Accessible GLSL Shader Programming

Antoine Toisoul[1], Daniel Rueckert[1] and Bernhard Kainz[1]

[1]Department of Computing, Imperial College London, UK

Abstract

Teaching fundamental principles of Computer Graphics requires a thoroughly prepared lecture alongside practical training. Modern graphics programming rarely provides a straightforward application programming interface (API) and the available APIs pose high entry barriers to students. Shader-based programming of standard graphics pipelines is often inaccessible through complex setup procedures and convoluted programming environments. In this paper we discuss an undergraduate entry level lecture with its according lab exercises. We present a programming framework that makes interactive graphics programming accessible while allowing to design individual tasks as instructive exercises to solidify the content of individual lecture units. The discussed teaching framework provides a well defined programmable graphics pipeline with geometry shading stages and image-based post processing functionality based on framebuffer objects. It is open-source and available online.

Categories and Subject Descriptors (according to ACM CCS): K.3.2 [Computers and Education]: Computer and Information Science Education-Computer Science Education—D.2.3 [Software Engineering]: Coding Tools and Techniques—

1. Introduction

Teaching the principles of Computer Graphics beyond traditional classroom based lectures is especially challenging because of high entry barriers formed by high-level graphics Application Programming Interfaces (APIs) and specific hardware requirements.

Practical lab exercises and small tasks related to taught content are essential to solidify knowledge and to practice new skills. Especially on undergraduate level, limited knowledge of programming environments and programming languages make it almost impossible to design a lab exercise that fit the needs of every student while including state-of-the-art technologies. Therefore, Computer Graphics related courses are very often at a late stage and only sparsely covered in undergraduate curricula and advanced programming models are usually only available to postgraduate students. Furthermore, despite great interest amongst high-school students in Computer Graphics rooted in the popularity of computer games, there are almost no tools available that would allow an accurate but simple way for experimentation with graphics APIs.

Only recently, tools like the Raspberry Pi [Gay14] and languages like python [van95] made general purpose programming accessible to the masses. Game-engines like Unity [Sta13] are currently making Computer Graphics programming and game design accessible in a similar way. However, while general purpose programming interfaces often lack advanced real-time graphics capabilities without the use of high-level graphics API libraries, full scale game-design tools usually include a lot of additional features like physics and artistic tools, which require again a high learning effort.

In this paper we present *ShaderLabFramework* as an easy to use OpenGL 4 framework with an IDE-like interface to program and compile GLSL shaders. This framework is used during our Computer Graphics course at the Department of Computing, Imperial College London, UK, for lab sessions and individual coursework exercises to teach students how to code GLSL shaders and apply the theory from the lectures. The framework is open-source and available online.

2. Related Work

A few frameworks are available to teach the basics of Computer Graphics.

GlMan [Bai] is a software that loads a scene description and renders it using specified shaders. It can be used for teaching purposes to help students understand GLSL shaders. ShaderMaker [Kra08] is a cross platform open-source framework written in C++ using the Qt library [Dig] and designed to teach students the basics of Computer Graphics. It allows to edit and compile shaders directly from its interface and visualize the result in an OpenGL window. The user interface also contains many widgets to modify the camera characteristics as well as the scene description. ShaderMaker was written in 2007, hence it does not support OpenGL 4 as well as the last Qt features that help writing code specifically for 3D renderings. ShaderLabFramework's GLSL shader editor is inspired by the one that has been implemented in ShaderMaker.

Unity [uni17] is a popular game engine that can be used to write shaders and visualise renderings easily. The shaders are written in

the ShaderLab language which is specific to unity. Our course is a general graphics course. As a result our framework uses OpenGL and teaches students GLSL to avoid being specific to a given game engine.

Lambers [Lam16] proposed a simple framework that can be used in virtual reality courses. It helps students to practice VR without the requirement of writing a specific graphics application or knowing high level libraries. With such a framework students can focus on the course instead of spending time on software related problems. In this paper we propose a framework with a similar purpose i.e help students learning the key concepts of Computer Graphics.

Also related is the work of Fink et al. [FWW13] who developed a Java 3D renderer to help students understand shaders and how to implement them in Java. Their framework is designed for an introductory course on Computer Graphics and does not teach OpenGL API and GLSL.

