Interactive Computer Graphics Coursework

Bernhard Kainz, Antoine Toisoul, and Daniel Rueckert

November 7, 2016

Important

- The Computer Graphics coursework MUST be submitted electronically via CATE. For the deadlines and the required files for the assessed tasks see CATE. The files you need to submit are described later in this document.
- This document includes the specifications for all coursework exercises. Some tasks are instructional, not assessed but highly recommended to fully understand the course content. Others solidify key concepts and are assessed. You should solve one task per week, usually after the according content has been covered in the lecture.
- Before starting the assignments please make sure you have read the description of the programming environment and data formats below.
- All PCs in the lab are fully supported by the framework. We only support using the framework on the lab machines. This means that we will not support you when try to run the framework on you own machine.
- The framework for all exercises will be available on the lab machines and on gitlab. It might be updated during the course. Make sure to update any copies or you pull a potentially updated framework before starting with a new task.
- you can execute the framework on the lab machines directly through **TODO:** link or download and build the framework with

```
git clone https://gitlab.doc.ic.ac.uk/bkainz/ShaderLabFramework.git
cd ShaderLabFramework
cmake .
make
./ShaderLab
```

- Save your solution as *.xml file using 'File' → 'Save Pipeline'

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

This coursework exercise is a practical programming exercise, which should be done using the Open Graphics Library (OpenGL)[1] [2] and the OpenGL Shading Language (GLSL)[3] [4]. To keep the overhead as low as possible, we provide a comprehensive framework to manage basic I/O, shader editing and compilation functionalities. As requested by previous years students we are focusing only on the most recent version of OpenGL, *i.e.*, online resources that use older GLSL specifications will not be useful for the following tasks.

Before you start the task make sure you have fully read and understood this section.

1 Task 1: Framework and GLSL shader basics

This course provides a framework written in C++ using the popular Qt library[5] and modern OpenGL. It provides a convenient interface to all shader required in this exercise. The framework's shader and rendering hierarchy is shown in Figure 1. In the beginning of the coursework all of these shaders are simple pass-through shaders. The resulting scene has no illumination or other more sophisticated Computer Graphics effects. You will develop simple rendering engines using the provided shader framework during this coursework.

[1] http://www.opengl.org/wiki/Getting_Started
[2] http://www.opengl.org/sdk/docs/man/
[3] http://www.opengl.org/documentation/glsl/
[4] http://www.lighthouse3d.com/opengl/glsl/
[5] https://www.qt.io/

