UNIVERSITY of ABERTAY DUNDEE

HONOURS PROJECT DISSERTATION

CMP403

An evaluation of the memory bandwidth of the Visibility Buffer rendering technique in real-time graphics applications.

Author:
Alberto TAIUTI

Supervisor:
Dr Paul ROBERTSON

Industry mentor:
Dr Matthäus G. CHAJDAS

26th April 2017

CONTENTS

Abstract

In this paper the Visibility Buffer rendering technique is explored and its memory bandwidth usage is evaluated and compared against the memory bandwidth usage of Deferred Rendering. The main advantage of the Visibility Buffer rendering system is in reducing the memory footprint of a renderer, when compared to the memory footprint of a g-buffer in deferred shading, by using a smaller buffer and by keeping its elements format small, without affecting the frame time performance. This advantage in terms of memory footprint has already been assessed in previous works; however in these same works the memory bandwidth usage is simply stated as better than the one of Deferred Rendering but no measurements are provided as to quantify the claim. Hence we set out to perform measurements which will provide the data necessary to reason on the memory bandwidth usage of the Visibility Buffer and its implications compared to the one of Deferred Rendering. Different implementations of the Visibility Buffer are evaluated and one is selected for implementation, giving specific reasons for this choice. The same process is performed to choose a Deferred Renderer implementation. These implementations are then used to collect memory bandwidth performance data using using a specific instrumentation injection library which adds bandwidth counters to the SPIR-V modules compiled from the GLSL shaders. The results are presented and discussed in view of the previous works and of the chosen rendering architecture for both renderers.

Keywords: 3D real-time rendering, Visibility Buffer, rendering techniques

1 INTRODUCTION

Computer graphics is a field which is in constant evolution. New technologies and techniques are developed every year and there is a constant push for innovation both in terms of performance and of visual quality.
However, one specific area of improvement being constantly researched is how to reduce the memory footprint of the rendering framework used by applications so that more GPU memory is available for other purposes, such as storing textures or general usage buffers. This is especially relevant when put in the context of the games industry's push to achieve real-time 4K high-resolution rendering in computer games. Due to the high pixel density of the images being rendered at such

resolution, a large amount of the memory available on GPUs is used for storing the buffer images used by the rendering engine. This leaves less space for the texture maps used by the 3D models which also need to be of higher resolution to avoid aliasing with the mentioned higher resolution render targets. More so, with the advent of Virtual Reality headsets and their high rate of adoption by customers, it becomes even more important to explore how to reduce the memory footprint of the available rendering techniques given that to render a 3D VR environment, the whole scene needs to be rendered twice and the results of the rendering stored in memory for both the left and right eye, basically doubling the amount of memory required by the renderer.

A solution to this problem is to simply increase the memory available on the GPU, which some hardware vendors have started implementing. However this approach requires the end users to buy a new GPU and reduces the reachable audience if your game or 3D application requires such hardware. This aspect is even more evident on consoles, where the available hardware is fixed for a whole console cycle. Another solution is to develop and assess new software approaches

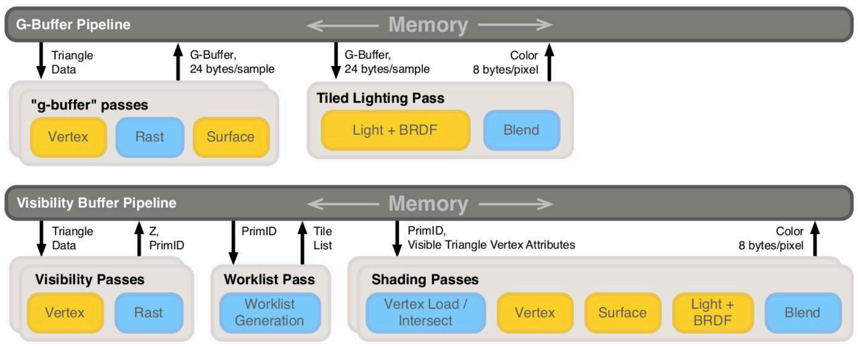


Figure 1 – *Comparison of the deferred rendering pipeline with the Visibility Buffer pipeline as shown by Burns and Hunt (2013). For this paper a different visibility buffer implementation is used but this diagram gives a good idea of the core differences between deferred rendering and visibility buffer rendering.*

which allow the memory efficiency of 3D renderers to improve with the currently available hardware without affecting the frame time performance. There has recently been an example of such a technique being brought forward, named in various ways by different authors but more generally referred to as the *visibility buffer*.

The visibility buffer rendering technique proposes to reduce the number of intermediary image buffers required to render a scene at the expense of more complex real-time calculations on the GPU, thus reducing the overall memory footprint of the renderer implementation. See Figure 1.

There have already been some developments on the usage of the technique, with various works proposing different variations of the same core concept. These works outline the benefits in terms of memory footprint of the visibility buffer compared to deferred rendering, and mention that there is also an improvement in terms of the memory bandwidth used.
However, they do not quantify this gain in memory bandwidth, which makes this subject a great candidate for an in-depth research on its impact on the performance of the visibility buffer technique. The visibility buffer technique requires to read the albedo maps in the second pass, which could potentially result in a higher memory bandwidth usage than the one for deferred rendering. Hence the question which arises is whether the gain in terms of memory footprint presented by previous works is met by an increase in the memory bandwidth used compared to deferred rendering. This investigation will set out to understand the implications in terms of memory bandwidth of the visibility buffer technique.

Therefore the research question for this project is:

> *How does the visibility buffer technique compare to deferred rendering in terms of memory bandwidth usage?*

The aim is to develop a sample rendering application which implements the visibility buffer technique and a deferred renderer so that these can be used to measure their memory bandwidth. A support library will be developed to aid in implementing a memory bandwidth usage system in the renderers.

The key objectives are:

- Further research the Visibility Buffer technique and relevant previous work.
- Assess whether additional techniques are required, e.g. shadow mapping, SSAO. Assess whether additional culling techniques are required, e.g. depth pre-pass, GPU triangle culling.
- Develop a sample application using a novel graphics API to demonstrate the techniques.

- Develop a library which allows to inject custom SPIR-V code in a binary SPIR-V module compiled from GLSL.
- Retrieve bandwidth performance data from both renderers and present it in an understandable manner for evaluation. Utilise more than one 3D scene.
- Compare the memory bandwidth usage of the visibility buffer and deferred renderer techniques utilizing multiple 3D scenes ranging in complexity and resolution.

2 Literature review

In the real-time graphics programming field there has always been great effort put into developing and researching new rendering pipelines and rendering techniques which would improve the performance of a given real-time application without excessive compromise in terms of final image quality.

Forward rendering is the standard, out-of-the-box rendering pipeline which is implemented via hardware by GPUs and exposed to the user via specific APIs (Akenine-Möller, Haines and Hoffman 2008; Zink, Pettineo and Hoxley 2011). These pipelines process and compute the shading of screen-space samples in triangle submission order. This technique however has several limitations. As the triangles are processed and their visibility computed, some fragments could end up being replaced by another fully-shaded fragment of another primitive which was submitted later to the pipeline, hence resulting in a phenomenon known as overshading (Akenine-Möller, Haines and Hoffman 2008; Schied and Dachsbacher 2015): fragments which will not end up being visible are fully shaded and resources are spent in their processing, even if they will not contribute to the final image. Another major issue with forward rendering is the small number of lights which can be considered for lighting if performance is of concern. Because of overshading, fragments which potentially will not contribute to the final image will still have the lighting calculations performed, resulting in wasted computations which, for a large set of lights, would result in a frame render time which is too large (Zink, Pettineo and Hoxley 2011).

Deferred shading improves over forward rendering by decoupling the lighting calculations from the geometry and shading computations, which are performed in two separate passes (Saito and Takahashi 1990). The first pass generates a series of buffers storing the surface attributes, and these are then used to perform the second pass, during which the surface fragments are shaded and illumination is taken into account. This way only fragments which are visible in the final image will be shaded. Thanks to this it is possible to have a larger number of lights in a scene compared to a classic forward renderer, since only visible fragments will be shaded (Zink, Pettineo and Hoxley 2011) and hence the irradiance is calculated less times. However, deferred shading presents various drawbacks.

For example, with deferred rendering lighting is still coupled with the shading of the fragments (Harada, McKee and C.Yang 2012). The light pre-pass renderer (Engel 2009) decouples these two operations by using a z-prepass which also outputs additional geometric surface information. It then uses a second pass to evaluate albedo surface attributes and a final one to compose the data and evaluate irradiance. The downside of this approach is that it requires submission of the scene geometry multiple times.

Another issue with deferred rendering has to do with the inability to easily integrate the use of different BRDFs for different parts of the scene during the same render pass. This is due to the shading pass being decoupled from the geometry submission and to the use of a single fullscreen pass for its evaluation. Forward + (Harada, McKee and C.Yang 2012) addresses this issue by combining forward rendering, for which complete material information is available, with light culling. It runs a z-prepass which is used to perform a fullscreen second pass during which the screen space is subdivided into tiles. The lights bounding boxes are tested against the tiles and assigned to each tile they intersect with. The list of per-tile lights is then used during the shading. This way the number of lights needed to evaluate irradiance per-fragment is potentially greatly reduced, making the technique feasible. However, this technique also requires submission of the scene multiple times which can present an issue in high-end game productions.

Deferred rendering also presents drawbacks in terms of memory usage, especially at high resolution render targets with MSAA enabled. With the advent of VR and 4K resolution TVs, there is now a high demand for being able to render to high resolution targets without losing performance and memory available for the high-resolution textures needed for such targets. Deferred rendering scales poorly as resolution increases, hence novel techniques are required if these higher frequency targets are to be supported on most recent hardware (Engel 2016).

Finally, deferred rendering cannot efficiently render transparent geometry due to the loss of geometry order of submission and of depth information: the information about what geometry was behind the topmost fragment is lost. The solution is to run a forward rendering pass with geometry sorted back-to front (Zink, Pettineo and Hoxley 2011; Burns and Hunt 2013; Akenine-Möller, Haines and Hoffman 2008)

The Visibility Buffer technique was first introduced by Burns and Hunt (2013). In their paper they propose to replace the G-buffer with a visibility buffer, which holds the visibility part of equation (1) (Akenine-Möller, Haines and Hoffman 2008) for every sample in screen-space. The main scope of their work is to present a novel technique which aims at providing an alternative renderer which uses less GPU memory when compared to a deferred renderer while still keeping performance at a satisfactory level.

$$L_0(\mathbf{v}) = \int_\Omega f(\mathbf{l},\mathbf{v}) \bigotimes L_i(\mathbf{l})(\mathbf{n}\cdot\mathbf{l})d\omega_i \tag{1}$$

They use a first rendering pass to store a triangle index and instance ID per screen-space sample, encoded as an integer of four bytes and subsequently use this information in a second rendering pass to compute the barycentric coordinates of the triangles by retrieving the vertices using the triangle index store in the first pass. Once the barycentric coordinates are available, they are used to interpolate the vertex attributes to the sample location and perform shading. This allows them to more clearly separate the visibility determination phase from the shading phase and has the benefit of reducing the memory footprint of the buffers necessary to implement this rendering technique compared to a deferred renderer. Burns and Hunt (2013) compare their implementation using different types of GPUs, varying between Dedicated and Integrated ones. They show that their implementation of the visibility buffer technique has a positive impact in terms of cache hits, especially on SoC GPU architectures. However, the memory bandwidth required in sampling the albedo maps from the second pass is not taken into account.

Schied and Dachsbacher (2015) expand on the work by Burns and Hunt (2013) and recognise the advantages of the Visibility Buffer over deferred shading. Their solution proposes a different structure of the Visibility Buffer than the one by Burns and Hunt (2013). Their Visibility Buffer uses multiple passes to store the triangle data and their buffer address in two separate memory locations, making use of a memoization cache (Liktor and Dachsbacher 2012) to achieve it. Furthermore Schied and Dachsbacher (2015) use the partial derivatives with respect to screen-space of the barycentric coordinates in x and y to calculate the vertex attributes at the shading point. They obtain this data using a separate compute pass. Their results show that they avoid redundant shading by means of the additional data they store to identify duplicate visibility samples, which results in a performance improvement over deferred shading while at the same time reducing the GPU memory usage. In comparison to the work done by Burns and Hunt (2013), their implementation consumes more memory, but can achieve higher performance improvements (Schied and Dachsbacher 2015). This work, like the one by Burns and Hunt (2013), does not account for the memory bandwidth required in sampling the albedo maps from the shading pass.

