School of Computing: assessment brief

Module title	High Performance Graphics
Module code	COMP5822M
Assignment title	Coursework 1
Assignment type and description	Programming assignment: Vulkan Infrastructure and Debugging/Analysis
Rationale	COMP5822M introduces modern projective graphics through Vulkan. In this coursework, you will implement a Vulkan-based renderer to demonstrate your ability to work with the Vulkan API. Custom rendering and visualization modes are n important tool for development and debugging. You may further implement two such visualizations.
Page limit and guidance	Report: 8 pages with 2cm or larger margins, 10pt font size (including figures). You are allowed to use a double-column layout. Code: no limit. **Please read the submission instructions carefully!**
Weighting	40%
Submission deadline	2024-03-07 15:00
Submission method	Gradescope: code and report
Feedback provision	Written notes
Learning outcomes assessed	Graphics programming in a modern API; shader programming knowledge; ability to develop and debug graphics applications.
Module lead	Markus Billeter

1. **Assignment guidance**

   In Coursework 1, you are tasked with writing a small Vulkan-based renderer and implement a few different rendering/visualization modes in this renderer. You will use the visualizations to further analyse the scene and renderer. Specifically, you will

   - Implement the basic Vulkan rendering infrastructure
   - Load and render multiple objects with texturing
   - Implement two visualization modes and use these for analysis.

   Before starting your work, please study the coursework document in its entirety. Pay special attention to the requirements and submission information. Plan your work. It might be better to focus on a subset of tasks and commit to these fully than to attempt everything in a superficial way.

2. **Assessment tasks**

   Please see detailed instructions in the document following the standardized assessment brief (pages i-iv). The work is split into five tasks, accounting for 40% of the total grade.

3. **General guidance and study support**

   Coursework 1 relies on materials covered by Exercises 1.X. The exercises introduce the Vulkan rendering API and guide you through creating a simple renderer with Vulkan. It is recommended that you complete the exercises before attempting Coursework 1.

   Additional support will be provided during scheduled lab hours. Further support may be provided throught the module's "Teams" channel (but do not expect answers outside of scheduled hours).

4. **Assessment criteria and marking process**

   Submissions take place through Gradescope. Valid submissions will be marked primarily based on the report and secondarily based on the submitted code. See following sections for details on submission requirements and on requirements on the report. Marks and feedback will be provided through Minerva (and not through Gradescope - Gradescope is only used for submissions!).

5. **Submission requirements**

   Your coursework will be graded once you have

   (a) submitted project files as detailed below on Gradescope.

   (b) successfully completed a one-on-one demo session for the coursework with one of the instructors.

   (c) If deemed necessary, participated in an extended interview with the instructor(s) where you explain your submission in detail.

Details can be found in the main document.

Your submission will consist of source code and a report. *The report is the basis for assessment. The source code is supporting evidence for assertions made in the report.*

Submissions are made through Gradescope (do *not* send your solutions by email!). You can use any of Gradescope's mechanisms for uploading the complete solution and report. In particular, Gradescope accepts `.zip` archives (you should see the contents of them when uploading to Gradescope). Do not use other archive formats (Gradescope must be able to unpack them!). Gradescope will run preliminary checks on your submission and indicate whether it is considered a valid submission.

The source code must compile and run as submitted on the standard SoC machines found in the 24h teaching lab (2.15 in Bragg). Your code must compile cleanly, i.e., it should not produce any warnings. If there are singular warnings that you cannot resolve or believe are in error, you must list these in your report and provide an explanation of what the warning means and why it is acceptable in your case. This is not applicable for bulk warnings (for example, type conversions) – you are always expected to correct the underlying issues for such. *Do not change the warning level defined in the handed-out code. Disabling individual warnings through various means will still require documenting the warning in the report.*

Your submission must not include any "extra" files that are not required to build or run your submission (aside from the report). In particular, you must *not* include build artifacts (e.g. final binaries, `.o` files, ...), temporary files generated by your IDE or other tools (e.g. `.vs` directory and contents) or files used by version control (e.g. `.git` directory and related files). Note that some of these files may be hidden by default, but they are almost always visible when inspecting the archive with various tools. Do not submit unused code (e.g. created for testing). Submitting unnecessary files may result in a deduction of marks.