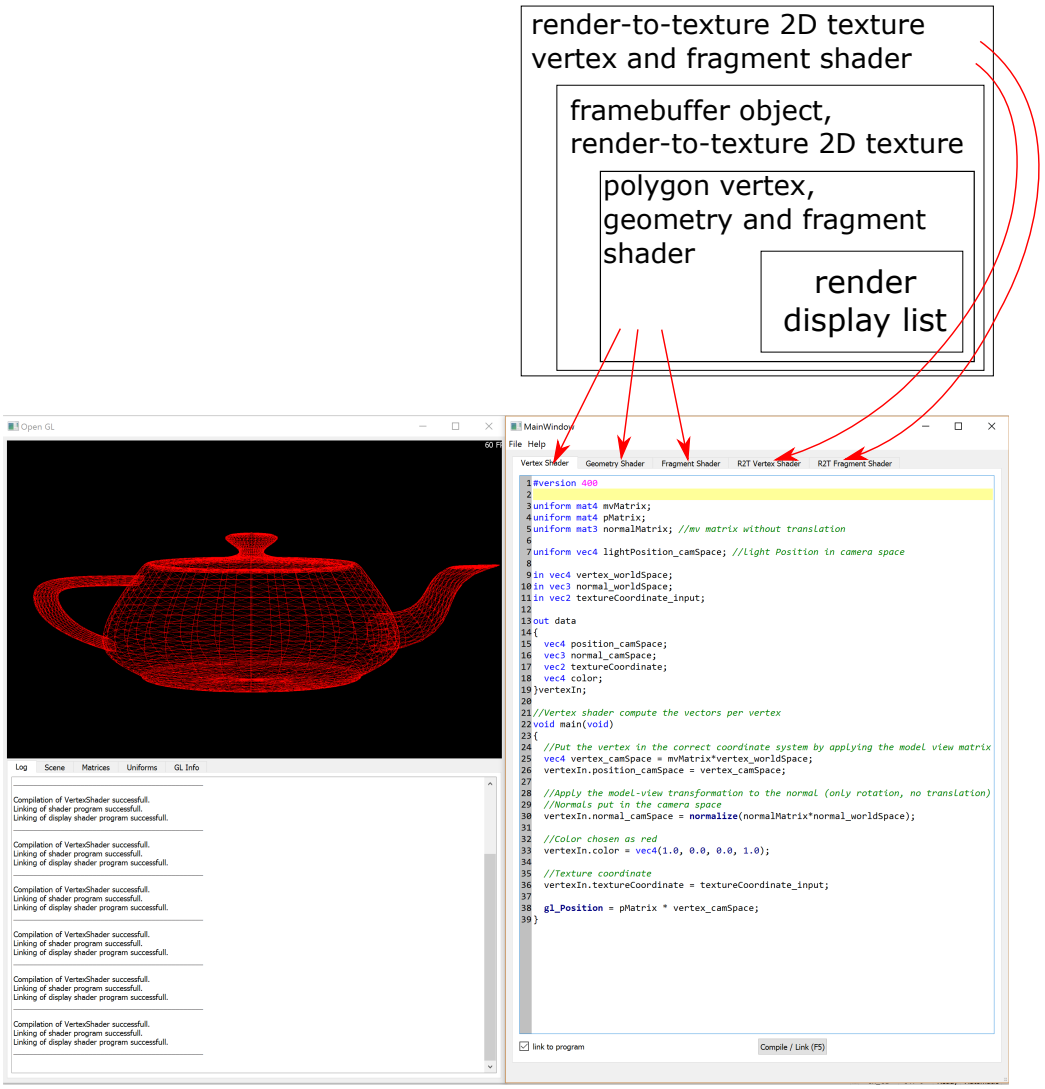


Figure 1: In this framework, from inside out in the figure, the polygons stored in a display list are first shaded in object space using a vertex, geometry and fragment shader. The result is rendered to a 2D texture of exactly the same size as the camera plane (= the render window.). This texture is passed through an additional vertex and fragment shader to achieve image based effects.

The framework provides a direct interface to the used matrices (see Task 2), *uniform* variables, which define the interface between the host programme and the shader and texture samplers that allow to access texture images stored in graphics memory (more about this later in Task 6). The values for these interface variables are mapped to fields in the provided widgets of the GUI.

The framework also provides a `Log` widget which shows the result of the shader compilation and linker stages ('Compile and Link' with the according button in the `Editor` widget or use `F5`).

Since the initial shaders are pure pass through shader using a hard-coded constant color for shading, the scene has not much appeal yet. The default model is a cube but we cannot see its shape yet because of missing illumination. The check the geometry besides the lack of a proper lighting model the framework provides a `Wireframe` mode in the `Scene` widget. (*Scene* → *Enable wireframe*)

Your tasks are:

- Write some rubbish in either the Fragment or the Vertex shader and hit Compile and Link. Check the `Log` widget to see what the GLSL compiler thinks about your syntax. Revert your changes and compile again.
- Find the used constant default hard-coded RGBA color value (pure 'red' per default) and change it to pure green.
- define a `uniform` vec4 variable and use this through the GUI to define the color of the object.
- change to `Wireframe` mode, get an overview over the scene and explain what you are seeing in this render mode.
- In `Wireframe` mode, klick the checkbox for `Back Face Culling`. Explain what is happening (if you for example turn around the object wit activated and deactivated `Back Face Culling`).

