School of Computing: assessment brief

Module title	Advanced Rendering
Module code	COMP5892M
Assignment title	Assignment 1
Assignment type and description	Programming assignment: Ray Tracing and Rasterization basics.
Rationale	You will be implementing the two most common approaches to rendering 3D scenes, and applying most of the foundational content from this module. Completing this assignment demonstrates achieving your planned learning outcomes for this module.
Page limit and guidance	Assessed via in-person demo with quiz. This is a "pass/fail" assignment where you will receive the full marks if you complete all tasks successfully. You will obtain feedback during lab classes and have multiple opportunities to complete the assignment via the in-person demo and Q&A session up until the deadline.
Weighting	50%
Submission deadline	11/12/2024 (last timetabled lab)
Submission method	In-person demo *during scheduled lab hours*
Feedback provision	Oral feedback
Learning outcomes assessed	LO1-LO6 (basic level)
Module lead	Rafael Kuffner dos Anjos & Markus Billeter

1. **Assignment guidance**

   This assignment is divided into two parts, a raytracing and a rasterisation component. Please see the individual description of tasks for each one.

2. **Assessment tasks**

   There are two blocks of tasks to be completed on separate code bases: CPU-based Raytracing (with a support framework in OpenGL), and Rasterisation with Vulkan.

3. **General guidance and study support**

   Consult the Minerva page to obtain the slides for this module, and references in the reading list. Your main source of support for this assignment is the laboratory sessions. There are two formative feedback occasions.

4. **Assessment criteria and marking process**

   Assessment takes place during the scheduled labs, thus, the deadline for completing this assignment is the last scheduled lab hour. However, it is strongly encouraged that you take an iterative approach to this, work your way through the tasks as quickly and regularly as possible, and attempt demos early. Start working on the assignments as early as possible. If you have a problem that you have been unsucessful in debugging/solving yourself and do not know how to progress, ask for support during the labs at the earliest opportunity, and we will help you continue.

   Demo: When you believe you have completed all the exercises for one of the set of tasks (e.g., raytracing or rasterization), inform the module team, and we will commence the in-person demo. Here, we verify that you have completed everything to specification by inspecting your working implementation. If there are any problems, we will inform you so you can continue working and fix the issues.

   Q&A: We will ask you a few questions about your code that may include explanation about what you did, theoretical questions about the material you had to use to implement it, and requests to change some functionality. The goal of this Q&A session is to verify that you understand the code thoroughly. If you fail to respond to any questions, we will stop the Q&A and give you time to further study. Expect questions related to any one of tasks and the surrounding theory..

5. **Submission requirements**

   After each demo and associated Q&A has been completed, you will be given a "digital receipt" that you have been assessed.

   Further, you are required to submit your solutions online:

   - Part 1 (raytracing): submit a zip (.tar, .7z, .rar are not supported by Gradescope!) of your src folder to the gradescope submission point.

- Part 2 (rasterisation): Submit code to Gradescope. You can either use Gradescope's built-in support to submit code hosted on Github or Bitbucket, or submit via a .zip file (do not upload files individually, as this will strip the directory information from them!). If submitted successfully, Github will show you a list of individual files. Additionally, it will run a set of automated tests to verify the correctness of your submission. *Make sure you check the output from Gradescope and fix problems as reported! Submissions that do not pass critical tests may receive zero marks!*

You should make sure to submit a "clean" solution. It only include files necessary for your solution. This includes source code, shaders (if any), third party software (including license statements) and so on. You should under no circumstances submit generated/temporary files, artefacts from the build, caches from your development environment or similar. (Makefiles and/or VisualStudio project files are considered generated files, so they should not be included!)

When submitting to Gradescope, there are a few limitations. Specifically, uploads via a zip archives have a 255 file limit and some (ever-changing) file size limits. In Part 2 you are likely reach some of these limits. You can therefore omit the following (and only these!) in your submission: the `third_party` folder and the `assets-src` folder. (The `assets` folder contains mostly generated files, so those should not be submitted by default!)

6. **Presentation and referencing**

   Questions will be asked and answered in English. Comments on your code must be written in English.

7. **Academic misconduct and plagiarism**

   All of the code submitted will be verified using plagiarism detection software. You are not allowed to submit code that was not written by you. In this assignment, it is not allowed to use any external sources for the code you submitted, even with references. All the submitted code must be written from the start by you.

   The Q&A sessions will be also used to verify that you wrote the code yourself, and that you are familiar with all aspects of it. If any member of staff suspects of plagiarism at this point, you will be warned at this point and your submission will be closely investigated after submission to Gradescope.

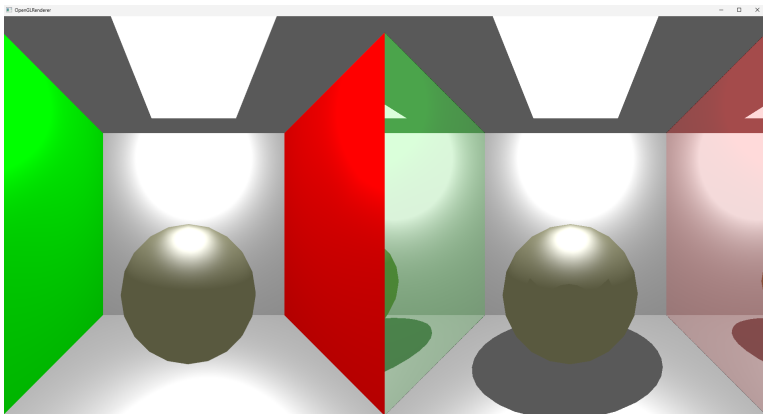
   While you are encouraged to use version control software/source code management software (such as git or subversion), you must not make your solutions publicly available. In particular, if you wish to use Github, you must use a private repository. You should be the only user with access to that repository.

8. **Assessment/marking criteria grid**

   50 points: Everything works, Q&As completed successfully, code submitted successfully.

Part 1: Raytracing

Contents



This is the raytracing part of the assignment that needs to be completed. The starting point for this part is the end of the Raytracing exercise which is a fully guided tutorial. If you haven't so, please do it now. The final goal is to have a working raytracer with phong shading, hard shadows, and mirrors implemented. This will be the output of correctly implementing the tasks described ahead

1 Task 1: Transformations

Your first task will be finishing the code that will perform the transformations. Implement a function that should return your modelview Matrix in the Scene class (`getModelview()`). This function is used on `updateScene▽ ▷ ()`, which is called when you call the `Raytrace()` function.

Does it work: Use the debugger to verify that the matrices are being updated accordingly when you use the interface. It will however be easier to tell in the when you are able to see things.

