Interactive Computer Graphics Coursework

Bernhard Kainz

January 2, 2014

Important

- The Computer Graphics coursework MUST be submitted electronically via CATE. For the deadline of the coursework see CATE. The files you need to submit are described later in this document.
- Before starting the assignment please make sure you have read the description of the environment and data formats below.
- If you are using a laptop with two graphics processors, make sure that the framework is executed on the high-performance GPU and not on the energy saving unit.
- The framework will be available on gitlab and will be updated during the course after the weekly lecture. Make sure you pull the updated framework before starting with a new exercise

```
git clone https://gitlab.doc.ic.ac.uk/bkainz/cgcoursework.git
git pull https://gitlab.doc.ic.ac.uk/bkainz/cgcoursework.git
```

Make sure that you give yourself enough time to do the coursework by starting it well in advance of the deadline. If you have questions about the coursework or need any clarifications then you should come to the tutorials!

This coursework exercise is a practical programming exercise, which should be done using the Open Graphics Library (OpenGL) and the OpenGL Shading Language (GLSL). The goal of the coursework is to get familiar with the

programming environment and to implement basic shader based computer graphics methods. Before you start the task, however, make sure you have fully read and understood this section.

Environment

The programming environment for the exercises is OpenGL with *The OpenGL Utility Toolkit* (GLUT), `http://www.opengl.org/resources/libraries/glut/`, which is available on the Linux/X11 machines of the department. Apple Macs may also provide a suitable Linux environment. You may use a Windows machine but then you need to make sure to use the correct built of GLUT (32 or 64 bit, etc.). While GLUT provides a pure C interface, you are free to program in either in C or C++.

The provided framework

The skeleton OpenGL program that is provided for you contains the following files:

- CMakeLists.txt,
- main*XX*.cpp,
- shader*XX*.vert,
- shader*XX*.geom,
- shader*XX*.frag,

where *XX* denotes the exercise number. The "shader..." files are GLSL shader files, which you will extend in the course of this exercise together with adjustments in main*XX*.cpp, whenever necessary.

To set up the framework, create a new directory and `git clone` all files into it. To compile the program, go to the new directory and type `cmake .` and `make` at the command line. This will compile the sample program for you. Executing `./cgExercise01` will open an OpenGL window as shown in Figure 1.

Figure 1: A very simple static OpenGL scene.

1 Exercise 1: Framework and Basic interaction

The scene shown in Figure 1 shows a famous object from Computer Graphics: "The Utah Teapot". However, the scene lacks of interaction and proper lighting, which you will implement in the following two exercises.

To get familiar with the framework you will first implement **basic mouse interaction**. Therefore we have provided two empty functions `mouseClick` and `mouseMotion`. These two functions should update the global variables `translate_z` for zoom, `rotate_x` and `rotate_y`, `move_x` and `move_y` for basic object rotation.

`mouseClick(int button, int state, int x, int y)` provides the clicked button binary encoded. If the **first bit** of `button` is set, then the left mouse button was pressed, if the **second bit** is set, then the left mouse button was pressed, if the **fourth bit** of is set, then the left mouse button was pressed. It also provides the state of the button in the form of predefined enumerations `GLUT_UP` and `GLUT_DOWN` together with the position of the click-event within the OpenGL window.

Your goal is to use the left mouse button for rotations of the object, the middle mouse button for moving the object in it's current xy-plane, and the right mouse button for moving the object along

the z-axis (zoom). When your calculations of `translate_z`, `rotate_x`, `rotate_y`, `move_x`, and `move_y` are ready, you can use the OpenGL functions `glTranslatef` and `glRotatef` in the main render function `renderScene` at the marked position to manipulate your teapot.

Note that we aim for object manipulation in this example and that the camera position will remain static. Make sure, that the interaction is intuitive, *i.e.*, the object should follow the mouse movements when a button is pressed. For example, when the middle mouse button is pressed and the mouse pointer moves to the left, the object should also move to the left with approximately the same speed as the mouse pointer.

1.1 Summary

- implement `mouseClick(int button, int state, int x, int y)`,
- implement `mouseMotion(int x, int y)`,
- extend `renderScene` with the appropriate translations and rotations of the object to provide mouse based scene interaction.
- execute `./cgExercise01`

Now you should be able to rotate your object with the left mouse button, to move it with the middle mouse button, and to zoom the object with the right mouse button.

2 Exercise 2: Illumination

In this exercise you will learn the basic use of vertex, geometry and fragment shader and how to build an interface between them. Finally, you will learn how to use these shaders for vertex-wise and pixel-wise scene illumination.