The Exploratories project [RRP00] contains a set of free java applets to help teaching graphics. These include lighting and shading, color theory and texture mapping. Although very useful to help explaining concepts in lectures these applets are not designed to teach students how to write modern OpenGL shaders.

Shadertoy [JQ13] is a tool to write pixel shaders (i.e a fragment shader applied on full screen quad) in a simple web interface. The result can be visualized in a browser with WebGL. Shadertoy has a community and can be used for learning purposes. However it is restricted to pixel shaders and rendering an arbitrary geometry with a regular rendering pipeline is not possible.

3. Course Syllabus

For our third year undergraduate Computer Graphics course we aim to provide a comprehensive lecture and a tailored lab framework in-between full scale game-design tools and low level graphics API. The course is usually attended by 120–150 students and runs for eight weeks. We base our lab exercise on one of the most common two-pass shader pipelines as shown in Figure 1.

During the lecture we connect this pipeline with the theory behind polyhedral rendering, illumination, image-based effects, and image generation methods like Ray tracing. Our detailed lecture syllabus is listed in the following. Each item is one lecture unit.

1. Projections and Transformations (week 1)
2. Transformations for Animation (week 1)
3. Clipping (week 1)
4. Graphics Pipeline and APIs (week 2)
5. Illumination, Shading (week 2)
6. Graphics APIs and Shading languages (week 2)
7. Colour (week 3)
8. Texture Mapping (week 3)
9. Rasterization, Visibility & Anti-aliasing (week 3)
10. Ray tracing (week 4)
11. Ray tracing (week 5)
12. Spline curves (week 5)
13. Spline surfaces (week 6)
14. Warping and Morphing (week 6)
15. Special effects (week 7)
16. Animation and Kinematics (week 7)
17. Radiosity (week 7)
18. Revision (week 8)

The Lab exercise is aligned with the content of the lecture. It covers the most essential topics, which are listed in the following.

1. Getting familiar with the course framework (week 1)
2. Transformations (week 2)
3. Illumination and Shading (week 3)
4. Geometry generation and processing (week 4)
5. Colour (week 5)
6. Texture & Render to Texture (week 6)
7. Simple GPU ray tracing (week 7-8)

While tasks 1, 2, and 5 are voluntary exercises, tasks 3, 4, 6, and 7 are assessed and contribute to the final mark of the student.

4. Simplified Shader Pipeline Programming

The core of the discussed Computer Graphics course is its comprehensive lab exercise. We have designed and implemented an integrated development environment (IDE) for a programmable shading pipeline with a fixed two shading pass layout. This design allows to cover most concepts that have been discussed during the lecture. Figure 1 provides an overview over the selected shading passes and Figure 2 shows an overview over our current ShaderLabFramework.

4.1. ShaderLabFramework architecture

The framework is implemented in C++ using the Qt library [Dig] and OpenGL 4 [Wol11]. It is divided in two parts: the shader editor and the rendering window. These two interfaces communicate with the Qt signal and slots system in order to update the renderings accordingly to the user modifications.

The rendering window contains an OpenGL widget and a set of tabs to modify the scene and camera descriptions. In these tabs the user can choose an object, change its properties such as the material it is made of (e.g ambient, diffuse and specular colours), load and apply textures and modify the camera properties (e.g field of view). The model, view and projection matrices can also be modified in the matrix tab and each modification can be directly visualized in the OpenGL widget. The wireframe and backface culling modes can also be enabled or disabled through the interface. Finally, the rendering window has a tab to display OpenGL information such as the GPU used and the GLSL compilation errors.

The shader editor can be used to code, compile, load and save the shaders that are applied in the rendering pipeline. It has a feature to directly save the entire two pass rendering pipeline to a XML file that can later be loaded. For the coursework students are asked to save and submit their XML file corresponding to each task. The XML format is very useful as it saves a lot of time for submission and marking. Indeed markers can easily load them, visualize the rendering and go through the code all within ShaderLabFramework.