2 Task 2: Projections and Transformations

In Computer Graphics transformations and projections are defined through matrix operations as discussed during the lecture. In this exercise you will

learn how to use these matrices. For this task you may want to use wireframe mode for better visibility.

A 3D point P is represented in homogeneous coordinates by a 4-dimensional vector

$$p = \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \tag{1}$$

A full 4×4 transformation matrix in homogeneous coordinates can be separated into individual parts steering translation T, rotation R, and the affine parameters scaling A_{sc}, reflection A_{re}, and shearing A_{sh} ($A = A_{sc}A_{re}A_{sh}$). The full transformation can be defined as

$$p' = T \cdot R \cdot A \cdot p. \tag{2}$$

Translation T can be defined as

$$T = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{3}$$

Rotation R can be defined as

$$R = R_x \cdot R_y \cdot R_z = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{4}$$

Scaling A_{sc} can be defined as

$$A_{sc} = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \tag{5}$$

where s_x, s_y, s_z are real values defining a scale factor along each axis.

Reflection through a specific plane A_{re} can be achieved by inverting components of the diagonal, *e.g.*, a reflection through the xy plane would look like this:

$$A_{re} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{6}$$

Shear effects can be achieved through manipulating the rotation parameters in a non-orthogonal way:

$$A_{sh} = \begin{pmatrix} 1 & a & b & 0 \\ c & 1 & d & 0 \\ e & f & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{7}$$

a, b, c, d, e, f changes each coordinate as a linear combination of all three.

A combined transformation matrix can be used as *ModelMatrix* to manipulate a 3D object in 3D space or to define the position of the camera plane as *ViewMatrix*.

The projection on the camera plane is defined through the *ProjectionMatrix*. In case of orthographic projection this matrix simply removed the z-coordinate and looks like

$$A_{re} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{8}$$

For perspective transformation we can add the focal length of the camera and use

$$A_{re} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/f & 1 \end{pmatrix} \tag{9}$$

as *ProjectionMatrix*.

The framework provides and interface to all of these matrices. For simplicity select the *box* model as object. Your task is to

- rotate the object 45° around axis $(0.5, 0.5, 0.75)$.

- scale the object by 50%.
- translate the object to $(0, 5, 0)$ followed by a 30° rotation around $(0, 0, 1)$.
- reflect the object through a plane defined by its normal vector $(0.7071, 0.7071, 0)$.
- shear the object along the x-axis to a general parallelepiped so that the top left edge of the cube is translated to $(1, 0, 0)$.
- change to orthographic projection.
- use perspective projection with focal length $f = 20mm$. The height and width of your field of view are shown on the perspective matrix widget.

Generate a separate screen shot for each of these tasks. Reset your matrices to *default* between the tasks.

3 Task 3: Illumination and Shading

Figure 2: Default scene shown by the framework: an unshaded teapod model.

In this exercise you will learn how to use vertex and fragment shaders for vertex-wise and pixel-wise scene illumination. We use the *per-polygon* vertex and fragment shaders for this task. The framework shows you an unshaded read teapod model per default as shown in Figure 2.

3.1 Per Vertex Gouraud shading*

For this exercise you will need to edit the file *per-polygon* vertex shader. This shader and performs operations on scene vertices in object space. It is already provided by the editor.

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 the *per-polygon* vertex shader we have already defined a basic interface to the other shaders to pass on specific information about the currently processed vertex:

```
out VertexData{
    vec4 texCoord;
    vec4 normal;
    vec4 color;
} VertexOut;
```

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()` 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 rendering pipeline:

```
VertexOut.texCoord = texCoord;
VertexOut.color = normalize(vec4(1,0,0,0));
VertexOut.normal = normalize(normalMatrix * normal);
gl_Position = projMat * modelViewMat * position;
```

`gl_Position` is the only output that is expected to be set by the glsl compiler. Everythin else can be freely defined. `modelViewMat` is the ModelView matrix and `projMat` is the projection matrix. These are updated by the main program using `uniform` variables. Please not that previous versions of OpenGL and GLSL had intrinsic variables for values like the ModelView and Projection matrix. Modern OpenGL is freely programmable and defines the use of the intrinsics as deprecated. Many examples on the internet might use old style OpenGL and intrinsics.

