Vulkan Renderer

Ahamd Shahbaaz Hussain

January 2025

Contents

List of Figures

1 Overdraw and Overshading

Overdraw and overshading visualizations can be incredibly insightful to find areas within the scene were the GPU is doing wasteful work by shading fragments that do not contribute to the final image. The GPU utilizes the depth buffer to determine screen visibility however, as meshes are drawn, the visibility of a fragment can change. A fragment which appears closer the viewer will overwrite the existing fragment. This leads to redundant computations by the GPU which we hope to identity through overdraw and overshading visualizations to better optimize the fragment shading process to prevent wastefully shading fragments which do not contribute to the final image [O'Conor and 2017, 2017].

The overdraw and overshading visualizations can be switched between using the `6` and `7` keys when using the *forward* renderer selected with key `9`.

Overdraw has been implemented by using the `VK_BLEND_OP_ADD` blend operation with `.depthEnabled` and `.depthWrite` set to `VK_FALSE`. This will allow us to visualize how many fragments have been generated per pixel without considering whether those fragments are eventually visible after depth testing. Overshading is implemented with a similar pipeline, except `.depthEnable` and `.depthWrite` are set to `VK_TRUE` to enable both early and late depth testing. This will ensure that fragments which pass the depth tests contribute to the final image.

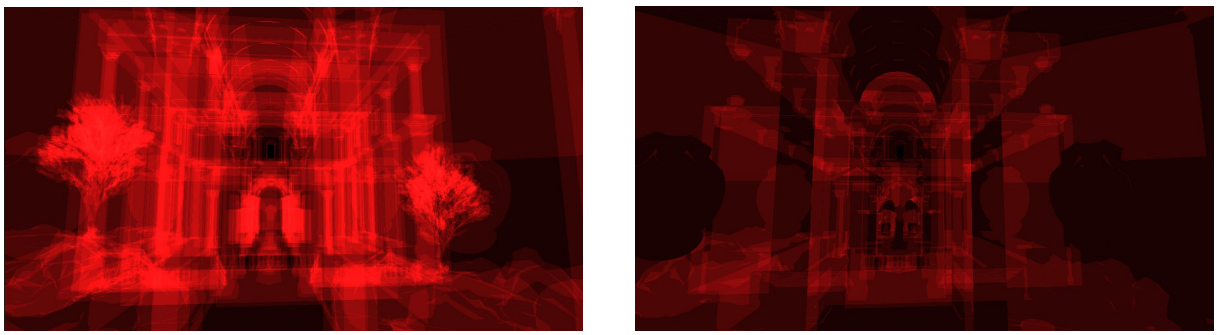
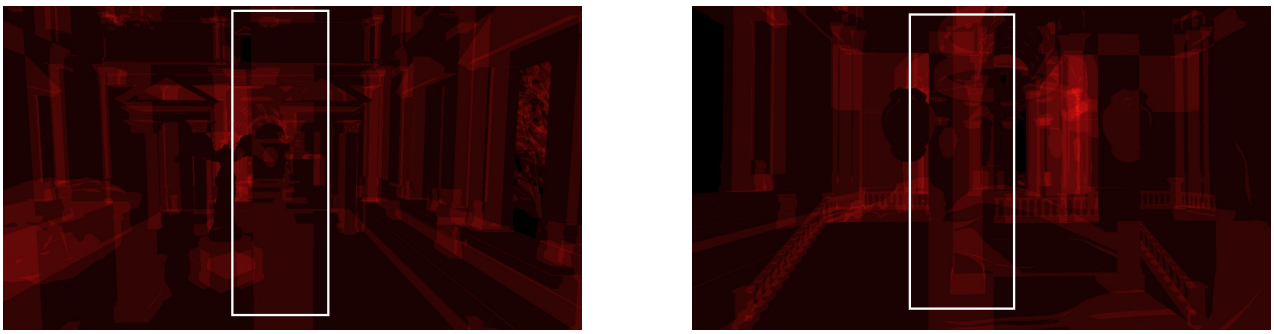


Figure 1: (left) Overdraw and (right) Overshading Visualizations.

With overshading enabled, we can explore the scene and observe something interesting. From one perspective Figure 2(b), we see a pillar inside the pillar in front of us. However, viewing from the opposite perspective Figure 2(a), we don't see the other pillar. While this might seem counterintutive, it aligns with how depth testing works and provides insight into the mesh rendering order. In Figure 2(a), many objects closest to the camera are rendered first, and those farther away are rendered later. Since depth testing discards fragments of farther objects when closer ones have already been drawn, overshading does not occur. In contrast, from the perspective of Figure 2(b), the rendering order is reversed. Distant objects are rendered first, followed by those closer. When the closer meshes are eventually drawn, they pass the depth test and overwrite the fragments of previously rendered distance objects, resulting in overshading.



(a) Perspective A.

(b) Opposite side of Perspective A.

Figure 2: (a) Closer pillar is drawn first which prevents overshading. (b) Closer pillar is drawn after pillar further away resulting in overshading.

1.1 Reducing overdrawn and overshading

We can reduce *baseline overdraw* (fragments generated without any depth testing) using a build-in GPU hardware feature known as depth testing. Depth testing occurs in two stages during the graphics pipeline; before the fragment is shaded (early depth testing) and after the fragment is shaded (late depth testing). A depth attachment needs to be provided during scene rendering, to which depth values are stored to check during the two stages whether the fragment will contribute to the image on screen. If it does not contribute, it is discarded. Early depth testing is particularly useful as it discards unnecessary fragments before the expensive fragment shader invocation occurs, significantly improving performance [Anagnostou, 2013] by reducing high overdraw-though it is not without it's limitations which will be discussed later. While depth testing does help reduce the overall overdraw problem, since the depth attachment used to determine fragment visibility is written to in real-time during rendering, it is not aware of the final order of objects which would end up on the screen as they are still being rendered and therefore will find itself overwriting many fragments which were previously determined closest, resulting in overshading and wasted GPU work.

In addition to depth testing, we have seen how the order in which meshes are rendered plays a crucial role in reducing overshading Figure 2(a). Intuitively we can observe that an effective method to optimize and reduce overshading is to render meshes in a **front-to-back** order, prioritizing those closer to the viewer [O'Conor and 2017, 2017]. This will allow early depth testing to be more effective in discarding fragments that overlap with previously rendered geometry Figure 2(a). While this technique can reduce both overdraw and overshading, it may not always be the most practical solution. For example, it can be quite difficult to predict the order of meshes in a dynamic scene and if the scene contains transparent objects, the rendering would need to occur back-to-front.

While mesh ordering can be an effective optimization to reduce overdraw and overshading, it's an optimization limited primarily to static opaque scenes as dynamic scenes are difficult to predict and transparency requires back-to-front rendering [O'Conor and 2017, 2017]. A more practical approach to handle both static and dynamic scenes would be to utilize a depth-prepass [Han et al., 2017]. Overdraw occurs when the GPU processes fragments that are covered by other fragments, leading to unnecessary work. During the depth-prepass, the geometry of the scene is rendered into the depth buffer, but no shading is performed. Instead, the GPU writes only depth information to depth target, without invoking the fragment shader. This process ensures that the depth buffer holds the depth values of all visible fragments. In the subsequent shading pass, the scene is rendered again using the depth-prepass depth buffer with *a less than or equal* depth compare operation to perform depth testing, allowing it to skip shading fragments that are hidden or occluded, thus preventing both overdraw and overshading [Anagnostou, 2020]. While this significantly reduces

overshading and overdraw, it requires the scene to be rendered twice (excluding shadow map pass), one for the depth-prepass and again for the shading pass which may be limited by hardware performance. Depth-prepass has been implemented and utilized with the *forward* rendering pipeline. Figure 3(b) demonstrates the effectiveness of the depth-prepass to reduce overshading.

Note: The depth-prepass is temporarily used here to demonstrate it's effectiveness. The submitted code has the depth-prepass in the source code but it is not used to meet the requirements of viewing overshading.

(a) Overshading visualization without depth pre-pass

(b) Overshading visualization with depth-prepass

Figure 3: Using depth-prepass to help eliminate overshading.

Depth-prepass is an effective optimization to reduce the overshading problem at the rasterization level. However, it requires all the objects to be rendered which can be quite costly. An alternative optimization to reduce overdraw and overshading would be to utilize *occlusion queries* [Sekulic, 2004]. This works by rendering a simple representation of the complex object, often a bounding-box [Wimmer and Bittner, 2005] and the number of visible pixels is tracked. If the number of visible pixels is greater than 0, we can render the entire object since we can be certain it appears on screen.

2 Naive Deferred Shading

For the naive deferred renderer, the following formats are used for each G-buffer target:

- **Albedo**: `VK_FORMAT_R8G8B8A8_SRGB`.
- **Normal**: `VK_FORMAT_A2R10G10B10_UNORM_PACK32`.
- **Metallic and Roughness**: `VK_FORMAT_R8G8_UNORM`.
- **Emissive**: `VK_FORMAT_R16G16B16A16_SFLOAT`.
- **Depth**: `VK_FORMAT_D32_SFLOAT`.