Implementation wise the code is divided two parts : one for the user interface and one for the description of the 3D scene and the

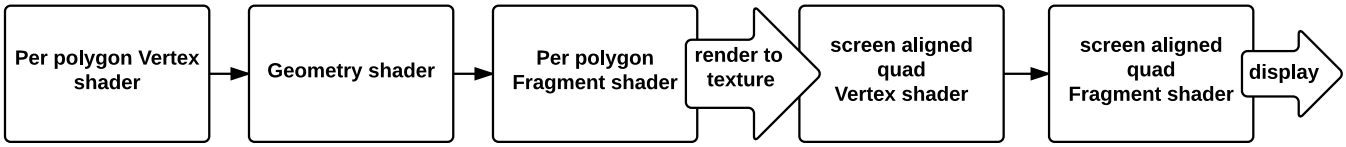


Figure 1: *The provided two-pass render pipeline architecture.*

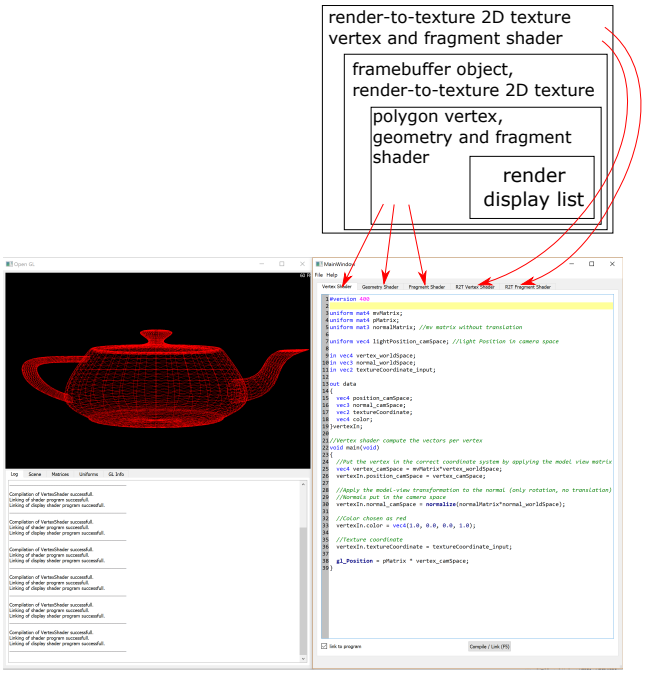
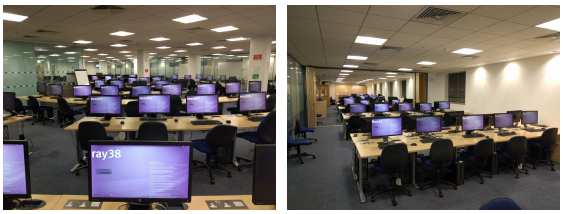


Figure 2: *In this framework, from inside out in the figure, the polygons stored in a display list are first shaded in object space using a vertex, geometry and fragment shader. The result is rendered to a 2D texture of exactly the same size as the camera plane (= the render window.). This texture is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



(a) *CG lab 1* **(b)** *CG lab 2*

Figure 3: *The students can work on 250 similar lab machines in two Computer Graphics labs, all equipped with Nvidia Graphics cards and identical operating systems, or on their own laptop.*

OpenGL classes. The 3D scene is divided in several classes called : *Scene*, *Object*, *Material*, *Mesh*, *Light*, *Camera* and *Texture*. These are self explanatory which helps the students to understand how the framework is made. In the user interface part we created a class inherited from *QGLWidget* with the main rendering loop. This class directly communicates with the user interface to change the 3D rendering depending on the user inputs.

The framework for all exercises is available on our unified lab infrastructure, running a customized Ubuntu operating system as shown in Figure 3, on *github:* `https://github.com/bkainz/ShaderLabFramework` and our departmental *gitlab* system. Thus the framework can either be directly executed on any lab machine or compiled using CMake on student's individual computers.

4.2. Task description

For each lab week the students are given a task tailored to the current week's lecture. After individual completion, the whole pipeline and all included shaders have to be saved as a XML file and submitted through our electronic submission system. Assessed tasks are marked within one week by the lecturers and the lab helpers. Feedback is given in the form of short statements about the provided solution. A sample solution is provided to the students two weeks after the submission deadline. The individual tasks are described in the following.

4.2.1. Task 1: The Framework

The framework provides a direct interface to the necessary matrices, *uniform* variables, which define the interface between the host program and the shader, and texture samplers that allow to access texture images stored in graphics memory. The values for these interface variables are mapped to fields in the provided widgets of the GUI.