We also provide a definition for a simple light source. Its properties are defined by:

```
vec4 diffuse;
vec4 ambient;
vec4 specular;
float shininess;
```

You can access the position of the single light source in this scene by `vec4 light_pos`. To convert a vector from `vec4` to `vec3`, you can either cast the vector, e.g., `vec3 vec = vec3(light_pos),`, or access the vector elements individually `vec3 vec = light_pos.xyz; float x = light_pos.x;` etc..

Your task for exercise 3.1 is to redefine `VertexOut.color` according to *Gouraud shading*. Use the provided *Utah Teapot* model for testing. (Scene Widget → Test Model → Utah Teapot). An example output for this task is shown in Figure **??**.

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

The result should look similar to Figure 3a. Note the illumination problems at the border of the specular highlight.

3.2 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 interpolated normal vectors for each fragment instead of interpolated colors as done for the previous task 3.1. The shading effect depends on the viewer's, the fragment's and the light position. In contrast to Exercise 3.1, this exercise operates directly on fragments and needs therefore the extension of the *per-polygon* fragment shader.

You can use the same interface definitions as provided in Exercise 3.1. However, note that in the fragment shader you get an input fragment instead of an input vertex:

```
in VertexData{
    vec4 texCoord;
    vec4 normal;
    vec4 color;
} FragmentIn;
```

Your task for exercise 3.2 is to redefine `colorOut` according to *Phong shading*. Use the provided *Utah Teapot* model for testing. (Scene Widget → Test Model → Utah Teapot.)

You can again access the light source position via `vec4 light_pos;`. If you test your programme, the result should look similar to Figure 3b. Note that the quality of the specular highlight is better than in Figure 3a.

3.3 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. This exercise also operates directly on fragments and needs therefore the extension of the *per-polygon* fragment shader. You can use preprocessor definitions as common in C-like language to switch between the shading types.

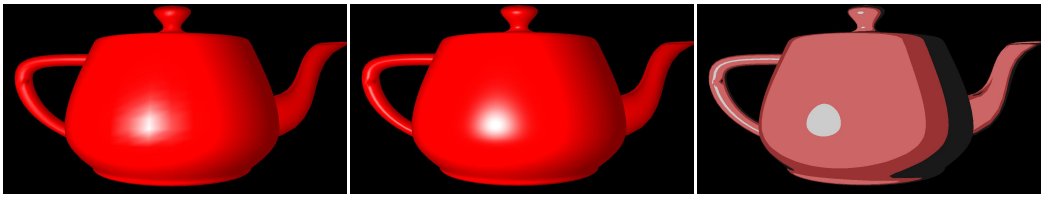
A toon shader can be defined per fragment as done in equation 10 and equation 11.

$$I_f = \frac{l}{||l||} \cdot \frac{n}{||n||} \tag{10}$$

$$I = \begin{cases} (0.8, 0.8, 0.8, 1.0), & \text{if } I_f > 0.98 \\ (0.8, 0.4, 0.4, 1.0), & \text{if } I_f > 0.5 \text{ and } I_f <= 0.98 \\ (0.6, 0.2, 0.2, 1.0), & \text{if } I_f > 0.25 \text{ and } I_f <= 0.5 \\ (0.1, 0.1, 0.1, 1.0), & \text{if } else \end{cases} \tag{11}$$

Your task for exercise 3.4 is to redefine `colorOut` according to *Phong shading*. Use the provided *Utah Teapot* model for testing. (Scene Widget → Test Model → Utah Teapot.)

The final result should look similar to Figure 3c.



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

Figure 3: Results of Exercise 2

Implement this task first using the provided light source position. The specular and illumination will stay constant relative to the position of the light source. Try to replace the static light source with a *head light*, *i.e.*, set the light source position equal to the position of the camera in camera coordinate space.

3.4 Task 3b: Blinn-Phong

Phong shading is not the most efficient way to approximate illumination. Blinn-Phong is a more efficent modification of Phong shading using halfway-vectors.

Your task for exercise 3.4 is to redefine `colorOut` according to *Blinn-Phong shading* as discussed during the lecture. Use the provided *Utah Teapot* model for testing. (Scene Widget → Test Model → Utah Teapot.) Figure 4 shows and example output.

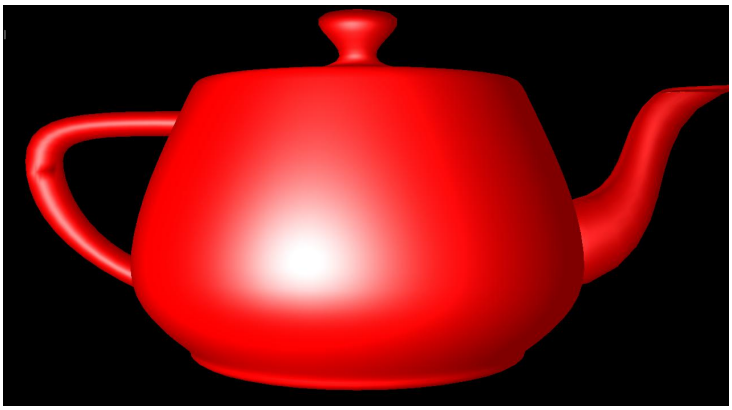


Figure 4: Result for Task 3b: Blinn-Phong shading

4 Task 4: Geometry

4.1 Task 4b: mesh subdivision*

In this exercise you will learn how to generate primitives within a geometry shader. So far, you have learned how to use a vertex shader and a fragment shader. In this task you will modify the *per-polygon* geometry shader. **To**

activate this shader in your rendering pipeline you need to tick the box beneath the geometry shader tab and rebuild the shader program (`F5`)!

While it is 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 very simple mesh subdivision algorithm as it is outlined in Figure 5.

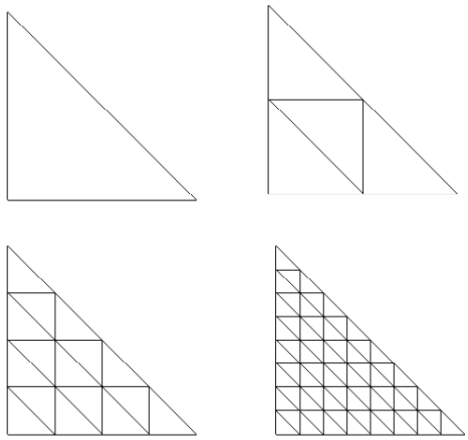
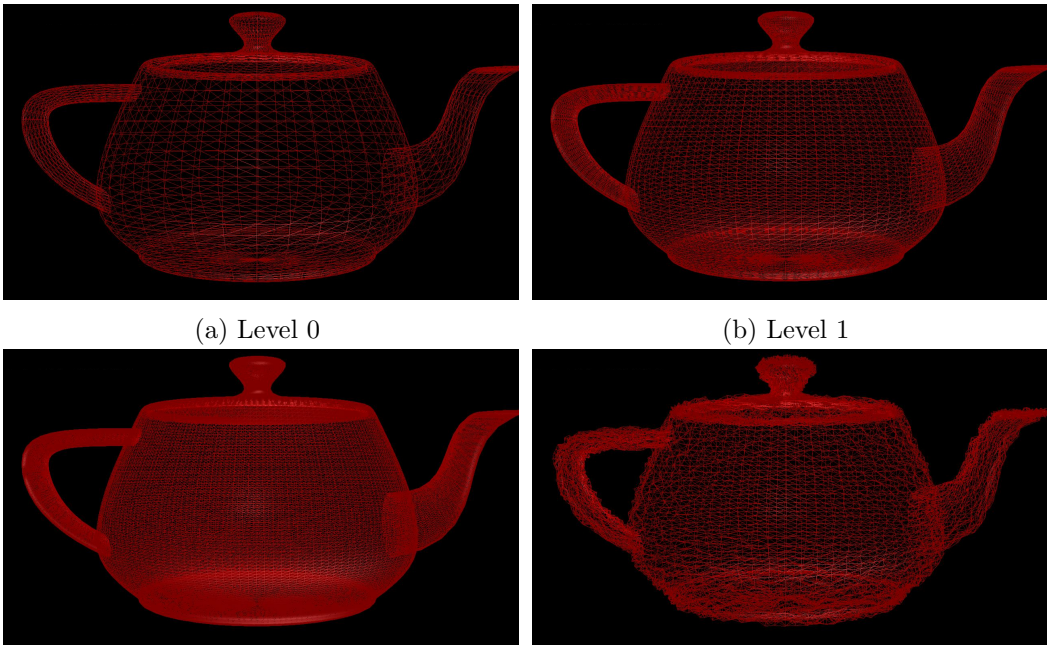


