School of Computing: assessment brief

Module title	High Performance Graphics
Module code	COMP5822M
Assignment title	Coursework 2
Assignment type and description	Programming assignment: Data and Shading
Rationale	Pre-computing, normalizing and pre-processing data into an efficient format ("baking data") is an essential step for high-performance applications. Shading, and especially evaluating lighting, is essential to realistic image generation. Lighting further relies on specific input attributes that may not be present in common mesh formats, so baking data ahead of time enables more advanced and efficient lighting.
Page limit and guidance	Report: 7 pages, 10pt font size. You are allowed to use a double-column layout. Code: no limit. **Please read the submission instructions carefully!**
Weighting	25%
Submission deadline	2022-05-09
Submission method	Gradescope: code and report
Feedback provision	Written notes
Learning outcomes assessed	Graphics techniques (lighting); Shader programming; High-performance graphics.
Module lead	Markus Billeter

1. **Assignment guidance**

   In Coursework 2, you will first complete a simple mesh-baking application that converts an input OBJ into a more efficient run-time format. You will then implement a physically-based shading model for lighting. In the baking step, add essential data to move to a more advanced lighting implementation. Finally, you will optimize the meshes to improve performance and reduce the amount of data.

   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 25% of the total grade.

3. **General guidance and study support**

   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.

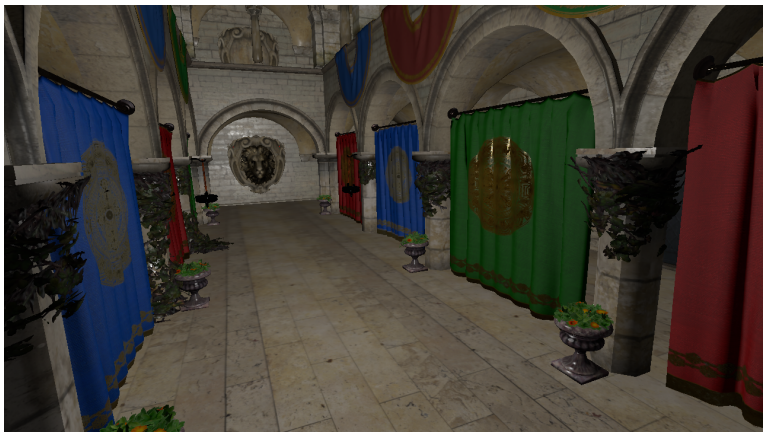
   By submitting this assignment you are confirming that the work is a true expression of your own work and ideas and that you have given credit to others where their work has contributed to yours.

8. **Assessment/marking criteria grid**

   (See separate document.)

COMP5822M Coursework 2

Contents



Coursework 2 revolves around two topics: "pre-baking" data and implementing a shading model with physically-inspired lighting. The baking step pre-computes additional data necessary for more advanced lighting and enables a number of optimizations.

If you have not completed Exercises 1.X and CW 1, it is highly recommended that you do so before attacking CW 2. When requesting support for CW 2, 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.

1 Tasks

The total achievable score for CW 2 is **25 marks**. CW 2 is split into the tasks described in Sections 1.1 to 1.5. 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.

"Baking"

Coursework 2 includes two applications. The first one, `cw2-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 as you've done in the exercises and CW 1.

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

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

Figure 1: *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?*

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 `cw2-bake/main.cpp` and `cw2/baked_model.cpp`).

Build and run the application convert the coursework's OBJ file to the binary runtime format that the task in Section 1.1 will use. The source OBJ files are located in the `assets-src/cw2` directory; the 'baked' output will be placed in the `assets/cw2` directory. The application also copies used textures to the destination 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).

1.1 Prep and Debug

6 marks

Before you start implementing either shading model, you will need to do some prep work. Implement a Vulkan renderer that loads and draws the binary file produced by the baking step (you may use the included `baked_model.{hpp,cpp}` for loading). Refer to your renderer from CW 1 if necessary. 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`).

Next, 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, i.e., in CW 2 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 $[0, 2, 0]$ in world space, and give it a color of $[1.0, 1.0, 1]$. 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 1.