Engel (2016) merges solutions from both Schied and Dachsbacher (2015) and Burns and Hunt (2013) to propose an efficient implementation of a renderer based on the Visibility Buffer technique. In their GDC presentation they show how they use a single visibility buffer coupled with GPU-side triangle filtering and culling to achieve optimal rendering performance while also keeping the memory usage low when compared to deferred shading at a 4K resolution and with MSAA enabled. The results presented highlight how the visibility buffer does not scale as fast as the deferred rendering G-buffer does as the resolution and MSSA scale from 1080p to 4K and from 1 to 4 samples respectively. In order to achieve a stable frame rate at that resolution, they rely on GPU-side geometry culling via the GeometryFX library (Chajdas 2016b) and the use of indirect draw calls exposed by the most recent rendering APIs such as Vulkan (The Khronos Group 2016a). Engel (2016) argues that these results are particularly important because they show how 4K resolutions and VR can be supported by lower-end hardware.

However, the previous papers on the Visibility Buffer do not precisely measure and assess the GPU memory bandwidth used by the technique. In these works it is stated that the Visibility Buffer pipeline reduces the GPU memory used to render the scene when compared to other rendering pipelines, (Burns and Hunt 2013; Schied and Dachsbacher 2015; Engel 2016) but the memory read and written to implement this technique is not accounted for.

Hence, as introduced in Section 1, this paper will assess whether the improvements in terms of the memory footprint of a renderer introduced by the Visibility Buffer technique also introduce a high memory bandwidth usage which could result in a drawback larger than the improvements.

3 Methodology

3.1 Framework

The renderers share common functionalities which are implemented and are available as a C++ framework. Some of these functionalities include model loading, shader loading and management, texture loading and management, and more.

3.1.1 Renderers

The main focus being on measuring the memory bandwidth of the renderers, these do not present any advanced rendering technique such as shadow mapping or SSAO. Implementing techniques which improve the visual quality of the final render was considered superfluous and would have not changed the outcome of the memory bandwidth measurements.

Shared functionalities The chosen BRDF is the Phong-Blinn (Blinn 1977) model with direct lighting from lights modelled as point lights for both renderers.
The framework functionalities used are the same across both renderers, even if used in different ways to accomplish the required rendering technique. For example the deferred renderer defines more render target textures than the visibility buffer renderer.
Shader compilation from GLSL to SPIR-V format is done on-line using the *shaderc* library (Google 2017).
Samples captures, explained in Section 3.1.4, perform both data measurement and screenshot taking.
The tonemapping operator used for both renderers in their tonemapping pass is the so-called Reinhard operator (Reinhard et al. 2002). Other operators would be available, such as, for example, the so-called Filmic operator (Hable 2010).

Both renderers always read from five different types of albedo maps to perform shading, even in the case where for a given material such maps would not be available. This way the results are consistent and the number of albedo maps being read does not vary across different sample captures. This is accomplished by passing a dummy `.ktx` texture to the shader resources creation where a given albedo map is missing at the time of model loading. The dummy texture is shared among all the materials which need one. Therefore certain materials could present artefacts due to reading a map which was not designed to be used with that material, however this does not represent a problem in the context of the objective of this project.

The early-z functionality is activated to avoid excessive overshading during the first pass by adding a specific layout specification in the fragment shaders as shown in Listing 1. It is also activated in the fragment shader of the second pass

```
...
layout(early_fragment_tests) in;
...
```

Listing 1 – *Activating the early fragment test from a GLSL fragment shader.*

and used in combination with the stencil buffer to mask the fragments which have been shaded during the first pass of both renderers and avoid running the fragment shaders of the second pass on fragments which are not covered by geometry, i.e. the parts covered by the skybox. The contents of the stencil buffer in a situation where some of the fragments have not been shaded during the first pass are shown in Figure 2.
Finally both scenes have 400 dynamic point lights.

Deferred renderer

G-Buffer The chosen implementation of a deferred renderer makes use of a G-Buffer with 4 buffers:

- Depth buffer, `DEPTH32_STENCIL8`

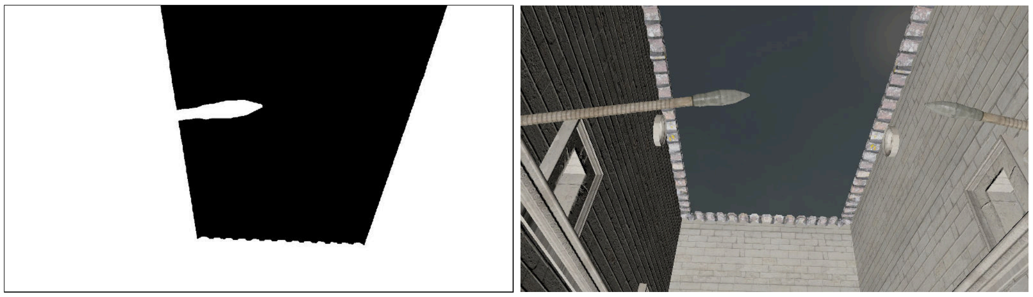

(a) *Stencil buffer.* **(b)** *Relative composed image.*

Figure 2 – *Captures showing the stencil buffer and the relative final, composed rendered image. The stencil buffer marks the pixels which pass the visibility and coverage tests during the first pass of both renderers, so that the sky section of a final image is not shaded in the second pass.*

- Normals buffer, `R16G16B16A16_FLOAT`
- Diffuse albedo buffer, `R8G8B8A8_UNORM`
- Specular albedo buffer, `R8G8B8A8_UNORM`

where the position is reconstructed from the depth buffer using the techniques explained by Zink, Pettineo and Hoxley (2011) and Pettineo (2009).

Pipeline The pipeline for the deferred renderer consists of 4 subpasses:

- G-Buffer construction
- Lighting and shading
- Tonemapping
- Skybox rendering

which are represented in the application by instantiations of the class *Renderpass* which represents a wrapper for the Vulkan definition of a renderpass (The Khronos Group 2016a). In order to reconstruct position from depth during the lighting and shading subpass, a vertex shader is also run which creates a view-space ray from the camera position to the back of the frustum (Zink, Pettineo and Hoxley 2011; Pettineo 2009).

Visibility buffer renderer

Buffers The chosen implementation of a visibility buffer renderer makes use of more than a single buffer as explained by Burns and Hunt (2013), instead doing something more akin to what introduced by Schied and Dachsbacher (2015). These buffers are:

- Visibility and barycentric coordinates: `R32G32_UINT`
- Texture UV derivatives in screen space X and Y `R32G32_UINT`
- Depth buffer, `DEPTH32_STENCIL8`

where the position is reconstructed from the depth buffer using the techniques explained by Zink, Pettineo and Hoxley (2011) and Pettineo (2009).

The visibility and barycentric coordinates buffer encodes the two different informations in its two different channels. The red channel stores the the ID of the triangle and the drawcall number, encoded into a 32-bit unsigned integer; this effectively represent the visibility buffer as described by Engel (2016). The green channel encodes the barycentric coordinates for the triangle to which the fragment belongs. Barycentric coordinates are part of the graphics processing pipeline and they are used to interpolate vertex attributes to the fragment location in screen space. They are defined with respect to the position of the fragment within the area covered by the triangle to which the fragment belongs and are calculated using what is commonly referred to as the *Edge Function*. Barycentric coordinates allow to define vertex attributes at the vertices of the triangle to which the fragment belongs and then interpolate their values for any fragment which the triangle covers in screen space (Scratchapixel 2016).
For this application the barycentric coordinates are acquired from the fragment shader of the first subpass using the AMD GCN Vulkan extensions (Chajdas 2016a). These extensions provide the barycentric coordinates either with correct

perspective interpolation or without. For this application, the perspective correct barycentric coordinates are used. The extensions expose only two of the three barycentric coordinates, since the third one can be calculated via the identity in equation 2 (Akenine-Möller, Haines and Hoffman 2008).
A workaround was also necessary to obtain the barycentric coordinates at the right respective vertices; see Listing 6 in Appendix A for the code used in the workaround.

$$\lambda_0 + \lambda_1 + \lambda_2 = 1 \tag{2}$$

Figure 3 shows the barycentric coordinates rendered to a debug output buffer, before encoding into a `UINT32` buffer.

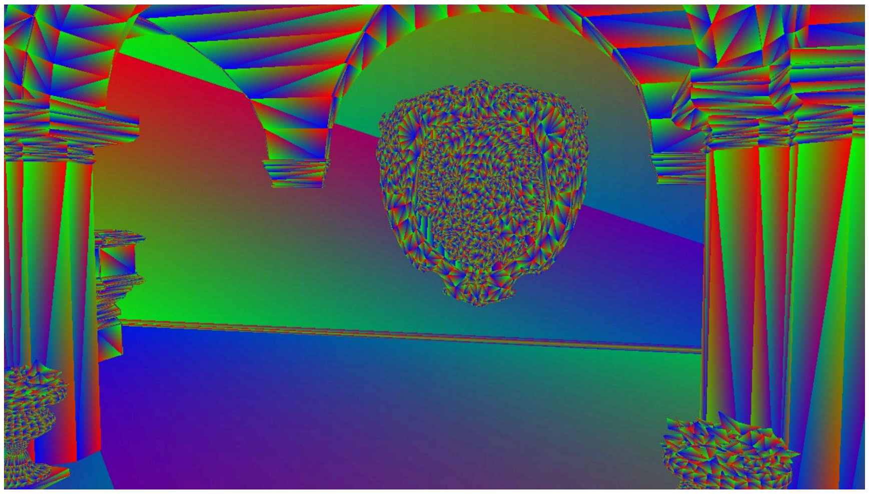


Figure 3 – *The raw barycentric coordinates before storage in the buffer. They have been output to a debug buffer during the first subpass before they are encoded and then stored in the visibility + barycentric coordinates buffer. These coordinates are provided by the AMD GCN GLSL extensions.*

The barycentric coordinates as provided by the AMD extension are floating point values. Hence in order to store them into a 32-bit unsigned integer, they are encoded by the GLSL instruction in Listing 2 which accepts a 2-component floating point vector and returns a 32-bit unsigned integer. The precision offered by 16-bit floating point values is enough for the usage of barycentric coordinates in this application. The same instruction is used to encode each derivative in the X and Y screen space directions of the UV coordinates into two 32-bit unsigned integers. This encoding system allows to achieve an acceptable precision while at the same keeping the size of the buffers to a relatively small size.

```
...
packUnorm2x16(vec2 val);
...
```

Listing 2 – *GLSL instruction for packing 2 16-bit floating point values into a 32-bit unsigned integer.*

Pipeline The pipeline for the visibility buffer renderer consists of 4 subpasses:

- Visibility pass
- Lighting and shading
- Tonemapping
- Skybox rendering

which are represented in the application by instantiations of the class *Renderpass* which represents a wrapper for the Vulkan definition of a renderpass The Khronos Group (2016a). In order to reconstruct position from depth during the lighting and shading subpass, a vertex shader is also run which creates a view-space ray from the camera position to the back of the frustum (Zink, Pettineo and Hoxley 2011; Pettineo 2009), in the same way done for the deferred renderer. The implementation of a visibility buffer renderer by Burns and Hunt (2013) and Engel (2016) does not make use of the additional barycentric coordinates and derivatives buffers and calculates them in the second subpass before shading. However, since the focus of this application is on the memory bandwidth rather than the memory footprint of the visibility buffer renderer, using additional buffers does not change the final outcome, as their presence is accounted for.

3.1.2 SPIR-V Instrumentation

One of the tools necessary to achieve the objective of being able to measure the memory bandwidth used by a given rendering technique is SPIR-V shader instrumentation. The reason behind this is that the renderer needs to be monitored so that every memory read or write can be logged using atomic instructions, however without having to manually modify the GLSL shaders to include atomic increments or declare the descriptor sets necessary to describe the buffers containing the logged data.

The final objective is to inject atomic increment instructions which will be executed by the SPIR-V module at runtime every time a specific instruction we are interested in counting is also executed. For this project the ones which are of interest are texture reads and writes. To achieve this, first an instruction is constructed using the class *Instruction* provided by the library shaderc (Google 2017). This class represents a SPIR-V instruction and implements an interface which allows to create an instruction and then retrieve its binary representation. Knowing what values to pass to construct a correct SPIR-V instruction, in this case an atomic increment instruction, is not trivial. Some instructions can be injected and work without needing additional instructions executed before them; however the majority of instructions will need type declarations, constant definitions, and more. This is in fact the case for atomic increment instructions. They do not need additional type and buffer definitions for themselves, rather descriptor sets describing the buffer to be incremented by the operation and their respective necessary definitions.
In order to know what parameters to pass and what instructions to execute to allow the atomic increment instruction to run, the steps followed are:

- Create a sample, bare bones GLSL fragment shader which executes an atomic increment.
- Compile it to human-readable SPIR-V using `glslangValidator`.
- Analyse the output module and recognise what are the required instructions.