Figure 5: Subdivision on the example of a single triangle.

For this task you should use *Wireframe* mode to see the result of your computation. (*Scene* → *Enable wireframe*) To define the desired number of levels for the primitive subdivision you should use a `uniform` variable. An example of the output (without subdivision) is shown in Figure 6a.

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. Do not forget the also interpolate the new vertex's normal vector. A pass-trough shader is already implemented in the *per-polygon* geometry shader, which shows the basic use of these functions:



(a) Level 0 (b) Level 1

(c) Level 2 (d) Level 2 Bonus

Figure 6: Results of Exercise 3

Results for the subdivision are shown for one level in Figure 6b and for two levels in Figure 6c 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.

4.2 Task 4b: vertex animation

Optionally 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 time since start of the shader program. 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 3a. You can implement any animation you like, for example, a melting teapot.

5 Task 5: Colour

Colour space conversion is an important tool in any modern computer graphics applications. The RGB colour space has the disadvantages that it is device specific, it is not useful for human description of colour (e.g., we do not describe color as RGB percentages) and it is highly redundant and correlated (*e.g.*, all channels hold luminance information, which reduces coding efficiency). In Computer Graphics is often required to separate colour from intensity information. This is difficult using the RGB model but straight forward when using an alternative color space like HSV. This space separates hue, saturation, and value, which makes it easier to classify colours irrespective of, e.g., local illumination conditions or to generate an adaptive target colour according to a changing projection surface.

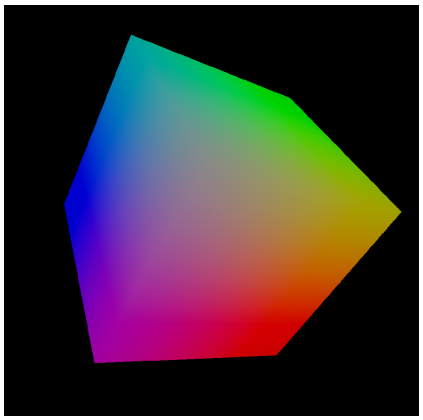


Figure 7: Example visualisation of the RGB colour space

Your task is to build a HSV to RGB converter as discussed during the lecture. You can do this in the *per-polygon* fragment shader and by defin-

ing a `uniform` variable as input HSV value. You can visualise the RGB colour space by using the currently rendered object's vertex position as output colour in the vertex shader (e.g. RGB colour space when rendering the cube model as shown in Figure 7). However, the cube for example is centred at the origin, hence you need to translate the position values with `vec4(0.5,0.5,0.5,0)` to get non-negative values.

Why is it difficult to visualise the HSV colour space using the cube model? Can you think of a different geometry that would be more suitable to sample the HSV space?

6 Task 6: Texture and render to texture

6.1 Task 6a: Texture*

Given that an object has defined *uv* texture coordinates, the texturing of an object can by done automatically in hardware. Simple texture coordinates can be generated automatically by OpenGL using spherical, cubical, cylindrical, etc. mapping. However, *uv* texture coordinates for more complex objects are usually generated by an artist, *e.g.*, for computer games using specialised tools.