2 Task 2: Casting a ray

Implement a function in the Raytracer class that given a pixel position, casts a ray towards the scene.

```
Ray calculateRay(int pixelx, int pixely, bool perspective);
```

Implement this function by following the process described on Lecture 4, slides 34-38. Call it inside your main raytracing loop, before calculating a colour for a given pixel.

Does it work: Use the breakpoints to see if the coordinates of your rays match the edges of your image plane. Those should be similar to what you have as the Left,Right,Bottom,Top from the left side. It will however be easier to tell in the next step when you are able to see things. Try it with a simple model such as triangle_backplane where you can do easy maths.

3 Task 3: Geometric Intersections

Calculate geometric intersections between camera rays and triangles in the input model. Use an auxiliary struct "CollisionInfo" to contain the intersection information, and implement the following function (starting point below).

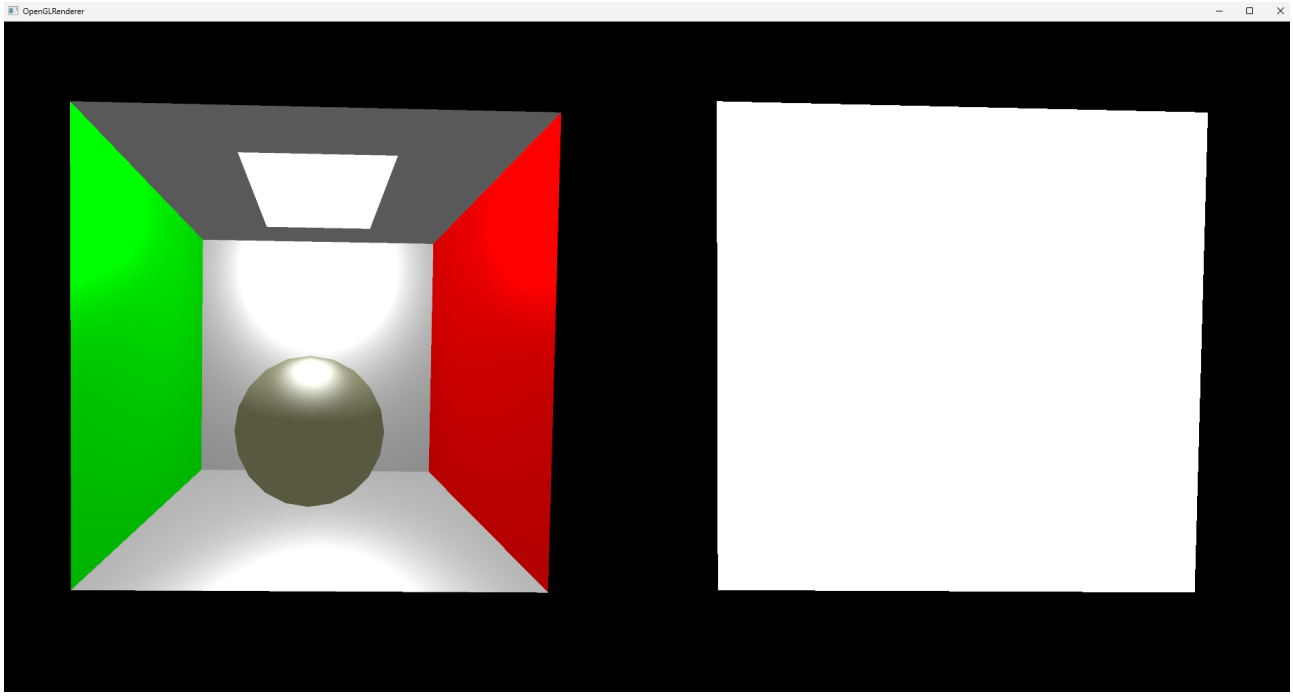
```
//scene.h
 struct CollisionInfo{
        Triangle tri;
        float t;
    };

CollisionInfo closestTriangle(Ray r);

//scene.cpp
Scene::CollisionInfo Scene::closestTriangle(Ray r)
{
    //TODO: method to find the closest triangle!
    Scene::CollisionInfo ci;
    ci.t = r.origin.x; // this is just so it compiles warning free
    return ci;
}
```

This function should iterate the list of triangles in the current scene, and test for intersection with them. The closest triangle given a ray will have the smallest t (See Lecture 3, Slides 35-38). To calculate intersections, implement a method in your Triangle Class.

```
float intersect(Ray r);
```

Follow the method described in the slides, and return the calculated t for a given intersection, as we can calculate the point of intersection given a ray's origin, direction, and t. We are only interested in cases where $t > 0$, as negative t would mean an intersection behind the camera. Use this to your advantage to encode "no collision" cases with a negative t value. To verify if the intersection point with the plane formed by the triangle is inside the triangle (thus, a valid intersection) implement the half plane test. (See Lecture 3, Slides 41-44).

Does it work: Call closestTriangle inside your main raytracing loop. If $t > 0$, assign the color white to the pixel. Use the mouse and keyboard to move the camera and objects in order to verify that your transformations are matching with the openGL renderer. Here is an example