In your report, document the following:

- Your choice of shading space

- Your choice of descriptor set layout and descriptor bindings. Motivate this choice *briefly*.
- Describe uniform input data (e.g. UBOs and similar). Motivate your choices briefly.

Include your own screenshots visualizing the normals, view direction, and light direction (similar to Figure 1) of the rendered scene.

You may be asked to briefly show the visualizations of the above values (and/or other debug visualizations) in the live version of your program in the demo session.

1.2 Lighting

7 marks

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 Blinn-Phong 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_p Material shininess
- M Material metalness
- $\mathbf{c}_{\text{emit}}$ Material emissive color - you may assume this is zero in CW 2
- $\mathbf{c}_{\text{mat}}$ 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

The PBR models provide *roughness* rather than *shininess*. The relation between shininess and roughness is [Boksansky, 2021; Burley, 2012]

$$\alpha_p = \frac{2}{r^4 + \epsilon} - 2,$$

where r is the roughness value between zero and one, and ϵ is a small value to avoid divisions with zero.

For the BRDF, we will use a general microfacet model for isotropic materials [Burley, 2012]:

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

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 2a) and the masking function G (Figure 2b).

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:

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

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 [Karis, 2013].

For the normal distribution function D, we will use the Blinn-Phong distribution. As the name indicates, it is very similar to to the Blinn-Phong specular term from the previous exercise

$$D(\mathbf{h}) = \frac{\alpha_p + 2}{2\,\pi}\ (\mathbf{n} \cdot \mathbf{h})_+^{\alpha_p} .$$

The masking term from the Cook-Torrance model [Cook and Torrance, 1982] 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:

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

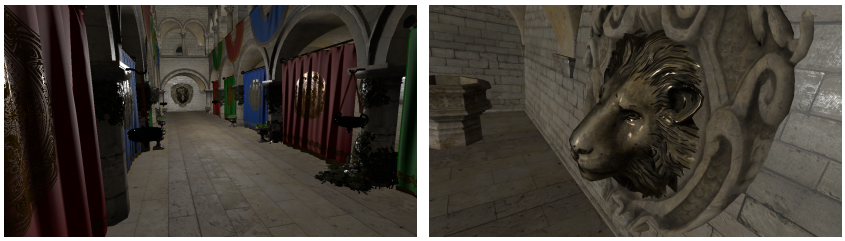
Implement the shading model for one light source (Equation (1)).

Compare your output to Figure 3. If you need to debug your shaders, see Figure 4 for visualization of some of the intermediate values (also check that the various values are behaving as they should!).

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.

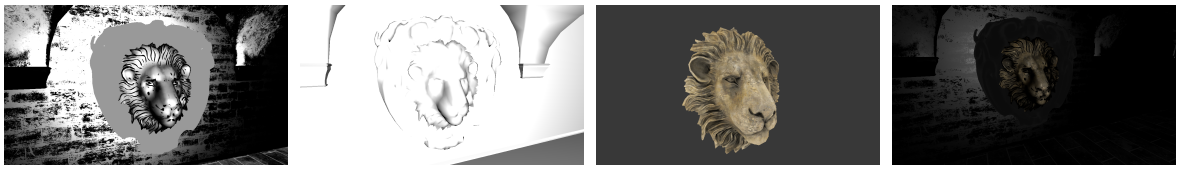
(a) *NDF* **(b)** *Geometry term*

Figure 2: *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)** *Lion Head*

Figure 3: *Shading with the PBR model. The Lion Head material has been changed to be a metal. Some features on the curtains are also metallic. You can view the corresponding texture in* `m-4661647079954366 22.jpg`.



(a) D **(b)** G **(c)** $\mathbf{F}$ **(d)** *PBR specular term*

Figure 4: *Visualization of the some of the components of the PBR model used in CW 2. The Fresnel term shows clearly that the Lion Head material is metallic.*

Animate the light position. For example, make the light orbit around the Y axis. Make sure that the animation is frame-rate independent (e.g., like the camera controls in CW 1). Let the user start/stop the animation by pressing spacebar.

In your report, include a few screenshots of your results. Include separate images that visualize the factors shown in Figure 4 and the diffuse contribution. Explain each contribution and discuss whether or not it behaves as you expect.