Once these have been identified, it is possible to inject the necessary instructions in the correct order into the original module which needs to have memory bandwidth measured using the library described in Section 3.1.3. Appendix B presents

Listings 7 and 8 which show an example of an instrumented SPIR-V module before and after it was instrumented as explained above.

The atomic instructions need to be provided with memory on which to operate. For this reason the injection can not be implemented completely opaquely with respect to the application as if it did not have instrumentation. The application thus defines Vulkan descriptor sets and descriptor layouts which are backed by host and device readable buffer memory. The descriptor sets created are bound to specific binding IDs which correspond to the IDs used by the injection instructions. Descriptor sets and layouts are the way to describe resources, e.g. memory or sampler resources, to be used by a rendering pipeline in Vulkan (The Khronos Group 2016b).

The interface of the framework's *Shader* class provides the ability to set whether to enable memory bandwidth measuring for a given GLSL shader described by an instantiation of that class. If enabled, this will perform the instrumentation injection as explained in Section 3.1.3.

3.1.3 INSTRUCTION INJECTION LIBRARY

The instrumentation tool is implemented as a C++ library which resides in a separate code repository from the one developed for the main application. This library provides two classes named *OpcodeStream* and *OpcodeIterator*. *OpcodeStream* represents a SPIR-V module which can be manipulated via *OpcodeIterator*s to insert or remove SPIR-V instructions in the module it represents.
When a new *OpcodeStream* is instantiated with a SPIR-V module in the form of a stream of bytes, it parses the module and builds a list of every instruction present in the module. To achieve this, the SPIR-V specification (Kessenich, Boaz and Raun 2017) was used to determine the structure of a given SPIR-V instruction so that an algorithm could be developed. Once the module has been parsed, *OpcodeStream* contains a list of the instructions for the module it represents in the form of an array of offsets measured in bytes between successive instructions. The class' interface resembles the one provided by the C++ STL containers, where the containers can be manipulated via iterators. Hence it is possible to use the provided iterator class to iterate over every instruction of the module, check its properties and act in accordance to these, for example by replacing the current instruction with another one or by inserting a new instruction before or after. Once

the stream has been modified, the object of type *OpcodeStream* can then emit the new module for usage by an application.

3.1.4 MEMORY BANDWIDTH MEASUREMENT

The bandwidth is measured at the end of each frame, after the whole pipeline has been executed and the shaders have completed working. At this point of the pipeline, if memory bandwidth data capture is enabled, the GPU memory buffers contain the measurements of the memory bandwidth used. Reading back this memory to the host for manipulation is achieved by synchronising the Vulkan command buffer submission, representing work sent to the GPU for processing, with the start of the reading phase by means of a Vulkan fence. Vulkan fences are one of the available synchronisation primitives in the API. They are used as a coarse way to synchronise work between the GPU and the host (The Khronos Group 2016b).
Once the fence is signalled, the application can proceed to map the Vulkan buffer using the provided API methods and read the data into the host application memory for later processing. Moreover, it is necessary to have the ability to compare the samples with the position in which the camera was at the time of sampling. Hence the renderers implement a function to take a screenshot of the first frame during which the bandwidth is read and sampled.

Captured data The memory bandwidth which needs to be measured is in terms of the memory reads and memory writes. That is, every time a texture is read or written in a shader, this needs to be logged and the counters incremented.
The counters take the form of one single contiguous Vulkan buffer which is visible both by the host and the device. This buffer is not shared among the two renderers; each renderer, which lives in its own application, declares one and manipulates its own one.

```
...
texture(...);
atomicAdd(first_pass[0]);
...
```

Listing 4 – *Example GLSL logging of number of memory reads.*

```
...
layout (std430, set = j, binding = k) buffer AtomicsBuff {
  uint first_pass[2]; // Reads 0, Writes 1
  uint second_pass[2]; // Reads 0, Writes 1
};
...
```

Listing 3 – *Example GLSL declaration for the buffers to be used to log the memory read or written by a shader.*

In the shaders it is declared as if separated into two arrays, as shown in Listing 3. The code in Listing 3 does not represent what is written in the GLSL shaders; instead it represents the equivalent of what is injected via the instrumentation library in form of binary SPIR-V instructions. The same applies for Listing 4, which shows what the GLSL equivalent of the injected SPIR-V instruction would be for logging the number of memory reads. The logging of memory reads would be similar, with the difference being in the index used into the buffer: 1 instead of 0.

Sample capture The sample memory bandwidth data is captured and stored into a data structure which grows as the number of samples increases. The processing is done at a later phase, during the application shutdown. Each sample consists of 20 rendered frames worth of memory bandwidth data: when the user requests a sample capture, the application will capture the data for 20 frames, then stop until a new sample is requested. If a shader has been enabled for capturing of memory bandwidth data as explained in Section 3.1.2, the atomic counters are run and reset every frame; however they are not read from the host unless a capture has been requested.

Screenshots When a sample capture is requested, the renderers will also run a function which takes a screenshot of the currently presented colour buffer. It accounts for gamma correction (Gritz and d'Eon 2008) and outputs the image to a `.ppm` file inside a specific folder inside the main build folder. It works by creating a host-visible Vulkan buffer and copying the color buffer into it, similarly to what shown by Willems (2016). The function is run after the semaphore synchronisation explained above, thus avoiding concurrency issues.

Sample data processing At the time of the application shutdown, both renderers will manipulate the gathered memory bandwidth data for output into `CSV` format. First the 20 frames worth of data is averaged to form one sample, per sample. Once the averaged data, forming samples, is available, the renderers will perform different manipulation operations depending on the structure of their buffers structure.
Since the data sampled directly from the Vulkan buffers does not represent the amount of memory in bytes but instead represents the amount of times a texture read or write has been called, this amount needs to be manipulated to yield the amount in `MiB`. This represents a more understandable amount which can be more easily compared and checked against the results from other papers and against expected outcomes. To do so, the reads and writes are divided by the number of buffers which can affect them. As mentioned above, this varies between the two renderers: the deferred renderer uses more buffers than the visibility renderer. It also varies because the two renderers sample the albedo maps at different subpasses with respect to each other. The single buffers values are then multiplied by the size in bytes of each buffer which contributed to the accumulation of this data and finally divided by the value of one `MiB` to transform the results from bytes to `MiB`. Listing 5 shows what this function looks like for the deferred renderer. In the visibility renderer it is implemented similarly with the differences as explained previously in this paragraph.

Once the data is available in `CSV` format it is processed by two different python scripts which produce bar graphs after further manipulating the data. Figure 4 shows an example of the graphs output by the python scripts.

```
...
for (eastl_size_t i = 0; i < mem_perf_data_reads_.size(); ++i) {
  average_reads[i] = {0, 0};
  average_writes[i] = {0, 0};

  // Calculate avg
  for (eastl_size_t j = 0; j < kFramesCaptureNum; ++j) {
    average_reads[i].first_frame += mem_perf_data_reads_[i][j].first_frame;
    average_reads[i].second_frame += mem_perf_data_reads_[i][j].second_frame;
    average_writes[i].first_frame += mem_perf_data_writes_[i][j].first_frame;
    average_writes[i].second_frame +=
        mem_perf_data_writes_[i][j].second_frame;
  }
  average_reads[i].first_frame /= kFramesCaptureNum;
  average_reads[i].second_frame /= kFramesCaptureNum;
  average_writes[i].first_frame /= kFramesCaptureNum;
  average_writes[i].second_frame /= kFramesCaptureNum;

  // Output 1st reads, maps
  ofs << ((SCAST_FLOAT(average_reads[i].first_frame) * 4.f) / kMebi) << ";";
  // Output 1st write, g Buffer
  ofs << ((SCAST_FLOAT(average_writes[i].first_frame) * 16.f) / kMebi) << ";";
  // Output 1st write, depth stencil
  ofs << ((SCAST_FLOAT(average_writes[i].first_frame) * 5.f) / kMebi) << ";";

  float reads_single_map = SCAST_FLOAT(average_reads[i].second_frame / 4U);
  // Output 2nd read g buffer
  ofs << ((reads_single_map * 16.f) / kMebi) << ";";
  // Output 2nd read depth buffer
  ofs << ((SCAST_FLOAT(reads_single_map) * 5.f) / kMebi) << ";";
  // Output 2nd writes
  ofs << ((SCAST_FLOAT(average_writes[i].second_frame) * 8.f) / kMebi)
      << "\n";
}
...
```

Listing 5 – *Output data manipulation for the deferred renderer. First the average over 20 frames is calculated and then these amounts are used to output the per-resource data, using the sizes of the buffer elements as a multiplier. When outputting the data for the first subpass writes there is no need to calculate a per-resource amount since the writes are not written by the counters per-resource but per-fragment shader invocation.*

3.2 EVALUATION METHODS

This project implemented the visibility buffer version proposed by Burns and Hunt (2013) with changes allowed by the AMD GCN Vulkan Extensions (Chajdas 2016a), which are explained in Section 3.1.1. This implementation assesses the memory bandwidth usage of a visibility buffer renderer compared to the one of a deferred renderer. The implementation is in C++ and Vulkan (The Khronos Group 2016a) and a framework was developed ad hoc for the project for the sake of code readability and as good industry-standard practice. After having implemented the visibility buffer a deferred shading system was developed with the aim of using it for comparison with the visibility buffer system, similarly to what is done by other papers on the visibility buffer (Burns and Hunt 2013; Schied and Dachsbacher 2015). The renderers use simple and standard rendering techniques for the lighting and shading. They do not present rendering techniques focussed on improving their images aspect since the focus of this paper is not on visual quality. In both renderers, the list of Albedo maps is:

- Diffuse
- Ambient
- Specular
- Roughness
- Normals

The choice of which and how many albedo maps to use was based on the necessity to produce meaningful memory bandwidth data. Hence the approach chosen was to use as many input albedo maps as possible in both renderers so that the amount of data would have made clear the advantages and disadvantages of both techniques.

Both rendering systems are modified to include memory bandwidth counters, implemented by means of atomic increments on pre-defined memory areas, which are then read by the application at the end of the rendering frame to produce output performance data. These bandwidth counters are added by means of injecting

SPIR-V binary code into a SPIR-V module so that the injection is opaque to a GLSL shader. These measure the memory read and written in `MiB`.

The data was captured under different situations to provide for a more precise result. For each renderer, the same same number of samples in the same situations were captured. The test cases were:

- Sponza at resolutions of:
  - 1280×720 pixels
  - 1366×768 pixels
  - 1280×1024 pixels
  - 1600×900 pixels
  - 1920×1080 pixels
- Rungholt at resolutions of:
  - 1280×720 pixels
  - 1366×768 pixels
  - 1280×1024 pixels
  - 1600×900 pixels
  - 1920×1080 pixels

These resolutions were chosen based on the most popular monitor resolutions (w3school 2017) in order to simulate the conditions in which a potential renderer using the visibility buffer or deferred rendering technique could run. Moreover having a breadth of different resolutions allows to determine how the techniques scale as the resolution increases or decreases.

For each test case above, 10 samples, or captures, were taken. A sample consists of 20 frames worth of memory bandwidth performance data, averaged to form one capture. This averaging is done at application shutdown as explained in Section 3.1.4. Sample capturing is triggered by the user, and can be either performed at the current camera location and orientation, or it can be automated to position and orientate the camera for 10 different locations in succession. In order to being able to correctly compare samples between the two rendering architectures, the samples were all taken at the same 10 locations for both renderers using the automated process described above. This means that for a given sample, e.g. sample 1, the same location and camera orientation was used when capturing in both renderers at every resolution. Of the 10 samples mentioned above at least one was taken in the following conditions:

- Mostly sky
- 50% sky, 50% geometry
- 100% geometry

This allows to determine the difference in memory bandwidth between cases in which different amounts of geometry are rasterized.

When background sky is present in the final image, that area is not accounted for in terms of memory bandwidth and it is not shaded in the shading passes of either renderer. This is achieved through the use of the stencil buffer. During application shutdown the data is formatted and output to file for further external processing. The second data processing stage consists in producing easily-interpretable bar charts. How this is accomplished is outlined in Section 3.1.4. After the data was gathered and presented, the two renderers' results were compared using a quantitative methodology. The results are first presented in Section 4.1 and then discussed in Section 5.

The choice of 3D scenes was determined by their availability in the academic field and by their properties. The Crytek Sponza scene (McGuire 2011) has 262,267 triangles and 184,330 vertices, compared to the Rungholt scene (McGuire 2011) which has 6,704,264 triangles and 12,308,528 vertices. This allows to gather results in two situations with a large difference in the count of the geometry being processed by the GPU.

The implementation of a visibility buffer renderer relies on AMD-specific extensions, and hence cannot be run on neither GPU cards by other vendors nor on pre-GCN (Chajdas 2016a) AMD GPUs.