2.1 Per Vertex Gouraud shading

For this exercise you will need to edit the file `shader02.vert`. This file is the Vertex shader and performs operations on scene vertices. It is already incorporated into the skeleton program and you don't need to edit main.cpp. However, you can have a look at the `setShaders` function, to get an idea about how to integrate shaders into an OpenGL program.

In this exercise you will implement *Gouraud shading* as discussed during the lecture. Gouraud shading is an interpolation scheme for the illumination based on the viewer's, the vertex' and the light position within the scene.

In `shader02.vert` we have already defined a basic interface to the other shaders to pass on specific information about the currently processed vertex:

```
out vertexData
{
    vec3 pos;
    vec3 normal;
    vec4 color;
    vec3 lightDir;
}vertex;
```

The keyword `out` specifies, that this struct will be passed on to the following shaders (the geometry shader and the fragment shader).

The function `main()` in `shader02.vert` defines already a simple pass-trough vertex shader. This means, that this shader does exactly the same as what would be done during the static graphics pipeline:

```
//xyz position of the vertex
vertex.pos = vec3(gl_ModelViewMatrix * gl_Vertex);
//normal vector of the vertex
vertex.normal = normalize(gl_NormalMatrix * gl_Normal);
//set the transformed position of the vertex
gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
```

As you can see, GLSL has already many useful variables built in. For a quick reference about all the built-ins in GLSL, please refer to `http://mew.cx/glsl_quickref.pdf`. Note that built in `gl_*` variables are valid for shaders before OpenGL version 4.0. Later you will learn how to use your own matrices, which is essential for programs using OpenGL 4.0 and higher.

We also provide `unifrom` variables, which are set by the main program:

```
uniform vec4 ambientColor;
uniform vec4 diffuseColor;
uniform vec4 specularColor;
uniform float specularExponent;
```

You can access the position of the single light source in this scene by `vec4 gl_LightSource[0].position`. To convert this vector from `vec4` to `vec3`, you can either cast the vector `vec3 v = vec3(gl_LightSource[0].position),`, or access the vector elements individually `vec3 v = gl_LightSource[0].position.xyz;` `float x = gl_LightSource[0].position.x;` etc..

Your task for exercise 2.1 is to redefine `vertex.color` according to *Gouraud shading*. You can define the attenuation coefficient directly via the in GLSL integrated properties of the light source.

```
float attenuation = 1.0 / (gl_LightSource[0].constantAttenuation
+ gl_LightSource[0].linearAttenuation * d
+ gl_LightSource[0].quadraticAttenuation * d * d);
```

$$
\begin{aligned}
I_a &= \texttt{ambientColor} && (1)\\
I_d &= \texttt{attenuation} * \texttt{diffuseColor} * (\vec{n} \cdot \vec{l}) && (2)\\
I_s &= \texttt{attenuation} * \texttt{specularColor} * (\vec{r} \cdot \vec{e})^{\mu * \texttt{specularExponent}} && (3)\\
I &= I_a + I_d + I_s. && (4)
\end{aligned}
$$

with

$$
\begin{aligned}
\vec{n} &:= \text{normal vector of vertex}\\
\vec{l} &:= \text{direction to light source}\\
\vec{e} &:= \text{normalized direction from light source}\\
\vec{r} &:= \text{reflected light vector}\\
I_a &:= \text{ambient color component}\\
I_d &:= \text{diffuse color component}\\
I_s &:= \text{specular color component}\\
I &:= \text{resulting color: } \texttt{vertex.color}\\
\mu &= 0.3 \text{ per definition, play around with it if you want}\\
\alpha &= attenuationcoefficient\\
d &= distancetothelightsource\\
* &= \text{component wise multiplication}\\
\cdot &= \text{dot product}
\end{aligned}
$$

Note that you only need to define one color per vertex. The interpolation between these vertices's is done by the graphics hardware.

Test your program now with

`./cgExercise02 1`

The result should look similar to Figure 2a. Note the illumination shortcomings at the border of the specular highlight.

2.2 Per Pixel Phong shading

In this part you will implement *Phong shading* as discussed during the lecture. Phong shading is an interpolation scheme for the illumination based on the viewer's, the fragment's and the light position. In contrast to Exercise 2.1, this exercise operates directly on fragments and needs therefore the extension of the fragment shader in `shader02.frag`.

You can use the same definitions as provided in Exercise 2.1. However, note that in the fragment shader you get an input fragment instead of an input vertex:

```
in fragmentData
{
     vec3 vpos;
     vec3 normal;
     vec4 color;
}frag;
```

You can again access the light source position via `gl_LightSource[0].position.xyz`.