Besides 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`) we also parse user defined `uniform` variables and make them availabe for manipulation in the *User uniforms* tab. To simplify the usage of OpenGL 4, which requires to handle all matrices as `uniform` variables, we have predefined a number of special `uniform` variables (*e.g.* ModelView matrix, Projection Matrix,*etc.*) that are not accessible through the *User uniforms* tab. These variables can be manipulated in the *Matrices* and *Material* tabs.

Since the initial shaders are pure pass through shader that replace the incoming colour with a hard-coded constant color, the scene has not much appeal as shown in Figure 4(a). The rendered model can be selected in the *Scene Tab* and the default model is

a Utah Teapot. However, its 3D shape cannot be perceived well at this stage because of missing illumination. To check the geometry besides the lack of a proper lighting model the framework provides a `Wireframe` mode in the `Scene` widget (*Scene Tab* → *Enable wireframe*)

The student's tasks are:

- to write some random text in either the Fragment or the Vertex shader and to hit *Compile and Link*. Then the student is asked to check the `Log` widget to see the GLSL compiler output. The changes need to be undone for error-free compilation.
- to find the used default (hard-coded) RGBA color value (pure 'red' per default) and change it to pure green.
- to define a `uniform` vec4 variable and to use this through the GUI to define the color of the object.
- to change to `Wireframe` mode, to get an overview over the scene and explain what they are seeing in this render mode.
- in `Wireframe` mode, to click the checkbox for `Back Face Culling` and to explain what is happening (if the object is for example turned around with activated and deactivated `Back Face Culling`).

4.2.2. Task 2: Projections and Transformations

In Computer Graphics transformations and projections are defined through matrix operations as discussed during the lecture. In this exercise the student will learn how to use these matrices. For this task the student is asked to use wireframe mode for better perception.

A 3D point p is represented in homogeneous coordinates by a 4-dimensional vector $p = (x, y, z, 1)^\mathsf{T}$. 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 $p' = T \cdot R \cdot A \cdot p$, where T is a 4×4 translation matrix, R a 4×4 rotation matrix, and A a 4×4 affine matrix.

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 a *ProjectionMatrix P*. In case of orthographic projection this matrix simply removes the z-coordinate and looks like :

$$P = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{1}$$

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

$$P = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/f & 0 \end{pmatrix} \tag{2}$$

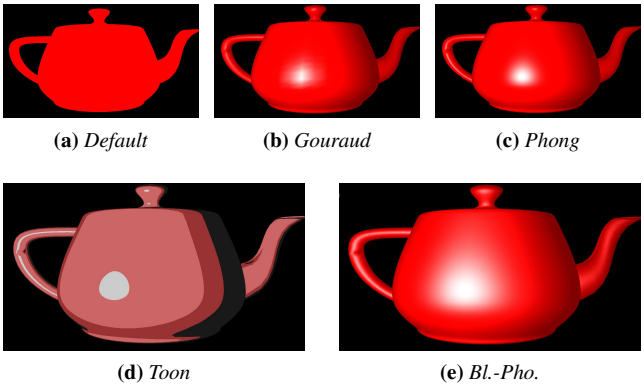
as *ProjectionMatrix*.

The framework provides an interface to all of these matrices. For simplicity the student is asked to select a *box* model as object. The student's task is to :

- rotate the object by 45° about the axis $(0.5, 0.5, 0.75)$,
- scale the object by 50%,
- translate the object to $(0, 5, 0)$ followed by a 30° rotation about the axis $(0, 0, 1)$,
- 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 the current field of view are shown on the perspective matrix widget.

4.2.3. Task 3: Illumination and Shading

In this exercise the students 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 presents to the student an unshaded *Utah Teapot* model per default as shown in Figure 4a. At the end of this task the student will be able to program different illumination models as shown in Figure 4.



(a) *Default* **(b)** *Gouraud* **(c)** *Phong*

(d) *Toon* **(e)** *Bl.-Pho.*

Figure 4: *(a) shows the default scene of the framework: a teapot model with no illumination and shading. (b-e) shows four different illumination models that are the results of Exercise 3.*

4.2.3.1. Task 3a: Per Vertex Gouraud shading For this exercise the students will need to edit the *per-polygon* vertex shader. This shader performs operations on scene vertices in object space. It is already provided by the editor. Students are asked to 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 define a basic interface to the other shaders to pass on specific information about the currently processed vertex (texture coordinates, normal vector, colour).