These formats were selected to balance precision and memory efficiency, optimizing both G-buffer read and write performance while staying true to the simplicity of a naive implementation

- **Albedo**: An 8-bit RGBA format is sufficient for storing color data, ensuring a low memory footprint.
- **Normal**: While `VK_FORMAT_R8G8B8A8_UNORM` (32-bit) may seem suitable, using a 10-bit RGB with a 2-bit alpha channel (32-bit) (`VK_FORMAT_A2R10G10B10_UNORM_PACK32`) provides higher precision for normals with the same memory usage.
- **Roughness and Metallic**: Stored in a single R8G8 texture which is sufficient for the data they store.
- **Emissive**: The `RGBA16F` format allows for high dynamic range (HDR) data, for bright emissive objects which is essential to apply post-processing such as bloom.
- **Depth**: A 32-bit floating-point format is used to ensure precision to reconstruct world position.

These formats seem most appropriate for the G-buffer in order to optimize it for both precision and performance while keeping memory usage low. Often a *positions* render target is also part of the G-buffer, however we have omitted this in favor of reconstructing world positions using the depth from the depth target. Further optimizations can be done such as omitting the normals render target and reconstruct world-space normals using the depth target. However, since this is a naive deferred renderer, and as result, we store normals.

The G-buffer is collectively 22 bytes. It could be reduced further to 18 bytes by reconstructing normals using depth information. Additional optimizations could be made to further reduce the size as seen in the frame breakdown of Cyberpunk 2077 [Pesce, 2020]. While the depth format is not specified in the analysis, assuming 32-bit depth format, the combined G-buffer used is 16-bytes per pixel. A similar 16-byte per pixel G-buffer is also seen in Metro Exodus [Schreiner, 2019]. However, the smaller G-buffer size in these games is likely due to hardware optimizations to meet specific target requirements. Our the 22-byte size is roughly what is expected for a G-buffer [Burns and Hunt, 2013].

Given a display with a screen resolution of 1920x1080, running at 60hz, we can compute the bandwidth for writing to the G-buffer targets and subsequently reading from them for shading. Given our G-buffer, we have a total of 22 bytes per pixel. The total bandwidth per frame can be computed as $22 \times (2.1 \times 10^6) = 46.2\,\text{MB/frame}$. We multiply this by 2 for both writing and reading. This means the bandwidth for our G-buffer is $92.4\text{MB} \times 60 = 5.544\text{GB/second}$ [Burns and Hunt, 2013]. This is of course, an approximation and does not take into account other factors involved in rendering to render the full-frame. However, it provides a good estimation of the bandwidth required by our G-buffer and the amount of data transfer the GPU needs to handle.

2.1 Forward vs Deferred compute cost

The standard approach to rendering rasterized triangles on the sceen involves taking some geometry data and passing it through the graphics pipeline. Starting with the vertex shader which processes input data by applying various transformations such as perspective projection to correctly position geometry. Relevant data required for shading are passed to the fragment shader, where the fragment is shaded before outputting to the screen. Doing this for all objects in our scene, the time complexity of lighting the scene is $\mathcal{O}(\text{objects} \times \text{numLights})$. If we write it as $\mathcal{O}(\text{numTriangles} \times \text{numLights})$ we can begin to understand how computationally expensive the fragment shader invocation becomes. If we have a single mesh with 1 million (1M) triangles and let's assume each triangle covers 8 fragments. This means for 1 million triangles the total number of fragments would be $\mathbf{1M}_{\text{triangles}} \times \mathbf{8}_{\text{fragments}} = \mathbf{8M}$ fragments, multiply by the number of lights `L = 4`, we have $8,000,000 \times \texttt{L} = 32,000,000$ lighting computations. Some of these fragments may not appear in the final image.

Deferred shading eliminates much of the wasteful GPU work that occurs through forward rendering by decoupling the work of processing geometry and shading [Thaler, 2011]. Deferred rendering is seperated into two passes: geometry processing and shading. The first stage, processes the geometry without shading. Relevant data required for shading is written to multiple render targets (MRT), collectively known as the *G-buffer*. These targets are passed to a subsequent pass which draws a full-screen quad/triangle and samples the G-buffer targets to retrieve data to compute shading. This dramatically reduces the compute cost to $\mathcal{O}(\text{screenResolution} \times \text{numLights})$. If we have a resolution of 1920x1080 and assume that each pixel has one fragment, that would be $\approx$ 2.07M fragments computed, multiplied by the number of lights we have $\approx$ 8.28M lighting computations-much less than the 32M from forward rendering.

For a performance comparison of forward and deferred rendering, Nvidia Nsight has been used to capture a frame with shading applied using both deferred and forward rendering.

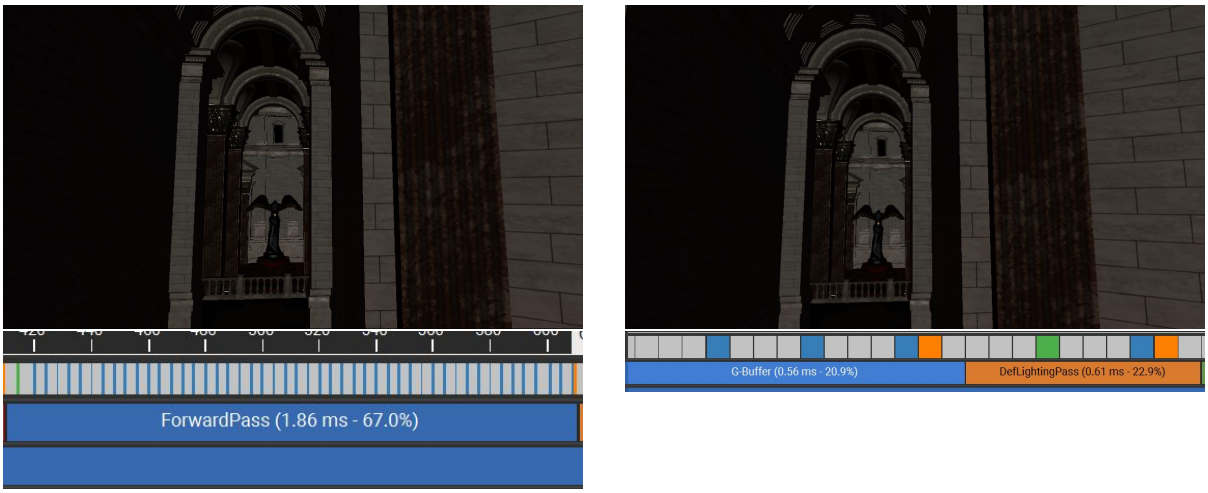


Figure 4: (Left) Forward pass shading output and compute time. (Right) Deferred rendering pass with compute time for G-buffer and lighting.

The forward rendering pass took 1.86ms to produce the scene with shading. In comparison, the deferred rendering pass took 0.56ms to generate the G-buffer targets and 0.61ms to apply shading, totaling 1.17ms to render the shaded scene.

2.2 Overshading

From our discussion in Section 1.1, we addressed the issues of overdraw and overshading, and explored strategies to mitigate them in the context of a forward renderer. We found that utilizing a depth-prepass can be an effective way to reduce the amount of wasteful computation performed by the GPU. To briefly recap, a depth target is used to store the depth of fragments that will appear on the screen. This allows the forward pass to discard fragments during both early and late depth testing that will not contribute to the final image. Without the depth-prepass in forward rendering, shading would incur the cost of processing fragments that ultimately won't be displayed. Deferred rendering indirectly applies this optimization, ensuring we only pay the cost once. In the G-buffer creation pass, the targets are updated to store only the data of fragments that appear on the screen, similar to a depth-prepass. When retrieving data from the G-buffer for shading, we can be certain that the fragment being shaded is guaranteed to appear on the screen, thus reducing both overdraw and overshading with the same effectiveness as the depth-prepass in forward rendering.

3 Mesh Density

The mesh density visualization can be activated using Key **8** with the forward renderer enabled Key **9**. Mesh density is defined using triangles.

When the GPU renders to the screen, for efficiency, pixel shading operations do not occur on a single pixel, instead the GPU performs shading on a 2x2 block of pixels, known as a quad. If a triangle's area is small and occupies only a small portion of the quad, significant computational waste can occur. For example, if a triangle edge covers only one region of the quad, the GPU must still process the entire 2x2 block, discarding 75% of the work where no triangle is present [Świerad, 2019]. This inefficiency becomes particularly costly with complex lighting shaders [Hill, 2012].

While triangle count is often associated with mesh density, modern GPUs can handle millions of triangles while maintaining real-time frame rates (60fps) with ease [Meysman, 2020]. As a result, the implementation prioritizes identifying long and thin triangles. This is more likely to contribute to inefficient computations and hinder performance unnecessarily, even if the overall triangle count seems acceptable [Meysman, 2020].

Knowing that the size of triangles can lead to wasteful computations on the GPU motivated the decision to focus on triangles rather than points or vertices, and to define density based on triangle size relative to pixel size. The method identifies problematic triangles by calculating their screen-space area in the geometry shader and comparing it to the size of a single texel (pixel). Triangles with areas smaller than or equal to a texel are flagged as problematic since they are likely to cause quad overdraw.

Additionally, this approach is extended to identify triangles that are small in area and also long. A "long triangle" is defined as a triangle that exceeds the size of a pixel as we hope to identify triangles which may exceed primitive/pixel. This will allow us to identify small, long and thin triangles which will contribute to overdraw. This is achieved by computing a screen-space bounding-box for the triangle and checking if it exceeds the size of a texel. If it does, we can assume the triangle is affecting more than the current pixel.

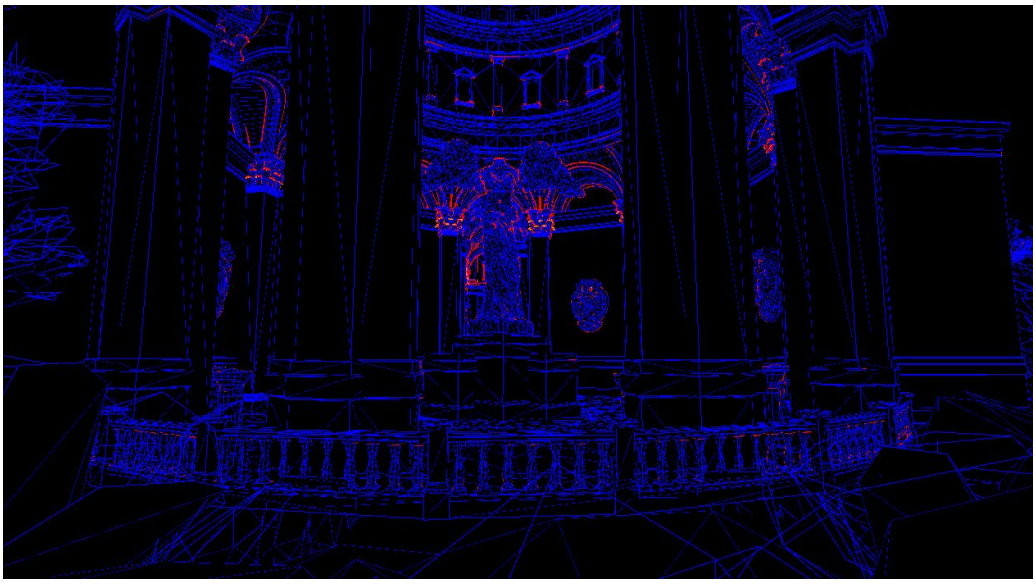


Figure 5: Mesh density visualization.

Problematic triangles are visualized using a range of colors: red represents the most problematic triangles-those that are small in area and long, while orange indicates triangles with a small area, and blue highlights the least problematic ones. This helps to identify and categorize problematic triangles.

3.1 Meeting target requirements

The visualization meets the target requirements laid out to ensure it is an effective visualization. The visualization was designed to be easily integrated into an existing engine and utilizes data which is already available. The method utilizes a depth-prepass buffer which all engines should already have whether they are deferred or forward renderers and per-vertex data which is passed to the geometry shader in groups of 3 to form a triangle. Calculations occur in the geometry shader and the appropriate colour is passed to the fragment shader. This keeps the integration of the visualization in other engines simple, utilizing already available rendering data.

In addition to this, the visualization focuses on displaying front-most visible surfaces to ensure that geometry which will affect the final image is considered. This approach focuses on geometries that appear on screen as they are the ones directly impacting performance.

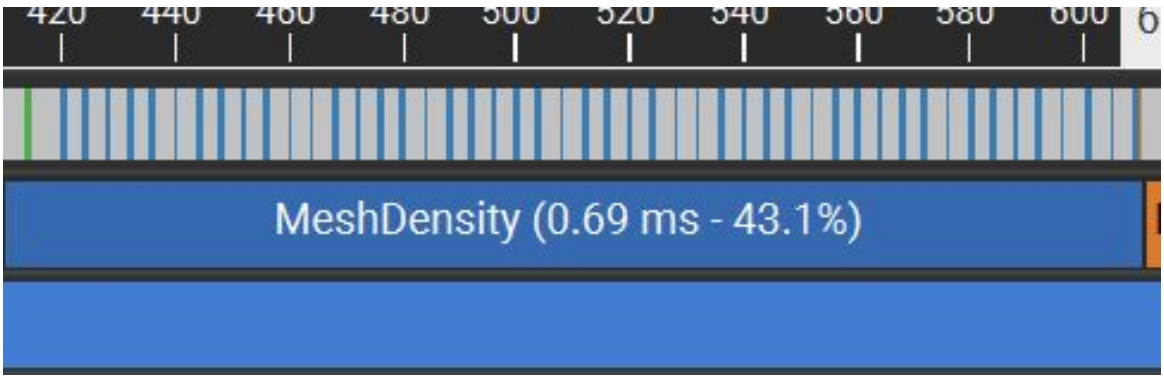


Figure 6: Mesh density compute time.

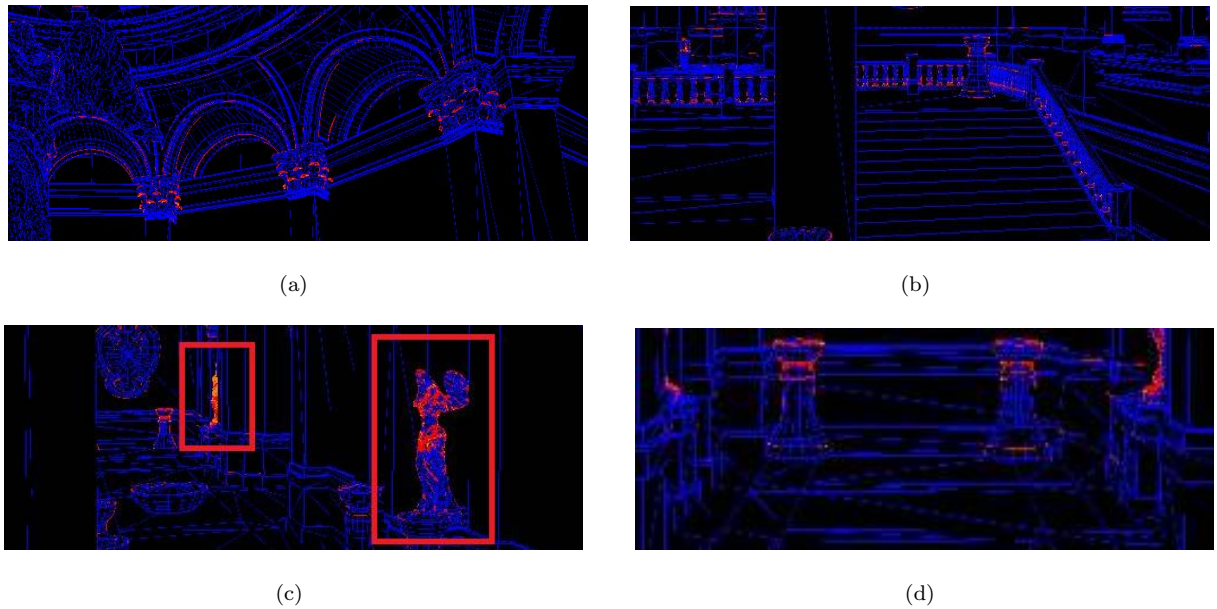
Furthermore, the visualization computes in 0.69ms, ensuring real-time performance. This enabled users to move around the scene iteratively in real-time and inspect the entire scene for problematic triangles.

The visualization also highlights triangles that may exceed the primitive-to-pixel requirement. Specifically, it identifies long triangles where either the width or height exceeds that of a pixel. These triangles are marked as candidates for exceeding the primitive-to-pixel ratio, as they may leak into nearby pixels.

3.2 Proposing scene changes

The scene suffers heavily from small and thin triangles which significantly contribute to quad overdraw. Using the visualization, it was easy to identify problematic areas and meshes.

The pillar construction at the top contains problematic triangles and appears repeatedly across all the rooms Figure 7(a). Similarly, the staircases and balcony in the main room feature vertical bars constructed with smaller triangles near curved sections, which further exacerbate the issue (Figure 7(b)). Additionally, the vase-shaped meshes exhibit small and thin triangles around their curved areas (Figure 7(d)). One of the largest contributors to the quad overdraw problem is the statue mesh, which contains several problematic triangles that remain problematic in size at varying distances. This issue becomes less noticeable only when the viewer is very close to the mesh. Spread across multiple rooms, the statue mesh significantly contributes to the overdraw problem, as seen in Figure 7(c).

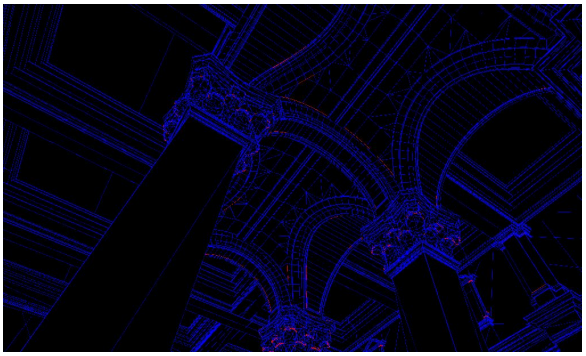


(a) (b)

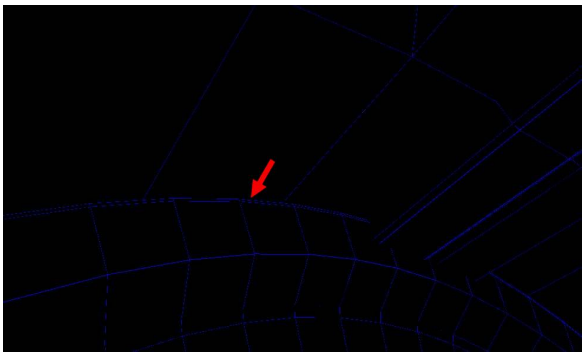
(c) (d)

Figure 7: (a) Top of pillar geometry with thin triangles. (b) Vertical bars on stair case with thin triangles. (c) Statue mesh suffers from a lot of problematic triangles. (d) Vase meshes with thin triangles near curved areas.

Further issues with the scene were found while investigating. The curved dome ceiling contains many thin triangles. The second room contains curved arches which have incredibly thin triangles which do not appear to contribute to forming the curved shape but rather are wastefully left there.



(a) Visualization highlighting the thin problematic triangles.



(b) Zoomed in area of the mesh displaying the thin triangles.

Figure 8: Many thin triangles can be seen wastefully placed in the scene which contribute to quad overdraw.

Collectively, the problemaic meshes identified throughout the scene, such as the pillar construction, staricase, balcony, vase-shaped meshes and curved ceilings, contribute significantly to quad overdraw. While the impact of each individual mesh might seem insignificant, their accumulative effect leads to significant performance impact. The collective impact of the meshes in the scene is demonstrated in Figure 9. It is clear the the scene suffers from many thin triangles, contributing to quad overdraw and impacting performance. The camera is moving forward to demonstrate that despite moving closer to these objects, many of these triangles remain thin and continue to be problematic.

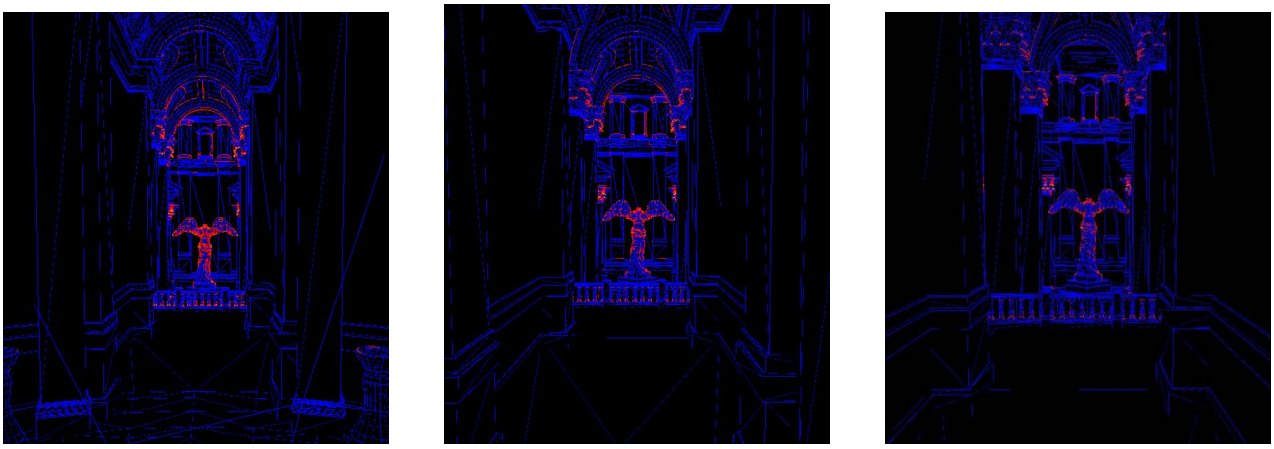


Figure 9: Collective impact the thin triangles from all meshes have on the scene. Camera is moving towards the statue.

3.2.1 Improving scene

We can address the issue of small and thin triangles in the scene by implementing a Level-of-Detail (LOD) system. LOD systems optimize mesh performance by dynamically adjusting the meshes triangle count based on various factors most notably adjusting based on the distance from the player [Edesberg, 2024]. The system generates progressively simplified versions of the original mesh ($\text{LOD}_1, \text{LOD}_2, \ldots, \text{LOD}_n$). As the player moves farther away. These simplified meshes replace the original, reducing both the polygon count and computational load [Nienaber, 2023]. More importantly, reducing the mesh detail introduces larger triangles to represent the geometry, effectively addressing the issue of small and thin triangles, reducing quad overdraw and further improving performance. This approach is comparable to replacing distant vegetation with 2D billboards.

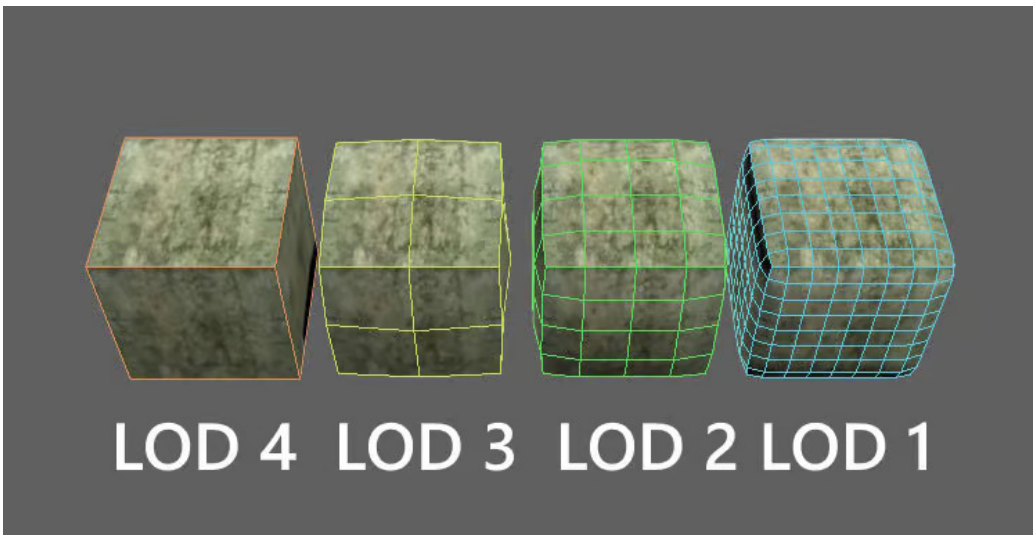


Figure 10: Level-of-Detail (LOD) of a mesh. LOD1 is most detailed [Nienaber, 2023].

In addition to this, improving the triangulation of some of the meshes in the scene can significantly help with performance. Figure 8(a) revealed some of the poor triangulation within the scene. Figure 11 demonstrates another example of poor triangulation within the scene.

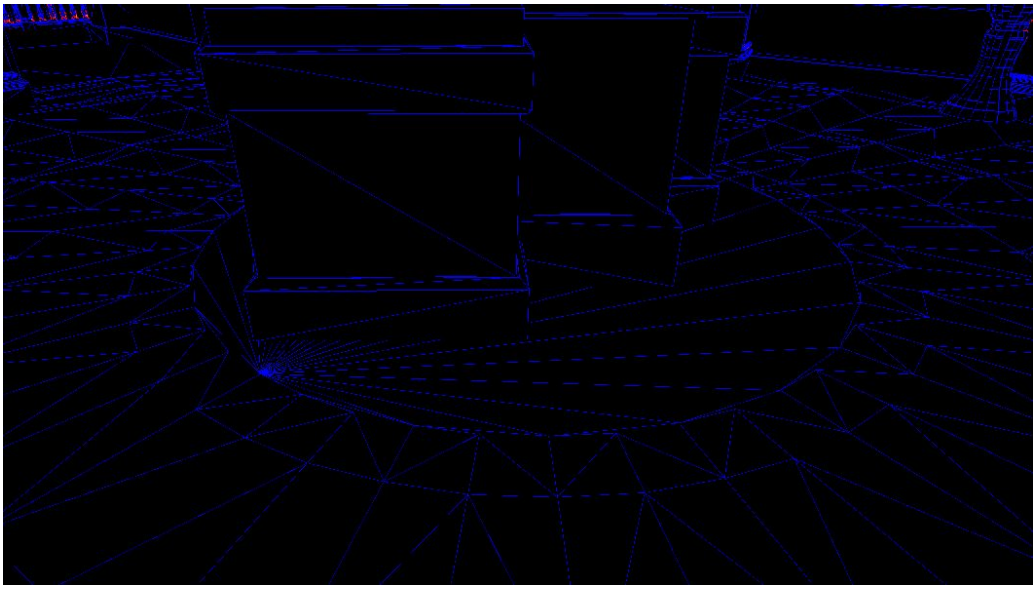
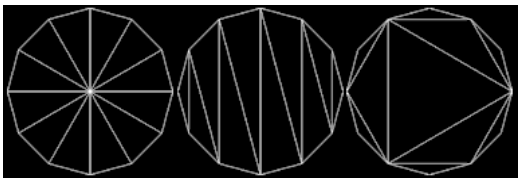


Figure 11: Poor triangulation of a mesh within the scene, producing inefficient triangles.