Test your program now with

`./cgExercise02 2`

The result should look similar to Figure 2b. Note that the quality of the specular highlight got improved.

2.3 Per Pixel Toon shading

In this part you will implement *Toon shading*. Toon shading is a simple lighting scheme, which allows you to achieve effects similar to hand drawn cartoons.

A toon shader can be defined as follows:

$$I_f = \frac{l}{||l||} \cdot \frac{n}{||n||} \tag{5}$$

$$I = \begin{cases} (0.8, 0.8, 0.8, 1.0), & \text{if } I_f > 0.98 \\ (0.8, 0.4, 0.4, 1.0), & \text{if } I_f > 0.5 \text{ and } I_f <= 0.98 \\ (0.6, 0.2, 0.2, 1.0), & \text{if } I_f > 0.25 \text{ and } I_f <= 0.5 \\ (0.1, 0.1, 0.1, 1.0), & \text{if } else \end{cases} \tag{6}$$

Test your program now with
`./cgExercise02 3`
The result should look similar to Figure 2c.

(a) Gouraud shading (b) Phong shading (c) Toon shading

Figure 2: Results of Exercise 2

2.4 Summary

- implement Gouraud shading in `shader02.vert`,
- implement Phong shading in `shader02.frag`,
- implement Toon shading in `shader02.frag`,
- execute `./cgExercise02 <illumintation model 1, 2, or 3>`

3 Exercise 3: Generating Primitives

In this exercise you will learn how to generate primitives within a shader. So far, you have learned how to use a vertex shader and a fragment shader. In this task you will program a geometry shader, whose base skeleton can be found in `shader03.geom`. While it is also possible to manipulate the position of incoming vertices in the vertex shader, a geometry shader is additionally

able to emit new primitives (i.e. vertices) into the pipeline and to transform them into different types.

In the following you will implement a simple mesh subdivision algorithm as it is outlined in Figure 3.

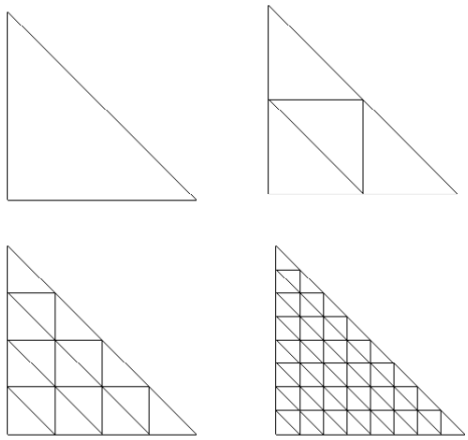


Figure 3: Subdivision on the example of a single triangle.

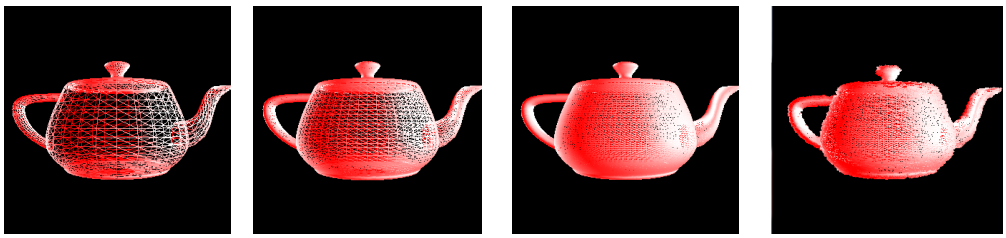
For this task we have changed the surface representation to wire-frame, so that you can see the result of your computations. You can try out this mode with

`./cgExercise03 2`,

where the second parameter is optional and by default set to 2. It is passed on to the geometry shader, to define the desired number of levels for the primitive subdivision. An example of the output (without subdivision) is shown in Figure 4a.

You should implement the subdivision using barycentric coordinates within a nested for-loop and use the functions `EmitVertex()` to generate a new vertex and `EndPrimitive()` to close the new triangle. You can also define new functions, e.g. for producing a new vertex in barycentric coordinates similar to a simple C program. A pass-trough shader is already implemented in `shader03.geom`, which shows the basic use of these functions:

```
int i;
for(i = 0; i < gl_in.length(); i++)
{
```



(a) Level 0 (b) Level 1 (c) Level 2 (d) Level 2 Bonus

Figure 4: Results of Exercise 3

```
frag.vpos = vertices[i].pos;
frag.normal = vertices[i].normal;
frag.color = vertices[i].color;
gl_Position = gl_in[i].gl_Position;
EmitVertex();
}
EndPrimitive();
```

`gl_in` defines a struct of the incoming base vertices and the struct `frag` is used to pass on further information to the fragment shader.