4 Task 4: Barycentric Interpolation

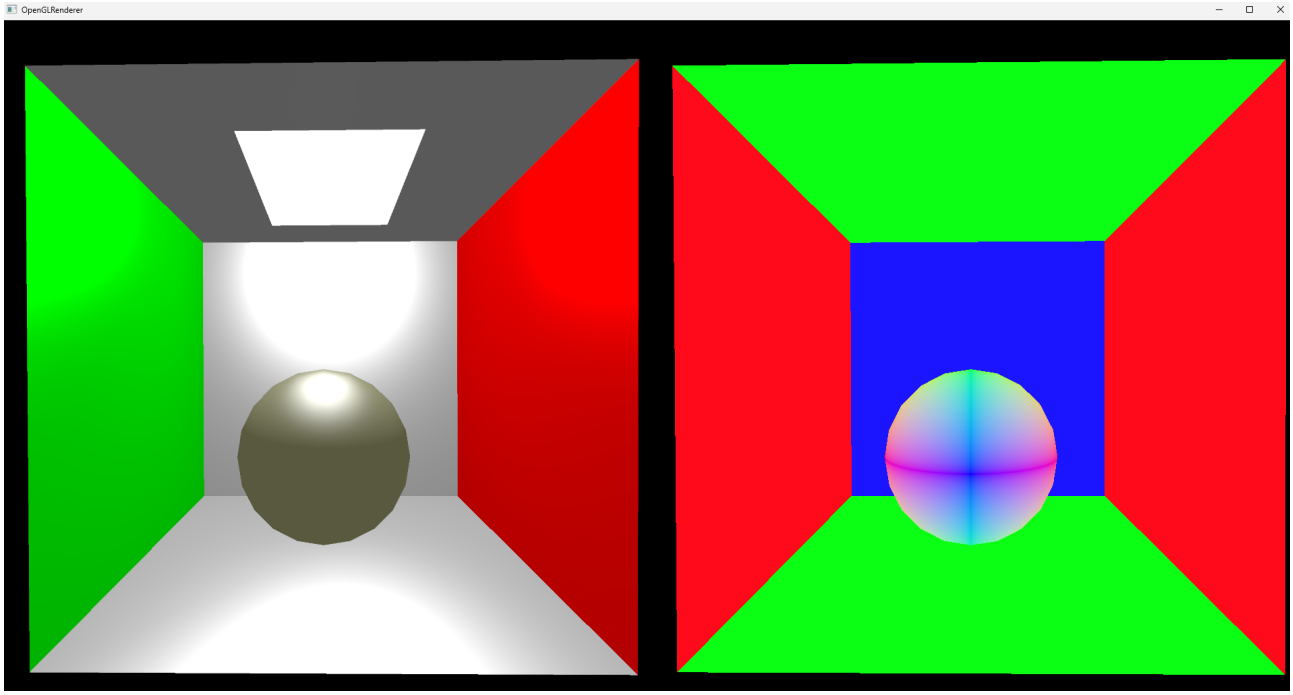
Implement barycentric interpolation to obtain the barycentric coordinates at the position of the intersection calculated in the previous task. Add a function to your triangle class that returns the "alpha, beta, gamma" as a Cartesian3. Here is a starting point

```
//add to .h
Cartesian3 baricentric(Cartesian3 o);
//add to .cpp
Cartesian3 Triangle::baricentric(Cartesian3 o)
{
    //TODO: Input is the intersection between the ray and the triangle.
    //o = origin + direction*t;
    Cartesian3 bc;
    bc.x = o.x ; // Just to compile warning free :)
    return bc;
}
```

Does it work: Use this function inside your main loop. Use the barycentric coordinates to calculate the normal vector for that point. Then set the color of the output pixel as the following, which should show the value of the normal if we have the checkbox "interpolation" checked:

```
if(renderParameters->interpolationRendering)
     return Homogeneous4(abs(normOut.x),abs(normOut.y),abs(normOut.z),1);
```

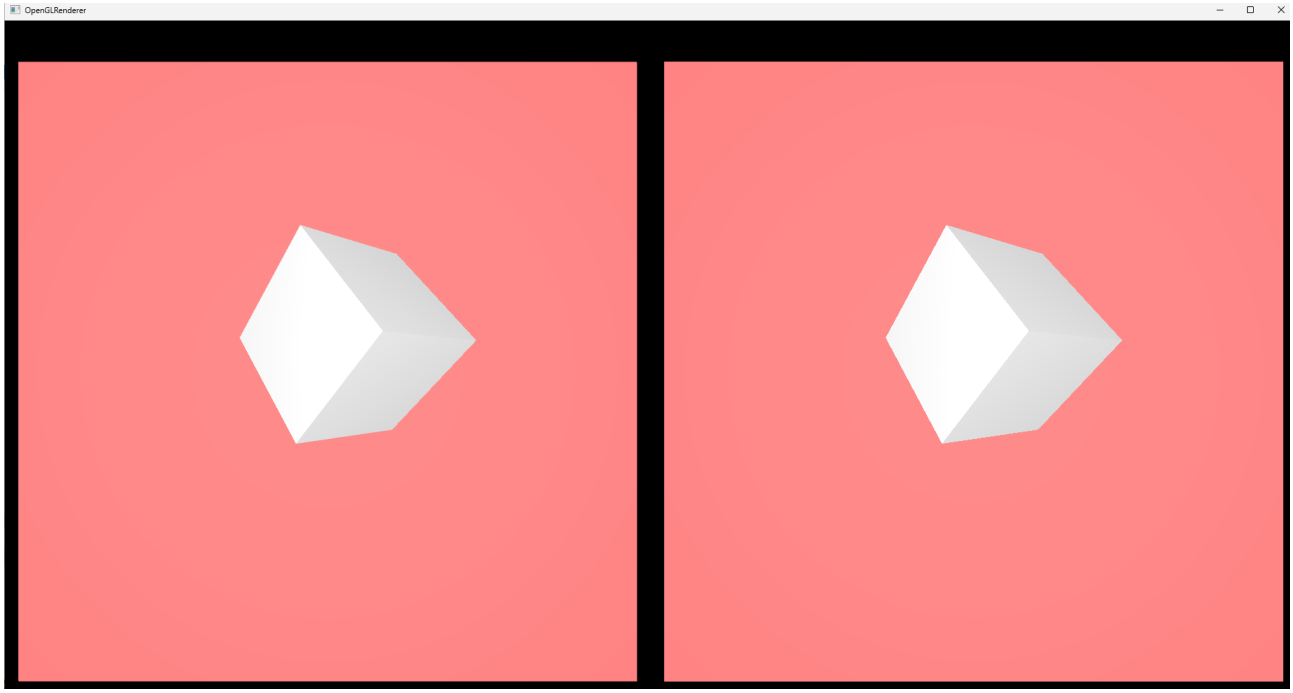
Here is an example:



5 Task 5: Blinn-Phong Shading

Calculate Blinn-Phong shading considering every light present in the scene. Use the lights in the vector that you populated in the tutorial, on the RenderParameters class. Remember to apply the modelview matrix to them as well, as they are tied to a physical object in the scene and these are subject to the mouse and keyboard controls. Implement the code to calculate the physically correct phong shading in a function of your Triangle class (Lecture 4, Slides 7-19, or the shader code for the rasterisation). Parameters should be the light position, light colour, and the intersection barycentric coordinates. Use the material properties for that given triangle to calculate the shading. Return the final colour as Homogeneous4 and apply it to the pixel.