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.*

The demo sessions will take place in person during the standard lab hours. You must bring a physical copy of the *Demo Receipt* page, pre-filled with your information. During the demo session, the instructor may ask questions to test your knowledge of the code. If satisfactorily answered, the instructor will sign both portions of the receipt, and keep the second half (the first half is for you).

6. **Presentation and referencing**

Your report must be a single PDF file called `report.pdf`. In the report, you must list all tasks that you have attempted and describe your solutions for each task. *Include screenshots for each task unless otherwise noted in the task description!* You may refer to your code in the descriptions, but descriptions that just say "see source

code" are not sufficient. Do **not** reproduce bulk code in your report. If you wish to highlight a particularly clever method, a short snippet of code is acceptable. Never show screenshots/images of code - if you wish to include code, make sure it is rendered as text in the PDF using appropriate formatting and layout.

Apply good report writing practices. Structure your report appropriately. Use whole English sentences. Use appropriate grammar, punctuation and spelling. Provide figure captions to figures/screenshots, explaining what the figure/screenshot is showing and what the reader should pay attention to. Refer to figures from your main text. Cite external references appropriately.

Furthermore, the new UoL standard practices apply:

The quality of written English will be assessed in this work. As a minimum, you must ensure:

- Paragraphs are used
- There are links between and within paragraphs although these may be ineffective at times
- There are (at least) attempts at referencing
- Word choice and grammar do not seriously undermine the meaning and comprehensibility of the argument
- Word choice and grammar are generally appropriate to an academic text

These are pass/ fail criteria. So irrespective of marks awarded elsewhere, if you do not meet these criteria you will fail overall.

7. **Academic misconduct and plagiarism**

   If you use any external resources to solve tasks, you must cite their source *both* in your report and in your code (as a comment). This applies both to code as well as to general strategies (e.g. papers, books or even StackOverflow answers).

   Furthermore, the new UoL standard practices apply:

   Academic integrity means engaging in good academic practice. This involves essential academic skills, such as keeping track of where you find ideas and information and referencing these accurately in your work.

   By submitting this assignment you are confirming that the work is a true expression of your own work and ideas and that you have given credit to others where their work has contributed to yours.

8. **Assessment/marking criteria grid**

   (See separate document.)

COMP5822M Coursework 1

Contents

For Coursework 1, you will create a Vulkan-based renderer. You will perform the necessary Vulkan setup to render to a window. You will then load and render a textured 3D scene. Finally, you may implement three additional visualizations that help with analysis of the rendering.

If you have not completed Exercises 1.X, it is highly recommended that you do so before attacking CW 1. When requesting support for CW 1, 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 CW submission. It is expected that you understand all code that you hand in, and are able to explain its purpose and function when asked.*

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.*

You must build on the code provided with the coursework. In particular, the project must be buildable through the standard premake steps (see exercises). Carefully study the submission requirements for details. If you want to use additional third party software/libraries, you must clear that with the module leader first (it is unlikely that you need this in CW1; using third party software to solve portions of the coursework will not give any marks for those portions).

CW1 uses the rapidobj library to load 3D models in the Wavefront OBJ format.

1 Tasks

The total achievable score for CW 1 is **40 marks**. CW 1 is split into the tasks described in Sections 1.1 to 1.6. Each section lists the maximum marks for the corresponding task.

Do not forget to check your application with the Vulkan validation enabled. Incorrect Vulkan usage -even if the application runs otherwise- may result in deductions. Do not forget to also check for synchronization errors via the Vulkan configurator tool discussed in the lectures and exercises.

1.1 Vulkan infrastructure

6 marks

Start by setting up the necessary Vulkan rendering infrastructure for a real-time rendering application. This likely includes:

- 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.

In your report, list the details of the Vulkan implementation that you've been using for this coursework: include Vulkan device name, the loader's and the device's Vulkan versions, and any extensions that you have enabled. Mention any optional features that you have enabled. List the color format of the swap chain images and the number of images that it contains.

†You are allowed to use Vulkan 1.2 or Vulkan 1.3 functionality or equivalent extensions (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 in your report!

1.2 3D Scene and Navigation

9 marks

CW1 includes a Wavefront OBJ file, `sponza_with_ship.obj` along with material definitions (`*.mtl`). Extend your application to load the OBJ file and render the corresponding 3D scene. You should not change the provided OBJ file (or its material definitions). Additionally, implement a "first person" camera with which a user can navigate the 3D scene (camera details below).