4 RESULTS

The various measurements per-resource assume the buffer sizes as explained in Section 3.1.1.

Hardware The application was developed and run on a machine with the specifications as shown in Table 1.

Component	Type/Amount
CPU	Intel Core i7 975 @ 3.33GHz
GPU	AMD Radeon RX 480
Memory	12GB DDR3
Motherboard	ASUS P6T7 WS SuperComputer
OS	Windows 10 64-bit Education

Table 1 – *Specifications for the test machine.*

4.1 AVERAGED SAMPLES

Total Figure 4 shows the total memory read or written for a given renderer implementation in various test cases. This shows that, on average, the amount of memory read by the visibility buffer renderer is less than the one read by the deferred renderer by around 30%. The amount of memory written does not differ as much as the amount of memory read by the two renderers does, with the deferred renderer using around the same amount of memory bandwidth as the visibility buffer renderer for writes, on average.
The graphs also show an approximation of the amount of memory bandwidth that would be used by a visibility buffer renderer implementation with a leaner visibility buffer. In this case the memory bandwidth used by the visibility buffer renderer is lower than the one used by the implementation of a visibility buffer renderer chosen for this paper, resulting in a around 50% less memory bandwidth also compared to deferred rendering. Furthermore, the memory written is also less, resulting in less memory around 50% less bandwidth used for writes compared to deferred rendering. Finally the graphs in Figure 4 also show that this

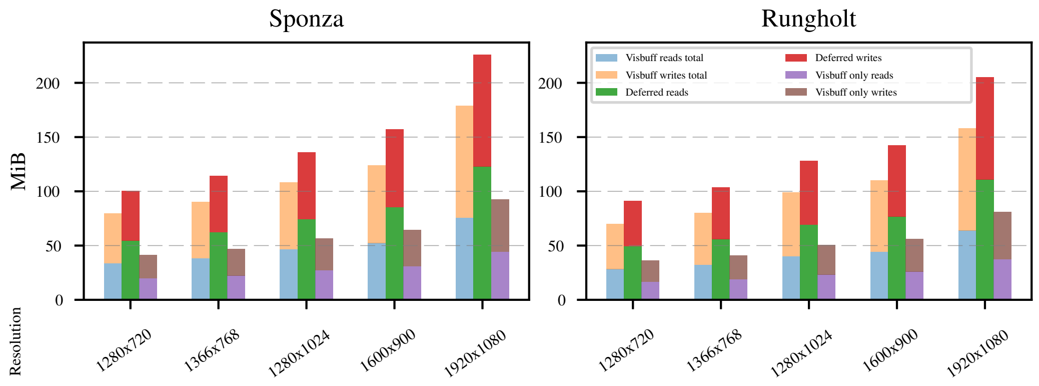


Figure 4 – *Total memory read and written for both renderers. The left hand image shows the data per-resolution taken in Sponza, whereas the right hand image shows the data from Rungholt. Notice the presence of the lines named "Visbuff only"; these represent the amount of data that would be read/written by a visibility buffer implementation without depth pre-pass as implemented by Burns and Hunt (2013) and Engel (2016).*

outcome remains consistent across various test scenes and final render target resolutions, as hypothesised in the previous sections. As the resolution increases, the results scale proportionally and the two renderers maintain the ratio between memory bandwidth read by the deferred renderer versus the one read by the visibility buffer.

Per-resource Figure 5 show the memory read and written in `MiB` by the visibility buffer renderer and the deferred renderer for the average of 10 samples worth of data, per-resource, per-scene and per-resolution (See Section 3.2 for the explanation about samples and how they were captured).

In terms of the visibiliy buffer renderer the results in Figure 5 outline that in the first subpass of the visibility buffer no memory is read since only the visibility of the fragments is computed, and that the memory written consists of 3 buffers for this implementation. The average memory written per-buffer resource does not exceed 40`MiB` in the visibility buffer renderer, and the memory written for both the visibility + barycentric coordinates buffer and the derivatives buffer is the same. Figure 5 shows that the second subpass of the visibility buffer renderer performs most of its reads by reading the albedo maps for all test cases. The albedo map reads do not exceed 40`MiB`.

In terms of deferred rendering the data displayed in Figure 5 shows that during the first subpass there is a high read rate of albedo maps for all test cases, coupled with a high write rate due to the writing of the G-buffer. The albedo maps reads amount to, on average, around 80`MiB` for the higher-resolution test case, and to around 45`MiB` for the lower-resolution test cases, with the writes being at around the same rate if the G-buffer and the Z-buffer writes are to be combined. During the second subpass, the deferred renderer performs, on average, most of its reads by sampling the G-buffer and Z-buffer, with the Z-buffer being part of the G-buffer in this implementation since it is used for position reconstruction. In any case, even in the higher-resolution test cases, their combined reads amount to a maximum of around 40`MiB`.

When considered together these results show that during the first subpass the visibility buffer renderer performs very similarly to the deferred renderer in terms of memory writes and has an improvement of 100% in terms of memory reads due to its lack of reads during this subpass. If, as explained in the above paragraph and in Figure 4, a leaner visibility buffer implementation is considered, then the visibility buffer improves by around 60% on average over deferred rendering in terms of memory writes.
During the second subpass the visibility buffer performs around the same amount of reads from the visibility, barycentric coordinates and derivatives buffer as the deferred renderer does from the G-buffer. However the visibility buffer implementation performs a large number of reads from the albedo maps during this subpass. This results in the deferred renderer performing around 50% less reads on average during this subpass. Again if a leaner visibility buffer implementation is considered, the number of reads performed by the visibility buffer renderer is lower and the advantage of the deferred renderer is around 15% instead.

Finally the charts in Figure 5 further show how the memory bandwidth used by both renderers in both scenes is directly proportional to the final render target resolution.

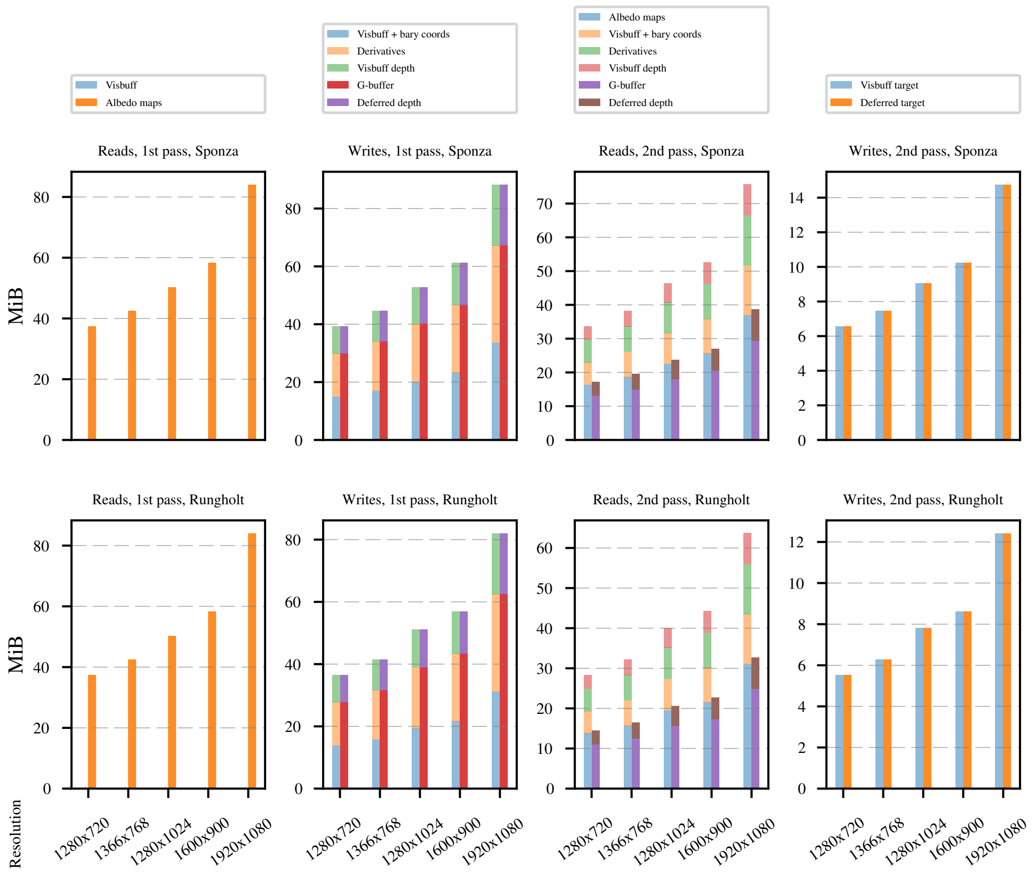


Figure 5 – *Average memory read and written per-resource and per-subpass, for both renderers and for both test scenes. During the first subpass, the visibility buffer renderer does not perform any reads, hence its bar is missing in the first column of charts from the left. In every chart, the bars on the left represent data for the visibility buffer, whereas bars on the right represent data for the deferred renderer. Notice the different scale of the Y axis depending on the graph.*

4.2 Per-sample

The graphs in Figures 6 and 7 show the difference in memory read and written, in `MiB`, with more granularity, for Rungholt at 1920×1080, Rungholt at 1280×720, Sponza at 1920×1080 and Sponza at 1280×720. Each value on the X axis corresponds to a sample. These particular resolutions were chosen for analysis because

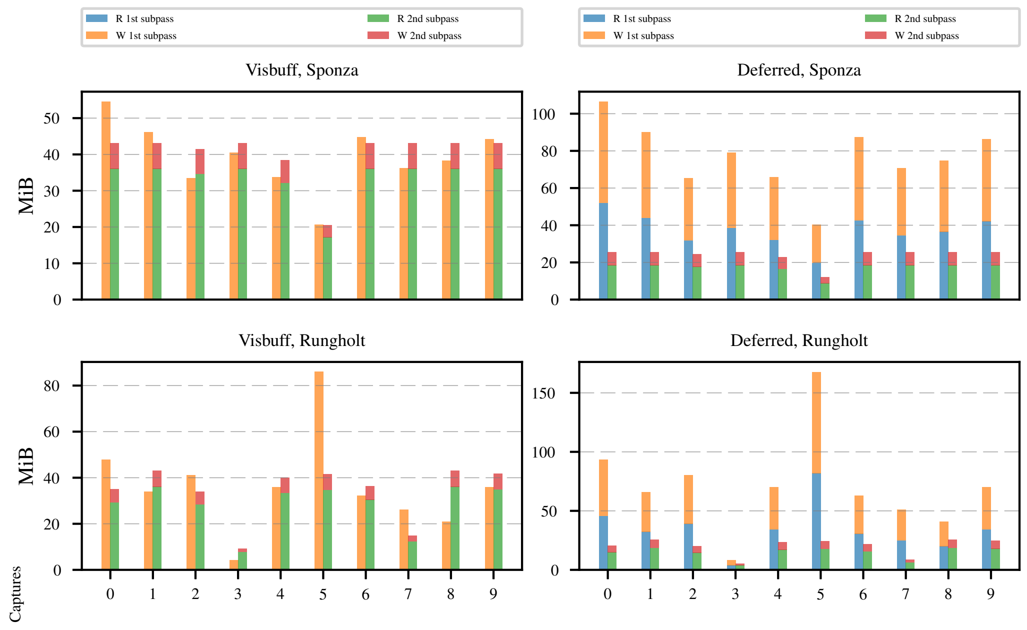


Figure 6 – *Memory bandwidth usage per-sample (hence per-location and camera rotation) and per-renderer at a resolution of 1280×720. Each entry corresponds to a sample capture and each sample is taken at a different location within the test scenes. In every chart, the bars on the left represent data for the first subpass, whereas bars on the right represent data for the second subpass. Notice the different scale of the Y axis depending on the graph.*

they are at the two ends of the range of resolutions analysed. Since, as stated in the previous paragraphs, the memory bandwidth is directly proportional to the final render target resolution, it is possible to analyse these two resolutions and apply the same deductions to the intermediate resolutions thanks to the results provided by the averaged results in the previous sections.