The function `main()` defines 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, except that it replaces the incoming color with a constant value :

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

`gl_Position` is the only output that is expected to be set by the GLSL compiler. Everything 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 note 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. We provide a definition for a simple light source as well as material properties with diffuse, ambient, specular and shininess terms.

The student's task for this exercise is to redefine `VertexOut.color` so that it forwards color values according to the *Gouraud* illumination model. The student may use the provided *Utah Teapot* model for testing. (Scene Widget → Test Model → Utah Teapot). An example output for this task is shown in Figure 4b.

The students are explicitly told that they only need to define one color per vertex. The interpolation between these vertices is done by the rendering pipeline.

4.2.3.2. Task 3b: Per Pixel Phong shading In this part the students 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 4.2.3.1. The shading effect depends on the viewer's, the fragment's and the light position. In contrast to Exercise 4.2.3.1, this exercise operates directly on fragments and needs therefore the extension of the *per-polygon* fragment shader.

The same interface definitions can be used as provided in Exercise 4.2.3.1. The students are instructed that the fragment shader gets an input fragment instead of an input vertex. The task is to redefine `colorOut` the forward colour values according to the *Phong* illumination model. The result is shown in Figure 4c. The students are asked to argue about the reasons for the quality differences between Figure 4b and Figure 4c.

4.2.3.3. Task 3c: Per Pixel Toon shading In this part the students will implement *Toon shading*. Toon shading is a simple lighting scheme, which allows to achieve effects similar to hand drawn cartoons. This exercise operates directly on fragments and needs therefore the extension of the *per-polygon* fragment shader. The students may use preprocessor definitions as common in C-like languages or `uniform` variables to switch between the shading types.

A toon shader can be defined per fragment through $I_f = \frac{l}{||l||} \cdot \frac{n}{||n||}$ and Equation 3.

$$I = \begin{cases} (0.8, 0.8, 0.8, 1.0), & \text{if } I_f > 0.98 \\ (0.8, 0.4, 0.4, 1.0), & \text{if } I_f > 0.5 \text{ and } I_f <= 0.98 \\ (0.6, 0.2, 0.2, 1.0), & \text{if } I_f > 0.25 \text{ and } I_f <= 0.5 \\ (0.1, 0.1, 0.1, 1.0), & \text{if } \textit{else} \end{cases} \quad (3)$$

l defines the vector from the light source and n the normal vector at the current fragment. The student's task is to redefine `colorOut`, so that it encodes illumination according to the *Toon shading* model. The final result is shown in Figure 4d.

The students are further asked to implement this task first using the provided light source position. The specular reflection will stay constant relative to the position of the light source. Then, they can 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.

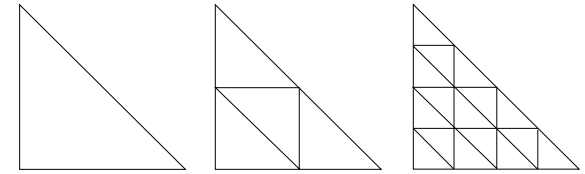
4.2.3.4. Task 3d: Blinn-Phong shading Phong shading is not the most efficient way to approximate per-fragment illumination. Blinn-Phong (Bl.-Pho. in Figure 4e is a more efficient modification of Phong shading using the halfway-vector. The student's task is to redefine `colorOut` according to the *Blinn-Phong* illumination model as discussed during the lecture. The result looks similar to Figure 4e.

4.2.4. Task 4: Geometry generation and processing

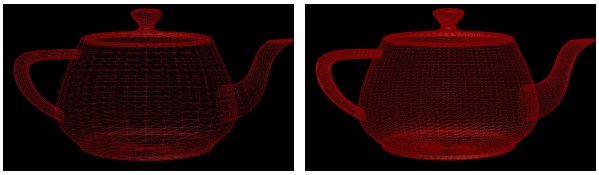
4.2.4.1. Task 4a: Mesh subdivision During this exercise the students will learn how to generate primitives within a geometry shader. To this point, we have covered how to use a vertex shader and a fragment shader. In this task, the lab participants will modify the *per-polygon* geometry shader. To activate this shader in the rendering pipeline it is required to tick the *include* box beneath the geometry shader tab and rebuild the shader program. 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. The students are asked to implement a very simple mesh subdivision algorithm as it is outlined in Figure 5a.

