DVA338 OpenGL Shader and Animation

2025

Formalities

This document is a collection of tasks that need to be done to complete the assignment. You will be given a high-level description of the tasks, but you are **required** to supplement this with relevant parts of the course book and lectures. You will need to learn the course content, think about the problems and solve them using your own ideas and thoughts.
The assignment is separated into three separate tasks for examination, marked as "Part 1", "Part 2" and "Part 3". Tasks 1 and 2 conclude with a series of *extra* assignments, which are optional but provide extra points (EP) for the written exam. To pass the assignment, "Part 1", "Part 2" and "Part 3" should each be submitted separately in Canvas and presented in the assigned Lab session.
While the tasks have a natural progression in terms of features, it is beneficial to read through the whole section before starting on the tasks.

1 Transformations and Simple Scene Rendering – Part 1

In this assignment, you will write a program that renders triangle meshes using OpenGL. It is recommended that you use the start-up code available on the course page, which includes a simple triangle mesh data structure, some triangle meshes, a small algebra package, and initializations. Run the program and make sure you understand how it works.

1.1 Understand the Code

In the **main.cpp** you might want to take a special look at the `display()` and `renderMesh()` functions. That is where most of the magic happens.
The `display()` function is called by OpenGL every frame. In addition to rendering all meshes, display prepares the camera transformation matrices and sends them to the OpenGL pipeline. Here we need to calculate a minimum of 2 matrices. These are called matrices **V** and **P** and at the moment we have given constant values to both of them.
Matrix **V** is the *View matrix* and represents the transformation that moves the object to the camera view. Part of your job in this assignment will be to calculate this matrix, given the location and direction of the camera.
Matrix **P** is the projection matrix and is responsible for projecting the 3D scene to a 2D space. Later you will need to create this matrix too.

The renderMesh() function is responsible for sending the mesh vertices and the transformation matrix to the OpenGL pipeline. At the moment, you do not need to understand the code much. Just remember that since this part of code is called for each object separately, later any local transformation will need to be passed to the pipeline from here.
Also take a look at **mesh.cpp**, this is where we store the mesh information from the input files into memory structures of our own. Try to investigate this code and also the mesh files to get a grasp of mesh structure. Notice that there are comments in the code to guide you where to work on the assignment.

1.2 Viewing Transformation

Extend the algebra module with your own functions for creating translation, scaling, and rotation matrices. Use these functions to define your view matrix. Let the camera location in the world be defined by three rotation angles α, β, γ and a position $c = (c_x, c_y, c_z)$. Then the transformation from world to camera coordinates can be defined as:

$$V = R_z(-\gamma)R_y(-\beta)R_x(-\alpha)T(-c)$$

To start using this camera transform during rendering, you must of course pass it to the vertex shader (look at the `renderMesh()` function. There are comments to help you out). Also, make it possible to interactively move the camera to demonstrate that the camera transform works as expected. For example, use the keys [x], [X], [y], [Y], [z], [Z] to change the position of the camera along the world's coordinate axes, and the keys [i], [I], [j], [J], [k], [K] to rotate the camera in the world coordinate system with respect to the x-, y-, and z-axes. Finally, note that although this rudimentary camera model works, the navigation is not very convenient. See suggestions for an improved camera model in Section 1.6.

1.2.1 Check your work

In the start-up code the camera is initially positioned at $(0, 0, 20)$ and all of its rotation values are set to 0 (if these are not the values that you have the change them to these). Before you start coding the aforementioned interactive camera, if all your viewing transformations are correct your result will look like as in Figure 1.Left.

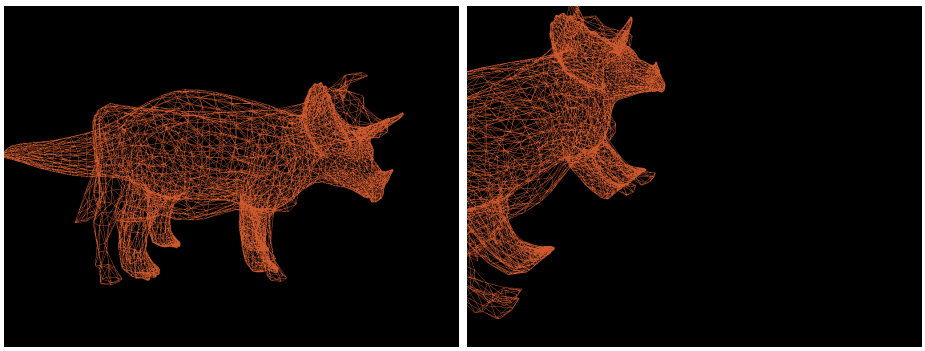

Figure 1: Result of part 1.2 with default camera values (Left), and with given values (right)

To check that the translations and rotations work, set the camera to the following and you should get the image shown in Figure 1.Right.

- Pos: $-5, -5, 20$
- Rotation: $-10, -30, -45$

1.3 Perspective and Parallel Projection

Extend the algebra module with your own functions for creating suitable perspective and parallel projection matrices. To demonstrate that the projection matrices created with your functions work as expected during rendering, make it possible for the user to interactively vary both the projection type and some of the parameters that define the projection matrices.

1.3.1 Check your work

Make sure that your code is rendering at 1024x768 resolution, and the camera is positioned exactly as default (at point $(0, 0, 20)$ with no rotation). Also comment out the lines that add the cow and triceratops and remove the comments for the knot.

- *Orthogonal Projection*: Use these values for left, right, top, bottom, near and far respectively: -20, 20, 10, -10, 1, 1000. The result is presented in Figure 2 (Left).
- *Perspective Projection*: Use the FOV version of the matrix and default camera settings. Use all the required values from the camera which is already defined in the code. The result is presented in Figure 2 (Right).

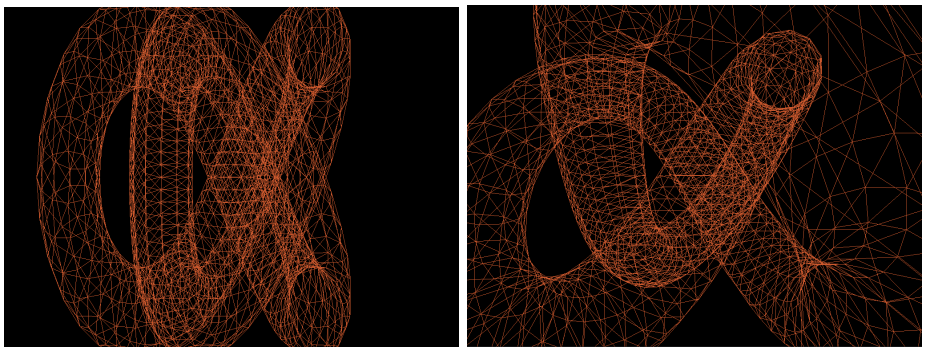

Figure 2: Orthogonal and Perspective project results.

1.4 Generating Vertex Normals

Calculate and store vertex normals for all triangle meshes during program initialization (look at **mesh.cpp**. There are comments to help you). These normals will be used later in the assignments to realize simple shading models. However, you should be able to see whether the normals seem correct or not, since the provided shader program renders them as if they were RGB colours over the surfaces of the meshes. To see this effect, change the polygon mode to get the triangles filled. You also need to enable the Z-buffer for proper hidden surface elimination. You can read about Polygon mode and Z-Buffer on the net.

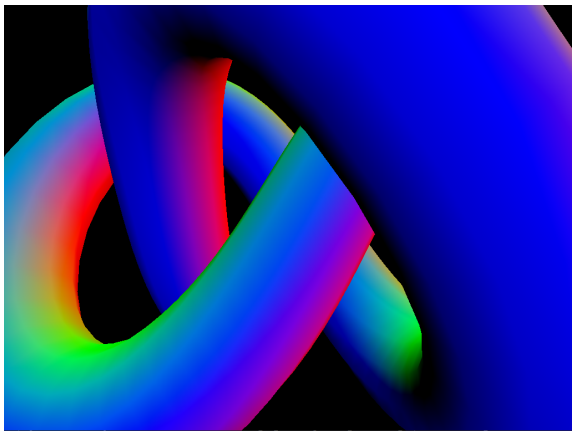

Figure 3: Vertex Normals calculated and visualized.

1.4.1 Check your work

Using the knot model and default camera position. The result should look like Figure 3:

1.5 Scene Composition

The geometry in the included 3D models is defined in their own local coordinate system. When several models are put into a virtual scene, we would like to place the objects at specific locations of our own choice. Make it possible to specify the position, orientation, and size of several models in your scene.
For each model, specify and store parameters for

- Scaling $s = (s_x,\ s_y,\ s_z)$
- Rotation $r = (r_x,\ r_y,\ r_z)$
- Translation $t = (t_x,\ t_y,\ t_z)$

Note that the transformation of the models, using these parameters, should take place when the objects are rendered, and not initially when the models are stored in the triangle mesh data structure. Each time a model is rendered, you compute the model's composite transformation matrix $\mathbf{W}$ directly from the scale, rotation, and translation parameters:

$$\mathbf{W} = T(t)R_x(r_x)R_y(r_y)R_z(r_z)S(s)$$

Then the combined transformation matrix $\mathbf{M} = \mathbf{VW}$ can be used during rendering to transform the vertices of a model directly from local to view coordinates.

1.5.1 Check your work

Using the default camera position, FOV projection from previous parts and the following values you should get the result as presented in Figure 4.

- Cow: Positioned at (0, 1.5, -8) and rotated (0, -90, 0)
- Teapot: Positioned at (0, -7, 0) and rotated (-90, 0, 180)
- Triceratops: Positioned at (-6.8, 0.8, 5) and uniformly scaled by 0.1

Note: If you've changed it, the default scaling on the meshes are 20.0 for the cow and 3.0 for the triceratops and teapot.

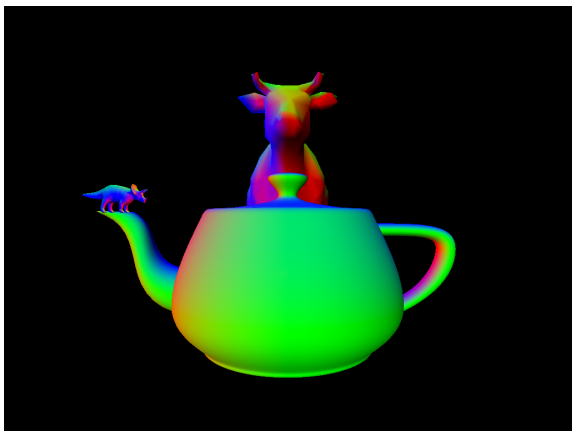

Figure 4: The result of part 1.5.1.

The extras for section one start here, if you do not want to do the extras, then jump to Shader Programming (Section 2).

1.6 Extra: Navigation (2 EP)

Implement a camera model which makes it possible for the user to interactively navigate along the principal axes of the current view, rather than along the axes of the world coordinate system. Support a full 6-DOF interactive camera model, i.e., it should be possible to move the camera forward/backwards, up/down, left/right as well as rotating the camera using pitch, yaw, and roll rotations.

Hint: Define the camera coordinate system using a camera position, $\boldsymbol{c}$, a normalized view direction vector, $\boldsymbol{v}$, and a normalized up vector, $\boldsymbol{u}$. Note that the third axis of the camera coordinate system is implicitly given as the cross product of the view direction and the up vector. To render the current view, you also need to create a `lookAt()` function that, given $\boldsymbol{c}$, $\boldsymbol{v}$, and $\boldsymbol{u}$, sets the view matrix appropriately.

To move the camera, translate and rotate the camera parameters appropriately. For example, to move the camera along the line of sight, we can compute a new camera position as $c_0 = c + v$, which will move the camera coordinate system forward 1 length unit. Or to rotate the camera sideways (yaw), we can simply rotate the view direction, $\boldsymbol{v}$, a few degrees using $\boldsymbol{u}$ as the rotation axis. For this operation, you need to define a function that generates a matrix for rotation about an arbitrary axis.

1.7 Extra: Use the GLM Library (1 EP)

Use the GLM (OpenGL Mathematics) library instead of the small algebra module you have used so far, i.e., remove all dependencies of **algebra.h** and **algebra.cpp** and use the corresponding GLM functions instead. Test that all the features you have implemented still work as expected together.

2 Shader Programming – Part 2

Current graphics hardware supports the execution of several types of user-defined programs such as vertex, fragment, geometry, tessellation, and compute shaders. This assignment will give you some practical experience in writing your own shaders using the OpenGL Shading Language (GLSL). In this assignment you will focus on vertex and fragment shaders.

2.1 Loading Shaders from Files

The startup code comes with basic a basic vertex shader and a basic fragment shader. The GLSL code for these shaders is defined in the "**shaders.h**" file as string arrays and then later used in the "**main.cpp**" file (look at the init() function in **main.cpp** and follow the call to prepareShaderProgram() function).
Of course, this is not a good way of doing this. Your first task is to implement a system to load the shader code from text files to a related string for each shader. Then type in the provided shader code into text files (one for vertex and one for fragment shader) and load them into separate strings and use the strings to call the `prepareShaderProgram()`. This will make the rest of the assignment much easier. Pay special attention to the error messages printed during shader compilation and link, as they will help you find the errors you might have.

