ECM3423: Computer Graphics
Worksheets 6&7 - Local Illumination and Shaders

Getting the new code

For this worksheet, I have uploaded a new version of the codebase onto the VLE. You should re-download all files, as I have updated them compared to WS5. The main addition to the program are the following files:

- `lightSource.py`: This file contains a basic class for handling light source information.
- `mesh.py`: This file contains the mesh class, used to load the mesh from file, not much change here.
- `material.py`: This file contains a class to hold the material information loaded from file.
- `bunny_world.mtl`: This file is a complement of the Blender file `bunny_world.obj` containing the material information.
- `shaders/`: This folder contains the GLSL source code for each shader program. You will see a folder: Flat. This folder contains two files: `vertex_shader.glsl` and `fragment_shader.glsl` that contain the programs for the vertex and fragment shaders respectively.

Some files have been significantly updated for this workshop:

- `blender.py`: The methods for loading Blender's `.obj` files are now capable of handling material and returns a list of mesh objects for more complex files.
- `shaders.py`: This class now allows to load the shader code from GLSL files located in a specific directory.

If you run this code, you will see a bunny mesh rendered using Flat shading. The following key control the rendering:

- 0: Switch the rendering between wireframe and polygon fill.
- 1: Render using flat shading (default)
- 2: Render using Gouraud shading
- 3: Render using Phong shading (will be done in the next session)

The aim of this workshop (and also the next one) is to:

1. Fill the GLSL programs for the vertex and fragment shader to implement Gouraud shading.
2. Fill the GLSL programs for the vertex and fragment shader to implement Phong shading.
3. (bonus) Add a completely new shader option to the program for Blinn shading.

If you run the code, you should get the following output: flat shading.

Shader programs: The OpenGL Shading Language (GLSL)

As will be discussed during the lecture, OpenGL 2.0 has introduced a *programmable pipeline*, where some programmable stages, called *shaders*, can be programmed using the *GL Shading Language (GLSL)*(the blue boxes in Figure 1). In OpenGL 4.0, the programmable shaders are:

- Vertex Shader: this shader is instantiated for each vertex, and is concerned mainly with the geometric transformations due to camera viewpoint.
- Tesselation Shader: can be used to subdivise primitives on the fly [**advanced topic**]
- Geometry Shader: is instantiated for each primitive, and can output zero or more primitives [**optional,advanced topic**]
- Fragment Shader: the fragment shader is instantiated for each fragment (pixel) of the polygon being rendered (typically in parallel) and it determines the appropriate colour for the pixel.

A few things are important to remember about shaders:

- Shaders are loaded, compiled and linked at runtime;
- Data needs to be transferred between the main OpenGL program and the shaders;
- Shaders are programs that are instantiated multiple times and run in parallel, e.g., for all vertices or fragments;
- Shaders are intended to run on GPUs.

All this means that special attention needs to be paid to how the data is uploaded to shaders, using VAO and VBOs.

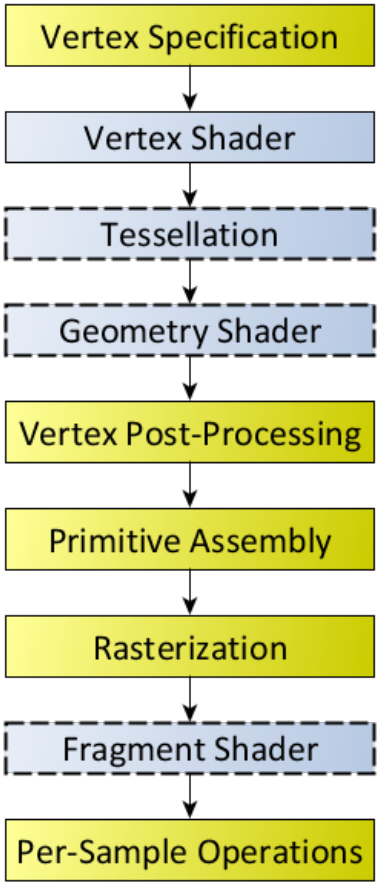

Figure 1: OpenGL pipeline

Loading GLSL Shaders into OpenGL program

Both the `vertex` and `fragment` shaders can be programmed using the GL Shading Language (based on the C language). In order to use such a *shader program*, it needs to be compiled on the fly by your program. In the provided code, this is handled by the `Shaders` class in `shaders.py` [l.181]:

```
self.program = shaders.compileProgram(
    shaders.compileShader(self.vertex_shader_source, shaders.GL_VERTEX_SHADER),
    shaders.compileShader(self.fragment_shader_source, shaders.GL_FRAGMENT_SHADER)
)
```

This code compiles the GLSL source for both shaders and assembles them into a shader program that can be used to render a scene, or part of a scene. The program can then be used at any point by calling:

```
glUseProgram(self.program)
```

In order to use the program, we will need to define a way for information to communicate between the Python main program and the GLSL shaders.

Data exchange between program and shaders

Remember that shaders are small programs that will be instantiated for each vertex/fragment:

Uniforms: are variables that will have the same value for *All* instances, i.e., all vertices or all fragments;

Inputs: are variables that will be different in each instance, i.e., vertex location or normal; and

Outputs: are the values that each instance of the shader will transfer down to the rest of the pipeline.

Uniforms

Uniforms are program-wide variables that are shared by *all instances* of the shader (i.e., same for all vertices or fragments).[1]. One example in our code is the PVM matrix, that is the same for the whole model. In the code, this is handled by the `Uniform` class in `shaders.py`.

Because uniforms need to be accessed by the shader program, usually running on the GPU, they need to be declared appropriately:

The first step is to find the *location* of the uniform in the linked shader program:

```
self.location = glGetUniformLocation(program=program, name=self.name)
```

This location can then be used to set the value of a uniform from the main program, using one of the `glUniform` functions. For example:

```
glUniform1i(self.location, self.value)
glUniform1f(self.location, self.value)
glUniform2fv(self.location, 1, value)
glUniform3fv(self.location, 1, value)
glUniform4fv(self.location, 1, value)
glUniformMatrix4fv(self.location, number, transpose, self.value)
```

Variants of `glUniform` exist for different uniform types, or for passing the values as arrays of integer or floats, check the documentation for details.[2]

Input attributes: vertex shader

The input variables to a shader are denoted by the keyword `in` and are different for every *instance* of the shader. To take the example of the vertex shader, the inputs could be:

- The vertex coordinates: (x, y, z);
- The vertex normal vectors: (n_x, n_y, n_z);

In this case, it is clear that these three vectors will be different for each vertex in the mesh. In practice, these inputs could be declared in the shader's GLSL code by:

[1] `https://www.khronos.org/opengl/wiki/Uniform_(GLSL)`

[2] See `http://pyopengl.sourceforge.net/documentation/manual-3.0/glUniform.html` for a description

[shaders/gouraud/vertex_shader.glsl]

```
in vec3 position;        // the position attribute contains the vertex position
in vec3 normal;               // store the vertex normal
in vec3 color;                // store the vertex colour
```

When the program renders the scene, one instance of the shader will be created for each vertex, and each instance will receive input values corresponding to this vertex. In practice, this means that the data for all vertices need to be uploaded to the GPU in buffers before in order for the shaders to access them.

To this end, the vertex data is stored buffers:

- as a linear (C-style) array
- made of floats (i.e., 4 bytes each)
- and contains 3 values per vertex (i.e., x, y, z)

The OpenGL program needs to prepare and upload the data in the correct structure to allow vertex data to be streamed to the correct instances of the vertex shader.[3]

As seen before, the following lines, in `BaseModel.py`, load all data in a Vertex Buffer Object (VBO) that can be uploaded to the GPU:

```
self.vbos[name] = glGenBuffers(1)
# and bind it
glBindBuffer(GL_ARRAY_BUFFER, self.vbos[name])

# enable the attribute
glEnableVertexAttribArray(self.attributes[name])

# Associate the bound buffer to the corresponding input location in the shader
# Each instance of the vertex shader will get one row of the array
# so this can be processed in parallel!
glVertexAttribPointer(index=self.attributes[name], size=data.shape[1], type=GL_FLOAT, normalized=
                      stride=0, pointer=None)

# ... and we set the data in the buffer as the vertex array
glBufferData(GL_ARRAY_BUFFER, data, GL_STATIC_DRAW)
```

We also need to inform the program to which `in` attribute in the GLSL vertex shader is the array associated to:

```
glBindAttribLocation(self.scene.shaders.program, self.attributes[name], name)
```

This will bind the attributed at index `self.attributes[name]` to the `in` attribute `name` in the vertex shader code. Note that the attribute also needs to be activated using:

```
# enable the attribute
glEnableVertexAttribArray(self.attributes[name])
```

Let's look at this in more details:

Vertex Buffer Objects (VBO) A Vertex Buffer Object (VBO) is an optimised manner for uploading data to the GPU. It can be used with the following commands:

- `glGenBuffers()` is used to create a new buffer and return its ID (an integer). This ID can then be used to interact with the created buffer. If size is larger than 1, then this creates that many buffers and expects `vbo` to be an array of integers to be filled with the buffers IDs.
- `glBindBuffer(type,vbo)` this binds the buffer specified by the ID `vbo` as the current buffer for all subsequent OpenGL function calls. Two main buffer types are of interest for this course:
  - `GL_ARRAY_BUFFER` is used for shaders' input variables, i.e., vertex position, normal and texture coordinates.
  - `GL_ELEMENT_ARRAY_BUFFER` is used for vertex index arrays when using a shared vertex representation. It is used automatically by the primitive assembly stage and is not associated to a shader variable.
- `glBufferData(type,data,usage)` is used to actually allocate the currently bound buffer and initialise its data from a local array. `type` is the same parameter as for `glBindBuffer` and `data` is a pointer to the start of the array. `usage` is a hint to OpenGL on how the buffer is going to be used, we will assume `GL_STATIC_DRAW`.[4]

[3] `https://www.khronos.org/opengl/wiki/Vertex_Specification`

[4] For more information, we refer to the OpenGL documentation `https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glBufferData.xhtml`

Specify buffer layout The layout of the data in the buffer needs to be specified in the OpenGL program to allow it to be distributed between instances. This is done using the function `glVertexAttribPointer`, for example:

```
glVertexAttribPointer(index, size, type, normalised, stride, pointer);
```

This function specifies:

- The index of the input variable, as set before;
- The size of the input variable, e.g., 2 for a `vec2` object or 4 for `mat2` object;
- The data type: e.g., `GL_FLOAT`;
- Whether the data needs to be normalised, usually `GL_FALSE`;
- The stride, set to zero if the data is tightly packed;
- A pointer to an array of data, if `None` then a buffer object need to have been bound prior to the function call (this is what we will do in general).

Output variables: vertex shader

The output variables are the result of the computations of the shader, hence are computed for each vertex and are propagated further down the pipeline. The values are then interpolated across primitives (triangles or quads) for each fragment location and sent to the fragment shader.

`[shaders/gouraud/vertex_shader.glsl]`

```
out vec3 fragment_color;  // the output of the shader will be the colour of the vertex
```

Because the shaders are part of a pipeline, the fragment shader's input variables need to be declared as output variables of the vertex shader:

`[shaders/gouraud/fragment_shader.glsl]`

```
in vec3 fragment_color;
```

Because we render the scene directly to the screen, the output of the fragment shader only needs to be the final colour of the pixel as a four values vector for the red, green and blue components:

```
out vec3 final_color;
```

but it is also possible to render in buffers as we will see later on, where the fragment shader can output more data.

Task 1: Implement Gouraud shading

The first task is to implement Gouraud shading, following information in the lecture slides. The provided code gives you already a skeleton for this program and indications what to fill. Note that with Gouraud, the shading is calculated *per vertex* and therefore all the work is done in the vertex shader.
Hint: Study carefully the comments in the code, and make use of the provided uniforms and attributes.
Hint: Look into the provided flat shader for inspiration (except for the normals!).

Task 2: Implement Phong shading

Phong shading is very similar to Gouraud, with the exception that the normals are interpolated instead of the intensity. This means that normals need to be passed onto the fragment shader, and that the shading calculations need to be done in the *fragment shader*.

Task 3: Implement Blinn shading

Blinn shading is a small adaptation of Phong to reduce computations. This code skeleton is similar to Phong shading.