Your task is to apply your own texture to the test objects. You can use the texture management capabilities of the framework and define a 2D texture sampler `sampler2D` object as `uniform` varible in the *per-polygon* fragment shader.

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

Now apply Phong illumination from Exercise 3.2 to the result of the texture lookup. The result should look like Figure 8b.

6.2 Task 6a: Bump mapping

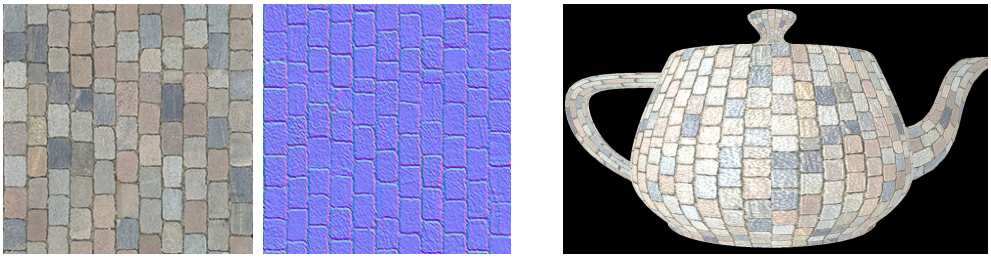
Bump mapping can be used to reduce geometric complexity why generating the impression of highly tessellated surfaces. The idea of bump mapping is simply to use another lookup texture which encodes surface normals instead of RGB colour values. The normals are still encoded as RGB values but can

(a) texture and ambient only (b) texture and Phong

Figure 8: Textured and Phong shaded teapot.

be interpreted during the illumination step as surface normal instead of the real, from the vertex shader coming, interpolated normal.

Your task is to use a second texture sampler in the *per-polygon* fragment shader and to use one of the provided normal maps as additional input texture. Use the sampled normals in the Phong illumination model. The result should look like shown in Figure 9(c). Figure 9 shows also the used texture and normal map.



(a) texture (b) normal map (c) Bump mapping

Figure 9: Bump mapping textures and result.

Task 6b: Render to Texture*

For this task you will use the render-to-texture 2D fragment shader instead of the per-polygon shaders. These shaders are applied to a screen aligned quad that is rendered in front of the camera. The quad is textured with the scene you have been working with so far and serves as an intermediate representation to allow image-based operations.

The framework 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.

The render-to-texture 2D fragment shader and render-to-texture 2D vertex shader are available in the editor and act in their plain version as pass-through shader for the screen-aligned textured quad.

Your task is to extend the render-to-texture 2D fragment shader, 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 1.0]. 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:

$$rgb_{blur} = \frac{1}{n} \sum_{i=0}^{n} (tex(p + \vec{p} * d_i)), \tag{12}$$

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

$$d_i = s_i * d_{max}. \tag{13}$$

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 $d_{max} = 1.0$ and, the resulting scene should look similar to Figure 10.

Figure 10: Very simple radial blur effect.

7 Task 7: GPU ray tracing

7.1 Task 7a: Simple GPU ray tracing *

In this exercise you will implement a very simple ray tracer. Since the scene from Exercise 6.2 provides already a rather static camera setup for rendering a 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*. Note that you will overwrite the per-polygon shaders in this exercise and that they will become useless in the pipeline. You can therefore deactivate them.

We have predefined a scene consisting of spheres and one plane in `SimpleSceneRayTracing.frag`.

The render-to-texture 2D vertex shader already defines positions and for rays in camera direction for a 16:10 aspect ratio. Without any implementation the render window will show a statically white render window.

Your task is to implement the ray tracing algorithms in the render-to-texture 2D fragment shader. Therefore, you will need to follow every ray until it reaches a maximum ray tracing depth. This value could be for example 42.

For simple ray tracing you may test every ray 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 ray-trace depth.