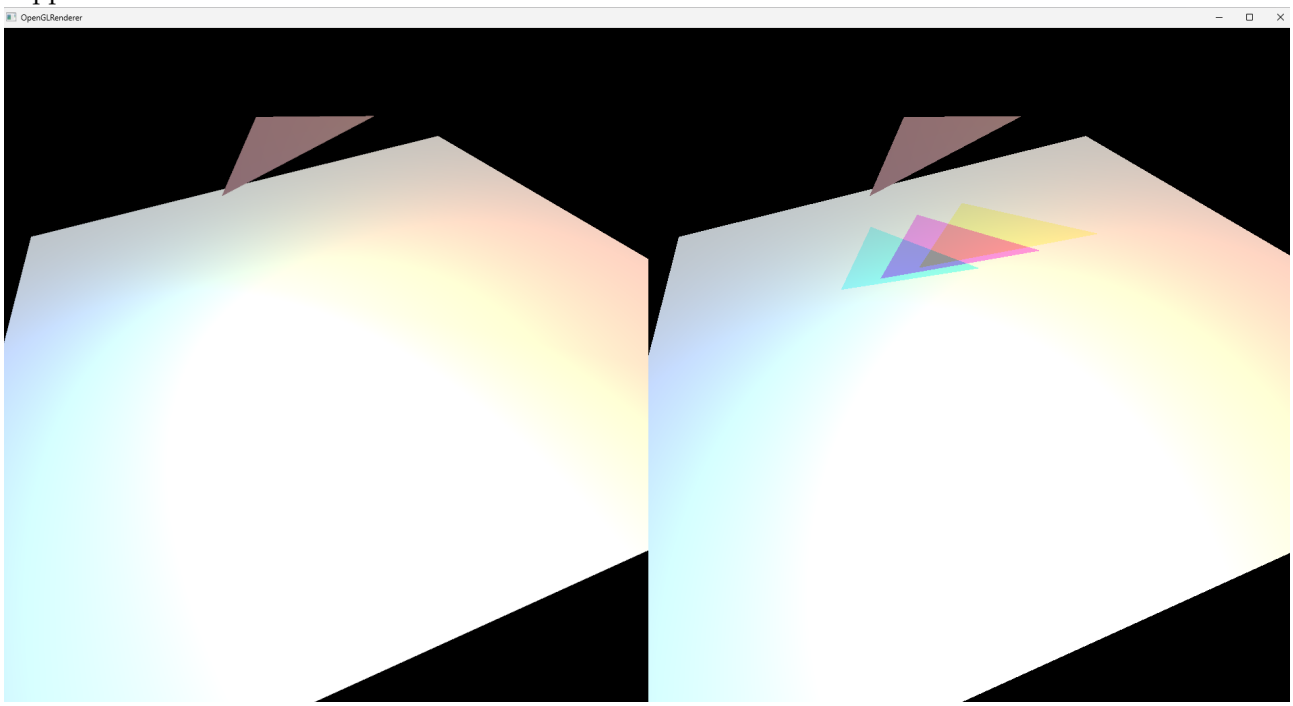
Does it work: Compare it to the shading that happens on the left side. Change the mtl file of objects you try to see if your implementation responds properly to it.



6 Task 6: Shadow Rays

Include shadows in your shaded result, also considering every light source in the scene. Follow the instructions on the slides (Lecture 5, slides 4-8). Use the `closestTriangle()` function you implemented before to perform intersection tests, and the t value to check if there was a valid intersection. Do also check that you are not intersecting with the light geometry itself (easy by checking the material). Pay attention to shadow acne, correctly displacing the starting point of your shadow ray. Adjust your shading function to only apply ambient and emissive colour if the object is in shadow.

Does it work: Move the object around, zooming in, and using different angles, ensuring shadow acne never happens.



7 Task 7: Impulse Reflection

Use the material properties to verify if the object is mirrored. If it is, calculate the reflected value which should be combined with the object's colour accordingly (values are 0 to 1). This will require you to change your raytracer into a recursive function. Move out all of your shading code after calculating the ray out of the main raytracing loop into a separate function:

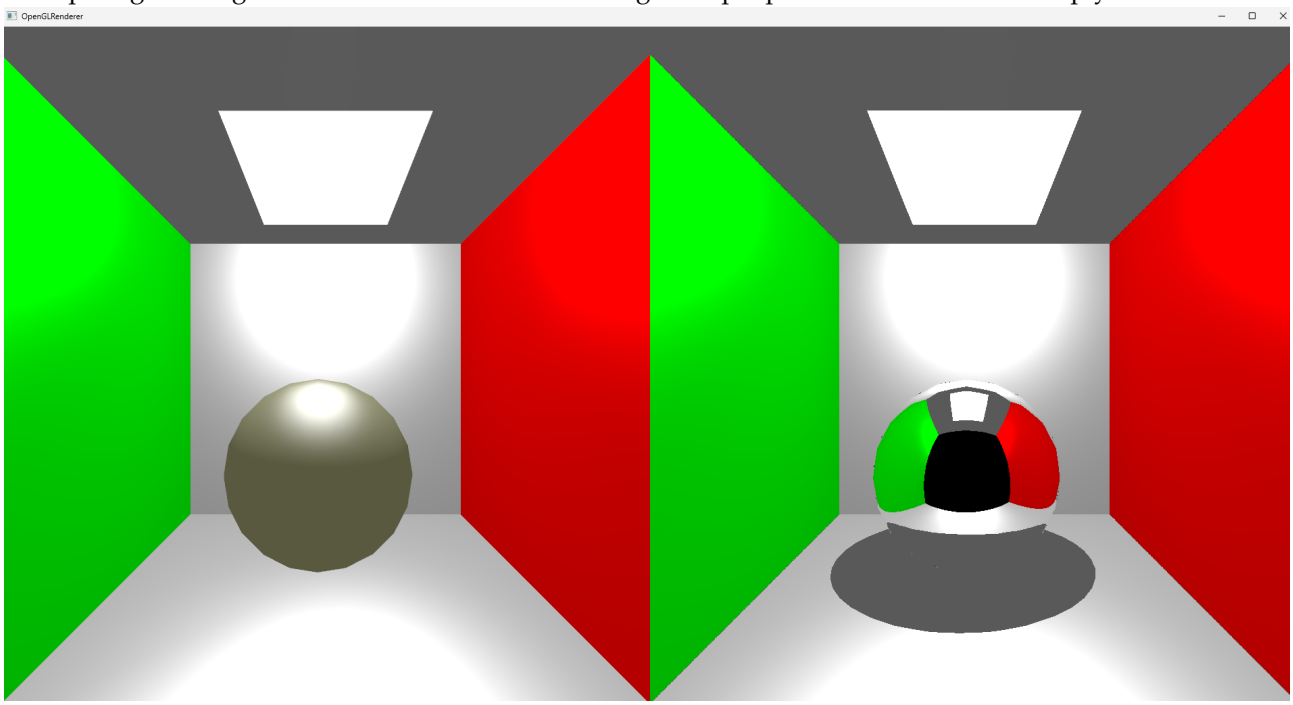
```
Ray r = calculateRay(i,j,!renderParameters->orthoProjection);
Homogeneous4 color = TraceAndShadeWithRay(r,N_BOUNCES,1.0f);
//... applying to pixel, gamma correction etc
```

