Project Report: Fancy Alpine Maps

Gerald Kimmersdorfer 01326608

November 27, 2023

Abstract

In this report, I detail the significant enhancements and optimizations done to the Alpine Maps renderer. Advanced techniques in shading and shadow mapping were integrated to boost visual effects and performance. Benchmarks on two GPUs, an NVIDIA GeForce RTX 3060 Ti and AMD Radeon HD 7700M, demonstrated the effectiveness of these updates. In addition to the current updates, the report also outlines the future work aimed at further enhancing the efficiency of the Alpine Maps renderer.

1 Introduction

The Fancy Alpine Maps project aims to increase visibility on an existing OpenGL renderer by adding additional geometry based effects. Specifically Shading and screen space ambient occlusion (SSAO) are intended to highlight geometry that was not visible when just using the orthographic images.

2 Implementation

During implementation and testing of the proposed effects further tasks and features arose to be necessary. This report will highlight all of the changes made to the Alpine Maps renderer during the course of this project. The order of the following sections resembles the order of implementation, not necessarily the order of importance of the respective changes.

2.1 Uniform Buffer Objects

To send small blocks of data to the GPU a base class for Uniform Buffer Objects was implemented. An alternative would have been to bind seperate uniform objects, but since UBOs can easily be bound to multiple shaders we opted for this method. Furthermore by allowing the UBO-struct to be a `Q_GADGET` it is possible to send the whole configuration block to the gui thread, edit it via QML, and send it back to the rendering thread. The use of signals and slots guarantees thread-safety for those transfers. The `gl_engine::UniformBuffer`-class is a templated class which takes an arbitrary struct

as type. This type is the data-container for the specified UBO. The possible options for the template are defined in the `gl_engine/UniformBufferObjects.h`-file.

As of the time of writing the following UBOs are in use:

1. *uboSharedConfig* (GL-Name: `shared_config`, size: 152 bytes)
   Contains all editable configuration parameters for the users. Can be serialized and used in QML.
2. *uboCameraConfig* (GL-Name: `camera_config`, size: 416 bytes)
   Contains various camera matrices and their inverse as well as view-port properties.
3. *uboShadowConfig* (GL-Name: `shadow_config`, size: 336 bytes)
   Contains lightspace-matrices and cascade information for CSM. Default implementation uses 4 cascades.

2.2 GUI Rework

In order to be able to adjust parameters for the various effects it was necessary to implement QML Components that can easily be used and extended in a modular fashion. Existing components were changed to fit the material design optics and to make them more responsive when used on targets with different screen sizes.

2.3 Frame Profiler

The newly created `TimerManager` class allows to capture time differences both on the CPU and on the GPU side. GPU-Timing is limited to targets that use an OpenGL context, as WebGL and OpenGL ES 3.0 do not have support for GPU timing queries. At the end of the frame the TimerManager sends captured data via a signal to the `TimerFrontendManager` who is in charge to store the data inside `TimerFrontendObjects`. The amount of frames stored can be individually configured for each timer. The appropriate frontend elements inside the **Statistics-Window** are spawned using QML. Timings can be compared using either a pie- or a line-chart and by their exact or averaged values.

2.4 Shading

For proper shading effects surface normals are necessary. You can either calculate them in the fragment shader[1] by utilizing the OpenGL functions `dFdx, dFdy` or you calculate them per vertex and let the GPU interpolate it for each fragment. The second solution allows for smooth normal transitions between faces. It is computationally more expensive as we need to sample the height texture 5 times for each vertex. For shading itself we use the computationally cheap Blinn-Phong model. Shading properties may be adjusted via the `shared_config`-ubo.

[1] `https://www.enkisoftware.com/devlogpost-20150131-1-Normal-generation-in-the-pixel-shader`

2.5 Deferred Shading

Calculating ambient occlusion in screen space needs position and normal data. To keep the computational loss of overdraw as small as possible a Deferred Shading pipeline is advised. For this matter the tile-stage of the rendering pipeline has been refactored as the geometry-stage where material properties are written inside the g-buffer. The backbuffer-stage was refactored and is now in use as the lighting-stage where shading is calculated.

2.6 GBuffer Layout

The final gbuffer contain 4 textures as described in table 1 resulting in a size of 27 byte per pixel.