Results for the subdivision are shown for one level in Figure 4b and for two levels in Figure 4c Note that the number of possible additional primitives that can be emitted by a geometry shader is limited and hardware-dependent. Therefore, you should limit your number of levels to 2 or 3.

3.1 Exercise 3 Bonus: vertex animation

If you want to acquire some bonus points, you may implement vertex animation at the geometry shader stage. Therefore you can choose any time-dependent displacement of the resulting vertex in direction of it's normal vector. To help your with this task, we provide a uniform variable `time` in the geometry shader, which is set by the host program to the current system time. You can also use pseudo-number generation functions initialized by, e.g. the vertex xy position, similar to:

```
float rnd(vec2 x)
{
    int n = int(x.x * 40.0 + x.y * 6400.0);
```

```
    n = (n << 13) ^ n;
    return 1.0 - float( (n * (n * n * 15731 + 789221)
    + 1376312589) & 0x7fffffff) / 1073741824.0;
}
```

A snapshot of the animation may look like shown in Figure 2a. You can implement any animation you like, for example, a melting teapot.

3.2 Summary

- implement vertex subdivision in `shader03.geom`.
- Bonus: implement vertex animation in `shader03.geom`,
- execute `./cgExercise03`

4 Exercise 4: Texture

Given that an object has defined *uv* texture coordinates, the texturing of an object can by done automatically by the hardware as you can see when starting

`./cgExercise04`.

Simple texture coordinates can also be generated automatically by OpenGL:

```
glEnable(GL_TEXTURE_GEN_S); //enable texture coordinate generation
glEnable(GL_TEXTURE_GEN_T);
//draw some object without texture coordinates
glDisable(GL_TEXTURE_GEN_S); //enable texture coordinate generation
glDisable(GL_TEXTURE_GEN_T);
```

However, *uv* coordinates for more complex objects are usually generated by an artist, *e.g.*, for computer games.

Your task is in this exercise to extend the interface structures between vertex, geometry and fragment shader with texture coordinates. Therefore you can get the texture coordinates of a vertex in the vertex shader from `gl_MultiTexCoord0`. As soon as you have send these coordinates though the pipeline, you can access their interpolated results via the integrated GLSL function `texture2D(sampler2D tex, vec2 texureCords)`.

Note that `gl_MultiTexCoord0` is a `vec4`. However, to access the correct pixel position in the texture, you only need the first two components of this vector, *i.e.*, `frag.texCoords.st`.

Now try to set the `ambient` component from your Phong illumination from Exercise 2.2 to the result of the texture lookup. The result should look like Figure 5b.

(a) texture only (b) texture and Phong

Figure 5: Textured and Phong shaded teapot.

4.1 Exercise 4 Bonus: Your own texture and further considerations

Try to figure out how the texture is loaded and replace the texture with your own image. The new texture should *seamless* for an appealing visual quality.

Additionally, try to implement a more vivid representation of the texture or write down your thoughts about how the appearance of the texture could be improved to get more plastic. Add your thoughts to the end of `shader04.frag`.

4.2 Summary

- implement texturing in `shader04.vert`, `shader04.geom`, and `shader04.frag`.
- Bonus: use your own image as texture and improve or write down possible improvements of the texture representation in `shader04.frag`.

- execute `./cgExercise04`

5 Exercise 5: Render to Texture

For this exercise we have modified the main program slightly. You can find the modified version of the host program in `main05.cpp`. The modifications are an extended `renderScene` function, which renders the scene first into a so called framebuffer. A framebuffer is basically a texture image similar to the one that you used in the previous exercise. However, this object has the additional capability to capture the output of your render window. This function is currently one of the most important ones in applied Computer Graphics because many different image procession algorithms can be applied to this 2D texture image as post processing step. `renderScene` displays finally a screen aligned quad, which is textured with the previously generated scene texture.

For this task we have prepared `shader05.frag` and `shader05.vert`, which act in their plain version as pass-through shader for the screen-aligned textures quad. Your task is to extend `shader06.frag`, so that it produces a simple blur effect.