Where N_BOUNCES is defined in your .h file as the maximum number of recursions allowed in your raytracer. An object is reflective if the mirror property in the material is different than 0. If that's the case, you should find the color at the end of this reflection. Implement a "reflectRay" function that reflects a ray according to a surface normal.

```
reflectRay(r,normal,hitPoint);
```

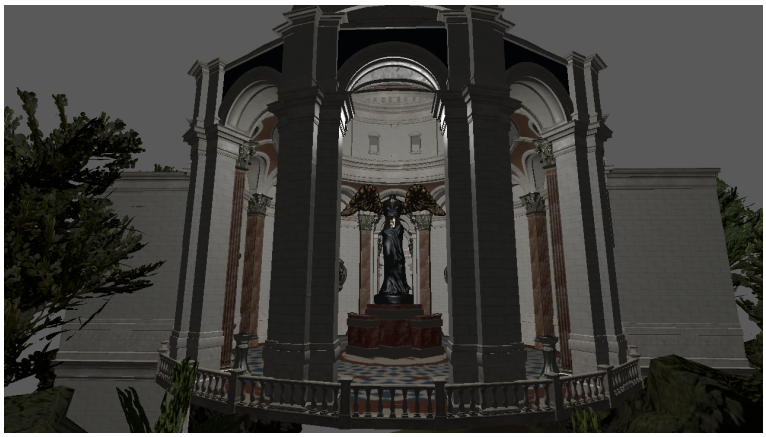
And recursively call TraceAndShadeWithRay, as described in the slides (Lecture 5, slides 10-13). In the material model we are using in this assignment, the "mirror" property in the material file is a floating-point value meaning how much of the total energy should be from the reflected ray. Use this to linearly combine the resulting color of the mirror ray, and the color of the current surface, making sure that no energy is lost, or added to the system.

Does it work: Change the material properties to verify that when mirror = 1.0, all you see is a reflection. When mirror = 0.0 you should not see any reflection. Anything else should be a combination of colours from phong shading and the reflection. This following example plus the teaser should help you understand.



Part 2: Vulkan

Contents



Part 2 revolves around a basic renderer using the Vulkan API with physically-inspired shading for lighting. In this part, you will render the UE4 Sun Temple model. The model is sourced from NVIDIA ORCA and is available under a CC-BY-NC-SA license. It contains about 1.6M vertices. The original model comes in the FBX format with DDS textures. It has been converted to OBJ with PNGs for this assignment. The OBJ model is compressed with ZStandard compression, which the baking software (Task 1.1) handles for you. This reduces the OBJ file size from ~180MB to around 20MB.

If you have not completed the Vulkan exercises, it is highly recommended that you do so before attacking this part. When requesting support, it is assumed that you are familiar with the material demonstrated in the exercises! As noted in the module's introduction, you are allowed to re-use any code that ***you have written yourself*** *for the exercises in the assignment. It is expected that you understand all code that you hand in, and are able to explain its purpose and function when asked.*

1 Tasks

Part 2 of this assignment includes two applications. The first one, `a12-bake`, is a non-graphical tool that reads an OBJ file and "bakes" it into a simple binary format. The second one is the renderer, which you may build around the same code-base that you are familiar with from the tutorials.

1.1 "Baking"

Study the baking application. It reads an OBJ file and converts the resulting triangle soup into an indexed mesh. It then writes a simple binary file with the necessary data. In essence, the generated file contains three regions: a list of textures, a list of materials (with references to the textures), followed by a list of meshes. Each mesh refers to a material and includes an array of vertex positions, an array of normals, an array of texture coordinates and an array of indices. The exact file format is documented in the code (see both `a12-bake/main.cpp` and `a12/baked_model.cpp`).

Build and run the application convert the coursework's OBJ file to the binary runtime format that you will use in the following tasks. The source OBJ files are located in the `assets-src/a12` directory; the 'baked' output will be placed in the `assets/a12` directory. The application also copies used textures to this directory.

Verify that the outputs were successfully created in the `assets/a12` directory.

The application will not overwrite existing textures – if it is re-run, it will report that it failed to copy the textures. This is on purpose (to avoid overwriting existing files).

1.2 Prep and Debug

Start by setting up the necessary Vulkan rendering infrastructure for a real-time rendering application. You can reuse your own solutions from the Vulkan Exercises to get the following:

- Creating a Vulkan instance
- Enabling the Vulkan validation layers in debug builds
- Creating a renderable window
- Selecting and creating a Vulkan logical device
- Creating a swap chain
- Creating framebuffers for the swap chain images†
- Creating a render pass†
- Repeatedly recording commands into a command buffer and submitting the commands for execution

You must also implement any necessary synchronization. You should use the `FIFO` presentation mode and ensure that the application waits for V-sync. Your application must allow the window to be resized and handle the resizing appropriately (i.e., it must recreate the swap chain and related resources when necessary). If you can draw anything, including a solid color, on the screen, you have likely completed this task.

†You are allowed to use Vulkan 1.2 or Vulkan 1.3 functionality or equivalent extensions (specifically, imageless framebuffers or dynamic rendering) instead. In this case these two steps change slightly. If you decide to go down this route, make sure to mention this during your demo session!

1.3 3D Scene and Navigation

Extend your application to load the baked model data from the binary file produced in the baking step (Task 1.1) and render the corresponding 3D scene. Note that the binary format uses indexed meshes instead of triangle soups, so you must use `vkCmdDrawIndexed` instead of `vkCmdDraw`. You must additionally provide the indices via an index buffer (`vkCmdBindIndexBuffer`).

The model is textured with textures provided in the PNG format (`.png`). The `stb_image.h` library introduced in the exercises can load PNG files (as well as JPGs). Texture sampling should use trilinear (=mipmapped) filtering. Generate mipmaps on the fly, using Vulkan.

Import the "first person" camera shown in Exercise 2.5. With it, you (and your users) will be able to navigate the scene, which will be important for debugging. The first-person camera is controlled with the keyboard and mouse. Control position/movement with the WSAD and EQ keys:

- `W` - forward
- `S` - backward
- `A` - move/strafe left
- `D` - move/strafe right
- `E` - move up
- `Q` - move down

(directions relative to the camera's orientation).

Movement speed should be increased by a constant factor when holding the Shift-key, and decreased by a constant factor when holding the Ctrl-key. Use reasonable default speeds.

The camera is also for your use during development. Generally, you want to be able to move across the whole scene in about 5 or so seconds with shift pressed. Slow movement (ctrl pressed) is useful for inspecting something up close (without accidentally moving through a surface). Make sure you pick reasonable settings.

Use the mouse to control the camera's orientation (think first-person shooter camera). Mouse navigation should be activated when the right mouse button is clicked and deactivated again when it is clicked a second time.