For this task, the students are instructed to use *Wireframe* mode to see the result of the computation. (*Scene* → *Enable wireframe*) To define the desired number of levels for the primitive subdivision a `uniform` variable is used. An example of the output (without subdivision) is shown in Figure 5b.

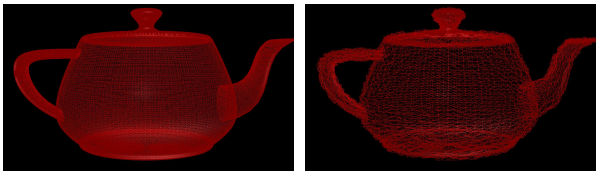
The students are required to implement the subdivision using barycentric coordinates within a nested for-loop and to use the functions `EmitVertex()` to generate a new vertex and `EndPrimitive()` to close the new triangle. It is also possible to define new functions, *e.g.*, for producing a new vertex in barycentric coordinates similar to a simple C program. It is also required to interpolate the new vertex's normal vector. A pass-trough shader is already implemented in the *per-polygon* default geometry shader and provided by the editor.



(a) *triangle subdivision strategy*



(b) *Level 0* **(c)** *Level 1*



(d) *Level 2* **(e)** *animation*

Figure 5: *(a) Subdivision strategy for a single triangle. (b-e) Example results for task 4: Different levels of triangle subdivision and vertex animation in (e).*

Results for the subdivision are shown for one level in Figure 5c and for two levels in Figure 5d. Note that the number of possible additional primitives that can be emitted by a geometry shader is limited and hardware-dependent. Therefore, it is recommended to limit the number of levels to two or three.

4.2.4.2. Task 4b: Vertex animation Optionally the students may implement vertex animation at the geometry shader stage. Therefore it is possible to choose any time-dependent displacement of the resulting vertex in direction of it's normal vector. To help 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. We also provide a pseudo random 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 single frame of the animation may look like shown in Figure 5e. The students can implement any animation they like, for example, a melting teapot.

4.2.5. 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 it is often required to separate colour from intensity information. This is difficult using the RGB model but straightforward 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.

The students' task is to build a HSV to RGB converter as discussed during the lecture. It is possible to do this in the *per-polygon* fragment shader and by defining a `uniform` variable as input HSV value. The students are also asked to 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 a cube model.). However, the cube for example is centered at the origin, hence the students are instructed to translate the position values with `vec4(0.5,0.5,0.5,0)` to get non-negative positions. Furthermore, we ask the students to argue about why it is difficult to visualise the HSV colour space using the cube model and if it possible to find a different geometry that would be more suitable to sample the HSV space.

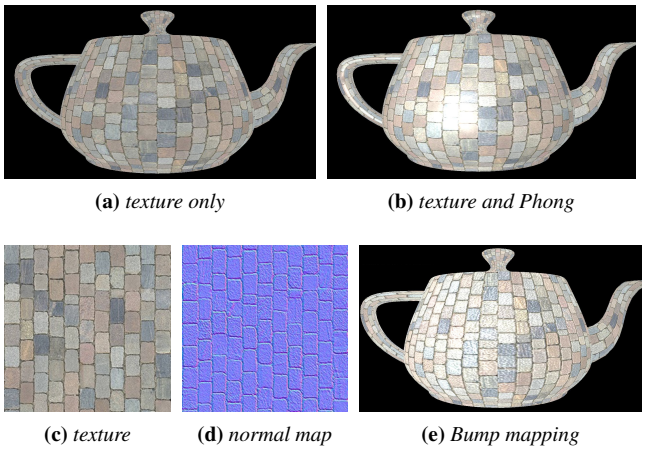
4.2.6. Task 6: Texture and render to texture

4.2.6.1. Task 6a: Texture Given that an object has defined *uv* texture coordinates, the texturing of an object can be 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.

The students' task is to apply their own texture to the test objects. It is possible to use the texture management capabilities of the framework and define a 2D texture sampler `sampler2D` object as `uniform` variable in the fragment shader. The required textures can then be set in the *User uniforms* tab. Furthermore we ask the students to apply Phong illumination from Task 3 to the result of the texture lookup. The result is shown in Figure 6b.