2.2 Change Shaders Interactively

Apart from the shader code that you moved from the **shaders.h** file to separate text files, we have provided you with a new set of shaders in the `InterlacedVertexShader.glsl` and `InterlacedFragShader.glsl` files. First use your newly developed shader loading function to load and use these shaders instead. If all goes fine the result should be as presented in Figure 5.
As now you have multiple shaders to use, you should be able to switch between them at runtime. Make an appropriate key binding to let the user change the active shader.
NOTE: It is important that you do not load and initialize the shader every time the user changes the shader program. The loading and initialization must only be done once.

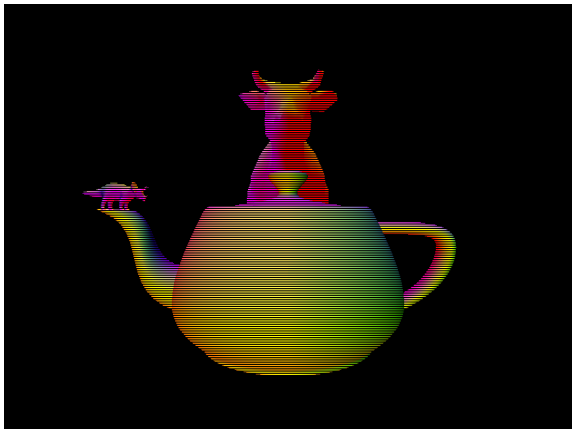

Figure 5: Output of the Interlaced Shaders.

2.3 Phong Shading

Write a shader program that accomplishes Phong Shading, i.e., vertex normals are interpolated over the faces. In this way, the Phong Reflection Model can be used for each pixel. Try to visualize each component of the Phong shading separately first, before adding them up for the final result (Figure 6).

2.3.1 Checking your work

Make the following scene:
Light:

- Pos: $(x, y, z) = (-20, 10, 10)$
- Ambient: $(r, g, b) = (0.2, 0.2, 0.2)$
- Diffuse: $(r, g, b) = (0.9, 0.9, 0.7)$
- Specular: $(r, g, b) = (1.0, 1.0, 1.0)$

Camera:

- Pos: $(x, y, z) = (0, 0, 20)$
- Rotation: $(x, y, z) = (0, 0, 0)$
- FOV: 60 degrees

Objects:

- Teapot:
  - Position: $(0, -7, 0)$
  - Rotation: $(-90, 0, 180)$
  - Ambient: $(0.19225, 0.19225, 0.19225)$
  - Diffuse: $(0.50754, 0.50754, 0.50754)$
  - Specular: $(0.508273, 0.508273, 0.508273)$
  - Shininess: 51.2
- Cow:
  - Position: $(0, 1.5, -8)$
  - Rotation: $(0, -90, 0)$
  - Ambient: $(0.329412, 0.223529, 0.027451)$
  - Diffuse: $(0.780392, 0.568627, 0.113725)$
  - Specular: $(0.992157, 0.941176, 0.807843)$
  - Shininess: 27.9
- Triceratops:
  - Position: $(-6.8, 0.8, 5)$
  - Scale: $(0.1, 0.1, 0.1)$
  - Ambient: $(0.05, 0, 0)$
  - Diffuse: $(0.6, 0.4, 0.4)$

- Specular: $(0.7, 0.04, 0.04)$
- Shininess: 10

For other examples of materials to play around with, see e.g.: OpenGL Material Samples