These per-sample results highlight the difference in memory bandwidth between samples taken at different locations, hence with a varying amount of geometry within the view frustum. For samples with a high amount of geometry within the frustum, the results tend to show a high memory bandwidth usage. Captures 0, 1 and 6 for deferred rendering in Sponza in Figure 7 present an example of such case: there is a high rate of reads, especially during the first subpass, accompanied by a much lower rate of reads in the second subpass. These samples also show a high rate of memory writes, coming mainly from the first subpass. However for

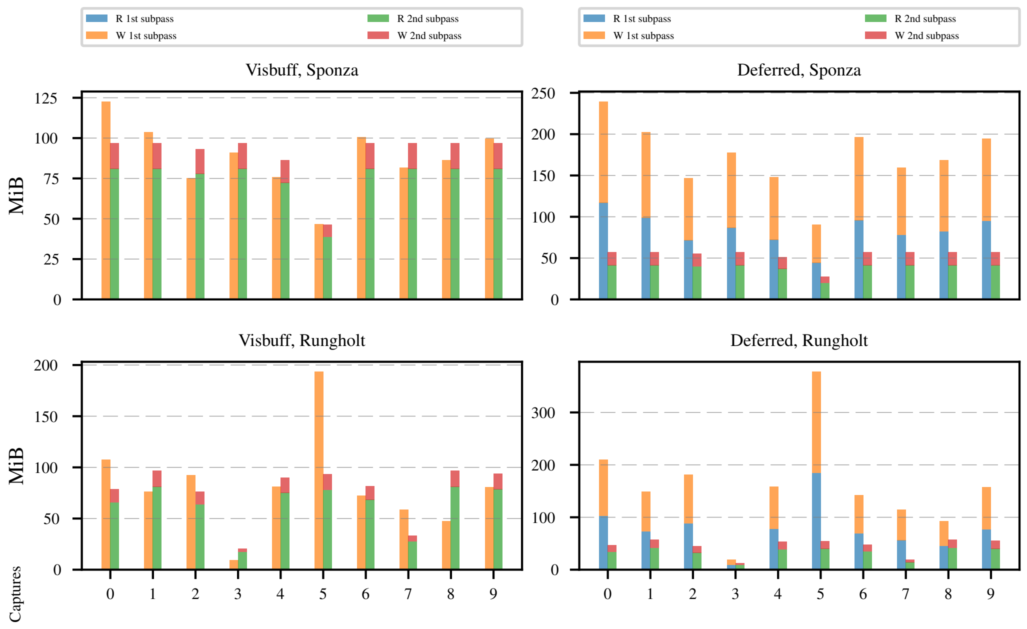


Figure 7 – *Memory bandwidth usage per-sample (hence per-location and camera rotation) and per-renderer at a resolution of 1920×1080. Each entry corresponds to a sample capture and each sample is taken at a different location within the test scenes. In every chart, the bars on the left represent data for the first subpass, whereas bars on the right represent data for the second subpass. Notice the different scale of the Y axis depending on the graph.*

the Rungholt scene samples, which present a higher sky to geometry ratio, there is a larger difference between situations with high overshading and ones which suffer less overshading.
A similar outcome can be observed in the results for the visibility buffer renderer

in Figure 7: there is a higher rate of memory writes and writes for samples taken in situations presenting a high amount of geometry, such as samples 0 and 6 for Sponza.

Figures 8 and 9 show the same data as Figures 6 and 7 but after it was manipulated as to highlight the difference in total memory read and written by both renderers in each sample. These graphs highlight how, on average, the visibility buffer performs less memory reads and writes than the deferred renderer. However, in some cases, such as in sample 8 from Rungholt, the two renderers perform very similarly. Finally there is a larger difference in memory bandwidth used by the two renderers between samples with high overshading and ones which suffer less from overshading.

The results also show that this trend for which the rate of memory writes and reads is tied to the geometry complexity within a given sample's view frustum applies also to the remaining scenes and test cases: their interpretation can be extended to the other scenes and resolution test case combinations.

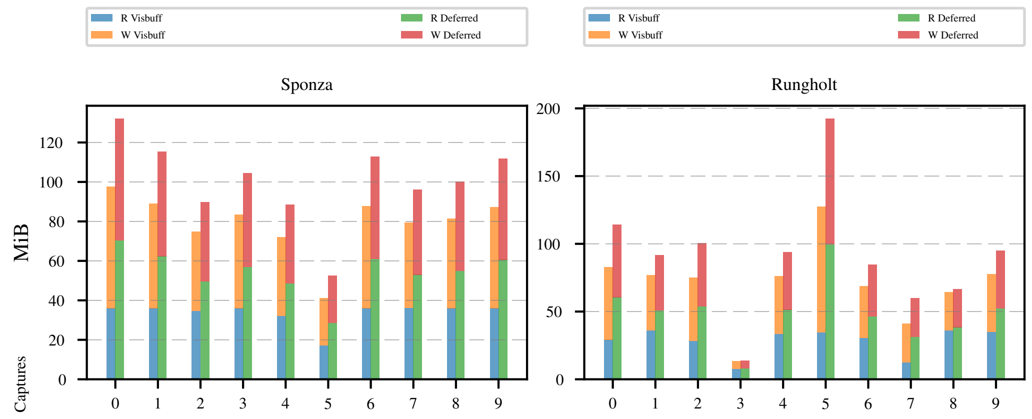


Figure 8 – *Memory bandwidth usage per-sample (hence per-location and camera rotation) and per-scene at a resolution of 1280×720. Each entry corresponds to a sample capture and each sample is taken at a different location within the test scenes. In each chart, the bars on the left represent data for the visibility buffer, whereas bars on the right represent data for the deferred renderer. Notice the different scale of the Y axis depending on the graph.*

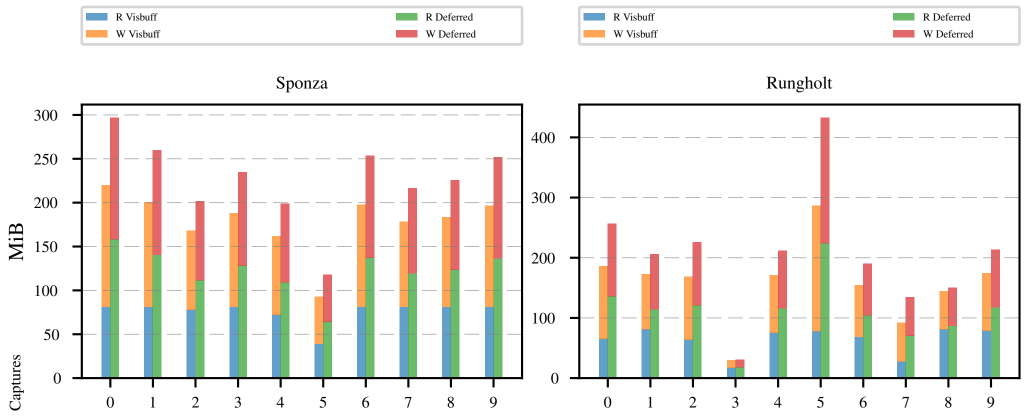


Figure 9 – *Memory bandwidth usage per-sample (hence per-location and camera rotation) and per-scene at a resolution of 1920×1080. Each entry corresponds to a sample capture and each sample is taken at a different location within the test scenes. In each chart, the bars on the left represent data for the visibility buffer, whereas bars on the right represent data for the deferred renderer. Notice the different scale of the Y axis depending on the graph.*

5 Discussion

5.1 Interpretation of results

5.1.1 Averaged samples

Total The results presented in Section 4.1 show that the visibility buffer renderer uses around 30% less memory bandwidth than the deferred renderer, on average, in all of the test cases. In terms of memory reads there is a difference of around 40%, on average, between the two renderers. This can be attributed to mainly two factors: overshading and how deferred rendering operates.

Deferred rendering performs albedo maps fetches during the first subpass, which is also the subpass which suffers from overshading. The result is a large number of texture fetches during the first pass for the deferred renderer. This phenomenon could be alleviated using a depth prepass (Burns and Hunt 2013), however since the visibility buffer renderer suffers from the same overshading phenomenon as the deferred renderers, the outcomes level each other out. In terms of memory writes the results show that the deferred renderer uses around the same bandwidth as the visibility buffer, on average, in all situations, which is caused by the same phenomena which causes the deferred renderer to use a larger memory bandwidth for reading: overshading and the operating structure of the G-buffer. Since during the first subpass the deferred renderer suffers from overshading it ends up writing to a given buffer location more than once.

This first set of results hence prove what had been suggested by previous works: the visibility buffer has a lower memory bandwidth requirement than deferred rendering, although not by a large amount, even with a rather memory footprint-heavy visibility buffer implementation such as the one chosen for this paper. However, if the leaner version of a visibility buffer renderer is taken into consideration and compared to deferred rendering, the result is that the visibility buffer is a clear winner in terms of memory bandwidth usage, with an improvement of around 50% or more, in total, compared to deferred rendering. This also confirms the statements made in previous work on the visibility buffer regarding the improvements in terms of memory bandwidth of the visibility buffer (Schied and Dachsbacher 2015; Burns and Hunt 2013).

Per-resource As outlined in Section 4.1 all the reads for the visibility buffer are performed during the second pass of its pipeline. The results in Figure 5 clearly show how for the visibility buffer, since for it to work it is necessary to read all the albedo maps during the second pass, the memory bandwidth required for that operation is on average as much as the one required for reading the G-buffer. When the need to also read the visibility buffer and other possible buffers, as it is the case for the visibility buffer implementation chosen in this paper, is taken into account the graphs show that the visibility buffer performs 50% more reads than the deferred renderer, on average, during the second subpass. In terms of memory writes, the renderers perform similarly during the first subpass, which is to be expected due to their buffer formats and the fact that they both suffer from overshading.

One of the aims was to understand whether the need of the visibility buffer rendering to read all the albedo maps during the second subpass would have negatively affected its memory bandwidth performance compared to the one of deferred rendering, where the G-buffer is read instead during the second pass. The results in Figure 5 highlight how the majority of the reads are performed on the

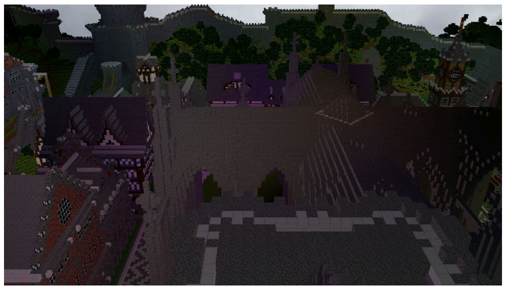
albedo maps for both renderers: for the visibility buffer that happens during the second subpass and for the deferred renderer during the first subpass. However the deferred renderer performs more albedo map reads than the visibility buffer on average if both subpasses are considered at the same time, with a difference of around 50% on average between the two when considering this resource. This is to be expected for the reasons presented in the above paragraphs: the deferred renderer suffers from overshading during the first pass, during which it performs the reads, whereas the visibility buffer instead performs albedo maps sampling during its second pass. Hence the visibility buffer renderer will perform, in the worst case scenario of a visibility buffer completely covered by geometry, as many reads as the number of albedo maps times the resolution of the final target, whereas the deferred renderer could do more due to overshading.
Summarising, though the graphs show how during the second subpass the visibility buffer is affected by the need to read the albedo maps, when the results are considered across the two subpasses the visibility buffer performs less memory reads than deferred rendering even with a visibility buffer implementation which presents a rather memory footprint-large buffer. These results would be in favour of the visibility buffer even more in the case of a leaner visibility buffer implementation.

5.1.2 Per-sample

The outcome outlined in the previous sections is particularly significant if the results per-sample presented in Section 4.2 are also taken into consideration. These results show that in some cases the visibility buffer performs similarly to the deferred renderer but never worse, especially in terms of the amount of bandwidth used for reading textures.
This can be attributed to the lower memory consumption of the visibility buffer compared to the G-buffer: this buffer in fact stores only the visibility of the fragments, compared to the G-buffer which stores additional surface information, e.g. the normals, the albedos. Another important factor is, again, overshading, which is discussed further at the end of this paragraph.

The data charts in Figure 7 show the captured data for Rungholt and Sponza at a resolution of 1920$\times$1080, where each entry is a different sample, comparing the memory usage of the first subpass against the one of the second subpass.