Movement speed should be independent of the frame rate. Camera rotation speed should be determined by the magnitude of the mouse movement. Use GLFW's event driven framework with callbacks to receive input. Avoid polling keyboard/mouse state.

For this task, implement a renderer that can draw the model with textures and has a usable camera for navigation. See Figure 1 for renders of the scene with textures but without illumination.

Figure 1: *View of the Sun Temple scene without textures. Note that the scene is created with the interior in mind. The outside is not textured completely.*

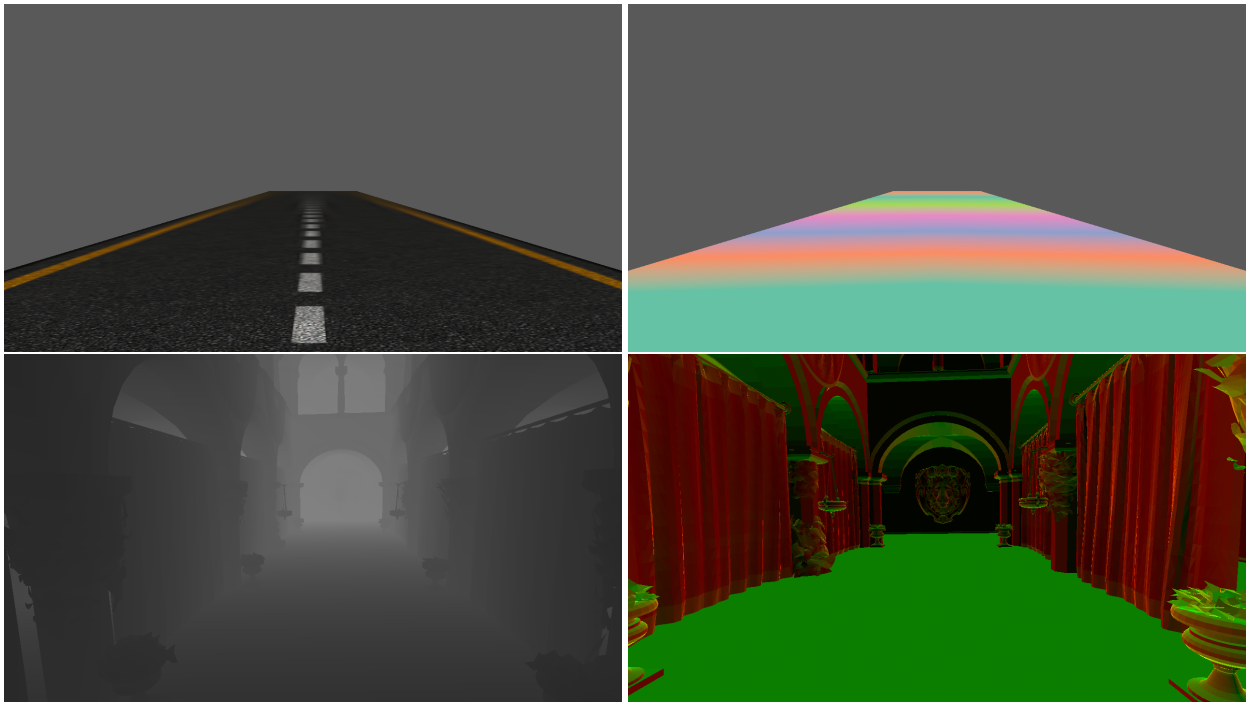


Figure 2: *Top-Left: Textured plane with mipmap filtering. Top-Right: Visualization of the mipmap levels. The color indicates which mip-levels were used when sampling the texture. Bottom-Left: Linearized depth. Bottom-Right: Partial derivatives of the per-fragment depth (scaled to be easily visible). You should implement this for the Sun Temple scene used in the assignment (not for the road/Sponza shown here)!*

1.4 Debug visualization

Implement a set of rendering modes that lets the user visualize properties related to fragments. Specifically, you should visualize the following:

1. Utilization of texture mipmap levels
2. Fragment depth
3. The partial derivatives of the per-fragment depth

You will need to find appropriate ways of visualizing the above properties, which includes mapping of the relevant quantities to colors. Figure 2 shows an example of such a visualization. For example, for mipmap levels, the rendered color displays which mip-levels were used when sampling the texture. (You will want to disable anisotropic filtering for this task.)

For the mipmap visualization, experiment with different mipmapping modes and with different values for the mipmapping bias. You can affect the bias both when configuring a `VkSampler` and through an optional parameter to the GLSL `texture` function.

Implement the tasks such that you can switch between them with the main number keys (not the numpad!). Let `1` be the default rendering mode and use `2` ... `4` select the different debug visualization modes.

You must implement the debug visualization in a separate Vulkan pipeline with separate shaders. You can use one or more pipelines for the different visualizations (your choice), but they must not be in the "main" shader.

(a) *Normal* ($\mathbf{n}$) **(b)** *View direction* ($\mathbf{v}$) **(c)** *Light direction* ($\mathbf{l}$)

Figure 3: *Visualization of the (normalized) normal vector, view direction and light direction. The vectors are visualized with their world space values. You can use the fact that red maps to the x coordinate, green to y and blue to z to verify that the displayed values make sense – e.g., do the normals point into the direction you expect at each point?*

1.5 PBR Shading

First, decide which space you want to perform your shading computations in. The document will refer to this space as the *shading space*. The recommendation is to perform shading in world space, but you can pick a different space if you wish (make sure you've studied all tasks before deciding). All shading should be done per fragment, we will not perform any per-vertex shading.

Make sure you get the necessary data into the right place.

- Make sure you pass the normals through the vertex shader to the fragment shader. (You may continue to assume that the models are defined in world space, and hence omit the model-to-world transform.)
- Make sure that the fragment shader has access to the fragment's position in the shading space.
- The fragment shader will need to have access to the camera's position in the shading space. (Note: you can just extend the per-scene transform information to include the camera's position and make sure it is accessible in both the `SHADER_STAGE_VERTEX` and `SHADER_STAGE_FRAGMENT` shader stages.)
- The fragment shader will need to have information about the scene's lighting data (e.g., a per-scene ambient light value and information about one or more light sources. Light sources are defined by their position and color).
- The fragment shader will need to get the correct material information.

For now, place the light source at the coordinates $[-0.2972, 7.3100, -11.9532]$ in world space, and give it a color of $[1, 1, 1]$. Screenshots in this document will use those settings. Information about the light must be passed to the relevant shaders using an uniform buffer/interface block (i.e., do not hard-code the light's properties in the shaders).

Decide on a reasonable setup with descriptor sets and descriptor bindings. Introduce additional uniform buffer objects as necessary. Recall the `std140` layout rules (see lecture slides if you need a quick refresher).