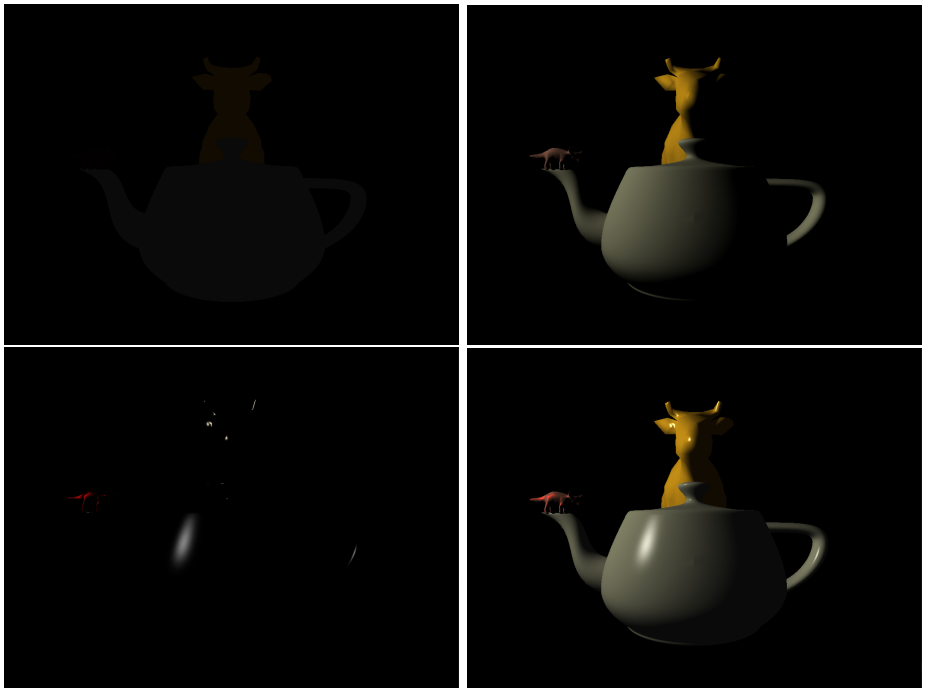

Figure 6: Ambient (top, left), Diffuse (top, right), Specular (bottom, left) components of Phong and full Phong render (bottom,right).

2.4 Cartoon Shading

Implement a simple cartoon shader, which gives a 3D object a kind of 2D look by only using a few levels of colour for the shading. This can be achieved by calculating the final diffuse + ambient colour and then scaling it by an intensity value which is calculated to be at specific levels (i.e. 0.2, 0.4, 0.6, 0.8). The results are shown in Figure 7.

> The extras for section two start here, if you do not want to do the extras, then jump to Animations (Section 3).

2.5 EXTRA: Multiple Light Sources (1 EP)

Extend your previous shader programs by adding support for at least two point light sources.

2.6 EXTRA: Flat Shading (1 EP)

Design a shader program that emphasize the actual faceted look of the polygon meshes by using the true mathematical polygon normal in the lighting computations, rather than constructed vertex normals.

Figure 7: Cartoon Shader using 5 levels of shading for each colour.

2.7 EXTRA: Morphing (2 EP)

Write a shader program that renders a morphing model, that is, a model trans-formed smoothly from a source model to a target model, and back again. Note that you only need to care about morphing between triangle models which have same number of vertices, triangles, and mesh topology.

2.8 EXTRA: One-pass Wireframe Rendering (1 EP)

Write a shader program to accomplish high-quality wire frame rendering of polygon models with hidden line removal. See the one-page article Single-pass Wireframe Rendering by Bærentzen et al. published in ACM SIGGRAPH 2006 Sketches [1] for more information on how this can be done.

3 Animations – Part 3

In this part we will learn the basics of frequent updating of the scene and practice it on simple animations.

3.1 Periodic Update

Animations are simply just a sequence of image frames that change at a frequency that make it appear fluid, where 30 to 60 frames per second are traditionally used in media. To make the frames update a certain number of times within each second, glut has a built-in function called `glutTimerFunc()`. The function receives a function pointer and a time offset, it then waits for the amount of time and then calls the function which it has received. Using `glutTimerFunc()` allows us to update the values and render the scene on specific intervals without blocking the main thread.
First of all, check that this is working by implementing a simple function that prints something to the console. Then make it run with some periodicity (e.g., once per second) and confirm that the printout appear (roughly) when you expect it. Note that each call to

`glutTimerFunc()` only registers one execution of the callback function. It is your job to make this happen periodically.

3.2 Simple Animated Scene

In order to create a simple animation, every time the timer callback function is called by glut, we can use `glutPostRedisplay()` to force a redraw of the scene.

3.2.1 Check your work

Make this scene:
Light:

- Pos: $(x, y, z) = (-20, 10, 10)$
- Ambient: $(r, g, b) = (0.2, 0.2, 0.2)$
- Diffuse: $(r, g, b) = (0.9, 0.9, 0.7)$
- Specular: $(r, g, b) = (1.0, 1.0, 1.0)$

Camera:

- Pos: $(x, y, z) = (5, 7, 20)$
- Rotation: $(x, y, z) = (-13, 13, 0)$
- FOV: 60 degrees

Objects:

- Cube:
  - Position: $(-10, -1, -10)$
  - Scale: $(4, 0.1, 4)$
  - Material: same as the triceratops from the previous part.
- Knot:
  - Position: $(0, 4, 0)$
  - Scale: $(0.4, 0.4, 0.4)$
  - Material: Same as the cow from the previous part.

Animation: Rotate the knot around its y-axis by 2 degrees each frame. Choose your timer values so that you achieve a framerate of 50.

References

[1] Andreas Bærentzen, Steen L. Nielsen, Mikkel Gjøl, Bent D. Larsen, and Niels Jørgen Christensen. Single-pass wireframe rendering. In *ACM SIGGRAPH 2006 Sketches*, SIGGRAPH '06, page 149–es, New York, NY, USA, 2006. Association for Computing Machinery.