ID	Format	Size	Usage
1	RGB8	3 byte	albedo
2	RGBA32F	16 byte	position and distance
3	RG16UI	4 byte	octahedron encoded normals
4	R32UI	4 byte	encoded distance for readback

Table 1: G-buffer Texture Layout

Various attempts were made to reduce the size of the gbuffer even further, especially by compressing the positions. All of those attempts had drawbacks, summarized in the following paragraph:

1. **Reconstruction of position using the z-buffer**: Even the reconstruction using a floating point depth buffer is not accurate enough given the huge camera spectrum. Reverse-Z was also attempted and betters the situation significantly, but is not supported on WebGL and OpenGL ES 3.0 as they store the depth within the bounds of $[-1.0, +1.0]$ opposed to $[0.0, +1.0]$.
2. **Use RGB32F for position**: The distance can theoretically be derived by the position, but RGB floating buffers are not supported on WebGL and OpenGL ES 3.0.
3. **Use RGBA16F for position**: This is indeed possible, but the reduced resolution will result in visible artefacts for points far away from the camera.
4. **Reconstruction of position using the readback-buffer**: The readback buffer is saved with a custom encoding function that actually doesn't work for positions further away than 442413 length units in world space. It's optimised in a way that the reconstruction of closer points have more accuracy. This is good enough for picking, but reconstruction for shading is not possible using this approach.

2.7 Screen Space Ambient Occlusion (SSAO)

The implementation is based on the tutorial [2] by Joey de Vries. It is supposed to be a crude check to see if ssao can yield better visual results. It's a very straightforward

[2] `https://learnopengl.com/Advanced-Lighting/SSAO`

implementation with one ssao screen pass to calculate the ssao buffer, and a blur pass to hide the high frequency artefacts that we get through the random rotation of the vectors. The algorithm itself could be made faster by several times. Please see section 4 for more information.

2.8 Cascaded Shadow Mapping

The cascaded shadow mapping is based on the tuorial [3] by Márton Árbócz. In the default configuration we render 4 shadow maps withing the following bounds (distances in world space to camera):

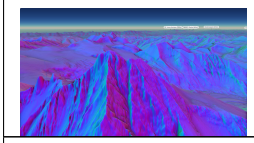
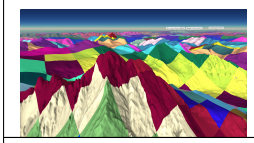
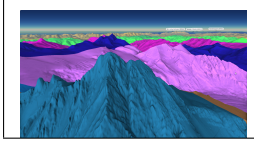
- Map 1: 1.0 (near-plane) - 2.000
- Map 2: 2.000 - 4.000
- Map 3: 4.000 - 10.000
- Map 4: 10.000 - 100.000

The shadowmaps itself are rendered in a resolution of 4096x4096 pixel. As opposed to the original solution proposed by Márton Árbócz we don't store the individual shadowmaps in a `GL_TEXTURE_2D_ARRAY` but in individual textures. Even though OpenGL ES 3.0 would support this method `QOpenGLTexture` does not.

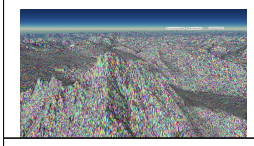
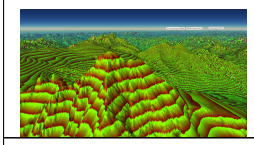
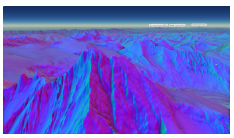
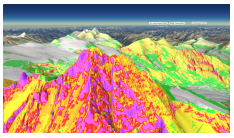
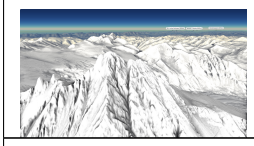
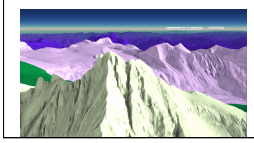
2.9 Further implementations

The following subsections contain information about, what we consider smaller changes for the Alpine Maps Renderer.

2.9.1 Overlays

Preview	Name	Description
	Normals	Shows the calculated normals per fragment where the neutral normal is encoded as rgb(0.5, 0.5, 1.0)
	Tiles	Shows the different tiles and tile borders rendered in the current view.
	Zoomlevel	Shows the zoomlevel of the tiles in the current view.