It is a good idea to verify that things are working as they should. Change your shader to visualize the per-fragment normals, view direction and light direction. Make sure the values behave as expected (e.g., should they change with the camera position or not?). You can compare your results to the screenshots in Figure 3.

You will now implement a physically-inspired model that has a Lambertian diffuse component and a specular component based on a microfacet BRDF using the Beckmann normal distribution function.

To introduce the model, we will start with the Rendering Equation, as shown in the lectures:

$$\mathbf{L}_o = \mathbf{L}_e + \int_\Omega f_r \, L_i(\omega) \, (\mathbf{n} \cdot \mathbf{l}) \, d\omega$$

We are dealing with discrete point lights, which are the only points in space that emit light, and ambient light. Consequently, we can sum over the light sources instead of integrating over a hemisphere (where the ambient light approximates all indirect illumination):

$$\mathbf{L}_o = \mathbf{L}_e + \mathbf{L}_{\text{ambient}} + \sum_{n=0}^{N-1} f_r \, \mathbf{c}_{\text{light},n} \, (\mathbf{n} \cdot \mathbf{l}_n)_+$$

For now, we'll focus on a single light source, meaning we can drop the sum (and the index n) all together:

$$\mathbf{L}_o = \mathbf{L}_e + \mathbf{L}_{\text{ambient}} + f_r \, \mathbf{c}_{\text{light}} \, (\mathbf{n} \cdot \mathbf{l})_+ \tag{1}$$

The general form of the equation is already somewhat reminiscent of the modified Blinn-Phong model introduced earlier. In short, the BRDF (f_r) describes how much of the incoming light ($\mathbf{c}_{\text{light}}$) is reflected towards the camera/viewer.

This model relies on a set of material parameters:

- α Beckmann roughness, which is equal to texture_roughness^2
- M Material metalness
- $\mathbf{c}_{\text{emit}}$ Material emissive color - you may assume this is zero in Assignment 1
- $\mathbf{c}_{\text{mat}}$ Material base color
- $\mathbf{c}_{\text{light}}$ Light color
- $\mathbf{c}_{\text{ambient}}$ Scene ambient light color
- $\mathbf{n}$ Surface normal (normalized)
- $\mathbf{l}$ Light direction (normalized), pointing *towards* the light
- $\mathbf{v}$ View direction (normalized), pointing *towards* the camera/viewer
- $\mathbf{h}$ Half vector (normalized), computed from the light and view directions

Important: The roughness used by the Beckmann NDF, α, is equal to the roughness value from the texture squared.

The baking software includes information about emissive textures. This will be used in Assignment 2. You can ignore it for now and just assume that all materials have zero emissive.

For the BRDF, we will use a general microfacet model for isotropic materials **[?]**:

$$f_r(\mathbf{l},\mathbf{v}) = \mathbf{L}_{\text{diffuse}} + \frac{D(\mathbf{n},\mathbf{h})\ \mathbf{F}(\mathbf{l},\mathbf{h})\ G(\mathbf{n},\mathbf{l},\mathbf{v})}{4\ (\mathbf{n}\cdot\mathbf{v})_+\ (\mathbf{n}\cdot\mathbf{l})_+},$$

where $\mathbf{L}_{\text{diffuse}}$ is the diffuse contribution, and the specular contribution is constructed from the Fresnel term F, the normal distribution function D (Figure 4a) and the masking function G (Figure 4b).

Here, $(\mathbf{a}\cdot\mathbf{b})_+$ denotes a "clamped" dot-product, $(\mathbf{a}\cdot\mathbf{b})_+ = \max(0,(\mathbf{a}\cdot\mathbf{b}))$. Furthermore, the $\otimes$ operator will be used to denote element-wise multiplication of two vectors/tuples.

Some care must be taken if one attempts to evaluate the above term directly. The denominator can become zero, which, in practice, produces a NaN value. This then cascades through the following computations. A simple workaround is to add a small ϵ' to the denominator before the division. Alternatively, one can try to cancel out some of the terms in the division (e.g., compare to Equation (1)), and potentially avoid problems in the first place.

In theory, we need to differentiate between metals and dielectrics (non-metals), as these behave quite differently. Metals only reflect on the surface, meaning that they have a zero diffuse contribution. Additionally, the (specular) reflection from a metal is tinted. In contrast, dielectrics have both a diffuse and specular contribution, where only the diffuse one is colored. The specular reflection tends to have the same spectrum/color as the incoming light.

In practice, it is possible to merge the two cases into a single approximative method. The underlying idea revolves around the observation that the amount of specular base reflectivity. F_0, of dielectrics is *relatively* similar for most materials. The value 0.04 is frequently used as an approximation across the board. For dielectrics, the material's color then determines the diffuse color. For metals, the material's color is instead used to control specular base reflectivity:

$$\mathbf{F}_0 = (1-M)\ [0.04, 0.04, 0.04] + M\ \mathbf{c}_{\text{mat}}$$

M describes the *metalness* of the material. In reality, a material is either a metal ($M = 1$) or a dielectric ($M = 0$). However, in computer graphics, we may encounter cases where this is not true – for example, when the material properties of a metal and dielectric are interpolated between. The above formula deals with non-binary metalness.

The amount of specular reflection is given by the Fresnel term F, which we evaluate using the Schlick approximation:

$$\mathbf{F}(\mathbf{v}) = \mathbf{F}_0 + (1-\mathbf{F}_0)\ (1-\mathbf{h}\cdot\mathbf{v})^5.$$

For the diffuse term we will just use a simple Lambertian. Only light that that wasn't reflected specularly will participate in the diffuse term (hence the $\mathbf{1}-\mathbf{F}$ term). Additionally, as previously mentioned, metals have a zero diffuse contribution. We model this with the $1-M$ term:

$$\mathbf{L}_{\text{diffuse}} = \frac{\mathbf{c}_{\text{mat}}}{\pi} \otimes ([1,1,1] - \mathbf{F}(\mathbf{v}))\ (1-M).$$

(a) *NDF* **(b)** *Geometry term*

Figure 4: *Illustration of the D and G components of the microfacet model. The NDF, $D(\omega)$, describes the distribution of microfacets. It specifically tells us the density of facets that are oriented such that their normal points in the direction ω. The masking function $G(\omega_i, \omega_o)$ describes self-shadowing where microfacets block each other.*

(a) *Overview* **(b)** *Shield* **(c)** *Wing*

Figure 5: *Shading with the PBR model. The statue in the middle is declared fully metal. See its textures in the files* `M_Statue_Inst_0_BaseColor.jpg` *(base color),* `m-M_Statue_Inst_0_Specular.png` *(metalness) and* `r-M_Statue_Inst_0_Specular.png` *(roughness). The shields are also metallic. You can view the shield textures in the files containing the name* `*M_Shield_Inst_0*`*. Should the statue be fully metallic? You can try changing the texture to be more selective.*

More complex diffuse reflectance models exist. However, for now, the Lambertian is sufficient. Other models can be quite a bit more expensive and may only contribute with minor improvements [**?**].