You should also compute shadows by using additional *shadow rays*. You can do this calculation in a separate `computeShadow(in Intersection intersect)` function. When computing shadow rays, you should slightly move the ray origin outwards of the object along the surface normal 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 can implement as for example

`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 be able to produce an image similar to Figure 11.

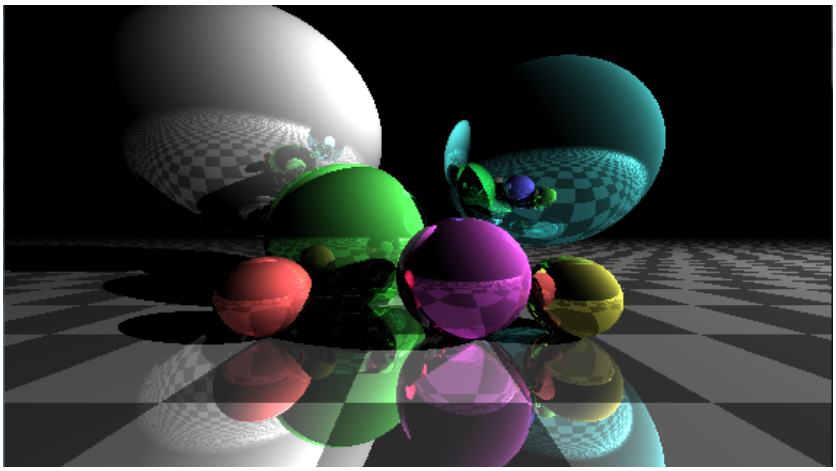


Figure 11: Example result from simple geometric ray tracing.

You should also implement mouse based scene interaction as you have been using it throughout the exercises. You can do this with the `uniform` matrices `projMat` and `modelViewMat`. These matrices will change when you use the mouse interaction scheme from as used in the framework. You may use them to *either* manipulate the initial ray direction *or* to alter the object's

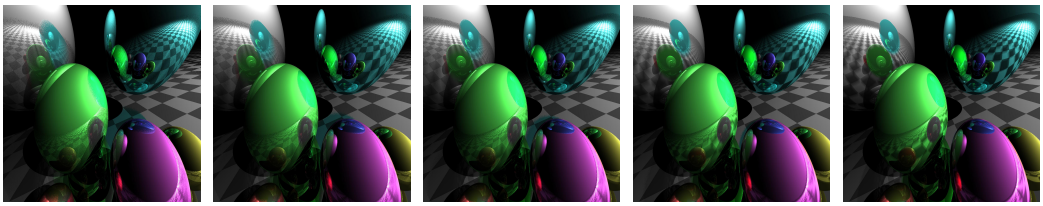
position. Think about the smarter option and make the right choice!

If you implemented the above described simple ray-tracer with shadow-rays and mouse based interaction successfully you will gain 75% of the maximum points for this task. The remaining 25% can be achieved by extending this exercise with your own ray-tracing extensions. For example you could implement transparent and refracting objects, light scattering effects, caustics, soft shadows, and many more. It is up to you which extension you choose!

7.2 Task 7b: Simple Monte-Carlo Path tracing

Monte-Carlo Path tracing is used to simulate global illumination. The algorithm aims to integrate over *all* the illuminance arriving to a single point on the surface of an object. A simple approximation of the algorithm can be achieved by sending several, slightly tilted rays instead of a single ray into the scene and to split them into more than one secondary ray following a random direction at each intersection point. This is an open ended task and you can implement as many path tracing features as you like. Be aware of the limitations of the used hardware. Processing many rays may quickly exhaust your GPU, which might lead to a crash of the rendering system.

Figure 12 shows an example for Monte-Carlo Path tracing with different numbers of secondary path rays.



(a) 5 path rays (b) 10 path rays (c) 15 path rays (d) 30 path rays (e) 50 path rays

Figure 12: Example result from Monte-Carlo ray tracing with a different number of Monte-Carlo rays.

HAVE A LOT OF FUN!!