(a) *Sample 3*


(b) *Sample 5*

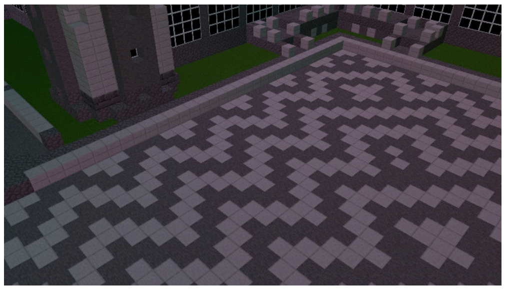

(c) *Sample 8*

Figure 10 – *Screenshots of Rungholt at 1920×1080 relative to samples shown in Figure 7 and discussed in Section 5.1.2. They present a variety of mostly-geometry and mostly-sky samples so that the memory bandwidth usage in these two cases can be assessed. Since the 10 captures per resolution are all done at the same camera locations, these two screenshots are valid for both renderers.*

These graphs are useful for noticing how the statements made in the previous sections regarding how the amount of memory read by the visibility buffer renderer is related mostly to the coverage of the visibility buffer by geometry, and hence not related to overshading, are correct. The graphs for Sponza in Figure 7 show how for most captures the amount of reads by the visibility buffer does not exceed a certain threshold, which represents the amount of reads for a fully-covered visibility buffer, i.e. a situation where the geometry fills the whole final render target, whereas for deferred rendering the amount varies depending on the sample and hence on the amount of overshading. The same results can be noticed in the graphs for Rungholt, although there is a larger variance between different samples due to a higher variance in the sky-to-geometry rate for the Rungholt scene.
Analysing capture 3 for the visibility buffer for Rungholt in Figure 7 for which the screenshot is shown in Figure 10a is possible to notice how the memory band-

width consumption is lower compared to sample 5, shown in Figure 10b of the same scene. This is due to there being more sky in sample 3 than in sample 5 and to the fact that the position of the camera was so that less geometry was in front of it, resulting in less overshading. This shows that for the visibility buffer the performance in terms of memory writes scales proportionally to the amount of coverage by geometry in the final render target. The same applies to the deferred renderer: comparing sample 3 and 5 shows how this renderer is also affected by overshading.
Figure 8 presents data which is similar to the one presented in Figure 9, only with smaller amounts of memory bandwidth usage due to the lower resolution used. This is in line with the statements in previous sections about how the memory bandwidth usage is directly proportional to the target resolution.

Comparing sample 8 of the visibility buffer with the same one for the deferred renderer for Rungholt, for which the screenshot is shown in Figure 10c, using Figure 9 as a reference highlights that there is not much difference between the two renderers when the amount of overshading is low, with the two techniques performing almost the same amount of reads and writes. However in samples such as number 5 for Rungholt the difference in memory bandwidth between the two renderers is much larger, with the visibility buffer outperforming the deferred renderer. Sample 5 also reveals how the visibility buffer does around 60% less memory reads than the deferred renderer, but that at the same time it performs as many memory writes as deferred rendering. This is because the visibility buffer suffers from overshading during the first subpass as much as the deferred renderer does, but that does not affect its memory reads performance, since the memory reads for visibility buffer rendering are all done in the second subpass.
Figure 9 also shows how the reads for the visibility buffer never exceed a certain threshold. This behaviour is described also in previous sections, and hence this result further strengthens the observation that although the visibility buffer renderer performs more memory reads during the second subpass than deferred rendering, since these reads are performed in screen-space they do not affect the final performance as much as the reads done by the deferred renderer in the first subpass do. Hence the visibility buffer has an advantage over deferred rendering for the implementations chosen for this paper since its reads are not coupled with the amount of geometry processed during the first subppas, i.e. they are decoupled from overshading.

In summary the graphs outline how the visibility buffer is less susceptible to overshading for memory reads than deferred rendering. The amount of memory read is constant on average if the coverage of the final render consists of mostly geometry and not sky. This contrasts with the deferred renderer which, performing the albedo map reads during the first subpass, is more susceptible to overshading in terms of memory reads: the more overshading there is, the more the memory bandwidth for reads and writes increases. This information shows that although the performance in terms of memory bandwidth varies depending on the position of the camera and what is being contained within the view frustum (i.e. more or less geometry), the visibility buffer performs at worst very similarly to the deferred renderer and on average better than the deferred renderer, further strengthening the conclusion that the claims made in previous work regarding the benefits of the visibility buffer due to its small footprint compared to deferred rendering.

5.2 DESIGN IMPLICATIONS

The implementation of the visibility buffer developed for this paper makes use of additional buffers and additional texture channels for storing the extra information represented by the barycentric coordinates and the texture coordinates gradients. Hence it could be argued that the results are incorrect, however this is taken into account when discussing the results. In fact, even when the additional buffers are taken into account to calculate the averages and single samples, the results are still in line with the main hypothesis.
Using the additional information to implement the visibility buffer is not necessary if the implementation proposed by Burns and Hunt (2013) is developed; however since it was possible to simply discard, when necessary, the additional data deriving from using additional buffers from the calculations, the visibility buffer was implemented as outlined in Section 3.1.1.

Moreover a similar critique could be made of the implementation of a deferred renderer: the size of the G-buffer per-fragment could be reduced by employing different channel formats or by using other encoding formats for the information that the G-Buffer stores, e.g. the normals could be encoded in polar format (Zink, Pettineo and Hoxley 2011). However when combined with the larger size of the visibility buffer highlighted above the results can be considered correct as they both present an increase in size.

Since neither renderer implementation makes use of a depth pre-pass, they both suffer from overshading during the first subpass, even with the early-z optimisation enabled.

5.3 VISIBILITY BUFFER ARTEFACTS

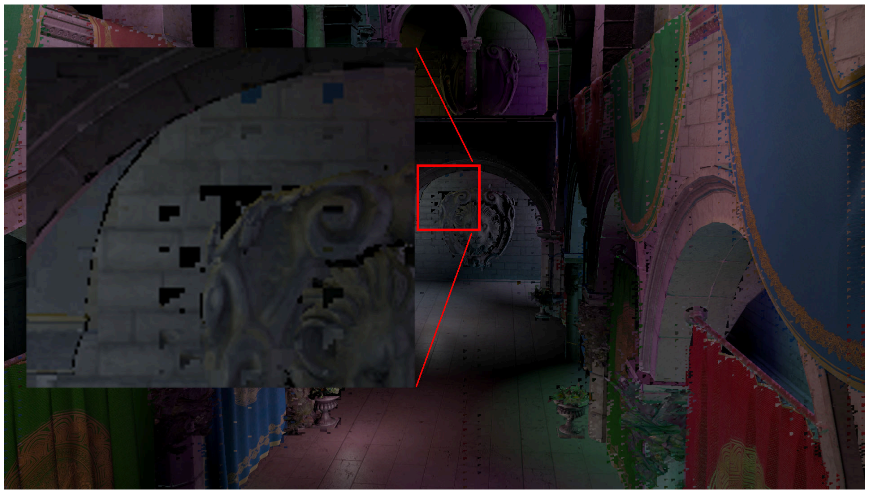


Figure 11 – *Screenshot showing how the final render of the visibility buffer renderer artefacts appears at run-time. The artefacts are mainly located at the intersection of two different materials in screen space due to the nature of their origin, as explained in Section 5.3.*

Due to a limitation in Vulkan, the visibility renderer presents artefacts in the final rendered image. This limitation stems from the inability of different threads within a pixel shader wavefront to each fetch a different texture (Tovey 2017).

However these artefacts do not affect the memory bandwidth measurements and hence, although a workaround is available but not implementable (Tovey 2017), they are present in the final implementation. These artefacts can be noticed either by running the visibility buffer renderer or by looking at the screenshots provided in the appendix. Figure 11 shows what the artefacts look like at run-time. In Rungholt the artefacts are not present because that 3D model uses a single texture for all the materials.

6 CONCLUSIONS

The memory bandwidth of a visibility buffer renderer has been compared to the one of a deferred renderer, within the constraints and rules defined by the chosen implementation.
The results show that, for the test case of a visibility buffer and a deferred renderer without depth pre-pass, the visibility buffer technique uses less memory bandwidth than a deferred renderer, on average. In the best case scenario, e.g. situations with low overshading, the implementation of the visibility buffer performs similarly to a deferred renderer, but in situations with high overshading the visibility buffer has a lower memory bandwidth usage due to it performing the albedo map reads during the second subpass instead of during the first subpass, which in practice decouple the albedo textures reads from the overshading and makes them constant in screen-space.

This confirms that the statements regarding the advantages in terms of memory bandwidth of the visibility buffer over deferred rendering made in previous works on the visibility buffer are valid for the implementation chosen and also for an implementation similar to the one used in the previous works. These works proved that the visibility buffer can offer improvements in memory footprint over deferred rendering without losing frame rendering time performance. However they did not present precise measurements for the improvements of the memory bandwidth usage of the visibility buffer, although such improvement was claimed. This investigation hence complements the previous work on the subject by providing such missing memory bandwidth usage data, by defining with more certainty the improvements which are available by using a visibility buffer renderer over deferred rendering and helping to clarify what the best use cases are for the visibility buffer and how it compares to deferred rendering.

Hence the answer to main research question:

> *How does the visibility buffer technique compare to deferred rendering in terms of memory bandwidth usage?*

is that the visibility buffer rendering technique performs better in terms of memory bandwidth than deferred rendering on average and for the implementations chosen. Furthermore, it has been estimated and shown that under the conditions given by a visibility buffer implementation with a smaller visibility buffer the results would be even more in favour of the visibility buffer, at the expense of an increased computational cost in the second subpass.

Future work and suggestions The implementation of the visibility buffer could be changed to the one proposed by Burns and Hunt (2013) or the one by Schied and Dachsbacher (2015) and the tests could be repeated under the conditions introduced by these different visibility buffer designs. The visibility buffer renderer would have a smaller memory footprint, thus producing a smaller memory bandwidth usage. However this would result in more time spent in the second subpass due the implementation having to calculate the barycentric coordinates. With the implementation proposed by Schied and Dachsbacher (2015) this performance implication would be alleviated at the expense of memory footprint.

The deferred renderer implementation could be changed to use a G-buffer with a smaller memory footprint, i.e. different data encodings, different buffer formats, etc. Various solutions to achieve this are proposed by Zink, Pettineo and Hoxley (2011), e.g. encoding the normals of the normal buffer into two channels by means of polar coordinates, or avoiding the use of three RGB channels to store the specular albedo in favour of a grayscale specular albedo which would thus occupy a single channel. This would result in a smaller memory footprint for the deferred renderer and consequentially in a smaller memory bandwidth usage.

Both renderers could be integrated with a form of geometry culling, so that less geometry is sent to the GPU for processing per-frame, akin to what done by Engel (2016) using a library similar to GeometryFX (Chajdas 2016b). Having less geometry to process during the first subpass would mainly advantage the deferred renderer since it is the renderer which suffers the most from overshading, resulting in a smaller memory bandwidth usage for it. However there would be a similar drop in memory bandwidth usage for the visibility buffer, hence the results would not be extremely different from what discussed in this paper.

The test scenes could be varied further to test the implementation under more conditions. Bigger or smaller scenes could be used, which would mainly affect the overshading which affects both renderers. There would therefore be a variance in the results similarly to what shown in this research with samples taken with different geometry complexity within the view frustum.

The memory bandwidth measurement tests could be run on mobile to assess this performance aspect on the such embedded GPU architectures which generally present tiled rendering architectures, and desktop GPUs from other vendors, e.g. NVidia, could be tested as well.

Other types of renderer could be developed and compared against the visibility buffer renderer, e.g. Forward +, Forward and Light pre-pass. Such renderers, making use of a depth pre-pass, would integrate well in a test with a visibility buffer implementation making use of a depth pre-pass such as the one by Schied and Dachsbacher (2015). The deferred renderer would especially benefit from the depth pre-pass which would make it avoid overshading in the subpass used to construct the G-buffer. This would result in a lower memory bandwidth usage. However, the visibility buffer would benefit in terms of less overshading for the same reasons.

7 Acknowledgements

Many thanks to AMD for providing the GPU used in this paper and for supporting this project.

References

Akenine-Möller, T., Haines, E. and Hoffman, N., 2008. *Real-Time Rendering*. vol. 3rd. Book, Whole. A K Peters/CRC Press. URL: https://www.crcpress.com/Real-Time-Rendering-Third-Edition/Akenine-Moller-Haines-Hoffman/p/book/9781568814247;.

Blinn, J. F., 1977. 'Models of Light Reflection for Computer Synthesized Pictures'. *Proceedings of the 4th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '77. New York, NY, USA: ACM, pp. 192–198. DOI: 10.1145/563858.563893. URL: http://doi.acm.org/10.1145/563858.563893 [visited on 18/11/2016].

Burns, C. A. and Hunt, W. A., 2013. 'The Visibility Buffer: A Cache-Friendly Approach to Deferred Shading'. *Journal of Computer Graphics Techniques (JCGT)* 2 (2), pp. 55–69. ISSN: 2331-7418. URL: `http://jcgt.org/published/0002/02/04/`.

Chajdas, M., 2016a. *GCN Shader Extensions for Direct3D and Vulkan*. GPUOpen. URL: `http://gpuopen.com/gcn-shader-extensions-for-direct3d-and-vulkan/` [visited on 26/03/2017].

Chajdas, M., 2016b. *GeometryFX*. URL: `http://gpuopen.com/gaming-product/geometryfx/`.

Engel, W., 2009. 'Designing a renderer for multiple lights: The light pre-pass renderer.' *ShaderX7: Advanced Rendering Techniques*. Shaderx series. Course Technology/Cengage Learning, pp. 655–666. ISBN: 978-1-58450-598-3. URL: `https://books.google.co.uk/books?id=iZjrPAAACAAJ`.

Engel, W., 2016. *The filtered and culled Visibility Buffer*. Web Page. URL: `http://www.confettispecialfx.com/gdce-2016-the-filtered-and-culled-visibility-buffer-2/`.

Google, 2017. *shaderc*. URL: `https://github.com/google/shaderc` [visited on 15/11/2016].

Gritz, L. and d'Eon, E., 2008. 'The Importance of Being Linear'. Nguyen, H. *GPU Gems 3*. GPU Gems. Addison-Wesley, Chapter 24. ISBN: 978-0-321-51526-1.

Hable, J., 2010. *Filmic Tonemapping Operators*. Filmic Worlds. URL: `http://filmicworlds.com/blog/filmic-tonemapping-operators/` [visited on 27/03/2017].

Harada, T., McKee, J. and C.Yang, J., 2012. 'Forward+: Bringing Deferred Lighting to the Next Level'. *Eurographics Short Paper* (Journal Article).

Kessenich, J., Boaz, O. and Raun, K., 2017. *SPIR-V Specification*. URL: `http://www.khronos.org/registry/spir-v/specs/1.1/SPIRV.html`.

Liktor, G. and Dachsbacher, C., 2012. 'Decoupled Deferred Shading for Hardware Rasterization'. Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games. I3D '12. Conference Proceedings. ACM, pp. 143–150. ISBN: 978-1-4503-1194-6. DOI: 10.1145/2159616.2159640. URL: http://doi.acm.org/10.1145/2159616.2159640.

McGuire, M., 2011. *Computer Graphics Archive*. URL: http://graphics.cs.williams.edu/data.

Pettineo, M. J., 2009. *Scintillating Snippets: Reconstructing Position From Depth*. The Danger Zone. URL: https://mynameismjp.wordpress.com/2009/03/10/reconstructing-position-from-depth/ [visited on 17/11/2016].

Reinhard, E., Stark, M., Shirley, P. and Ferwerda, J., 2002. 'Photographic Tone Reproduction for Digital Images'. *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '02. New York, NY, USA: ACM, pp. 267–276. ISBN: 978-1-58113-521-3. DOI: 10.1145/566570.566575. URL: http://doi.acm.org/10.1145/566570.566575 [visited on 15/11/2016].

Saito, T. and Takahashi, T., 1990. 'Comprehensible Rendering of 3-D Shapes'. Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques. SIGGRAPH '90. Conference Proceedings. ACM, pp. 197–206. ISBN: 0-89791-344-2. DOI: 10.1145/97879.97901. URL: http://doi.acm.org/10.1145/97879.97901.

Schied, C. and Dachsbacher, C., 2015. 'Deferred Attribute Interpolation for Memory-efficient Deferred Shading'. Proceedings of the 7th Conference on High-Performance Graphics. HPG '15. Conference Proceedings. ACM, pp. 43–49. ISBN: 978-1-4503-3707-6. DOI: 10.1145/2790060.2790066. URL: http://doi.acm.org/10.1145/2790060.2790066.

Scratchapixel, 2016. *Scratchapixel*. URL: https://www.scratchapixel.com/ [visited on 02/12/2016].

The Khronos Group, 2016a. *The Vulkan API*. URL: https://www.khronos.org/vulkan/.

The Khronos Group, 2016b. *Vulkan - A specification*. Web Page. URL: https://www.khronos.org/registry/vulkan/specs/1.0/xhtml/vkspec.html#queries.

Tovey, S., 2017. *Visbuffer artefacts*.

w3school, 2017. *Browser Display Statistics*. URL: https://www.w3schools.com/browsers/browsers_display.asp [visited on 19/04/2017].

Willems, S., 2016. *Vulkan examples and demos*. URL: https://github.com/SaschaWillems/Vulkan [visited on 15/10/2016].

Zink, J., Pettineo, M. J. and Hoxley, J., 2011. *Practical Rendering and Computation with Direct3D 11*. vol. 1st. Book, Whole. Natick, MA, USA: A. K. Peters, Ltd. ISBN: 1-56881-720-7 978-1-56881-720-0.

BIBLIOGRAPHY

Advanced Micro Devices Inc., 2016. *CodeXL*. URL: http://gpuopen.com/compute-product/codexl/.

Berglund, C. and Geelnard, M., 2016. *GLFW*. URL: http://www.glfw.org/ [visited on 26/03/2017].

Chapman, J., 2013. *SSAO Tutorial*. Web Page. URL: http://john-chapman-graphics.blogspot.co.uk/2013/01/ssao-tutorial.html.

Chapman, J. *Per-Object Motion Blur*. URL: http://john-chapman-graphics.blogspot.com/2013/01/per-object-motion-blur.html [visited on 16/11/2016].

Cook, R. L., Carpenter, L. and Catmull, E., 1987. 'The Reyes Image Rendering Architecture'. *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '87. New York, NY, USA: ACM, pp. 95–102. ISBN: 978-0-89791-227-3. DOI: 10.1145/37401.37414. URL: http://doi.acm.org/10.1145/37401.37414 [visited on 14/11/2016].

Cornut, O., 2016. *imgui*. URL: https://github.com/ocornut/imgui [visited on 27/11/2016].

Dippé, M. A. Z. and Wold, E. H., 1985. 'Antialiasing Through Stochastic Sampling'. *Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '85. New York, NY, USA: ACM, pp. 69–78. ISBN: 978-0-89791-166-5. DOI: 10.1145/325334.325182. URL:

http://doi.acm.org/10.1145/325334.325182 [visited on 14/11/2016].

Fujita, S., 2016. *tinyobjloader*. URL: https://github.com/syoyo/tinyobjloader [visited on 27/11/2016].

Hyslop, J. and Sutter, H., 2000. *Conversations: Virtually Yours*. Dr. Dobb's. URL: http://www.drdobbs.com/conversations-virtually-yours/184403760 [visited on 27/11/2016].

Kaneko, T., Takahei, T., Inami, M., Kawakami, N., Yanagida, Y., Maeda, T. and Tachi, S., 2001. 'Detailed shape representation with parallax mapping'. In Proceedings of the ICAT 2001. Conference Proceedings, pp. 205–208.

Karlsson, B., 2016. *RenderDoc*. URL: https://github.com/baldurk/renderdoc.

Lauritzen, A. *GPU Gems 3 - Chapter 8. Summed-Area Variance Shadow Maps*. URL: http://http.developer.nvidia.com/GPUGems3/gpugems3_ch08.html [visited on 07/01/2017].

Lengyel, E., 2011. *Mathematics for 3D Game Programming and Computer Graphics*. Third. Cengage Learning. 566 pp. ISBN: 978-1-4354-5887-1.

Lighthouse3d, 2014. *Geometric Approach – Extracting the Planes » Lighthouse3d.com*. URL: https://web.archive.org/web/20140702124616/http://www.lighthouse3d.com/tutorials/view-frustum-culling/geometric-approach-extracting-the-planes/ [visited on 17/11/2016].

Lottes, T., 2009. *FXAA*. URL: http://developer.download.nvidia.com/assets/gamedev/files/sdk/11/FXAA_WhitePaper.pdf.

Mayer, T., 2007. 'The 4K Format Implications for Visualization, VR, Command \& Control and Special Venue Application'. Proceedings of the 2007 Workshop on Emerging Displays Technologies: Images and Beyond: The Future of Displays and Interacton. EDT '07. Conference Proceedings. ACM. ISBN: 978-1-59593-669-1. DOI: 10.1145/1278240.1278249. URL: http://doi.acm.org/10.1145/1278240.1278249.

Meyers, S., 2005. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs*. Google-Books-ID: Qx5oyB49poYC. Pearson Education. 320 pp. ISBN: 978-0-13-270206-5.

Microsoft Inc, 2011. *DirectX 11*.

Myers, K., 2007. *VarianceShadowMapping*. URL: `http://developer.download.nvidia.com/SDK/10.5/direct3d/Source/VarianceShadowMapping/Doc/VarianceShadowMapping.pdf` [visited on 07/01/2017].

NVIDIA, 2016. *NSight*.

Pawel, L., 0206. *API without Secrets: Introduction to Vulkan*. Web Page. URL: `https://software.intel.com/en-us/articles/api-without-secrets-introduction-to-vulkan-part-1#`.

Pettineo, M. J., 2010. *Position From Depth 3: Back In The Habit*. The Danger Zone. URL: `https://mynameismjp.wordpress.com/2010/09/05/position-from-depth-3/` [visited on 19/11/2016].

Pettineo, M. J. *Bindless Texturing for Deferred Rendering and Decals*. The Danger Zone. URL: `https://mynameismjp.wordpress.com/2016/03/25/bindless-texturing-for-deferred-rendering-and-decals/` [visited on 11/12/2016].

Ragan-Kelley, J., Lehtinen, J., Chen, J., Doggett, M. and Durand, F., 2011. 'Decoupled Sampling for Graphics Pipelines'. *ACM Trans. Graph.* 30 (3), 17:1–17:17. ISSN: 0730-0301. DOI: `10.1145/1966394.1966396`. URL: `http://doi.acm.org/10.1145/1966394.1966396` [visited on 14/11/2016].

Reed, N. *GPU Profiling 101*. Web Page. URL: `http://www.reedbeta.com/blog/gpu-profiling-101/`.

Reed, N. *Understanding BCn Texture Compression Formats – Nathan Reed's coding blog*. URL: `http://www.reedbeta.com/blog/understanding-bcn-texture-compression-formats/` [visited on 23/11/2016].

Riccio, C., 2016a. *GLI*. URL: `http://gli.g-truc.net/0.8.1/index.html` [visited on 18/09/2016].

Riccio, C., 2016b. *GLM*. URL: `http://glm.g-truc.net/0.9.8/index.html` [visited on 18/09/2016].

Sellers, G., 2013. *The Road to One Million Draws*. Web Page. URL: `http://www.openglsuperbible.com/2013/10/16/the-road-to-one-million-draws/`.

Sellers, G. and Kessenich, J., 2016. *Vulkan Programming Guide: The Official Guide to Learning Vulkan*. Google-Books-ID: XVBxDQAAQBAJ. Addison-Wesley Professional. 885 pp. ISBN: 978-0-13-446468-8.

Strugar, F., 2014. *An investigation of fast real-time GPU-based image blur algorithms — Intel® Software*. URL: `https://software.intel.com/en-us/blogs/2014/07/15/an-investigation-of-fast-real-time-gpu-based-image-blur-algorithms` [visited on 09/01/2017].

Sutter, H., 2001. *Virtuality*. URL: `http://www.gotw.ca/publications/mill18.htm` [visited on 27/11/2016].

The Kronos Group, 2014. *OpenGL Wiki*. URL: `https://www.opengl.org/wiki` [visited on 18/10/2016].

Vandevenne, L., 2013. *LodePNG*. URL: `http://lodev.org/lodepng/` [visited on 27/11/2016].