4.2.6.2. Task 6b: Bump mapping Bump mapping can be used to reduce geometric complexity by 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 be interpreted during the illumination step as surface normals instead of the real, from the vertex shader coming, interpolated normals. The 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. The sampled normals are also required for the Phong illumination model. The result looks like shown in Figure 6e. Figures 6c and 6d show also the used texture and normal map.

4.2.6.3. Task 6c: Render to Texture This task leads the student from simple object texturing to using textures as framebuffer,



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

(c) *texture* **(d)** *normal map* **(e)** *Bump mapping*

Figure 6: *Textured (a) and Phong shaded (b) teapot from task 6a. (c) result from task 6b. Textures obtained from [3D].*

which is essential for image-based post processing. The students are instructed to 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 and serves as an intermediate representation to allow image-based operations. The framework renders the scene first into a framebuffer. A framebuffer is basically a texture image similar to the one used in the previous exercise. However, this object has the additional capability to capture the output of the render window. This function is currently one of the most important functions in applied Computer Graphics because many different image processing algorithms can be applied to this 2D texture image as post processing step. Furthermore, fragment *gathering* and *scattering* operations are enabled through such a two-pass rendering setup.

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. The students' task is to extend the render-to-texture 2D fragment shader, so that it produces a simple blur effect.

Simple radial 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 centre is located at $c = (0.5, 0.5)$ and the vector to the image centre from any position p can be calculated by $\vec{p} = c - p$. By accumulating colour values from the input texture tex parallel to the normalized $\vec{p}$, one can define a blurred color value for the current pixel according to it's distance d to the current pixel position p :

$$rgb_{blur} = \frac{1}{n}\sum_{i=0}^{n}(tex(p + \vec{p} * d_i)), \tag{4}$$

where d can be limited to a maximum range d_{max} and sampled within this range by fixed distances s_i. Therefore, $d_i = s_i * d_{max}$. We ask the students to use predefined $n = 12$ factors s_i to determine the samples within d_{max}. For $d_{max} = 1.0$ and $s_i = (-0.10568, -0.07568, -0.042158, -0.02458, -0.01987456, -0.0112458, 0.0112458, 0.01987456, 0.02458, 0.042158, 0.07568, 0.10568)$ the resulting scene looks similar to Figure 7.

Figure 7: *Simple radial blur effect from task 6c.*

4.2.7. Task 7a: GPU ray tracing

In this exercise the students implement a very simple ray tracer. Since the render to texture shader as used in Task 6 provides already a static camera setup for rendering a screen aligned textured quad, one 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 are easy to describe analytically: *planes* and *spheres*. Note that the student can deactivate the initial three per-polygon shaders of the pipeline, since their calculations will not affect the output. For this task we provide an example scene and basic shader setup, which can be downloaded. In this example, the render-to-texture 2D (R2T) vertex shader already defines positions and rays in camera direction for a 16:10 aspect ratio and the R2T Fragment Shader defines a simple scene. Without any implementation the render window will show a statically black render window.

The students' task is to implement the ray tracing algorithms in the R2T Fragment Shader. Since recursions are not allowed in GLSL, one needs to follow every ray until it reaches a maximum ray tracing depth or until it leaves the scene. For simple ray tracing, one 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. We also ask the students to compute shadows by using additional *shadow rays*. One can do this calculation in a separate `computeShadow(in Intersection intersect)` function. When computing shadow rays, we advise the students to 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 in a provided `rnd()` function to avoid numerical problems. The plane intersection should additionally vary the ray hit color, so that a checkerboard pattern results. A correct ray tracing implementation is able to produce an image similar to Figure 8.