The scene consists of several sub-objects that each can have different materials. Hence, you will need to deal with multiple materials. In particular, some materials contain textures, while others use solid colors only. You will need to handle both cases. You may base your loader on the included `simple_model.hpp` and `load_model_obj.{hpp,cpp}`.

Using a `if` statement to choose between two code paths in the fragment shader may be tempting, but is likely a suboptimal choice. Try to come up with an alternative choice.

You do not have to implement any lighting. Do enable back face culling and (of course) depth testing. Your results should look similar to the teaser image.

The textures are provided in the JPEG format (`.jpg`). The `stb_image.h` library introduced in the exercises can load JPEG files as well as PNGs. Texture sampling should use trilinear (=mipmapped) filtering. Generate mipmaps on the fly, using Vulkan.

Pay attention to relative cost of operations when picking a strategy. For this, you should assume that switching pipelines is more expensive than binding a different descriptor set or input vertex buffers, both of which are more expensive than issuing draw calls.

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).

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.

In your report, describe how you've chosen to handle the different meshes and materials. Motivate your choice. Discuss the efficiency of your strategy. Outline the costs in terms of required Vulkan resources and Vulkan operations. What do you do at load time and what do you do for each frame? Include relevant screenshots.

1.3 Anisotropic filtering

2 marks

Extend the texturing to use anisotropic filtering. You will need to enable anisotropic filtering first when you create the logical Vulkan device, and later when creating the `VkSampler`.

Make sure that you only enable anisotropic filtering when it is supported by the Vulkan implementation and that you respect the limits that the Vulkan implementation defines.

In your report, include two screenshots that illustrate the differences resulting from enabling anisotropic filtering. List Vulkan-related limits and their values for your implementations. Mention and motivate any choices that you've made.

1.4 Visualizing fragment properties

6 marks

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 1 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.

In the report, describe how you have implemented the visualizations for this task. Describe and discuss your findings (with supporting screenshots). Are all mipmap levels used? What happens if you render to a smaller/larger window (or just a larger surface)? How does the bias affect the level selection?

You can choose how you want to implement this task in practice. It is not necessary to be able to switch between rendering modes at runtime (though you may do so if you wish). A compile-time flag (`#define`+`#if`/`#ifdef`) is a perfectly fine choice. You can also define a separate application for this by extending the `premake5.lua`.
Mention your choice in the report and include instructions on how to run the rendering mode. In the report, mention where your implementation's code can be found. However, code must absolutely be handed in and be somewhat conveniently runnable (e.g., having to comment/uncomment different regions of code is not an acceptable solution).