Appendices

A AMD GCN Extensions workaround

```
...
layout (location = 0) flat in uint draw_id;
layout (location = 1) in vec2 uv_in;
layout (location = 2) flat in vec4 pos0;
layout (location = 3) __explicitInterpAMD in vec4 pos1;
...
vec4 v0 = interpolateAtVertexAMD(pos1, 0);
vec4 v1 = interpolateAtVertexAMD(pos1, 1);
vec4 v2 = interpolateAtVertexAMD(pos1, 2);
if (v0 == pos0) {
  debug_out.y = gl_BaryCoordSmoothAMD.x;
  debug_out.z = gl_BaryCoordSmoothAMD.y;
  debug_out.x = 1 - debug_out.z - debug_out.y;
}
else if (v1 == pos0) {
  debug_out.x = gl_BaryCoordSmoothAMD.x;
  debug_out.y = gl_BaryCoordSmoothAMD.y;
  debug_out.z = 1 - debug_out.x - debug_out.y;
} else if (v2 == pos0) {
  debug_out.z = gl_BaryCoordSmoothAMD.x;
  debug_out.x = gl_BaryCoordSmoothAMD.y;
  debug_out.y = 1 - debug_out.x - debug_out.z;
}
...
```

Listing 6 – *Workaround to have the barycentric coordinates work correctly and appear at the vertex as expected. If the barycentric coordinates were used directly without manipulation, the invoking vertex would change depending on the view-space position of the vertices of the triangle. It makes use of the AMD GCN extensions for Vulkan and DX12.*

B INSTRUMENTATION EXAMPLE

```
...
       %OpName %__1 ""
       %OpDecorate %normals_map DescriptorSet 0
       %OpDecorate %normals_map Binding 15
       %OpDecorate %normals_map InputAttachmentIndex 4
...
 %64 = OpImageRead %v4float %61 %63
       OpStore %normal_specpower %64
...
```

Listing 7 – *Non-instrumented.*

```
...
       %OpName %__1 ""
       %OpDecorate %normals_map DescriptorSet 0
       %OpDecorate %_arr_uint_1_uint_1_2 ArrayStride 4
       %OpDecorate %_arr_uint_1_uint_1_2_0 ArrayStride 4
       %OpDecorate %258 Binding 12
       %OpDecorate %258 DescriptorSet 0
       %OpMemberDecorate %_struct_270 1 Offset 8
       %OpDecorate %_struct_270 BufferBlock
       %OpMemberDecorate %_struct_270 0 Offset 0
       %OpDecorate %normals_map Binding 15
       %OpDecorate %normals_map InputAttachmentIndex 4
...
 %64 = OpImageRead %v4float %61 %63
%276 = OpAccessChain %_ptr_Uniform_uint_1 %258 %int_4_0 %int_4_1
%275 = OpAtomicIAdd %uint_1 %276 %uint_1_1 %uint_1_0 %uint_1_1
       OpStore %normal_specpower %64
...
```

Listing 8 – *Instrumented.*

Figure 12 – *Listings 7 and 8 present sample SPIR-V code showing the comparison between a non-instrumented fragment module and one after instrumentation respectively. The module is created from compilation of the GLSL shader g_shade.frag, the fragment shader for the second subpass of the deferred renderer, which is then instrumented. Listing 7 is a line-by-line diff of Listing 8, where the highlighted lines represent the injected code.*

C Screenshots relative to samples

The subsequent figures present the screenshots for various test cases, e.g. Sponza with deferred rendering at 1280×720. For each test case 10 samples were taken, hence each figure presents 10 screenshots. In each figure, starting from the top left, the samples are ordered left-to-right, e.g. the screenshot relative to sample 3 is on in the first column of the second row of a given figure. Since the captures were all taken at the same locations per scene, not all resolutions are represented in these figures: only one resolution is used per scene since the screenshots for other resolutions would be identical with the differences only in a smaller pixel count. This amount of screenshots is enough to understand the conditions under which the memory bandwidth measurements were performed.

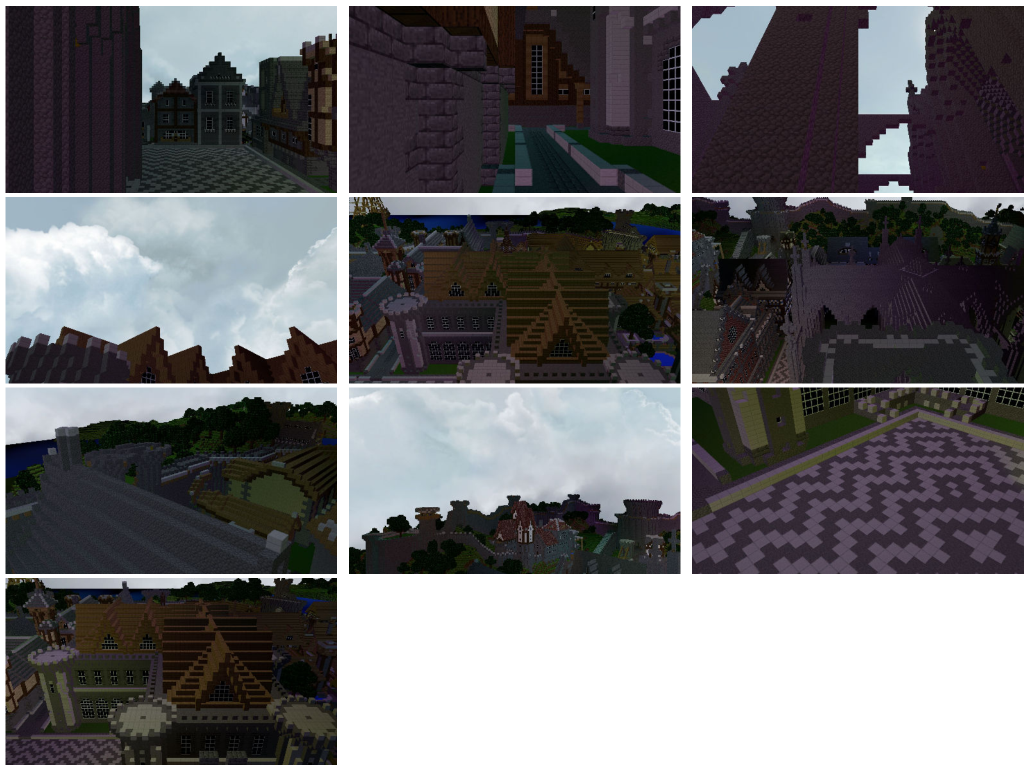


Figure 13 – *Screenshots for samples from deferred rendering at 1920×1080 in Rungholt.*

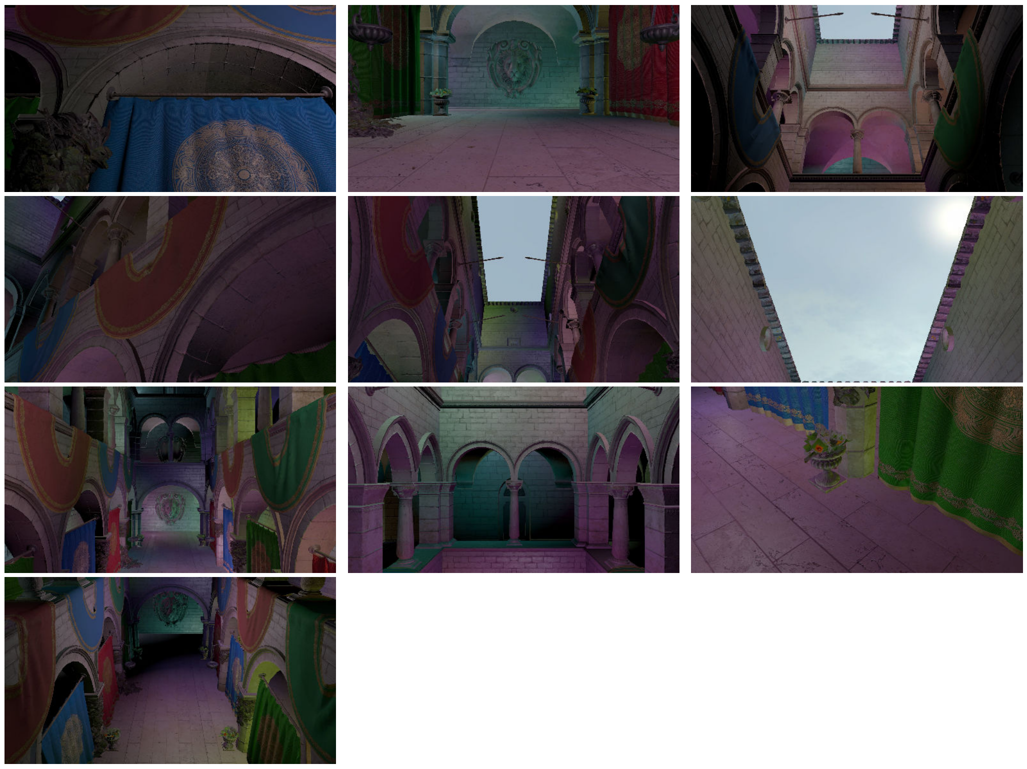


Figure 14 – *Screenshots for samples from deferred rendering at 1920×1080 in Sponza.*

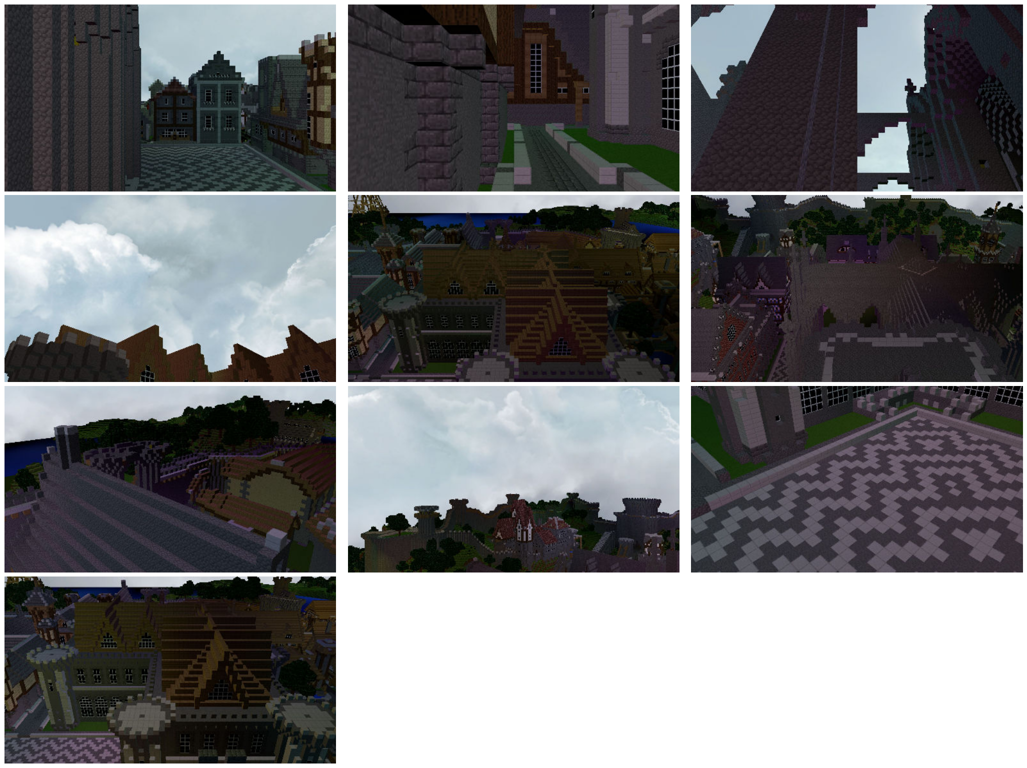


Figure 15 – *Screenshots for samples from visibility buffer rendering at 1920×1080 in Rungholt.*

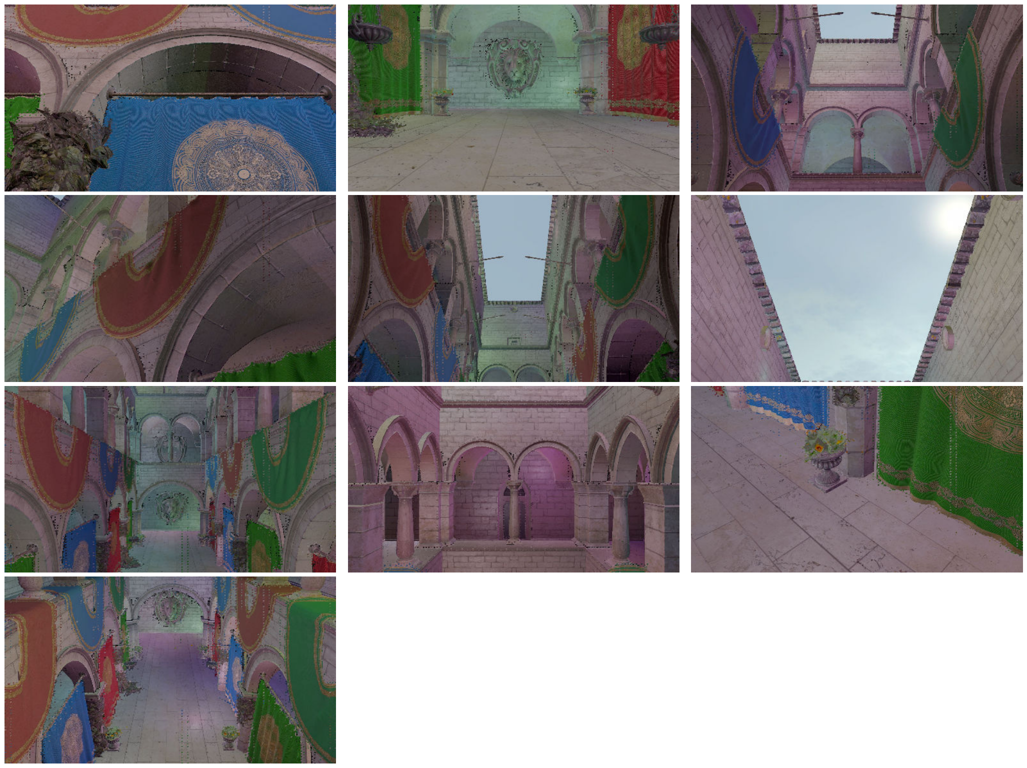


Figure 16 – *Screenshots for samples from visibility buffer rendering at 1920×1080 in Sponza.*