[3] `https://learnopengl.com/Guest-Articles/2021/CSM`

	Vertex-ID	Shows the vertex id and can be used to visualize the granularity of the underlying mesh.
	Vertex Height-Sample	This overlay represents the sampled height value out of the heightmap tiles.
	Decoded Normals	Normals are octahedron encoded in the g-buffer. This overlay shows the decoded normals in the same fashion as the other "Normals" Overlay.
	Steepness	This is supposed to be an example overlay for future overlays to come. It has a distance based falloff and depicts the steepness of the underlying fragment. The steepness gets evaluated based on the surface normal and gets binned in 9 differntly colored bins.
	SSAO Buffer	Shows the content of the SSAO buffer.
	Shadow Cascades	Highlights which shadow map is used for each fragment.

2.9.2 Shader precompilation

In the original renderer shaders could be attached to each other in the c++ code. To end up with a more standardised approach a shader precompilation step was implemented which supports the already well established `#include`-directive. Furthermore a helper function was introduced that catches the Qt OpenGL errors and additionally outputs the exact code line where the error appeared. Otherwise debugging is a hazzle as line numbers don't match anymore due to the include statements.

2.9.3 Shader download service

As the shaders get packed into a resource file when targeting the Web and Android, hot reload of shader files wasn't possible for those devices. Due to the long compilation time of the web version it was necessary to find an alternative to the old method. The shader download service is exactly this workaround and can be activated with the cmake option `ALP_ENABLE_SHADER_NETWORK_HOTRELOAD`. If this option is true shaders will be downloaded from the respective URL set by `ALP_SHADER_NETWORK_URL`.

Hosting shader files

The shader files are not automatically hosted on a local address. This needs to be done manually. There are millions of options on how to do so, but to name a few: local appache server, python http server or the live server addon for Visual Code. Make sure the hosted files exactly point to the shader directory at `gl_engine/shaders`

2.9.4 Height lines

A simple version of height lines are implemented as part of the compose step. The thickness of the line is derived by the fragment normal. Otherwise lines on flat surfaces will be much thicker than ones on steep surfaces. For proper height lines an additional rendering step might be necessary. More on optimisation ideas can be found in chapter 4.

2.9.5 Camera Presets

In order to benchmark the application the switching between camera positions was necessary. The set of predefined camera definitions can be found and edited in `nucleus::camera::PositionStorage`.

2.9.6 Sun Angle Calculation

The angle of the sun light can be calculated based on the geographical position and the time of the day. This calculation was implemented into the `nucleus`, based on the JS library suncalc. [4]

2.9.7 URL Parameters

Targeting web platforms, EMSCRIPTEN allows for easy access to the URL of the web page. Currently 5 different settings are saved and retrieved (on load) via url-parameters.

- **GL-Configuration** *config*: This parameter represents the whole `shared_config` ubo block serialized, compressed and converted to base-64.
- **Camera-Position** *campos*: This attribute holds the camera position in the format of `%latitude%_%longitude%_%altitude%`.
- **Camera-Lookat** *lookat*: This attribute contains a point that the camera is facing in the format `%latitude%_%longitude%_%altitude%`. This point will automatically be calculated based on the camera orientation and a fixed distance.
- **Simulation Date** *date*: Contains the datetime in urlsafe ISO format which can be set in the frontend.
- **Quality** *quality*: Contains the quality which represents the permissible screen space error when fetching tiles. It's the same value that can be adapted and set in the frontend.

[4] `https://github.com/mourner/suncalc`

3 Performance

The benchmarks were done with the following setup:

- GPU 1: NVIDIA GeForce RTX 3060 Ti *(Launch-Date: 19.06.2022)*
- GPU 2: AMD Radeon HD 7700M *(Launch-Date: 15.02.2012)*
- OS: Windows 10 Pro
- Exclusive Fullscreen with Full-HD resolution (1920x1080)
- Settings-Preset 1 (”Großglockner”, camera-preset: ”großglocker”):
- Settings-Preset 2 (”Karwendel”, camera-preset: ”karwendel”)
- DateTime (for shadowing): 20.11.2023 09:00

The deployed windows applications used for testing can be downloaded, up until the 07.May 2024, at the following url: `https://tuwienacat-my.sharepoint.com/:f:/g/personal/e1326608_student_tuwien_ac_at/EvivvJ3ucd9Io2Yx6E76CQ4BJFEkznK9pW3u9eqKNWf5Bw?e=Kcrfxl`.

About the benchmark target

Only desktop builds were used as there is no support for GPU timing on WebGL and Opengl ES. The rendering speed of the different rendering targets may vary significantly though as the underlying hardware can be quite different. (eg. Mobile targets often use tile base GPU architecture)

3.1 Tile-Sorting

One of the first improvements in performance was the implementation of a tile-sorting step to reduce overdraw. Meaning on the host side we sort the tiles, such that we send the tiles closest to the camera to the GPU first. This yields an increase in rendering speed depending on the current view. (see figure 1)

Even though in our benchmark the speed increase was quite low (in between 1%-10%), views with a lot of tiles being obscured can yield way higher performance gains.

3.2 Deferred vs. Forward-Rendering

The forward rendering version which was used for the comparison is based on the commit: ”change of camera presets in gui” *(0ac7a13c7aefee74dd1d522c8b765ffd5552c9c1)*. For the benchmark all shading and post processing effects were disabled. The results can be seen in figure 2.

The deferred renderer performs faster even without shading! One reasonable explanation is that the atmospheric calculations are heavy enough, such that the lack of overdraw in the deferred setup is resulting in a higher speed increase than the cost of the overhead of the deferred pipeline.

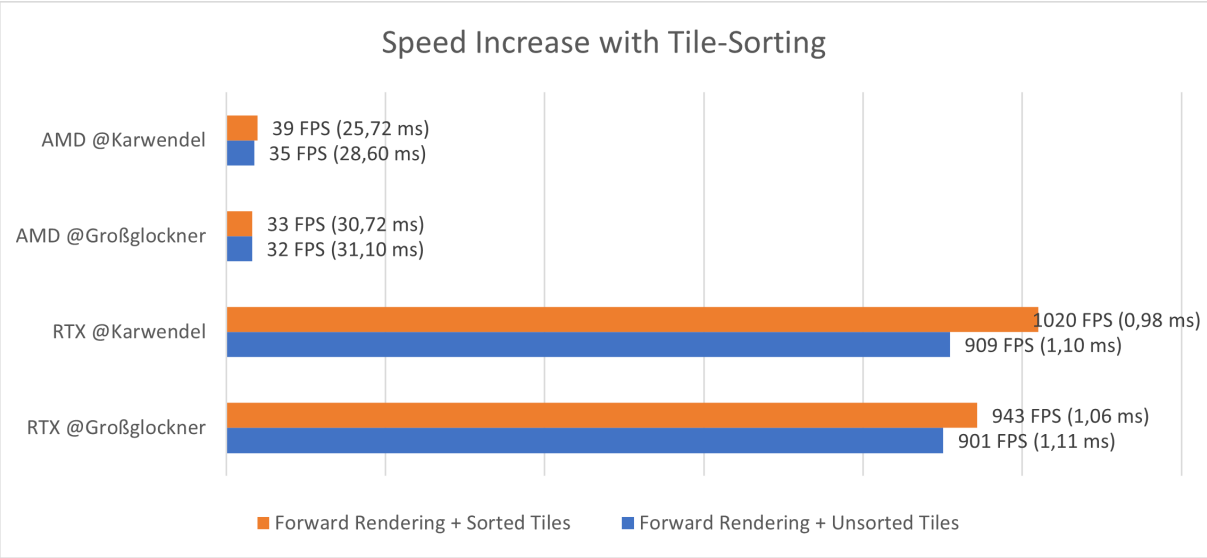


Figure 1: The speed increase by the introduced tile sorting depends solemnly on the current view. In our benchmark the speed increase lies in between 1%-10%

3.3 Rendering Stages in comparison

Figure 3 compares the impact of different post processing shading effects. It highlights the importance of better optimised algorithms especially for the ambient occlusion and shadow calculations, since those effects by themselves reduce the performance by about 80%. Substantial speedups are possible for both effects as discussed in section 4.

Figure 4 compares the individual stages on different machines. Interesting to notice is that the benchmark on the old AMD graphics card shows that the compose/shading step is coming at a much higher cost than on the RTX-card. That could be explained by either the shading or the atmosphere calculation getting differently optimised by both graphic cards. It also explains the huge difference in figure 2 for the AMD card.

4 Future work

4.1 Shader-Meta-Compiler

Currently a lot of if clauses based on the current `shared_config` are evaluated in the shaders during runtime. An alternative would be to extend the current approach of the pre-Compiler to also support injecting pre-compiler constants into the shader code before compiling the GL-program. That way we could use `#if`-statements and would also be able to forward global pre processor variables, such that for example we can disable code based on the build target. It would result into the necessity of recompiling the shader files in some cases (for example on overlay change), but may result into slightly better performance.

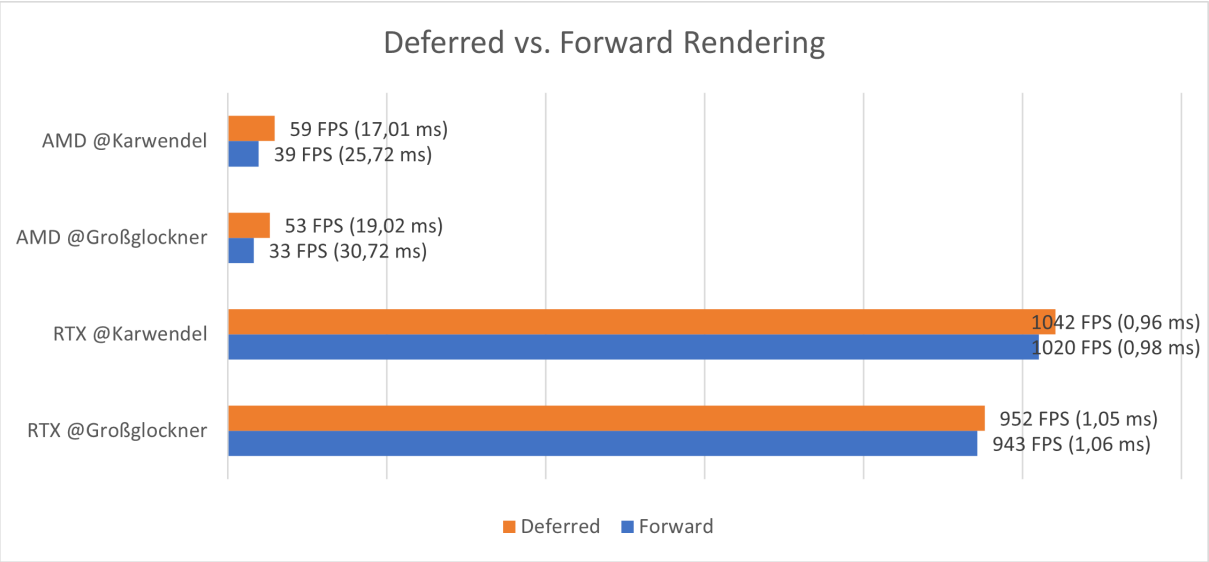


Figure 2: The deferred pipeline performs slightly better in all test scenarios.

4.2 Optimise SSAO

The SSAO step might be the easiest to optimise. Proper adaptions could result in a significant speed increase. The already implemented algorithm could be extended by the following steps:

- **Reduced Resolution**: Currently we Gaussian blur the resulting ssao buffer. An alternative might be to reduce the depth buffer to half the size, apply ssao and afterwards execute a depth aware upscale step. Even though this approach might lead to small geometry getting lost during the down sampling step, it could increase rendering speed significantly. The small error might be hidden by the low frequency nature of the ssao effect. A reference implementation (which uses HBAO though) can be found at: `https://github.com/nvpro-samples/gl_ssao`
- **Depth dependent Sample-Count**: Far in the distance proper SSAO is not as important as the primary target the user is looking at. We could therefore reduce the sample count as a function of the distance to camera. A similar approach is also proposed in Mcguire et al. (2011).

Alternatively more modern approaches to SSAO could be implemented.

- **DeepAO**: Zhang et al. (2020) proposes a compute step where a generative network combines normal and depth map into a depth buffer. As the 2.5d tiles currently in use have a very distinct shape and resolution, the training on this data might actually work quite well. They also found their solution to run faster compared to other state-of-the-art methods. This solution requires Compute-Shaders which are not accessible in the current OpenGL/OpenGL ES Renderer.
- **VAO++**: Bokšanský et al. (2017) proposes to use Volumetric Ambient Occlusion which first analyses the geometry in the depth buffer to realise a smaller sampling radius. That means less samples might result in ssao of similar visual quality.

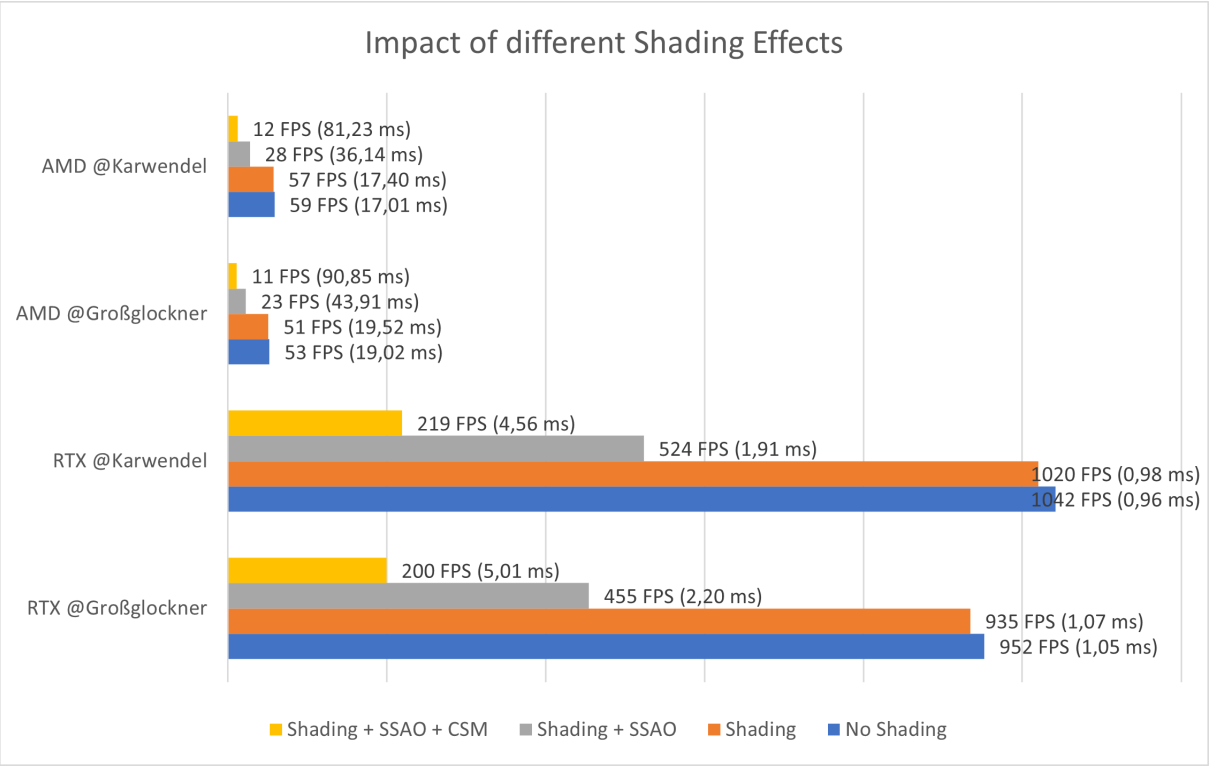


Figure 3: Activating Shadow-Maps and Ambient Occlusion leads to a significant drop in the frame rate.

4.3 Optimise shadow algorithm

In general the cascaded shadow mapping which is currently implemented might not be the ideal solution for Alpine Maps due to several reasons:

- Especially for top-down views most of the cascades are actually unnecessary and empty.
- With the fine granularity of our mesh, shadow map rendering is quite slow.

Shadow Tiles

Currently we use the same set of tiles for the rendering of the shadow maps as we do for the camera view resulting in popping of shadows when moving around. It might be feasible to use a lower quality tile set fetched with respect to the lights perspective.

Additionally the following algorithms might be a better fit or a good addition for the Alpine-Maps project:

4.3.1 Screen space shadows

Screen space shadows [5] can add fine small scale details purely by interpreting the depth buffer. Especially from top down views or views far away from the camera it might be

[5] `https://www.bendstudio.com/blog/inside-bend-screen-space-shadows/`

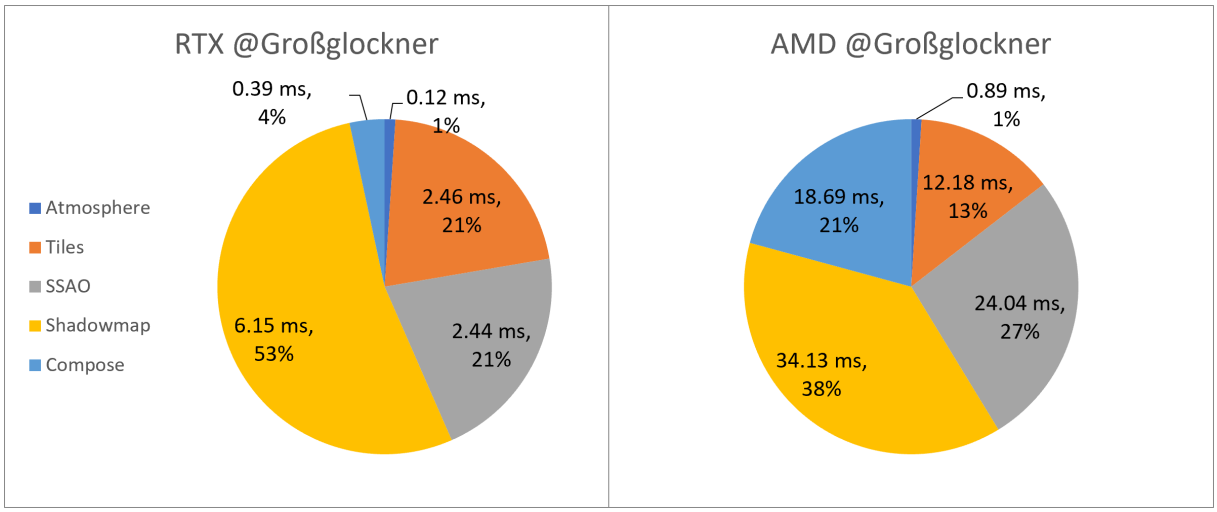


Figure 4: Detailed comparison of the processing times of the different stages of one frame. Even though both benchmarks were run on desktop GPUs the result in both cases is very different.

superior to traditional shadowing algorithms. It might be possible that with a proper screen space shadow step we will be able to reduce cascaded shadow mapping to less cascades. Furthermore bias issues that occur at the current version might be reduced.

4.3.2 Virtual shadow maps

This might be the best, but also most complicated shadowing solution for Alpine Maps. The great advantage is that Virtual Shadow Maps [6] (VSMs) provide a consistent shadow appearance across various distances, a significant improvement over the often low-resolution and blurry distant shadows produced by cascaded shadow mapping. VSMs require a bunch of modern graphics features though that might only be accessible with a native Vulkan/DirectX-Renderer.

4.4 Optimise height lines

An idea on how to gain height lines with a specific line width would contain two rendering steps. In this example I explain the idea on fixed height lines every 100 m in altitude, but the proposed solution could be extended to different scales.

- Render black/white alternately every 100 m into a heightline buffer (Meaning: First 100m are black, second 100m are white,...)
- Apply edge detection on the heightline buffer and alternatively increase black areas to achieve specific line width
- Merge with back buffer in compose step

[6] `https://ktstephano.github.io/rendering/stratusgfx/svsm`

References

Bokšanský, J., Pospíšil, A., and Bittner, J. (2017). VAO++: Practical Volumetric Ambient Occlusion for Games. pages 031–039. https://diglib.eg.org/bitstream/handle/10.2312/sre20171192/031-039.pdf.

Mcguire, M., Osman, B., Bukowski, M., and Hennessy, P. (2011). The alchemy screen-space ambient obscurance algorithm. pages 25–32.

Zhang, D., Xian, C., Luo, G., Xiong, Y., and Han, C. (2020). Deepao: Efficient screen space ambient occlusion generation via deep network. *IEEE Access*, 8:64434–64441.