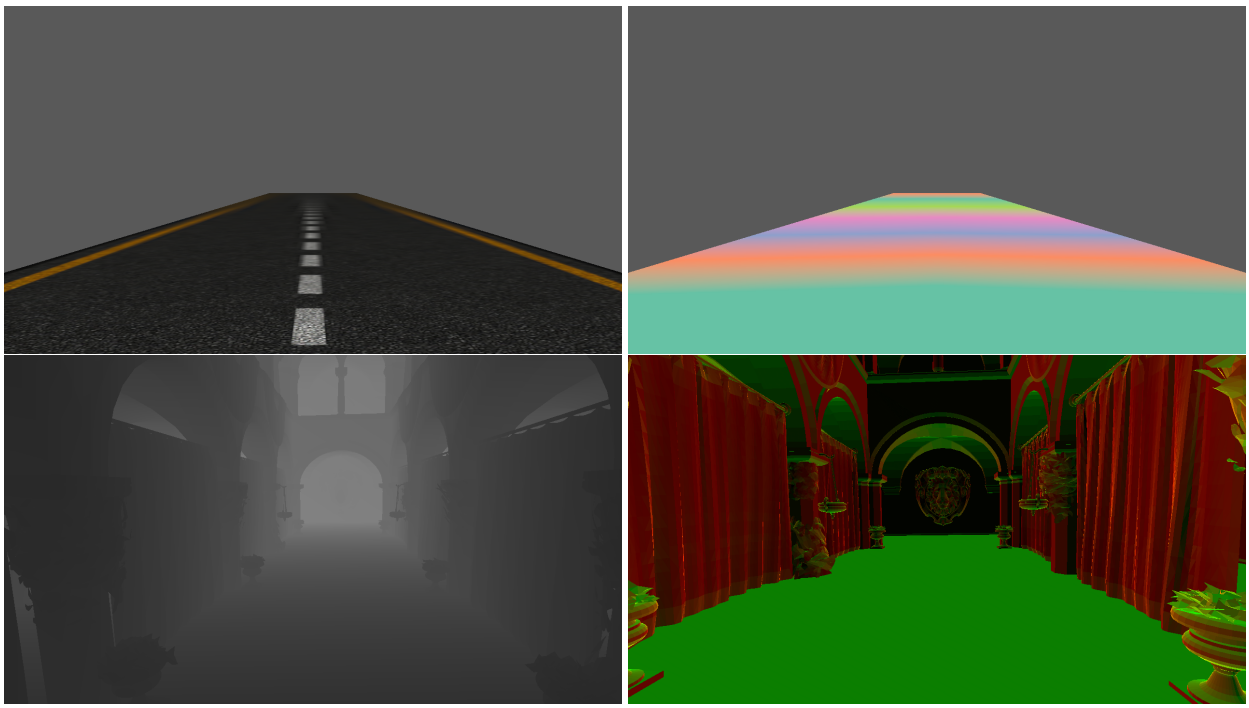

Figure 1: *Top-Left: Textured plane with mipmap filtering. Top-Right: Visualization of the mipmap levels. The color indicates which mip-levels were used when sampling the texture. You should implement this for the Sponza scene (not for the road shown here)! Bottom-Left: Linearized depth. Bottom-Right: Partial derivatives of the per-fragment depth (scaled to be easily visible).*

1.5 Visualizing overdraw and overshading

6 marks

Implement a rendering method that lets the user visualize overdraw and one that shows overshading. The terms overdraw and overshading are somewhat overloaded. For the purpose of Coursework 1, we will define them as following:

Baseline overdraw The number of fragments per pixel that would have been generated without any (early-)Z tests. This means each (front-facing) surface within the view frustum that covers a pixel should contribute by one fragment to that pixel.

Basic overshading The number of times the fragment shader runs per pixel. Fragments discarded by early-Z should not count. This should be lower than (or in the worst case, equal to) to the baseline overdraw.

The visualizations should produce false-color images that informs the views of the amount of overdraw and overshading, respectively. Choose an appropriate colormap (and document it!). Make sure the differences are clearly visible and that it is easy to interpret the results. Figure 2 shows an example of overshading.

In your report, describe your implementation. Discuss your findings relative to the Sponza scene. What do you observe as you move around the scene? For example, look down one of the row of columns from one side then the other. Is there any difference? Did you expect there to be? Why/why not? Discuss why such visualizations can be useful. How could one reduce the baseline overdraw? How could one reduce the basic overshading? Include relevant screenshots. Marks are mainly awarded for your discussion, supported by your findings.

1.6 Visualizing Mesh Density

9 marks

Formulate and implement a rendering method that lets the user visualize mesh density. The visualization should produce a false-color image to differentiate between high and low mesh/vertex densities. Choose an appropriate colormap (and document it!). Figure 3 shows an example of such a visualization.

You may define mesh density as either triangle density or vertex density (or perhaps another appropriate metric?). There are subtle differences, so make sure you can motivate your choice (e.g., why do you think you definition is more useful than the other/others?).

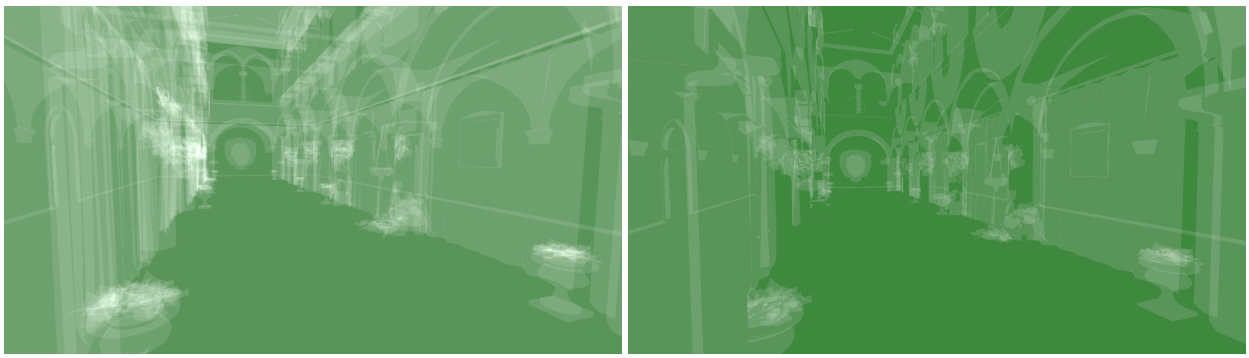

Figure 2: *Left: Baseline overdraw. Right: Overshading. Colors range from dark green (zero fragments) to white (20+ fragments).*

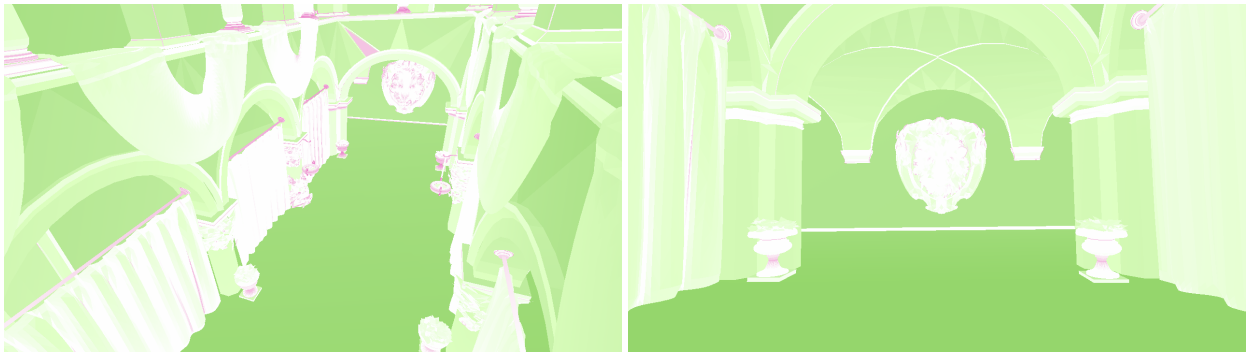

Figure 3: *Possible visualizations of a sample mesh density metric. Colors tending towards green indicate low density. High density regions tend towards purple. White is a neutral region in between. This example should only be used as inspiration and not be considered a ground truth.*

Ideally, the visualization should have the following properties:

- It should only show the front-most (=visible) surface. Hidden surfaces (e.g., from depth testing) should not affect the visualization.
- It should be easy to deploy into an existing application. For example, optimally, it should not require any additional substantial data (e.g., additional per-vertex attributes).
- The visualization should perform at interactive rates.
- The visualization should be able to highlight regions where the mesh density exceeds one primitive/pixel.

Partial solutions may still receive marks. You are allowed to use multiple render-passes. You are allowed to use additional shader stages (e.g., tessellation, geometry or compute shaders). You can use other Vulkan features. (As always, you cannot solve the problem with external software, though.)

In your report, describe how you have defined mesh density (and why) and then describe your method to visualize it. Mention which of the requirements it fulfills. Detail any special requirements it might have. Evaluate your results. Discuss edge- and problematic cases. Show results from your method with screenshots. Highlight any special findings in the test scene.

Discuss why such a visualization could be useful. Based on the method's results, are there any changes you would propose to the provided scene? Any key improvements to the rendering method? Marks are mainly awarded for your discussion, supported by your findings.

1.7 More overshading

2 marks

The overshading in Section 1.5, relating to overdraw, is not the only type of overshading. Another type results from helper invocations. Helper invocations are fragment shader invocations that only participate in shading to enable computing partial derivatives inside a shading quad (the smallest renderable unit is a 2×2 quad). Helper invocations themselves do not result in a visible fragment. You will want to use the `gl_HelperInvocation` built-in variable in GLSL, as well as some of the subgroup operations.

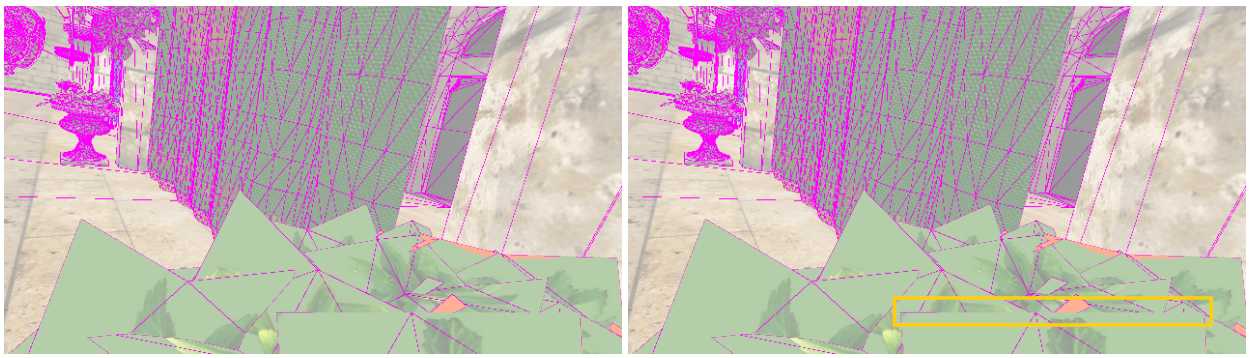

Figure 4: *Left: Overshading from shading quads. Right: Highlighted region (yellow) has no apparent overshading.*

In your report, describe how you have visualized this second type of overshading. Show results from your method with screenshots. Overshading mostly occurs at edges of polygons, but not all. Why? What else do you observe? Marks are mainly awarded for your discussion, supported by your findings.

2 Submission & Marking

In order to receive marks for the coursework, follow the instructions listed herein carefully. Failure to do so may result in zero marks.

Your coursework will be marked once you have

1. Successfully submitted project files as detailed below on Gradescope. (Do *not* send your solutions by email!)
2. Successfully completed a one-on-one demo session for the coursework with one of the instructors. Details are below - in particular, during this demo session, you must be able to demonstrate your understanding of your code. For example, the instructor may ask you about (parts of) your submission and you must be able to identify and explain the relevant code.
3. If deemed necessary, participated in an extended interview with the instructor(s) where you explain your submission in detail.

You do *not* have to submit your code on Gradescope ahead of the demo session.

Project Submission You will submit your solutions through Gradescope. You can upload a `.zip` file or submit through Github/Bitbucket. If successful, Gradescope will list the individual files of your submission. Additionally, automated tests will check your submission for completeness. Only solutions that pass these tests will be considered for marking.

The submission must contain your solution, i.e.,

- A report, named `report.pdf`. Only PDF files are accepted.
- Buildable source code (i.e., your solutions and any third party dependencies). Your code must be buildable and runnable on the reference machines in the Visualization Teaching Lab or in the 24h teaching lab.
- A list of third party components that you have used. Each item must have a short description, a link to its source and a reason for using this third party code. (You do not have to list reasons for the third party code handed out with the coursework, and may indeed just keep the provided `third_party.md` file in your submission.)
- The `premake5.lua` project definitions with which the project can be built.
- Necessary assets / data files.

Your code must be buildable in both debug and release configurations. It should compile without any warnings. Ask for assistance if you have trouble resolving a certain type of warning.

Your submission *must not* include any unnecessary files, such as temporary files, (e.g., build artefacts or other "garbage" generated by your OS, IDE or similar), or e.g. files resulting from source control programs. (The

submission may optionally contain Makefiles or Visual Studio project files, however, the program must be buildable using the included `permake5.lua` file only.)

If you use non-standard data formats for assets/data, it must be possible to convert standard data formats to your formats. (This means, in particular, that you must not use proprietary data formats.)

Demo Session The demo sessions will take place in person during the standard lab hours.

Bring a physical copy of the *Demo Receipt* page (Appendix A), pre-filled with your information. If successful, the instructor will sign both portions of the receipt, and keep the second half (the first half is for you).

Demo sessions will take place on a FIFO (first-in, first-out) basis in the scheduled hours. You might not be able to demo your solution if the session's time runs out and will have to return in the next session. *Do not wait for the last possible opportunity to demo your solution, as there might not be a next session in that case.*

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.

A Demo Receipt

Please bring this page on an A4 paper if you are demo:ing your coursework in-person. Fill in the relevant fields (date, personal information) in both halves below. If the demo session is successful, an instructor will sign both halves. The top half is for your record. The instructor will take the bottom half and use it to record that you have successfully demo:ed your CW.

Please write legibly. :-)

Coursework 1 (Student copy)

Date .....

Name .....

UoL Username .....

Student ID .....

The instructor(s) will fill in the following:

Instructor name .....

Instructor signature:

Coursework 1 (Instructor copy)

Date .....

Name .....

UoL Username .....

Student ID .....

The instructor(s) will fill in the following:

Vulkan Infra. 3D Scene Aniso. Properties Overdraw Density Overshade2

Instructor name .....

Instructor signature: