Bachelor Thesis

Dynamic real-time global illumination algorithms on modern GPU hardware and software

Author: Evgenii Afanasev
Student number: 444369
Study Program: Creative Media & Game Technologies
Study Coach: Paul Bonsma
Mentor: Malte Bennewitz
Date: June 2021

Acknowledgments

I would like to thank my mentor, Malte Bennewitz, for his time and help with various shader optimizations and my graphics-related questions. I would also like to thank Paul Bonsma and Hans Wichman for reviewing this document and giving me some tips on its structure, and Natalia Budnik for making the vectorized drawings.

Abstract

Many approximations for global illumination have been developed in the past decades for both real-time and non-real-time rendering. Those approaches were built for graphics software and hardware relevant to that time, which made some of the algorithms deprecated for modern computer graphics standards. Opposingly, by virtue of technological improvements, some methods now have the possibility of being enhanced for visuals and performance. This research focuses on some global illumination techniques from the past several years that are reimplemented for relevant *(to the time of the research)* software - a low-level graphics API, such as DirectX 12, and hardware - a modern GPU with hardware-accelerated raytracing, such as NVIDIA RTX2060. The algorithms are combined in a single open-source interactive demo and the comparisons are presented in this paper.

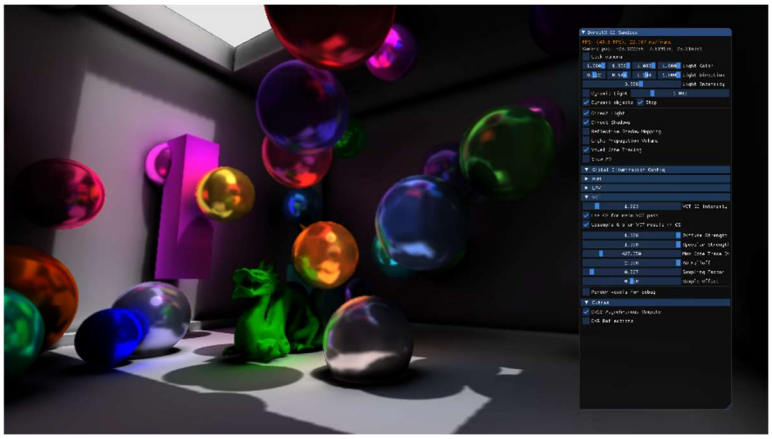


Figure 1. The final result from the project's demo application (https://github.com/steaklive/DXR-Sandbox-GI)

Notational Conventions

d – derivative
∂ – partial derivative
∂^2 – second partial derivative
$\langle \vec{a} | \vec{c} \rangle$ – dot product of two vectors
$||\vec{a}||$ – vector's length

Contents

I. Introduction

1.1 Background

One of the main challenges of computer graphics is the illumination of objects and their surfaces. In particular, rendering - the process of generating a digital image - often tries to solve that challenge as realistically as possible. Many scenes are not only lit from the direct sources (i.e., sun or lamp) but they also contain light contributions from the neighboring objects *(Figure 2)*. The phenomenon of calculating the total light energy gathered from different sources in the scene is known as *global illumination* in computer graphics.

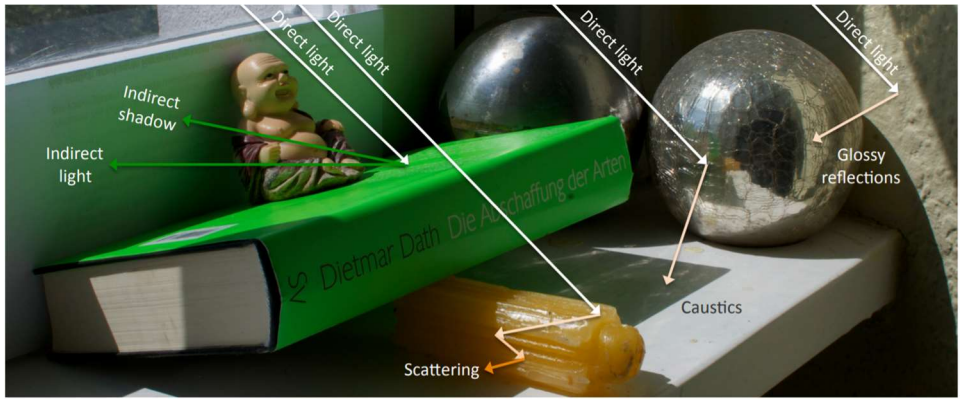


Figure 2. Global illumination effects in the real world (T. Ritschel, C. Dachsbacher, T. Grosch, J. Kautz, 2012).

Many researchers have been trying to solve that problem over the years with various approaches both on central processing units (CPUs) and on graphics processing units (GPUs). The latter became more popular in recent years due to its architecture which is more suited for *rasterization (B. Caulfield, 2018)*. This research focuses on real-time graphics approaches with the calculations that are run on GPU shaders and the commands for those calculations that are submitted from a CPU to a GPU with a dedicated graphics API (DirectX 12).

1.2 Main problem definition

Since one of the main goals of computer graphics is to render an image of the real-world scenario as close to it as possible, global illumination (often abbreviated as "GI") comes to our help with many separate partial solutions to that problem. On a practical side, it is worth mentioning that many complex phenomena from real life, such as reflections or shadows, fall into the subject of global illumination. The ways to approximate *direct* components of those effects were developed without proper GI in mind and have been pretty trivial for their computation in real-time. For example, methods like *shadow mapping (R. Dimitrov, 2007)* or *screen-space reflections (T. Stachowiak, 2015)* have gained popularity among developers for their simplicity and visually plausible results on the screen. However, due to their specifics, they are still not very accurate and cannot approximate the *indirect* component of the illumination at all: shadow mapping does not solve the indirect shadows problem, *ambient occlusion (J. Carlsson, 2010)*, and screen space reflections cannot do interreflections. These and other solutions are not unified – there is no convenient way to calculate interreflections and indirect shadows all together in one solution without advanced global illumination approaches. As a note, GI methods that are completely unified ("for everything"), such as *path-tracers*, *ray-tracers (M. Pharr, W. Jakob, G. Humphreys, 2016)*, are still not feasible for modern real-time graphics computation, although they are currently a subject of interest of many researchers and industry professionals thanks to the increasing capabilities of modern GPUs.

Figure 3. Offline path-traced image in "C-ray" (V. Koskivuori, 2021)

There is another differentiation between global illumination approaches in addition to unification – the ability to be *static* and/or *dynamic*. By *static,* we mainly refer to the solutions that pre-bake light data, store it in a specific way, and then process it. This approach can produce visually impressive results but needs to be redone every time the light's information or the objects' transformations are changed. One popular technique, that had been used in the game industry for many years in the past, is *lightmapping (K. Cupisz, T. Alexander Franke, 2018).* The main drawback of a static approach is memory consumption, which is often crucial in modern game development on consoles and handheld devices.

Dynamic techniques, on the other hand, have a much lighter impact on memory and allow GI to be recomputed in runtime. For this reason, they became popular in modern games with large scenes *(A. Yudintsev, 2019)* and were also chosen for this research. *Hybrid* approaches that combine both static and dynamic elements also exist and have been used successfully in AAA titles, such as *"Tom Clancy's The Division"* by Ubisoft *(N. Stefanov, 2016)*, although they are more useful for a big open-world engine rather than a demo application.

After giving a brief description of global illumination, one clear question can be raised that will also be the main question of this research: *how do popular dynamic GI techniques stand to each other on modern GPU hardware and software?*

The purpose of this research is an attempt to find a solution that would have the optimal balance between visual quality, scalability, and performance.

1.3 Sub-problems' definition

To answer the main question of the research, comparison and analysis of the existing approaches should be conducted. Due to the amount of GI techniques that exist, only the most relevant ones were used for this project: they had to be created in the 2000s at the earliest. An additional requirement for the selection was the accessibility of the algorithm – the technique should have a publicly available paper or presentation about its theory and implementation details. Undoubtedly, there are many techniques produced by game developers for their games and in-house engines that can compete with the algorithms from this research, however, they are usually not fully available and often not very "consumer-oriented" (code is not created for readability and for sharing with anyone unfamiliar with the codebase of a project). For this reason, such techniques are not taken into consideration for this research.

After browsing through the list of suitable GI techniques and estimating the timespan of the project, a decision to choose the following three has been made:

1) *Reflective Shadow Mapping (C. Dachsbacher, 2005)*
2) *Light Propagation Volumes (A. Kaplanyan, 2010)*
3) *Voxel Cone Tracing (C. Crassin, F. Neyret, M. Sainz, S. Green, E. Eisemann, 2011)*

All of them gained popularity and were made in the "prior-to-DX12" era, which made them possible for the reimplementation again and for many potential optimizations/improvements. Other candidates, such as *dynamically placed light probes (N. Stefanov, 2016)*, would require more engine work, or were not considered because of their quality level and potential artifacts that would be time-consuming to configure (i.e., *screen space global illumination (B. Peng, P. Poulin, 2019)*).

The sub-questions of the research are similar for every presented technique "A":

- *"How does "A" perform on modern hardware and software?"*
- *"How does "A" compare visually and technically to other techniques?"*
- *"Does "A" have any potential in the future or is it already deprecated?"*

A clear answer to the first sub-question answer will be based on the implementation of each technique in the project's demo engine, in-depth profiling with external tools, and added optimizations.

The second one will be based on the data gathered from those techniques after their implementation and their analysis by different criteria. For visuals, the algorithms will be compared to a set of offline-rendered images of the same test scene with path-traced global illumination (very close to a so-called "ground truth") for any existence of various artifacts, such as light leaking or shimmering, or the lack thereof. Technically, the comparison will be based on the CPU, GPU, and memory usages of the techniques and their final frame time.

The answer to the final sub-question will be a short conclusion and discussion where some thoughts will be given on the topic of every technique and on how an algorithm might or might not progress in future development scenarios.

1.4 Common theory

Simulating light accurately is a non-trivial task and requires an understanding of its properties together with some physical laws, such as the *conservation of energy*. Radiant energy is the energy graphics researchers and developers are mostly interested in when solving the problems of rendering. The term *radiant flux* Φ is an important measurement of radiant energy Q emitted, transmitted, received, or reflected, per unit time t:

$$\Phi = \frac{\partial Q}{\partial t} .$$

Radiant flux is an essential component in the definition of *radiance* which is often denoted as L and defines the density of flux per unit solid angle Ω per unit projected area $A \cos\theta$:

$$L_{\Omega} = \frac{\partial^2 \Phi}{\partial \Omega \, \partial A \cos\theta} .$$

Every point in the virtual 3D space that is projected on the 2D screen space can both emit and receive the radiance from a large number of directions surrounding that point until the equilibrium is reached. In principle, when radiance is conserved, the radiance emitted by a source is the same as the one received by the observer, or, in other words, by our virtual camera/eye.

Such a complex behavior was nicely put into one *rendering equation (J. Kajiya, 1986):*

$$L_{\mathrm{o}}(\boldsymbol{x}, \vec{\omega}_{\mathrm{o}}) = L_{\mathrm{e}}(\boldsymbol{x}, \vec{\omega}_{\mathrm{o}}) \; + \int_{\Omega+} f_{\mathrm{brdf}}(\boldsymbol{x}, \vec{\omega}_{\mathrm{i}} \rightarrow \vec{\omega}_{\mathrm{o}}) L_{\mathrm{i}}(\boldsymbol{x}, \vec{\omega}_{\mathrm{i}}) \langle \vec{\omega}_{\mathrm{i}} | \vec{\boldsymbol{n}} \rangle d\, \vec{\omega}_{\mathrm{i}} \; ,$$

where L_{o} is the outcoming radiance from the position $\boldsymbol{x}$ along the direction $\vec{\omega}_{\mathrm{o}}$, L_{e} is the emitted radiance, f_{brdf} is the bidirectional reflectance distribution function, $\vec{\omega}_{\mathrm{i}}$ is the negative direction of the incoming light, L_{i} is the incoming radiance from the direction $\vec{\omega}_{\mathrm{i}}$, and $\vec{\boldsymbol{n}}$ is the normal vector of the surface. The incoming light directions are coming from around the upper hemisphere $\Omega+$ which surrounds the point $\boldsymbol{x}$.

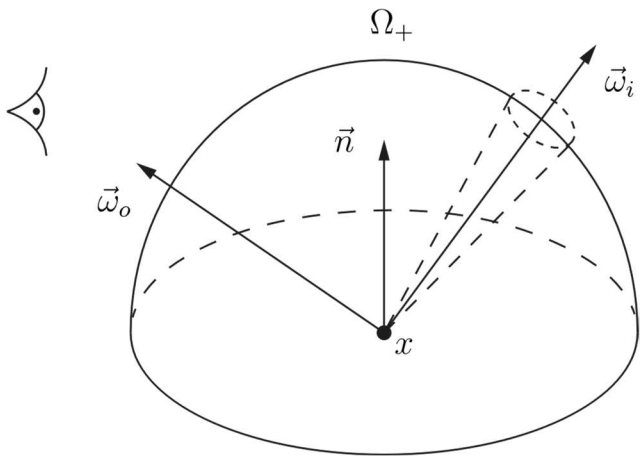


Figure 3. Geometry of the rendering equation

As it can be seen from the equation, the number of incoming light directions is not known, which makes the integral indefinite. Moreover, the light energy which comes from every direction $\vec{\omega}_{\mathrm{i}}$ completes a complex multi-bounce path before hitting the point $\boldsymbol{x}$. In other words, the path gathers contributions of outcoming radiance from every hit point along the way recursively. The number of recursions is also not defined and very non-trivial for being calculated numerically (the bounces stop when the equilibrium is reached: the light is fully absorbed). This fact makes the computation of the equation of a particular point $\boldsymbol{x}$ difficult for modern hardware, especially in real-time.

It should be mentioned that the presented equation contains a section that is not a part of this research – *bidirectional reflectance distribution function* or f_{brdf}. In short, this topic defines a material of the surface and its' relationship with incoming light. The topic is large by itself to study and is still active and popular among graphics developers and researchers. Many publications and theories exist which have been created and improved over the last decades. An interested reader might, for example, check *physically based rendering* which is the most popular subject of that area in current days *(M. Pharr, W. Jakob, G. Humphreys, 2016).*

This research, however, particularly focuses on the second part of the integral – incoming radiance of the point. We can separate the integral and only look at the following section:

$$L = \int_{\Omega+} L_{\mathrm{i}}(\boldsymbol{x}, \vec{\omega}_{\mathrm{i}}) \langle \vec{\omega}_{\mathrm{i}} | \vec{\boldsymbol{n}} \rangle d\, \vec{\omega}_{\mathrm{i}} \; .$$

The recursiveness of light bounces inside L_{i} that was mentioned before is exactly what we might intuitively think of as *global illumination.* In general, it can be divided into two groups: *direct* and *indirect* illumination:

$$L = L_{out} + L_{in} \; ,$$

where the first one L_{out} is very straightforward, since it works with a known direction of light and radiance it outputs, and the second one L_{in} , which defines the radiance received from other points, is less trivial to compute and is the main area of our interest. The following sections will focus on the ways to approximate L_{in} with different real-time techniques: reflective shadow mapping, light propagation volumes, voxel cone tracing.

II. Reflective Shadow Mapping

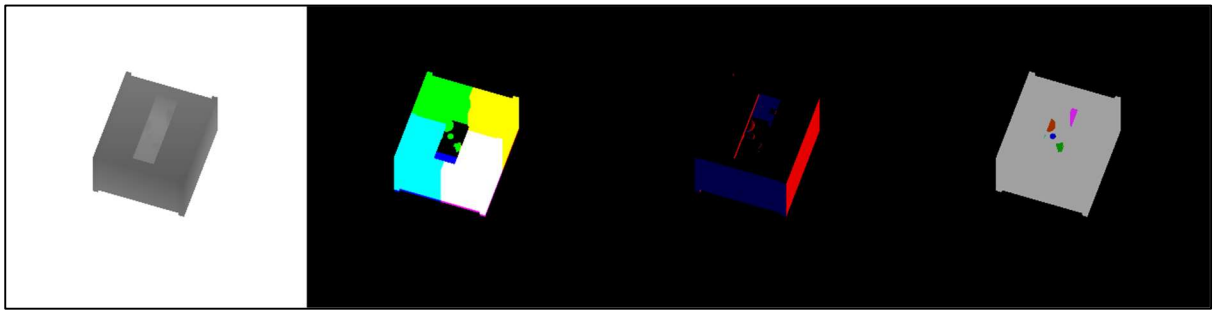


Figure 4. RSM textures of the scene from the directional light view: depth, world positions, normals, flux (in order)

2.1 Theory

The first algorithm that is implemented and analyzed is *reflective shadow mapping (C. Dachsbacher, 2005)*. The name comes from a similar common method of rendering shadows – *shadow mapping (L. Williams, 1978)*. In short, in shadow mapping we generate a shadow depth texture of the scene from the view of our light source and then use it later in lighting passes to shade a particular pixel on the screen. Similarly, *reflective shadow mapping* renders several textures from the light view: *world space positions, normals,* and *flux* which are presented on the screenshot above *(Figure 4)*. However, before going deeper into the reasons and explanations behind their usages, we should mention two essential radiant quantities: *radiant intensity* and *irradiance. Radiant intensity* is the amount of flux Φ (emitted, received, reflected, etc.) per unit solid angle Ω:

$$I_\Omega = \frac{\partial \Phi}{\partial \Omega} .$$

And *irradiance* is the amount of flux Φ received by a surface per unit area A:

$$L_{in} = \mathrm{E} = \frac{\partial \Phi}{\partial A} .$$

Both of these quantities describe flux density and are very useful when approximating the lighting in the scene. If we recall our theory, one of the main things we are interested in when shading a point is how much radiance it receives from the outgoing directions (or in other words, we want to calculate *irradiance*).

RSM (abbreviated for reflective shadow mapping) assumes that every point that the light "sees" from its view can be treated as an infinitely small source of light (VPL or "virtual point light") that affects its' neighboring points and contributes to their radiance. Essentially, we can create a set of textures with data of flux, world positions, and normals visible to the light. The radiant intensity of the point then can be approximated in the following way:

$$I_p(\vec{\omega}) \approx \Phi_p \max\{0, \langle \vec{\boldsymbol{n}}_p | \vec{\omega} \rangle\} ,$$

where p is the point, $\vec{\boldsymbol{n}}_p$ is the normal vector of that point and $\vec{\omega}$ is the emitted direction.

The original paper also suggests calculating the irradiance of a point like this:

$$E_p(x, \vec{n}) = \frac{\max\{0, \langle \overrightarrow{n_p} | (x - x_p) \rangle\} \max\{0, \langle \vec{n} | (x_p - x) \rangle\}}{||x - x_p||^4} ,$$

where x is an arbitrary point and x_p is a particular point in the world space. Then the total irradiance can be approximated as the sum of other pixels' irradiances with N defined as the number of neighboring pixels taken into consideration:

$$L_{in} \approx \mathrm{E}(x, \vec{n}) = \sum_{i=0}^{N} E_p(x, \vec{n}) \, .$$

The visualization of a possible scene environment is presented *(Figure 5)* for a better understanding of the light-rays traversal and the surface points' irradiance contributions to each other (the light source is described as L which is not the same as "radiance" from the previous equations).

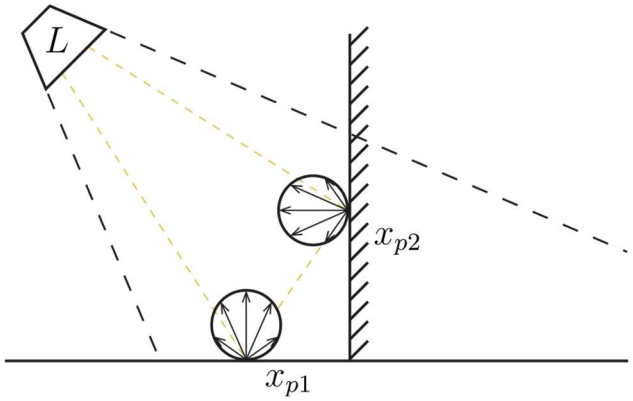


Figure 5. Scene view with points' irradiances

Calculating the sum for every single pixel of the texture for a particular point is quite expensive and does not significantly contribute to the final visual result, which is why we can limit N to a constant number and do N random samples for every point instead. The original paper suggests using polar coordinates for that with random uniformly distributed numbers:

$$p'(x, y) = \left(x + r_{max}\xi_1 \sin(2\pi\xi_2), y + r_{max}\xi_1 \cos(2\pi\xi_2)\right) \, ,$$

where $p'(x, y)$ is the sampling point, r_{max} is the maximum sampling radius, ξ_1 and ξ_2 are random uniformly distributed numbers in the range [0, 1].