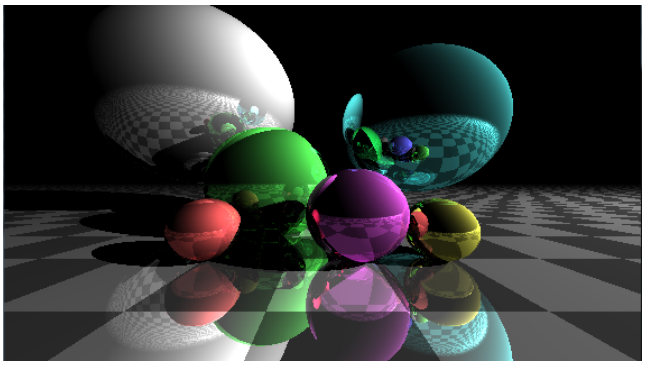


Figure 8: *Example result from simple geometric ray tracing from task 7.*

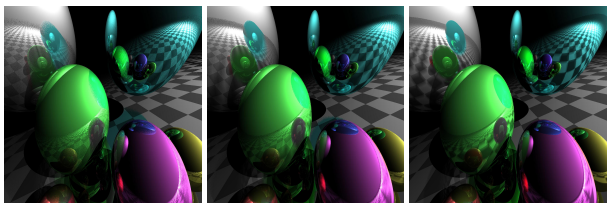
The students also need to implement mouse based scene interaction as they have been using it throughout the exercises. One can do this with the provided `uniform` matrices `projMat` and `modelViewMat`.

A fully correct implementation of the above described ray tracing algorithm will yield 75% of the maximum points for this task. The remaining 25% can be achieved by extending this exercise with their own ray-tracing extensions. For example one could implement transparent and refracting objects, light scattering effects, caustics, soft shadows, etc.. It is up to the students which extension they choose.

4.2.8. 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 the students can implement as many path tracing features as they like.

Figure 9 shows an example for Monte-Carlo Path tracing with different numbers of secondary path rays.



(a) *5 path rays* **(b)** *15 path ray* **(c)** *50 path rays*

Figure 9: *Example result from Monte-Carlo ray tracing with a different number of Monte-Carlo rays.*

5. Discussion & Conclusions

We have discussed an introductory Computer Graphics course and focused on a comprehensive lab exercise for undergraduate students. The provided coursework framework is open source and can be downloaded from *github.com*: https://github.com/bkainz/ShaderLabFramework. Our Computer Graphics course aspires to make high level graphics programming as accessible as possible for students with little to no previous knowledge about graphics APIs and limited programming knowledge. The discussed ShaderLabFramework IDE is able to hide complexity of modern graphics libraries and motivates students with an appealing and interactive development environment.

In future work we will evaluate the impact of using the presented teaching framework in class based on the achieved exam results and with our institution's central Student Online Evaluation tool (SOLE), which is a survey system collecting student opinions about every course.

References

[3D] 3D WAYFINDER: Normal maps in 3d wayfinder in webgl. http://3dwayfinder.com/bumps-and-dents-in-3d-with-normal-maps/. Accessed: 18-12-2016. 7

[Bai] BAILEY M.: Glman. http://web.engr.oregonstate.edu/~mjb/glman/. Accessed: 26-02-2017. 1

[Dig] DIGIA: Qt | cross-platform software development for embedded and desktop. https://www.qt.io/. Accessed: 18-12-2016. 1, 2

[FWW13] FINK H., WEBER T., WIMMER M.: Teaching a modern graphics pipeline using a shader-based software renderer. *Computers & Graphics 37*, 1–2 (Feb. 2013), 12–20. 2

[Gay14] GAY W.: *Mastering the Raspberry Pi*, 1st ed. Apress, Berkely, CA, USA, 2014. 1

[JQ13] JEREMIAS P., QUILEZ I. N.: Shadertoy: Live coding for reactive shaders. In *ACM SIGGRAPH'13 Computer Animation Festival* (2013), ACM. 2

[Kra08] KRAMER M.: Shader Maker – a cross-platform GLSL editor. http://cgvr.cs.uni-bremen.de/teaching/shader_maker/, 2007-2008. 1

[Lam16] LAMBERS M.: Lowering the Entry Barrier for Students Programming Virtual Reality Applications. In *EG 2016 - Education Papers* (2016), EG Association. 2

[RRP00] RAAB J., RASALA R., PROULX V. K.: Pedagogical power tools for teaching java. *SIGCSE Bull. 32*, 3 (2000), 156–159. 2

[Sta13] STAGNER A. R.: *Unity Multiplayer Games*. Packt Publishing, 2013. 1

[uni17] UNITY3D.COM: *Unity - Game Engine*, 2017. 1

[van95] VAN ROSSUM G.: *Python tutorial*. Report CS-R9526, Apr. 1995. 1

[Wol11] WOLFF D.: *OpenGL 4.0 Shading Language Cookbook*. Packt Publishing, 2011. 2