[Persson, 2009] identified that the triangulation of a mesh has a direct impact on performance. In their study, they triangulated a circle using three methods: a fan, a line strip, and a novel approach called *max-area*. The max-area method begins by placing an equilateral triangle at the center and recursively adds triangles along the boundary. This approach results in better-distributed triangles, minimizing thin, and inefficient triangles.



(a) Triangulation methods for a circle. (Left) Fan, (Middle) Line Strip, (Right) Max-Area, from [Persson, 2009]

(b) Performance results from [Persson, 2009], comparing triangulation methods. X-axis: vertex count, Y-axis: frames per second

Figure 12: [Persson, 2009] shows the impact of mesh triangulation on performance.

These findings are directly relevant to our scene, which contains meshes that would benefit from improved triangulation techniques such as max-area, specifically Figure 11 mesh which features the same shape as the one used in [Persson, 2009] testing. As seen in Figure 12(b), poor triangulation results in performance bottlenecks, reinforcing the need for optimized triangulation strategies to improve rendering efficiency for the scene.

3.3 Improving the visualization

While our mesh density visualization helps identify problematic triangles that contribute to quad overdraw, which negatively impacts rendering performance, the visualization could be further improved. Specifically, it should better display areas where the primitive/pixel limit is exceeded. Currently, the method relies on an assumption using the triangle's screen-space bounding box. However, a more effective approach would be to implement a visualization similar to Unreal Engine's quad overdraw debug view Figure 13, which uses a range of colours to highlight how many times a pixel has been re-rendered because multiple draw calls touch it. This would more accurately display areas where primitive/pixel overdraw is problematic. A possible implementation to achieve this would be to use atomic operations, specifically `imageAtomicAdd` and `imageAtomicExchange` in the fragment shader by enabling `VkPhysicalDeviceFeatures2.features.fragmentStoresAndAtomics`. With two storage images; one to store the ID of the primitive used to shade the current pixel, the other to store a counter. Store the `gl_PrimitiveID` value per-pixel when we shade the pixel and increment the counter. If the primitive ID changes next time we shade the pixel, we would have exceeded primitive/pixel.

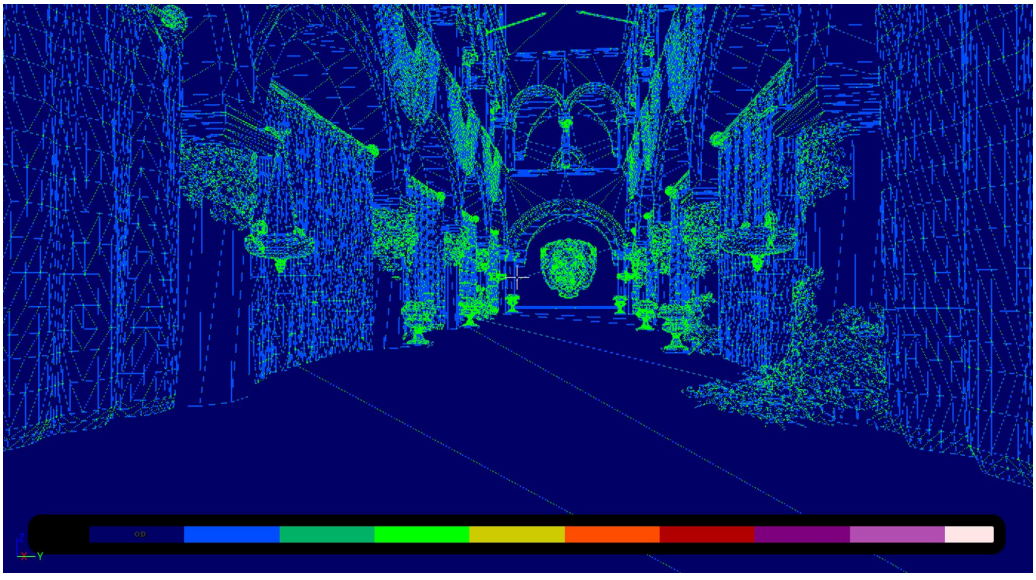
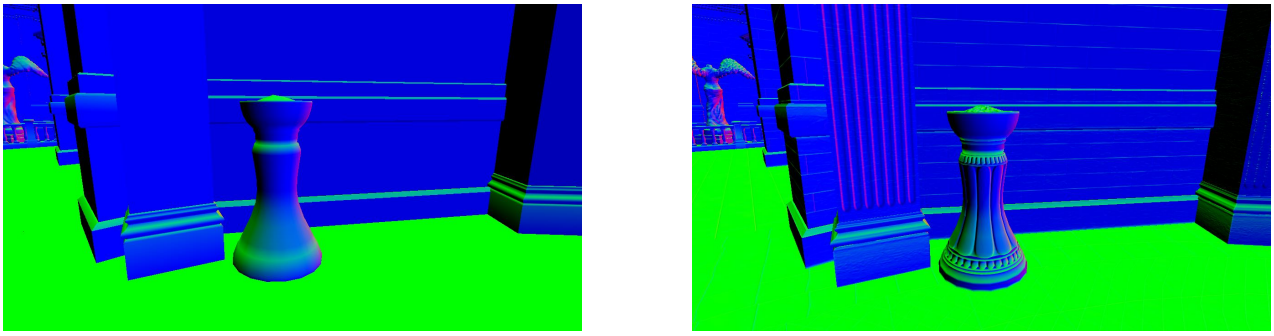


Figure 13: Unreal Engine Quad overdraw debug visual.

4 Normal Mapping



(a) Normals without normal mapping.

(b) Normals with normal mapping.

Figure 14: Mesh detailed enhanced using normal mapping.

4.1 Normal Format

When storing normal data, the goal is to achieve maximum accuracy while maintaining a low memory footprint. Although `RGBA8` might initially appear to meet these requirements, the `R10G10B10A2` format is a superior choice. Both formats share the same memory footprint of 32 bits, but `R10G10B10A2` provides 10 bits per channel for red, green, and blue, allowing for more precise storage of normal data compared to the 8 bits per channel in `RGBA8`. As a result, our implementation adopts `R10G10B10A2` to store normal data, maximizing accuracy without increasing memory usage.

The normal mapping approach used was inspired by [Frey, 2011] work on packing the Tangent, Bitangent, Normal (TBN) matrix. In their work, they introduced the idea of compressing the TBN matrix by converting it into a quaternion, specifically storing the x, y, z and reconstructing the w. This reduces the memory footprint of the 36-bytes 3x3 TBN matrix to a 12-byte representation. Although this is an improvement, the data can be further compressed. In our implementation, we compute tangent and bitangent vectors before baking the mesh and compute per-vertex TBN matrix which is then converted to a quaternion using `glm::quat_cast()`. We convert each x, y, z quaternion component into the range 0-1 and store each component as an 8-bit unsigned integer.

```
uint8_t packFloatTo8Bit(float f)
{
    float normalized = (f + 1.0f) / 2.0f; // normalize to 0-1
    uint8_t packed = static_cast<uint8_t>(normalized * 255.0f);
    return packed;
}

uint8_t xPacked = packFloatTo8Bit(qx);
uint8_t yPacked = packFloatTo8Bit(qy);
uint8_t zPacked = packFloatTo8Bit(qz);
```

The w component is reconstructed in the shader. As a result, our new compressed representation is only 24-bits large (3-bytes). This significantly reduces the memory footprint of the TBN from 36-bytes to 3-bytes. However, it is often the case that the entire TBN is not provided per-vertex. Instead, the tangent vector is provided, and the bitangent is reconstructed using a cross product of the normal and tangent.

Typically when a tangent vector is provided, it is a 4-component vector, with each component stored as a 32-bit float. This results in a combined 16 bytes for the tangent vector, which is much larger than our 3-byte compressed representation.

To compare memory consumption, we consider the best case where only a tangent vector is provided per-vertex. Suppose we have a scene with 2 million triangles, we have $2,000,000 \times 3 = 6,000,000$ vertices. A tangent vector is 16-bytes large $6,000,000 \times 16 = 96,000,000 \approx 91.66$MB. The compressed method utilizes only $6,000,000 \times 3 = 18,000,000 \approx 17.17$MB in comparison, a significant reduction in memory consumption.

4.2 Unpacking TBN matrix

The compressed representation is sent as per-vertex data to the vertex shader to be reconstructed. The w component is reconstructed as $w = \sqrt{1 - \text{dot}(q.xyz, q.xyz)}$ as suggested by [Frey, 2011]. The quaternion is used to reconstruct the TBN matrix, specifically we reconstruct the tangent and bitangent and use the world normal which is already available in the vertex shader. This saves us from having to reconstruct the normals from the quaternion, saving some compute time. The 3x3 TBN matrix is then sent to the fragment shader. The implementation reconstructs the TBN in the vertex shader as the fragment shader interpolates the attributes output from the vertex shader which ensures our values are correct for the current fragment [Giesen, 2011]. If we reconstruct our data in the fragment shader; not only will we not interpolate the attributes, we would introduce inaccuracies and overhead in the fragment shader which tends to be computationally more expensive than other stages.

