to the fragment shader. An example is the color of the fragment. The color that arrives at the fragment shader is the result of the interpolation of the colors of the vertices that make up the primitive.
- This type of variables, where the fragment receives interpolated, data are varying variables. GLSL has some predefined varying variables, such as the above mentioned color. GLSL also allows user defined varying variables. These must be declared in both the vertex and fragment shaders, for instance:

```
varying float intensity;
```

- A varying variable must be written on a vertex shader, where we compute the value of the variable for each vertex. In the fragment shader the variable, whose value results from an interpolation of the vertex values computed previously, can only be read.

Varying variables

- In order to compute values per fragment it is often required to access vertex interpolated data. For instance, when performing lighting computation per fragment, we need to access the normal at the fragment. However in OpenGL, the normals are only specified per vertex. These normals are accessible to the vertex shader, but not to the fragment shader since they come from the OpenGL application as an attribute variable.
- After the vertices, including all the vertex data, are processed they move on to the next stage of the pipeline (which still remains fixed functionality) where connectivity information is available. It is in this stage that the primitives are assembled and fragments computed. For each fragment there is a set of variables that are interpolated automatically and provided to the fragment shader. An example is the color of the fragment. The color that arrives at the fragment shader is the result of the interpolation of the colors of the vertices that make up the primitive.
- This type of variables, where the fragment receives interpolated, data are varying variables. GLSL has some predefined varying variables, such as the above mentioned color. GLSL also allows user defined varying variables. These must be declared in both the vertex and fragment shaders, for instance:

```
varying float intensity;
```

- A varying variable must be written on a vertex shader, where we compute the value of the variable for each vertex. In the fragment shader the variable, whose value results from an interpolation of the vertex values computed previously, can only be read.

Directional light

Vertex shader: directional_light.vert

```
/*
It is assumed that the OpenGL 2.0+ application provides:
        * projection and model view matrices;
        * a directional light using light index GL_LIGHT0;
        * a material for visible (front) faces; and
        * a normal vector for each vertex.
*/

varying vec4 diffuse, ambient;
varying vec3 normal, light_direction, half_vector;

void main()
{

        normal = normalize(gl_NormalMatrix * gl_Normal);

        light_direction = normalize(vec3(gl_LightSource[0].position));

        half_vector = normalize(gl_LightSource[0].halfVector.xyz);

        diffuse = gl_FrontMaterial.diffuse * gl_LightSource[0].diffuse;
        ambient = gl_FrontMaterial.ambient * gl_LightSource[0].ambient;
        ambient += gl_LightModel.ambient * gl_FrontMaterial.ambient;

        gl_Position = ftransform();
}
```

Directional light

Fragment shader: directional_light.frag

```
varying vec4 diffuse, ambient;
varying vec3 normal, light_direction, half_vector;

void main()
{
        vec3 n, halfV;
        float nDotL, nDotHV;

        vec4 color = ambient;

        n = normalize(normal);

        nDotL = max(dot(n, light_direction), 0.0);

        if (nDotL > 0.0)
        {
                color += diffuse * nDotL;
                halfV = normalize(half_vector);
                nDotHV = max(dot(n, halfV), 0.0);
                color += gl_FrontMaterial.specular * gl_LightSource[0].specular *
                        pow(nDotHV, gl_FrontMaterial.shininess);
        }

        gl_FragColor = color;
}
```

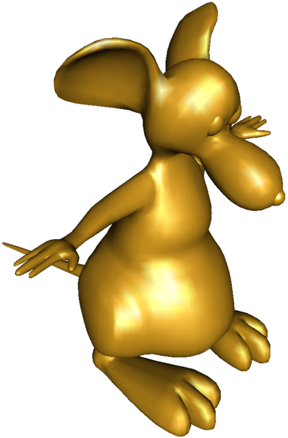

Fig. 4: Left: fixed pipeline functionality. Right: applying vertex and fragment shaders.