Simple blur can be achieved by sampling the available texture in the direction towards the image center. In this example we work with normalized texture coordinates, which means for GLSL, that every position within the input texture is encoded within [0.0 1.1]. Therefore the image center is located at $c = (0.5, 0.5)$ and the vector to the image center from any position p can be calculated by $\vec{p} = c - p$. By accumulating color values from the input texture tex parallel to the normalized $\vec{p}$ you can define a blurred color value for the current pixel according to it's distance d to the current pixel position p:

$$rgb_{blur} = \frac{1}{n} \sum_{i=0}^{n} (tex(p + \vec{p} * d_i)), \tag{7}$$

where d can be limited to a maximum range d_{max} and sampled within this range by fixed distances s_i. Therefore,

$$d_i = s_i * d_{max}. \tag{8}$$

You should use the following $n = 12$ factors s_i to determine your samples within d_{max}:

```
float s[12] =
    float[](-0.10568,-0.07568,-0.042158,
            -0.02458,-0.01987456,-0.0112458,
            0.0112458,0.01987456,0.02458,
            0.042158,0.07568,0.10568);
```

When defining $d_{max} = 0.3$ and executing `./cgExercise05`, the resulting scene should look similar to Figure 6.

Figure 6: Very simple blur effect.

5.1 Summary

- apply a simple blur to the framebuffer in `shader05.frag`.
- execute `./cgExercise05`

6 Exercise 6: Simple GPU ray tracing

In this exercise you will implement a very simple ray tracer. Since the scene from Exercise 5 provides already a rather static camera setup for rendering the screen aligned textured quad, you can use this camera also to virtually

shoot rays into a scene. However, since efficient ray tracing usually requires space partitioning for polyhedral geometry, we simplify the test scene in this exercise to objects that can be described mathematically: *planes* and *spheres*.

We have predefined a scene consisting of a spheres and one plane in `shader06.frag`. Please leave the scene configuration unaltered for marking reasons. The vertex shader `shader06.vert` already defines positions and for rays in camera direction for a 16:10 aspect ratio. Without any implementation the render window will show only white.

Your task is to implement the ray tracing algorithms in `shader06.frag`. Therefore, you will need to follow every ray until it reaches a maximum ray tracing depth `raytraceDepth`. We have set this value to 42 in the hope that the program will give the answer to the life the universe and everything.

For simple ray tracing you may test every tray for an intersection with every object in the scene until the ray does not hit any of the objects or until it reaches the maximum `raytraceDepth`.

You should also compute shadows by using additional *shadow rays*. You can do this calculation in the `computeShadow(in Intersection intersect)` function. When computing shadow rays, you should slightly move the ray origin towards the ray direction or alter the ray direction slightly using the pseudo random number generator provided in `rnd()` to avoid numerical problems.

The objects in the scene are defined by their intersect functions, which you should implement in

`shpere_intersect(Sphere sph, Ray ray, inout Intersection intersect)`

and

`plane_intersect(Plane pl, Ray ray, inout Intersection intersect)`.

The plane intersection should additionally vary the ray hit color, so that a checkerboard pattern results.

When your ray tracer is ready you should see an image similar to Figure 7.

You should also implement mouse based scene interaction as you have been using it throughout the exercises. For this the fragment shader provides the uniform matrices `projectionMatrix` and `modelViewMatrix`. These matrices still change when you use the mouse interaction scheme from Exercise 1. You may use them to *either* manipulate the initial ray direction and ori *or* to alter the object's position. Think about the smarter option and make the right choice!

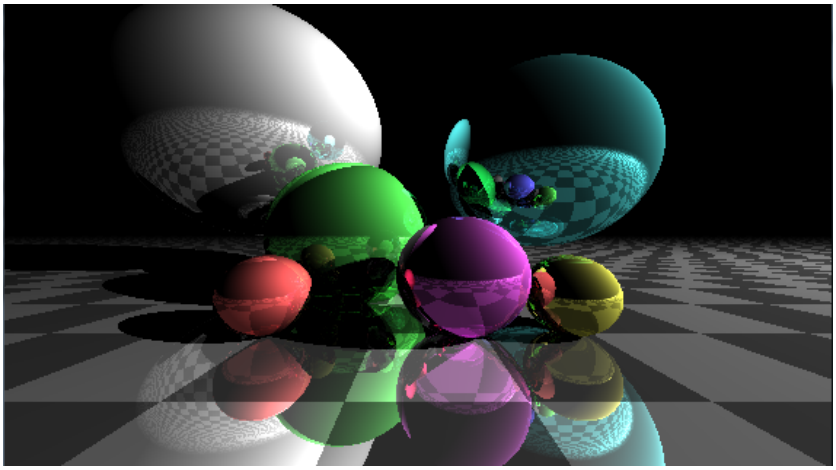


Figure 7: Very simple radial blur effect.

6.1 Summary

- implement ray tracing in `shader06.frag`.
- implement a mouse-based interaction using the provided matrices in `shader06.frag`.
- execute `./cgExercise06`

HAVE A LOT OF FUN!!