4.3 Evaluating Errors

While the method to compress the TBN significantly reduces the size of the matrix, because we packing a 4-byte float into a 1-byte, 8-bit unsigned integer, it is importance to evaluate the amount of error we may introduce as the packing and unpacking process will result in some numerical instability.

To evaluate the error introduced, a simple `Python` program is written to generate 50 numbers in the range [-1, 1] (since the quaternion values fall within this range) and pack them into 8-bit unsigned integers. Each compressed value is then unpacked, and the *Relative Error* is computed. Relative Error is used as the measurement of error as it allows us to compare the error to the original value [Helmenstine, 2018]. This is computed as $\text{Error} = \frac{\text{AbsoluteError}}{|\text{ActualValue}|}$. This error is then converted into a percentage to better understand the impact of the compression method.

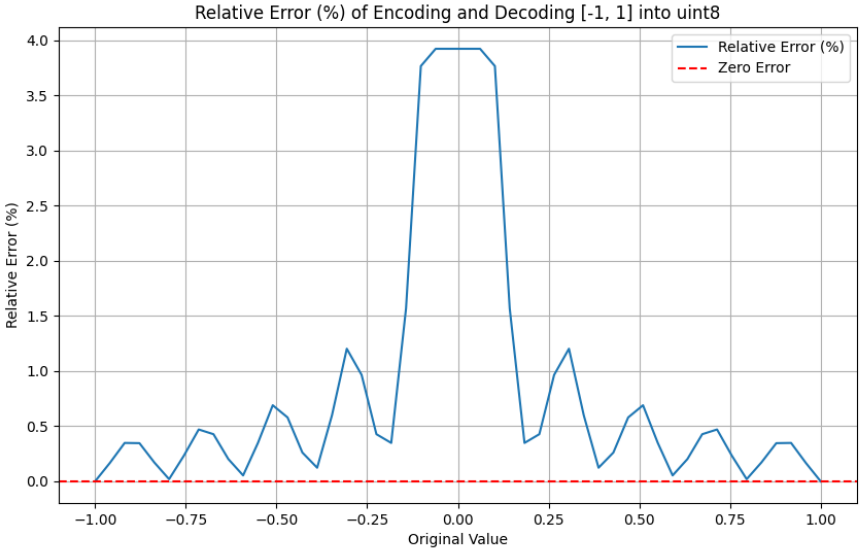


Figure 15: Results from testing 50 values in the range [-1,1] and reconstructing from a **uint8** to **float** and measuring relative error percentange.

From our test results, we observe that the error percentage increases for values closer to zero. Specifically, we introduce a maximum error of 1.3% for values further from zero, which is a very small margin. However, for values very close to zero, the error increases to almost 4%. While this might seem problematic, the introduced error is still relatively low. We would be concerned if we observed error values in the range of 25-50%.

4.3.1 4-component tangent

While we did not use the *tgen* library, a 4-component tangent is often produced as the **tangent.w** stores the handedness of the normals [Basler, 2023].

4.4 Final thoughts

Overall, given the substantial reduction in memory usage, minimal visual impact from the compression the compressed TBN representation is a highly effective optimization to reduce memory consumption.

5 Shadow Mapping

(a) Scene without shadow mapping. Lacks visual cues.

(b) Scene with shadow mapping.

Figure 16: The impact of shadow mapping.

Shadow mapping is a technique used to simulate shadows which are cast on surfaces when there is an occluder (object between shading surface and light) between the light and the object.

Similar to deferred shading, rendering shadows using shadow-mapping is a two-pass process. In the first pass the world is rendered from the light's point of view, similar to how we render a scene from the camera's perspective. We store the depth values of the scene from the light's perspective into a depth buffer.

Figure 17: First pass renders the scene to a depth buffer image from lights perspective.

The second pass would be our deferred/forward shading pass. In this pass, we use the shadow map depth buffer as a descriptor along with the light's view-transform matrix, commonly referred to as the *light-space matrix*. The light-space matrix allows us to transform position data from world space into light-space coordinates. For each point we shade, we: transform the position into light-space using the light-space matrix, perform-perspective division and transform to screen-space to map the position into the shadow maps texture-space. The shadow map is specified as a `sampler2DShadow` descriptor. This enables us to utilize GLSLs `textureProj` function, combined with a bilinear `sampler` allows us to perform hardware-accelerated `2x2` bilinear filtering. This filtering technique produces softer shadows Figure 18(b) and reduces the *stair-case* aliasing seen in Figure 18(a). We must ensure that the sampler we use for sampling the shadow-map utilizes `VK_COMPARE_OP_GREATER`. This is because we want to compare the depth values, specifically we want to check if the computed $z > depth$ which `textureProj` will handle for us.

(a) No hardware accelerated 2x2 PCF.

(b) Hardware accelerated 2x2 PCF.

Figure 18: Enabling hardware 2x2 PCF produces soft-shadows.

The implementation uses an orthographic projection for the light which will define the volume to be shadowed. An orthographic projection is used as a directional light source is infinitely far away, therefore it's light rays are all parallel [Vries, 2020]. Since this scene is mainly viewed from it's interior, we focused the shadow map on producing quality shadows inside the rooms. After some experimentation, to maximize the use of the shadow map, the following settings were utilized: $near = 0.1$, $far = 105$, orthographic view = 9.0. The chosen $near$ and far values allowed us to correctly encompass the scene in the view volume and capture details both in the main room and even further in other rooms. Together with the orthographic view, this enabled us to focus the shadow-map render on the interior.

(a) Shadows producing artifacts.

(b) Despite shadow-artifacts, we can observe the hardware filtering producing soft shadows.

Figure 19: (a) The raw output for shadows by simply checking if an object is or is not in shadow. Produces artifacts which need to be mitigated. (b) displays the hardware filtering producing soft shadows.

5.1 Shadow Artifacts

From our output Figure 19(a) we can observe significant artifacts resulting from our shadow calculation. This is commonly referred to as *shadow-acne* which occurs due to self-shadowing. This occurs when you transform a fragments position into light-space, small numerical inaccuracies due to the limitations of floating-point precision can occur [Microsoft, 2020]. This produces instances where the shadow map stores a depth value of 0.895 for some point and the transformed depth (z) value of the point might be computed as 0.91. Despite the fragment laying on the same surface-since $0.895 < 0.91$ the fragment is perceived to be in shadow.

To handle the *shadow-acne* we can apply a small offset (bias) to the z coordinate of the transformed position. This will ensure we only consider objects to be in shadow if there is some larger distance between objects.

(a) Shadows without a bias offset produces *shadow-acne*.

(b) *Shadow-acne* resolved by using a small (0.005) bias.

Figure 20: A small bias used to resolve the self-shadowing problem.

While the shadow bias can be an effective way to significantly reduce the shadow-acne, the bias value selected must be carefully tuned. A value that is too large will disconnect the shadow from the scene geometry, creating what is known as *peter-panning*. Rendering the shadow-map with front-face culling can help alleviate the peter-panning [Vries, 2020].

(a) Bias = 0.01

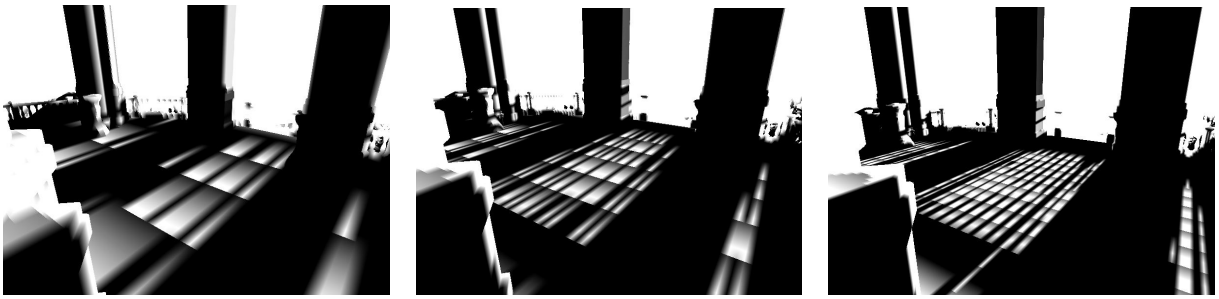
(b) Bias = 0.005

Figure 21: Tuning the bias value is crucial to ensure shadows do not disconnect (*Peter-panning*) from scene geometry.

5.2 Shadow-Map Resolution

Note: *Shadow-map resolutions are referred to using a single number e.g* 256, 512, 1024. *This refers to shadow-map resolutions* 256x256, 512x512 *and* 1024x1024 *respectively.*

Selecting an appropriate shadow-map resolution is crucial to get shadows that look realistic and produce the soft-shadows we want for a more realistic appearance. We tested several shadow-map resolutions to observe the shadows they would produce and select the one which produces the best result Figure 22.



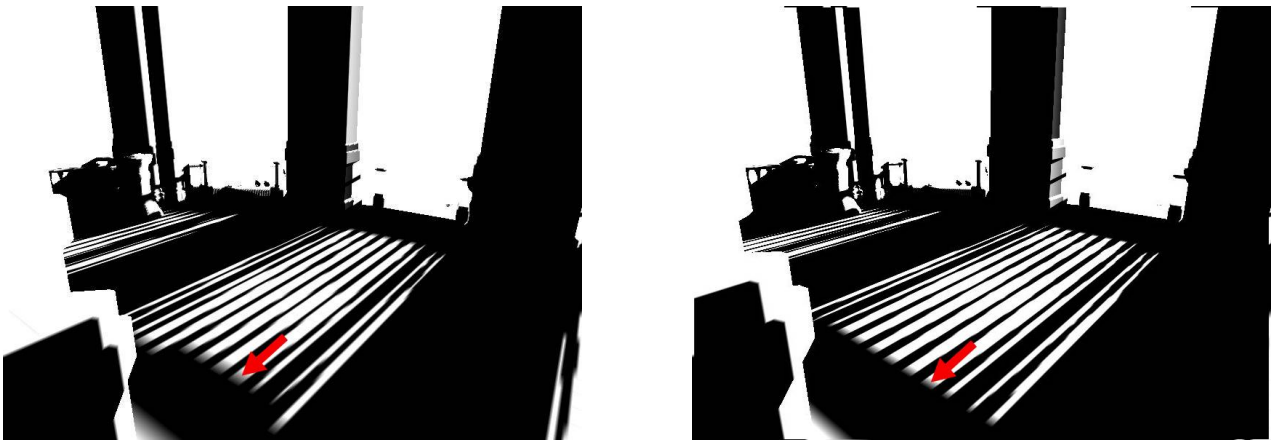
(a) 128. (b) 256. (c) 512.

(d) 1024. (e) 2048. (f) 4096.

Figure 22: Shadow output from different shadow-map texture resolutions.

We observe that lower resolutions (e.g., 128, 256, 512) produce a grid-like pattern in the shadows. This artifact is a form of shadow aliasing. This occurs because the depth map, which stores the scene's geometry, has a limited resolution. At lower resolutions, the depth map represents the geometry with fewer and less precise samples, resulting in blocky shadows. The depth buffer stores discrete samples, the lack of detail in these discrete samples results in large, visible blocks.

However, the high-resolution textures (1024, 2048, 4096) do not suffer from such artifacts as they provide more discrete samples. While high-resolution textures reduce artifacts, it increases memory usage and can have severe performance problems due to texture access problems as mentioned in [Microsoft, 2020]. Selecting a resolution involves finding a balance between shadow quality, performance and memory consumption. Naturally, 4096x4096 provides the most detail in our shadows however this resolution is wastefully too detailed for shadow mapping and does not meet our requirements of balancing memory consumption, performance and detail. 2048 produces great quality however, during our testing we noticed that 1024, while lower-resolution produces a more desirable soft-shadows appearance through hardware filtering which creates a more realistic scene.



(a) Soft-shadowing result using a 1024x1024 depth texture. (b) Soft-shadowing result using a 2048x2048 depth texture.

Figure 23: The 1024x1024 texture produces more visually desirable soft-shadows compared to 2048x2048 soft-shadowing result.

The 1024 shadow-map resolution provides the desired shadowing detail however, in some areas of the scene, we see noticeable aliasing referred to as *perspective aliasing* [Microsoft, 2020]. This occurs because a large number of eye-space pixels map to the same shadow texel [Astle, 2007]. Increasing the resolution of the texture seems the obvious choice to reduce the aliasing, however, while it does reduce the aliasing problem, it is very much still visible.

(a) (b)

Figure 24: (a) Perspective-aliasing from 1024x1024 shadow-map. (b) Perspective-aliasing from 2048x2048 shadow-map.

Increasing the shadow-map resolution further would continue to reduce the aliasing problem however, the shadow-map texture would be quite large and expensive to use. Several work has been done in the past decade to produce great looking shadows and solve such problems, notably [Dimitrov, 2007] work on *Cascaded Shadow-Maps* (CSM) which splits the view-frustum into *cascades* (sub-frustum). The frustum closest to the viewer has the highest resolution and those further have less. For each sub-frustum a shadow-map is rendered and the fragment shader will utilize the sub-frustum which matches the required resolution to shade the current fragment. This alleviates the aliasing problem associated with shadow-maps. While the solution proposed by [Dimitrov, 2007] reduces shadow aliasing by cleverly utilizing different shadow-map resolutions, [Bunnell and Pellacini, 2004] focus on implementing filtering techniques to achieve a similar goal.

6 Percentage-Closer Filtering (PCF)

To address shadow aliasing problem in our renderer, we implement Percentage-Closer Filtering (PCF) as a viable solution to address shadow aliasing and improving soft-shadows [Bunnell and Pellacini, 2004]. PCF works by sampling a region around the current pixel to approximate how many pixels are in shadow and averages the result. This is similar to the hardware bilinear filtering we discussed earlier Figure 18(b) with an extended kernel size to sample a larger region. [Bunnell and Pellacini, 2004] suggests utilizing a 4x4-texel sampling region which is adequate for quality and performance.

(a) No PCF filtering (Shadow-map)

(b) No PCF filtering (Shadow & Albedo) *stair-case* aliasing remains visible

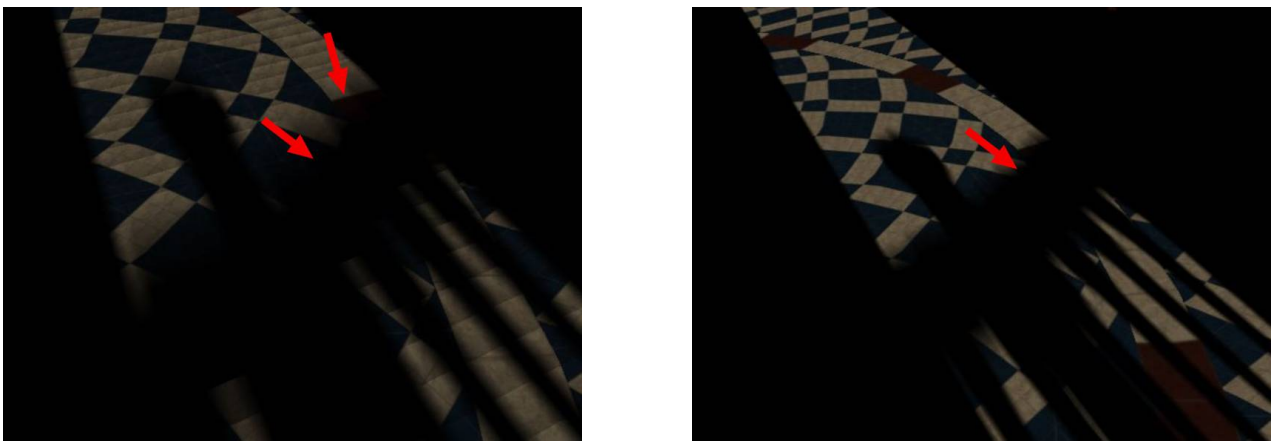
(c) PCF Filtering (4x4) (Shadow-map).

(d) PCF Filtering (4x4) (Shadow & Albedo).

Figure 25: Shadow output from different shadow-map texture resolutions.

The PCF filter has significantly reduced the aliasing. Rendering the scene with albedo and shadows without the filter Figure 25(b), we could see distracting aliasing on the shadows. However, with the PCF filter applied Figure 25(d), much of the aliasing has been significantly reduced. While still visible when rendering the shadows alone, combining the filtered shadows with the albedo colour hides a lot of the aliasing which was previously visible.

Since PCF filtering can hide a lot of the visible aliasing, we tested the filtering on a reduced shadow-map resolution (1024x1024) to see if we could reduce the memory of the shadow map.



(a)

(b)

Figure 26: (a) Perspective-aliasing from 1024x1024 shadow-map. (b) Perspective-aliasing from 2048x2048 shadow-map.

While much of the aliasing has been removed on the 1024x1024 texture, the blocky shape caused by aliasing remains visible with the filter enabled as seen in Figure 26(a) while the filtered result using the 2048x2048 texture suffers less from the visible block shapes.

The overall quality of the shadows is acceptable when using a 2048x2048 shadow-map resolution. Shading the scene with shadows takes only 0.49ms from our testing, the same as when using a 1024x1024 texture. Therefore, we use a 2048x2048 shadow-map resolution together with PCF filtering to render our shadows as it provides greater detail and reduced aliasing without a performance cost. We can extend the renderer with an optimized approach such as [Dimitrov, 2007] in the future to further reduce aliasing.

7 Bloom

To implement the bloom effect into our renderer, another render target was created during the deferred shading pass called the *Brightness* texture. This is a `RGBA16F` HDR texture which will be bound to the framebuffer as a render target and will store all the bright spots within our scene. Utilizing a HDR texture will allow us to store incredibly bright areas which exceed a luminance of 1.0.

During the deferred shading pass, we compute the shading for the current fragment and output this colour to the main deferred shading render target. Within the same fragment shader, we compute the luminance of the output colour to determine its brightness. If the luminance exceeds a specified threshold (1.0), the colour is also written to the brightness render target. We can see how the HDR texture works well here, as it can store colours values greater than 1.0. This will result in a render target which stores all the bright fragments in our scene which are precisely the areas of the scene the bloom effect should be applied.

7.1 Gaussian Blur

The bloom effect is created by taking the brightness texture and blurring the texture extensively which will create a *halo* like shape around the bright spots, resulting in our bloom effect [James and O'Rorke, 2004].

Gaussian blur can be implemented by applying a NxN kernel over the image and averaging the results with the precomputed weights. The weights can be computed as:

$$G(x, y) = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{x^2 + y^2}{2\sigma^2}\right) \tag{1}$$

However, this would be incredibly inefficient and costly. If we have a 22x22 kernel and an image with a 1280x720 resolution, that would be $\approx$ 446M texture fetches [Lisitsa, 2022].

7.1.1 Exploiting Separability

Gaussian filters are separable, meaning that instead of applying the 2D convolution directly, we can split the blurring process into two passes [James and O'Rorke, 2004]. The 1D Gaussian function used to compute the weights is given by the following formula:

$$G(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{x^2}{2\sigma^2}\right) \tag{2}$$

The Gaussian weights are computed on the CPU in `C++` using Equation 2 and the resulting weights are sent to the GPU through a uniform buffer. While pre-computing the weights and defining them in the fragment shader is more performant (as transferring data from the CPU to the GPU is slower) [Bickford, 2020], we only incur this cost once since the data remains constant across frames. This approach provides the flexibility to experiment with different kernel sizes and σ values. Once the desired blur settings have been found, the computed weights can be defined in the fragment shader when using the renderer in production.

Once the weights have been pre-computed we can begin our blurring passes. The first pass will blur the texture horizontally, and the second pass will take the horizontally blurred texture and blur it vertically, completing the blur. This significantly reduces the compute cost of the Gaussian blur from the naive implementation which was $O((\text{width} \cdot \text{height}) \cdot N^2) = O(N^2)$ to $O(2 \times N) = O(n)$.

7.1.2 Further optimizing

While this approach is already optimized, we can further enhance performance by exploiting GPU hardware bilinear filtering. This technique reduces a 43×43 Gaussian filter to just 22 taps. The optimization works in the following way: one texture fetch is used for the current pixel, and the remaining $N - 1$ taps $(43 - 1 = 42)$ are halved due to bilinear interpolation. This results in $\frac{42}{2} = 21$ additional taps, for a total of $21 + 1 = 22$.

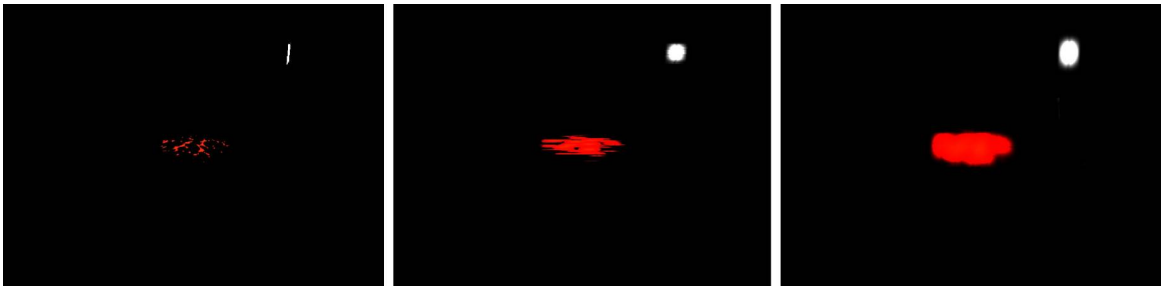
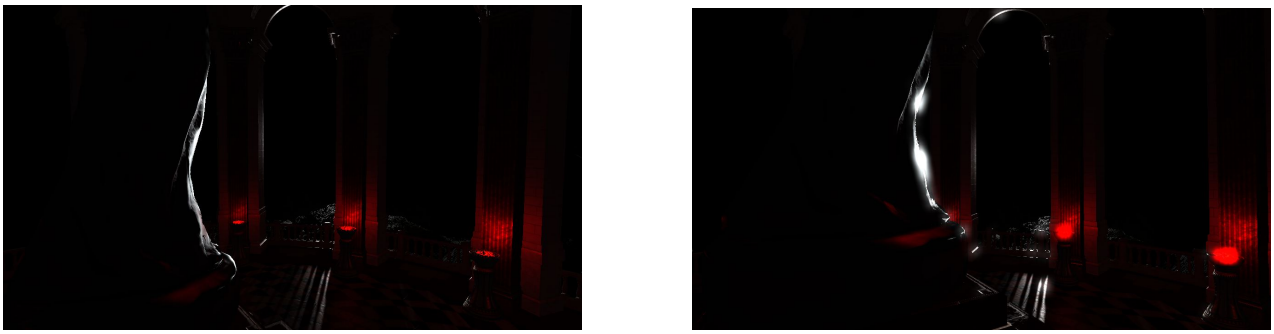


Figure 27: (Left) Brightness texture. (Middle) Horizontal Gaussian blur. (Right) Vertical Gaussian blur

After applying Gaussian blur, we have the Bloom effect applied to the bright regions of the scene. To combine the bloom with the deferred shading output target, a new render pass is introduced (*deferred composite*) which renders a full-screen triangle. This pass takes the deferred shading output target and the bloom output target as descriptors in the fragment shader. Each target is sampled and the colour is combined to produce a final image compositing both the deferred shading and bloom.



(a) Scene rendered without Bloom.

(b) Scene rendered with Bloom.

Figure 28: Bloom adds a real-time glow effect which adds a level of realism to the scene.

References

[Anagnostou, 2013] Anagnostou, K. (2013). Order and types of depth testing.

[Anagnostou, 2020] Anagnostou, K. (2020). To z-prepass or not to z-prepass.

[Astle, 2007] Astle, D. (2007). Shadow map aliasing.

[Basler, 2023] Basler, R. (2023). Three normal mapping techniques explained for the mathematically uninclined.

[Bickford, 2020] Bickford, N. (2020). $\mathrm{vk}_m ini_p ath_t racer$.

[Bunnell and Pellacini, 2004] Bunnell, M. and Pellacini, F. (2004). Chapter 11. shadow map antialiasing.

[Burns and Hunt, 2013] Burns, C. and Hunt, W. (2013). The visibility buffer: A cache-friendly approach to deferred shading. *The Visibility Buffer: A Cache-Friendly Approach to Deferred Shading*, 2.

[Dimitrov, 2007] Dimitrov, R. (2007). Cascaded shadow maps.

[Edesberg, 2024] Edesberg, A. (2024). Mastering level of detail (lod): Balancing graphics and performance in game development.

[Frey, 2011] Frey, I. (2011). Spherical skinning with dual-quaternions and qtangents.

[Giesen, 2011] Giesen, F. (2011). A trip through the graphics pipeline 2011.

[Han et al., 2017] Han, S., Kong, H., and Sander, P. (2017). Triangle reordering for efficient rendering in complex scenes. *Journal of Computer Graphics Techniques Triangle Reordering for Efficient Rendering in Complex Scenes*, 6.

[Helmenstine, 2018] Helmenstine, A. M. (2018). What is relative error?

[Hill, 2012] Hill, S. (2012). Counting quads.

[James and O'Rorke, 2004] James, G. and O'Rorke, J. (2004). Chapter 21. real-time glow.

[Lisitsa, 2022] Lisitsa, N. (2022). Compute shaders in graphics: Gaussian blur.

[Meysman, 2020] Meysman, D. (2020). Daan meysman - keeping your games 'optimized': Part 1 - triangles.

[Microsoft, 2020] Microsoft (2020). Common techniques to improve shadow depth maps - win32 apps.

[Nienaber, 2023] Nienaber, J. (2023). How lod systems work and why they are used.

[O'Conor and 2017, 2017] O'Conor, K. and 2017 (2017). Gpu performance for game artists.

[Persson, 2009] Persson, E. (2009). Humus - comments.

[Pesce, 2020] Pesce, A. (2020). Hallucinations re: the rendering of cyberpunk 2077.

[Schreiner, 2019] Schreiner, A. (2019). Metro exodus frame breakdown.

[Sekulic, 2004] Sekulic, D. (2004). Chapter 29. efficient occlusion culling.

[Thaler, 2011] Thaler, J. (2011). *Deferred Rendering*. PhD thesis.

[Vries, 2020] Vries, J. D. (2020). *Learn OpenGL: Learn modern OpenGL graphics programming in a step-by-step fashion*. Kendall Welling.

[Wimmer and Bittner, 2005] Wimmer, M. and Bittner, J. (2005). Chapter 6. hardware occlusion queries made useful.

[Świerad, 2019] Świerad, O. (2019). Pixel costs.