1.3 Alpha Masking

3 marks

As you may have seen, the Sponza 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.

In your report, include a screenshot that shows foliage before and after implementing the new pipeline.

1.4 Normal mapping

4 marks

Next, you will implement normal mapping. Aside from the normal maps, this requires additional vertex attributes, specifically a tangent vector (the bi-tangent can be reconstructed from the tangent and the normal).

The first step is to update the baking application to compute the per-vertex tangent vectors and store these in the run-time binary format (don't forget to update the `kFileVariant` for the generated output files; this will prevent mixups between files with and without the additional data, potentially saving quite a bit of debugging time). You may use the included tgen library to compute the tangents. Note that *tgen* computes 4-component tangents, where the final component indicates whether the TBN frame is mirrored. See *tgen*'s online documentation for additional information (hint: *tgen*'s public API defines four functions, you will want to call all of them in the declared order).

With the data in place, implement normal mapping. A useful trick to keep complexity down is to use a "dummy" 1×1 normal map for objects that otherwise do not have normal maps. (This is perhaps slightly suboptimal, but it avoids an combinatorial explosion in shader variants – if you want to use different shaders, you can do so, however.)

In the report, include screenshots that show results before and after normal mapping. Include a visualization of the normals before and after. Discuss why *tgen* produces a `vec4` tangent. Is this always required?

1.5 Mesh data optimizations

5 marks

Normal mapping adds a chunk of per-vertex data – each vertex now weighs in at 48 bytes ($3\times$ position, $2\times$ texture coordinates, $3\times$ normals and $4\times$ tangents, all stored as 32-bit `float`s). There is some redundancy in this data.

In particular, you should optimize the normal and tangent data. The normal and tangent form (together with the implicit bi-tangent) a TBN coordinate frame. The TBN frame generated by the *tgen* library is orthonormal. A orthonormal 3×3 matrix expresses a rotation (and potentially a mirroring).

We can express the rotation as a unit quaternion. Given that the quaternion is normalized, we can encode the quaternion into just three values (the fourth is reconstructed). Furthermore, for example, these three values can be discretized and stored in less than 32 bits. Briefly study the BitSquid Low-level Animation System document (particularly the second-to-last paragraph) for some additional inspiration. (But don't forget about having to store whether the TBN frame is mirrored!)

Implement an efficient packing of the TBN frame during the baking process. Decode the TBN frame in the your shader(s) when drawing (i.e., keep the compact form when storing data in Vulkan buffers).

In your report, describe your implementation and the data format that you've chosen. How much space does your selected coding save? Is the `R10G10B10A2` encoding recommended in the linked article a good choice or would you recommend a different format? How much data can you save with your coding? How large are the errors that you introduce? What is the overhead of this coding? Would you consider it to be worthwhile? Also discuss where (in which shader stage) you decode the TBN frame and why.

This task is intended to be somewhat more open. Partial solutions may still receive credit. In that case, it is even more important that you describe exactly what you have done and motivate your choices.

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. 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 mustbe 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.*

References

BOKSANSKY, J. 2021. Crash course in BRDF implementation. URL https://boksajak.github.io/blog/BRDF.

BURLEY, B. 2012. Physically based shading at disney. URL https://media.disneyanimation.com/uploads/production/publication_asset/48/asset/s2012_pbs_disney_brdf_notes_v3.pdf.

COOK, R. and TORRANCE, K. 1982. A reflectance model for computer graphics. URL https://graphics.pixar.com/library/ReflectanceModel/paper.pdf.

KARIS, B. 2013. Real shading in unreal engine 4. URL https://cdn2.unrealengine.com/Resources/files/2013SiggraphPresentationsNotes-26915738.pdf.

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 2 (Student copy)

Date .....

Name .....

UoL Username .....

Student ID .....

The instructor(s) will fill in the following:

Instructor name .....

Instructor signature:

Coursework 2 (Instructor copy)

Date .....

Name .....

UoL Username .....

Student ID .....

The instructor(s) will fill in the following:

Prep Lighting Masking Normal mapping Optimizations

Instructor name .....

Instructor signature: