School of Computing: assessment brief

Module title	Advanced Rendering
Module code	COMP5892M
Assignment title	Assignment 2
Assignment type and description	Programming assignment: Ray Tracing and Rasterization advanced tasks.
Rationale	You will implement a selection of more advanced rendering techniques, building on your work in Assignment 1. You can chose from tasks related to either raytracing or rasterization.
Page limit and guidance	You will submit your code solutions along with a report to Gradescope. 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	50%
Submission deadline	24/01/2025
Submission method	Gradescope
Feedback provision	Gradescope
Learning outcomes assessed	LO1-LO6 (advanced level)
Module lead	Rafael Kuffner dos Anjos & Markus Billeter

1. **Assignment guidance**

   You should complete Assignment 1 before attempting the second assignment. You may build on your code/solutions from Assignment 1. (Indeed, Assignment 2 assumes that you have the Assignment 1 solutions.)

   You can chose to attempt tasks related to either raytracing or rasterization.

   Important: In order to show a wide range of learning in this module, if you complete more than 10 points in either part, you must also complete at least 10 points in the other.

   Example: if you want to complete 20 points related to rasterisation, you need to complete at least 10 points in raytracing tasks (and vice versa).

2. **Assessment tasks**

   This is the summary of tasks that you can choose to attempt. Tasks are typically worth up to 5 points. Please see individual task specifications for details.

   Raytracing: Refraction and Fresnel, Monte-carlo sampling, Caustics, Area Lights and AA, Light absorbption (Beer's law).

   Rasterisation: Debug visualizations, deferred shading, shadows, normal mapping with compression & bloom.

   See detailed descriptions at the end of this document

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 taught content from the slides, and reading lists for the module.

   The module staff will be available for questions, but it is expected that students perform independent research on how to implement some of these tasks.

4. **Assessment criteria and marking process**

   Only the tasks that were implemented following the guidelines will be marked. If implemented correctly, you will be awarded the marks described in this document. Partial marks will be awarded according to how well it was implemented, and how well it supports all use-cases. For full marks robust high-quality solutions are required.

5. **Submission requirements**

   - Raytracing tasks: similarly to A1, only submit a .zip of your src folder. As no external codebases are allowed, and no changes in the shaders are needed, all of your code should be in the src folder so compilation can happen smoothly.

- Rasterization tasks: You are required to submit code and a report. Submit both to Gradescope. The report must be submitted as a single PDF file called `report.pdf`. Formats other than PDF are not accepted. Tasks specify what is expected in your report – focus on answering these questions (and only these).

  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!*

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. All the submitted code must be written from the start by you. You can use external references to understand a topic, not to copy paste a solution into your codebase. 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.

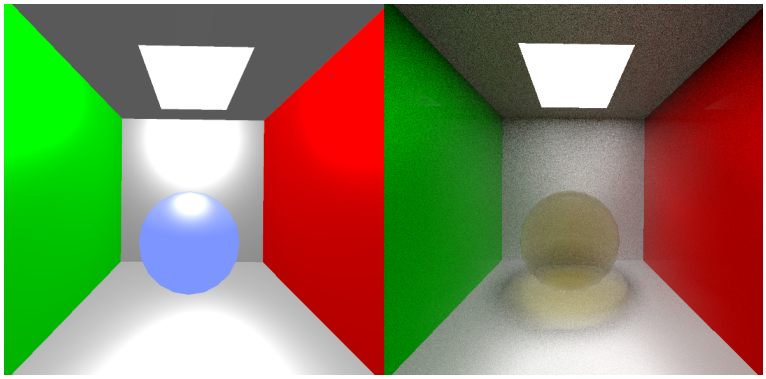
   While you are encouraged to use version control software/source code management software (such as git or subversion), you must not make your solutions publicly available. In particular, if you wish to use Github, you must use a private repository. You should be the only user with access to that repository.

8. **Assessment/marking criteria grid**

   Up to 5 points for each task completed. See detailed description below.

Part 2: Advanced Raytracing

Contents



The following tasks should be implemented on top of your raytracer. Using external codebases will not be accepted. Describe the implementation of each one of them in your report, making sure to cover the points suggested in each one.

1 Refraction and Fresnel effect:

5 marks

Use the material properties to verify if the object is transparent (values are 0 or 1). If it is, calculate the refraction ray keeping track of the correct IOR and the appropriate shaded result when it hits an opaque object. Pay attention to total internal reflection. Also implement the Fresnel equations (e.g. Schlick approximation), which will transform part of the refracted light into reflection, and also part of the diffuse (considered internal refraction) into reflection. Combine results accordingly.

In your report, clarify the model used to combine energy, how the fresnel effect is distributing it, and what does it mean for your scattering. Which energy is going where?

2 Montecarlo sampling for indirect lighting

5 marks

Replace the ambient component of your computation with a value calculated using monte-carlo sampling. For this, you will change the code to run as a path tracer when this checkbox is on, so a single indirect ray per loop should suffice. You should still use next event estimation (direct lighting rays), but being careful not to account for light sources twice. Moreover, pay attention to how the samples are generated so the result is unbiased.

In your report, describe how you have implemented next event estimation, and how many rays are you using. Evaluate the impact of your NEE to how well it converges. Clarify your termination conditions, and how was your code changed from a raytracer to become a path tracer.

3 Area Lights, Anti-aliasing

5 marks

This task is only available to be implemented if you have Montecarlo working. Change the code that samples the light sources when doing monte-carlo rendering, along the whole surface of the light source as defined in the obj files (and their sizes in the Light class). This will allow you to obtain soft shadows. Implement a function that provides this sampled position and use it accordingly. Add anti-aliasing to your path-tracer, with each primary ray being sent through a random position inside the pixel.

In your report, evaluate the impact of the anti aliasing technique implemented, and describe how the sampling techniques are similar.

4 Transparency with colour

5 marks

This task is only available to be implemented with refraction and fresnel working. Use Beer's law to simulate light absorbption when going through transparent media. It can be defined as an exponential with the product of the diffuse property of the material and the distance travelled inside the object.

$a = e^{-diffuse*distance}$

Be careful when computing the distance, as with cases of internal reflection that distance has to be tracked.

In your report, evaluate the effect of each parameter of the computation, and how different materials generate different looks for your rendering. Generate examples with variable amounts of absorbtion.

5 Caustics

5 marks

This task is only available to be implemented with refraction, montecarlo sampling, and area lights working. Now, you should see caustics being formed when light goes through transparent objects. I'd like to see this effect in your raytracer, with next event estimation still working. For this, you should change the way NEE works, where rays should refract, and a collision with the surface of the area light should be confirmed. In your report, show the most complex version you can with your implementation, and explain where energy in distinctive parts of your rendering is coming from.

Part 2: Advanced Vulkan

Contents

The advanced rasterization tasks build on part 2 of Assignment 1 and assume that you have successfully completed the tasks there. In particular, you will use the same baking mechanism for the input model and must use the PBR shading model detailed there for these tasks. You should disable the "mosaic" post-processing effect for the advanced tasks.

As noted in the module's introduction, you are allowed to re-use any code that ***you have written yourself*** *for the exercises or Assignment 1 in this second assignment. 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.*

1 Tasks

The advanced tasks are more open-ended. Partial solutions will often yield partial marks. For many tasks, the focus is more on your evaluation and analysis with/of your solutions than the implementation (although you will need to implement the tasks for this). For full marks, it is expected that you provide high-quality and well-engineered solutions, with a good analysis and discussion.

Before starting, it is recommended that you carefully study all tasks and then plan your work. You are not required to complete the tasks in order and it is not expected that you attempt all tasks. Please refer to the "Assessment guidance" section in assignment brief for details on marking.

1.1 Overdraw and -shading

3 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 this assignment, 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.

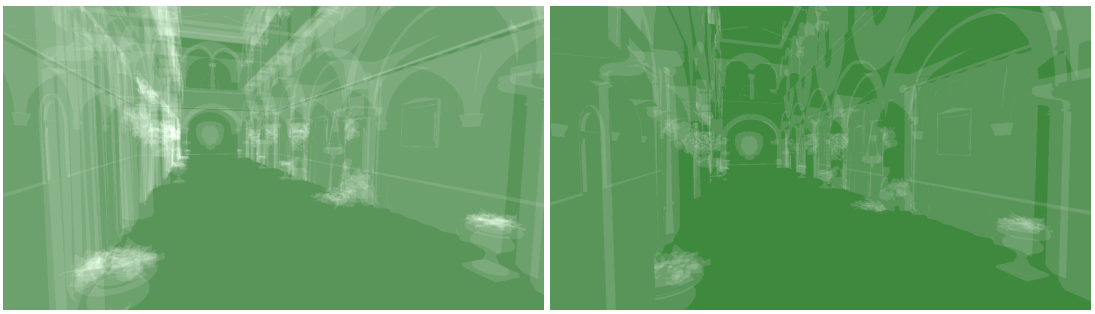


Figure 1: *Left: Baseline overdraw. Right: Overshading. Colors range from dark green (zero) to white (20+).*

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 1 shows an example of overshading.

In your report, describe your implementation. Document the keys to switch the visualization modes on and off. Discuss your findings relative to the Sun Temple 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.

Do this task before the deferred shading - in particular, it should consider overshading with respect to normal forward rendering and not deferred shading.

1.2 Naive Deferred Shading

3 marks

Implement naive deferred shading. Change the post-processing setup to output the necessary data into a G-Buffer in the first render pass and then move the shading into the full screen pass.

For full marks, you

- Most make reasonable choices in your G-Buffer format. Store only the data that you need. Minimize the number of attachments that you have. Pick appropriate formats for each attachment, based on the data stored therein. Using 32-bit floats for everything is not a good choice.
- In particular, you must reconstruct the 3D position of a sample in the full screen shading pass from the depth buffer and the fragment position. Do not store a separate 3D position!
- Support window resizing.

You should add multiple light sources (you can place lights e.g., at the various braziers in the scene, as shown in the teaser image).

You *can* use subpasses in your implementation of deferred shading if you wish. If you chose to do so, make sure you use them correctly. If necessary, check the documentation for subpasses and subpass inputs. If you use subpasses, make sure to mention this in the report. (Note: given that you already have a working post processing setup from Assignment 1, subpasses are unlikely to make this task much simpler.

In the report, document the G-buffer format that you have chosen. Discuss the advantages and disadvantages of the naive deferred shading approach compared the original forward shading approach. In this discussion, consider the following: overdraw, overshading, total number of lighting operations, memory use, memory bandwidth use.

1.3 Mesh Density

4 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 2 shows an example of such a visualization.

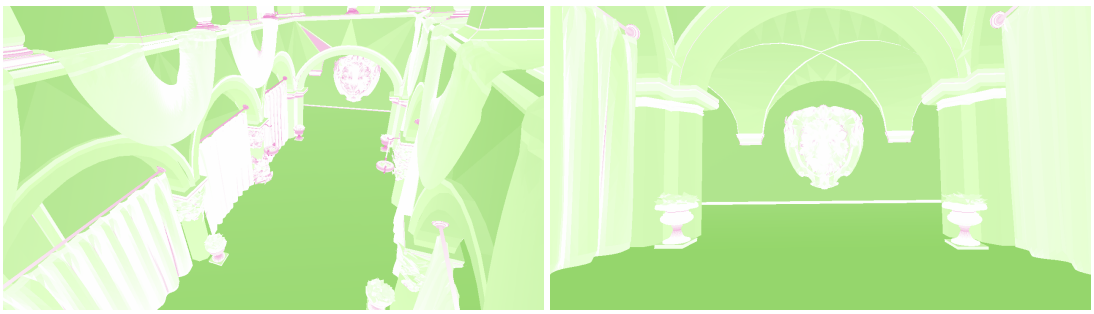


Figure 2: *Possible visualizations of a sample mesh density metric. Colors tending towards green indicate low density. High density regions tend towards purple. White is a neutral region in between. This example should only be used as inspiration and not be considered a ground truth.*

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?).

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. Document the key to toggle it on and off. 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.

1.4 Normal mapping

5 marks

First, implement normal mapping on top of the PBR shading from Assignment 1 (up to 3 marks). 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).

Begin by updating 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).

Do not compute the normals at runtime in the main application!

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.) See Figure 3.

(a) *Without Normal Maps* **(b)** *With Normal Maps*

Figure 3: *Left: Close up on statue wing without normal maps. Right: Wing with normal maps.*

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.

Optimize the normal and tangent data (up to 2 marks). 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).

> 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.

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?

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? Mention where (in which shader stage) you decode the TBN frame and why.

1.5 Shadow mapping

5 marks

The setup for shadow mapping is similar to the setup for post-processing. You will draw the shadow maps into separate textures before drawing the main scene. Unlike post-processing, shadow mapping requires multiple geometry passes (i.e., passes where you rasterize the 3D scene). A possible outline follows:

1. Render pass A: render 3D scene from the light's point of view.
2. Acquire next swap chain image
3. Render pass B: render 3D scene from the camera's point of view
4. Present swap chain image

Step 3 uses the shadow map from Step 1 to evaluate shadowing (light visibility) during the lighting computations. Since you are combining shadow mapping with your post processing implementation, there are additional steps between Step 1 and Step 2. Generally, it is possible to render shadow maps early, before any other drawing steps.

> It is even possible to draw shadow maps outside of the per-frame processing paradigm that we have applied so far. For example, shadow maps may be reused over several frames if the scene or light position does not change significantly.

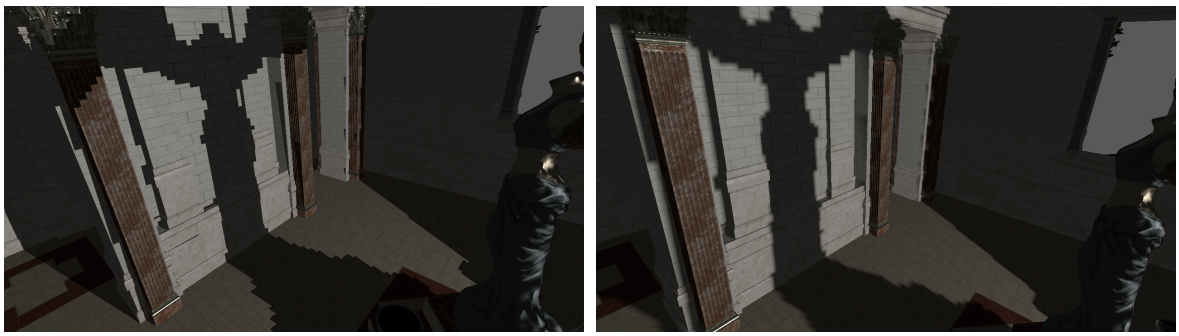


Figure 4: *Shadow map with and without PCF. The shadow map is low resolution on purpose, to make the effect of the PCF more visible. This uses just the simple built-in* 2×2 *filter.*

Start with a single 2D shadow map – this will result in a spot-light like light. One shadow-casting light source is sufficient. Determine a good shadow map resolution (note that 1024×1024 or larger might be necessary). Set up biases and related configuration to achieve good results and avoid e.g., surface acne and similar artifacts.

For full marks, you should consider the following:

- Shadow maps only use depth information. Ideally, you should create a framebuffer with only a depth attachment for Step 1. Consequently, the fragment shader for this pass will have no color outputs.
- Use built-in Vulkan and GLSL features where appropriate, as these might benefit from additional hardware acceleration. Relevant GLSL types/functions are: `sampler2DShadow` and `textureProj` (use the correct overload!); make sure to configure the Vulkan sampler correctly (see `VkSamplerCreateInfo`, specifically compare ops and border colors).
- Pre-compute the transformation into the light's projective space as an uniform value (ideally, the shadow lookup should be a single matrix multiplication and call to `textureProj`).
- Implement percentage closer filtering (PCF). Combine the built-in support for a 2×2 PCF filter with additional manual sampling to create a larger filter area. (You might want to test with lower shadow map resolutions to visualize the effects of this, see Figure 4 for an example.)
- Tweak the light's view frustum/projection to maximize the use of the available shadow map texels and the use of the depth-buffer resolution.

Renderdoc can be useful for debugging (e.g., looking at the shadow map contents).

In your report, describe your implementation (passes, pipelines/shaders, Vulkan resources, ...). Show your results. Include a comparison at different shadow map resolutions, and with/without PCF. Document your choices of parameters (e.g., shadow map resolution, projection settings, bias settings, ...), and motivate your choices. Discuss how these affect the results.

1.6 Bloom

5 marks

Bloom is a very common technique that is used for a variety of effects. The most common effect is to strengthen the impression of very bright/glowing objects. Figure 5 shows an example of this. The fire pits have an emissive texture (`M_FirePit_Inst_Glow_0_Emissive.png`). In the examples, the emissive contribution is included with a multiplier of 75.0, to boost its intensity (the 75 was determined empirically - you can experiment with other values).

You can also find a few examples of Bloom in action in e.g. Chapter 21 of the GPU Gems book and on the *Learn OpenGL* page on Bloom.

The bloom technique that you shall implement conceptually includes three steps:

- Filter out bright parts.
- Apply a blur to the bright parts.
- Combine the original image with the blurred results.

The example uses a threshold of 1.0 in the first step and keeps pixels if any of the RGB components are over the threshold (i.e., pixels are kept if $\max(r, g, b) > 1$). It uses a Gaussian blur with a 43×43 pixel footprint (with $\sigma = 9$, measured in pixels).

For full marks, you must do the following (partial marks possible):

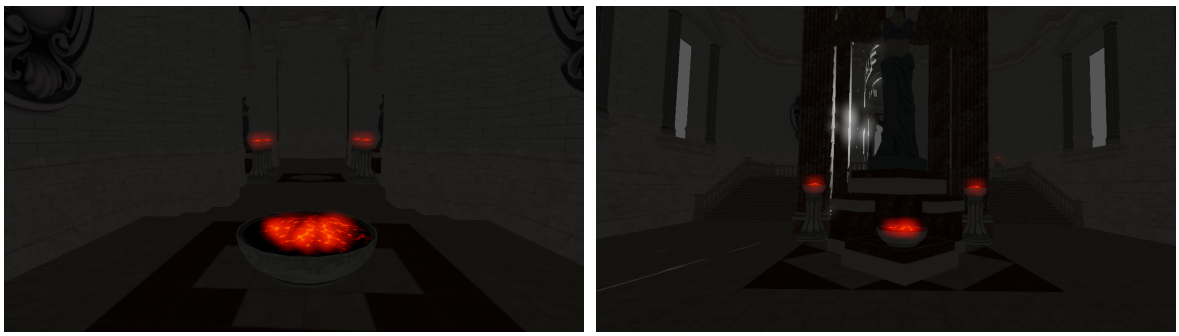


Figure 5: *Bloom effect from Section 1.6. Assignment 2 uses the emissive texture to the fire pits in the Sun Temple scene. Using the threshold, parts of the scene with strong illumination will also be affected by the bloom. The bloom creates a glow, but does not illuminate surrounding geometry. If you have time, you can experiment with adding light appropriately colored point lights (or -better- spot lights) to the fire pits and see the combination of the two (Figure 6).*

Figure 6: *Bloom with additional point lights (you are not required to implement this). Note that the smaller point lights do not cast shadows in this quickly put-together example.*

- Select appropriate texture formats for intermediate textures. Do not create unnecessary intermediate textures.
- Implement a 43×43 pixel footprint Gaussian blur filter.
- Derive your own weights for the filter. Use $\sigma = 9$ (in pixels). It is possible to perform the 43^2 Gaussian blur in just two passes with around 22 taps each.
- The Gaussian filter is separable, so it should be evaluated in two passes (a horizontal and a vertical). Figure 21-9 in Chapter 21 (GPU Gems) illustrates the idea; *Learn OpenGL*'s tutorial also discusses it.
- Use linear interpolation to reduce the number of taps (samples) that you need to take to evaluate. You can read about the optimization in blog post titled *Efficient Gaussian blur with linear sampling*.

In your report, show your results (screenshots). Outline your implementation, including what resources/intermediate textures are necessary (and why). Describe how you have calculated the weights. Describe how you make available the weights to the shader and why you have chosen this approach.

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.