For the normal distribution function D, we will use the Beckmann distribution [**?**]:

$$D(\mathbf{h}) = \frac{e^{\frac{(\mathbf{n}\cdot\mathbf{h})_+^2 - 1}{\alpha^2 (\mathbf{n}\cdot\mathbf{h})_+^2}}}{\pi \alpha^2 (\mathbf{n}\cdot\mathbf{h})_+^4}.$$

The masking term from the Cook-Torrance model [**?**] that you should use looks as follows:

$$G(\mathbf{l}, \mathbf{v}) = \min\left(1, \min\left(2\frac{(\mathbf{n}\cdot\mathbf{h})_+ (\mathbf{n}\cdot\mathbf{v})_+}{\mathbf{v}\cdot\mathbf{h}}, 2\frac{(\mathbf{n}\cdot\mathbf{h})_+ (\mathbf{n}\cdot\mathbf{l})_+}{\mathbf{v}\cdot\mathbf{h}}\right)\right).$$

For the ambient term, $\mathbf{L}_{\text{ambient}}$, you can assume a constant ambient illumination and just modulate it with the material's color:

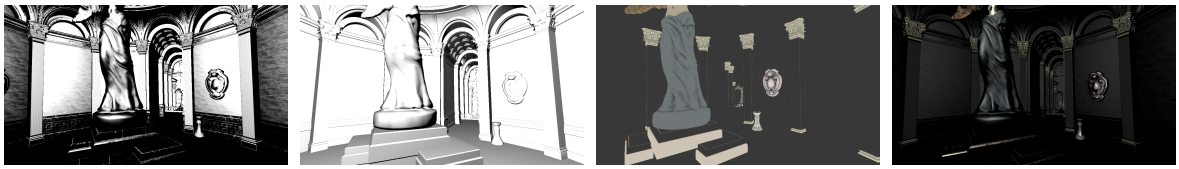
$$\mathbf{L}_{\text{ambient}} = \mathbf{c}_{\text{ambient}} \otimes \mathbf{c}_{\text{mat}}$$

Implement the shading model for one light source (Equation (1)).

Compare your output to Figure 5. If you need to debug your shaders, see Figure 6 for visualization of some of the intermediate values (also check that the various values are behaving as they should!). Make sure the shading is well-behaved when moving the camera around. In particular, pay attention to problems that may arise from dividing by (almost) zero.

The screenshots use a weak global ambient contribution of 0.02. Additionally, no falloff is applied to the light. You can experiment with the normal square-law falloff. In that case, you probably want to increase the light intensity beyond 1.0.

You must use the shading model described in the exercise. Implementing a different shading model, including GGX-based ones, will not be accepted.



(a) D **(b)** G **(c)** **F** **(d)** *PBR specular term*

Figure 6: *Visualization of the some of the components of the PBR model used in CW 2. The Fresnel term shows clearly which parts are metallic.*

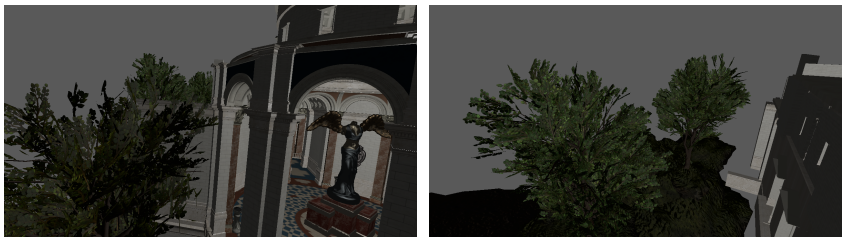


Figure 7: *Alpha-masked foliage.*

1.6 Alpha Masking

As you may have seen, the scene (teaser) contains some foliage. The foliage uses alpha-masked textures to define the high-frequency geometry.

Add a separate graphics pipeline to render materials with alpha-masked textures. The recommendation is to first render "normal" geometry with the default pipeline, and then switch to the alpha-masking graphics pipeline to render any geometry using alpha-masking. Note that foliage is two-sided. See Figure 7.

1.7 Post process

In this final task, you will implement a simple post processing filter – a simple pixelation / "mosaic". This requires setting up render-to-texture, which in turn requires changing the program structure slightly.

So far, we've been drawing directly into the swap chain images, which have subsequently been presented to the user:

(1) Acquire next swap chain image
(2) Render pass: render 3D scene to swap chain image via associated framebuffer
(3) Present swap chain image

In render-to-texture (RTT) methods, we introduce additional steps into this process:

(1) Render pass A: render 3D scene to intermediate texture image(s)
(2) Acquire next swap chain image
(3) Render pass B: perform post processing, using intermediate texture image as input and rendering results to swap chain image
(4) Present swap chain image

For some techniques, there are multiple steps in the post processing. These would function similarly to Step 3, but render to additional intermediate textures, and would likely take place between Steps 1 and 2. In this task, a single post processing step is sufficient.

This requires the following to be set up:

- Intermediate texture image(s) and associated framebuffer(s)
- Render passes A and B
- Full screen shader/pipeline that performs post processing/deferred shading.
- Descriptors, samplers, synchronization etc.

You should repurpose the render pass from previous tasks for use as render pass A, except that it should now render into a new intermediate image.

Implement an additional graphics pipeline for post processing. Draw a full screen triangle and perform any post processing computations in the fragment shader. The fragment shader will use the intermediate texture

Figure 8: *Mosaic / pixellation effect. You might need to zoom into the PDF to see the details. (The renderer also implements additional lighting not required in these tasks.)*

image drawn in the preceding steps as input. See lecture slides for an efficient way of setting up and performing the full screen pass. To draw a single triangle without vertex buffers, refer to Exercise 1.2 and/or see lecture slides.

In the post-processing fragment shader, create the mosaic effect. Make each 5×3 region of pixels use the value of one of the pixels inside this region. Ensure that the effect can be toggled on and off. See Figure 8 for an example.

Pay attention to synchronization, consider what resources (Vulkan images, framebuffers, ...) you need, and how many of each resource are required.

While it may be tempting to use subpasses, you cannot use them in this task. The pixelation post-process requires reading neighbouring pixel values, which is not possible with subpasses.

Acknowledgements

The document uses icons from https://icons8.com: , , . . The "free" license requires attribution in documents that use the icons. Note that each icon is a link to the original source thereof.