These calculations of the final irradiance should be done for every screen space pixel point of the final application's render target. The initial world positions can be gathered from the G-Buffer or reconstructed from the depth buffer.

For a better understanding of the RSM pipeline, the diagram of its original implementation in the demo engine is presented below:

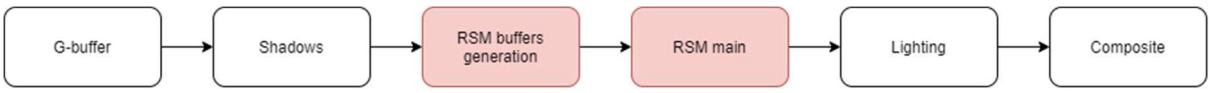


Figure 6. Rendering pipeline with RSM (original implementation); time order

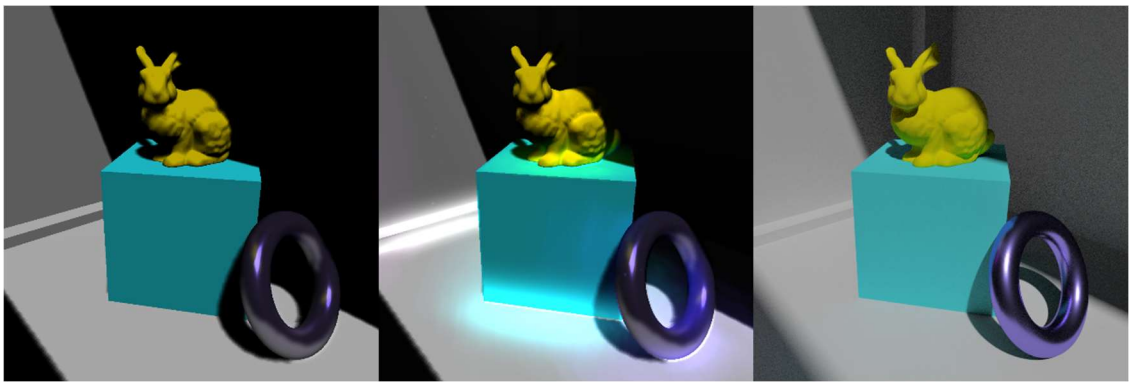


Figure 7. No GI (left), RSM (middle), path-traced (right)

2.2 Shader optimizations

Original RSM algorithm consists of two passes: generation of RSM textures and final indirect diffuse lighting calculation of the scene. The first one is very trivial and can be combined with the shadow mapping pass if it is used in the application (*flux* texture is nothing but a multiplication of the light's color and the object's surface color). The second pass, on the other hand, is trickier to implement optimally and took more creativity. This section focuses on two main shader optimizations that were specifically created for this demo and then gives a performance comparison with an unoptimized version of RSM.

The original technique suggests that N samples should be done for every single point on the screen in order to calculate the indirect global illumination of the scene. This is very expensive at full resolution and the performance of this approach is:

$$O(screenWidth \cdot screenHeight \cdot N)\,,$$

where $screenWidth$ and $screenHeight$ are the dimensions of the final render target (i.e., 1920x1080) and N is the number of samples per point, which is equal to "400" in the original implementation and "512" in this implementation.

Firstly, several lower-level experiments were carried for the optimization of `ReflectiveShadowMappingPS.hlsl` *(Appendix I)* to get rid of any costly ALU operations. One of the most interesting ones was the calculation of the divisor in this formula:

$$E_p(x,\vec{n}) = \frac{\max\{0, \langle \overrightarrow{n_p} | (x - x_p) \rangle\} \max\{0, \langle \vec{n} | (x_p - x) \rangle\}}{||x - x_p||^4}.$$

After the consultation with the mentor, the divisor's calculation was optimized for the calculation of length (square root operation) with dot products which decreased the number of shader assembly instructions on the GPU *(Figures 8, 9)* due to the vectorized architecture of modern graphics cards. This trick is quite original as it can be used in any other shader with similar math operations. An interested reader might find it useful for the scenarios where low-level shader performance is.

```
float3 res = flux * ((max(0.0, dot(vplNormalWS, vplPosDir)) * max(0.0,
dot(normal, -vplPosDir))) / (dot(vplPosDir, vplPosDir) * dot(vplPosDir,
vplPosDir)));

      dp3 r3.w, r7.xyzx, r6.xyzx
      max r3.w, r3.w, l(0)
      mov r7.xyz, -r6.xyzx
      dp3 r4.w, r0.xyzx, r7.xyzx
      max r4.w, r4.w, l(0)
      mul r3.w, r3.w, r4.w
      dp3 r4.w, r6.xyzx, r6.xyzx
      dp3 r5.w, r6.xyzx, r6.xyzx
      mul r4.w, r4.w, r5.w
      div r3.w, r3.w, r4.w
      mul r5.xyz, r3.wwww, r5.xyzx
```

Figure 8. DXBC/DXIL v. 5.0 shader assembly - without pow() function

```
float3 res = flux * ((max(0.0, dot(vplNormalWS, vplPosDir)) * max(0.0,
dot(normal, -vplPosDir))) / (pow(length(vplPosDir), 4)));

      dp3 r3.w, r7.xyzx, r6.xyzx
      max r3.w, r3.w, l(0)
      mov r7.xyz, -r6.xyzx
      dp3 r4.w, r0.xyzx, r7.xyzx
      max r4.w, r4.w, l(0)
      mul r3.w, r3.w, r4.w
      dp3 r4.w, r6.xyzx, r6.xyzx
      sqrt r4.w, r4.w
      mov r5.w, l(1.0000)
      mul r4.w, r4.w, r4.w
      mul r4.w, r4.w, r4.w
      mul r4.w, r4.w, r5.w
      div r3.w, r3.w, r4.w
      mul r5.xyz, r3.wwww, r5.xyzx
```

Figure 9. DXBC/DXIL v. 5.0 shader assembly - with pow() function

However, after testing the shader in the external tool (results are presented in the next sections) it quickly became clear that the performance was highly dependent on the resolutions of the final texture, no matter how many small code changes were added to the main shader. Other methods of optimization had to be investigated and their viability had to be checked.

2.2.1 Downsampling of the indirect illumination texture

Figure 10. RSM's indirect illumination texture (left) and final output texture with direct & indirect lighting (right); RSM values are exaggerated for better visualization.

An assumption was made that a downsampled version of the final GI render target *(Figure 10)* could be calculated instead in the main pass and then upsampled and filtered afterward without losing too much quality in visuals. That concept worked and gave a significant boost to the main pass' shader because the complexity became:

$$O\left(\frac{screenWidth}{k} \cdot \frac{screenHeight}{k} \cdot N\right),$$

where $k = 3$ is the down-sampling factor in the current implementation. Bigger factors produced results that were noticeably low in quality even after the filtering step. The filtering was done in `UpsampleBlurCS.hlsl` *(Appendix I)*. Simple 9x9 Gaussian blur *(A. Sharda, 2021)* was used as a filtering solution and a "groupshared" GPU memory was used for a very efficient temporary storage of the neighboring pixel during the sample *(A. Pranckevičius, 2018)*. After the extra pass was added, the total pipeline of a frame changed in the following way:

Figure 11. Rendering pipeline with RSM's indirect illumination texture upsampling and blur; time order

The performance increased drastically, and the scene started running within a 16ms frame time (~60 frames/second). The chart *(Figure 12)* shows the performance difference of the whole frame's time with and without downsampled indirect illumination texture:

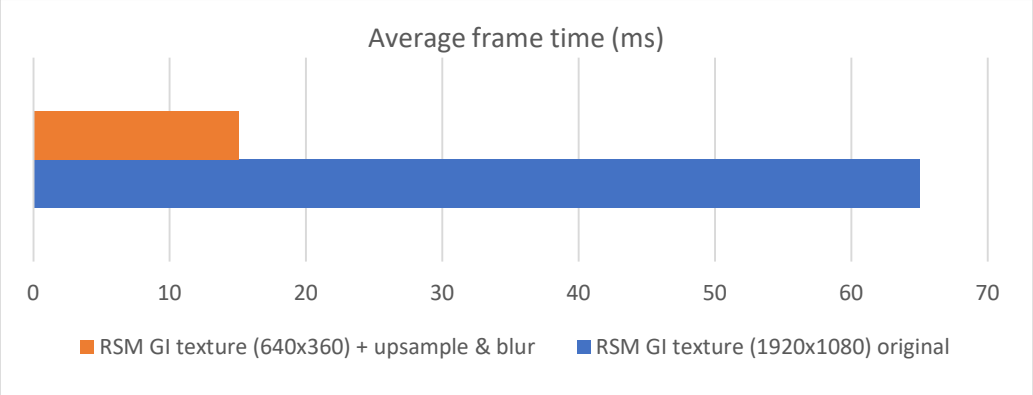


Figure 12. Average time of a full frame with different versions of RSM

The statistics from "NVIDIA Nsight Graphics" of the main calculation pass of RSM are presented below as well *(Figures 13, 14)*. It is noticeable that the number of shaded pixels was reduced drastically which resulted in the improvement of the L2-cache's and the VRAM's maximum throughputs, or, in other words, *SOLs (L. Bavoil 2019)*.

Range Info			
Draw Call Count	1	Dispatch Call Count	0
API Primitives (Total/Avg)	2.0 / 2.0	Threads (Total/Avg)	0.0 / 0.0
Shaded Pixels (Total/Avg)	2073600.0 / 2073600.0	Instructions (Avg Per Dispatch/Avg Per Thread)	0.0 / 0.0
GPU Adapter	NVIDIA GeForce RTX 2060		
Pipeline Overview			Pipeline Diagram
Top SOLs	TEX 77.9% | L2 24.1% | SM 23.1% | VRAM 0.4% | CROP 0.1%		
Graphics/Compute Idle	0	Wait For Idle Count	1
TSL2 Stall Cycles %	0.01	Pixel Shader Barrier Count	0
SM Section			
SM Active	99.06	SM Warp Long Scoreboard	12.66
SM Active Min/Max Delta	0.45	SM Warp Stall Short Scoreboard	0.74
SM Throughput For Active Cycles	23.36	SM Warp Stall Barrier	0
SM Occupancy (Active Warps Per Active Cycle)	91.75	SM Warp Stall Misc	0.02
Memory			
L2 SOL	24.10	Tex Hit Rate	88.38
L2 Hit Rate	99.23		

Figure 13. RSM original main pixel shader statistics from "NVIDIA Nsight Graphics"

Range Info			
Draw Call Count	1	Dispatch Call Count	0
API Primitives (Total/Avg)	2.0 / 2.0	Threads (Total/Avg)	0.0 / 0.0
Shaded Pixels (Total/Avg)	229401.0 / 229401.0	Instructions (Avg Per Dispatch/Avg Per Thread)	0.0 / 0.0
GPU Adapter	NVIDIA GeForce RTX 2060		
Pipeline Overview			Pipeline Diagram
Top SOLs	TEX 68.4% | L2 54.9% | SM 18.9% | VRAM 8.5% | CROP 0.1%		
Graphics/Compute Idle	0	Wait For Idle Count	1
TSL2 Stall Cycles %	0.02	Pixel Shader Barrier Count	0
SM Section			
SM Active	93.70	SM Warp Long Scoreboard	10.45
SM Active Min/Max Delta	15.21	SM Warp Stall Short Scoreboard	0.66
SM Throughput For Active Cycles	20.17	SM Warp Stall Barrier	0
SM Occupancy (Active Warps Per Active Cycle)	77.35	SM Warp Stall Misc	0.02
Memory			
L2 SOL	54.91	Tex Hit Rate	72.85
L2 Hit Rate	96.17		

Figure 14. RSM downsampled main pixel shader statistics from "NVIDIA Nsight Graphics"

As for the efficient upsampling and blur, that pass only added extra **0.46ms** of a frame time but at the same time filtered the downsampled result of the main RSM pass for slightly better visuals by reducing the aliasing of the GI output texture *(Figure 15)*. The upsampling and blur shader is universal and can be used in other similar passes which can be helpful in the engine (i.e., post-processing and other screen space effects).

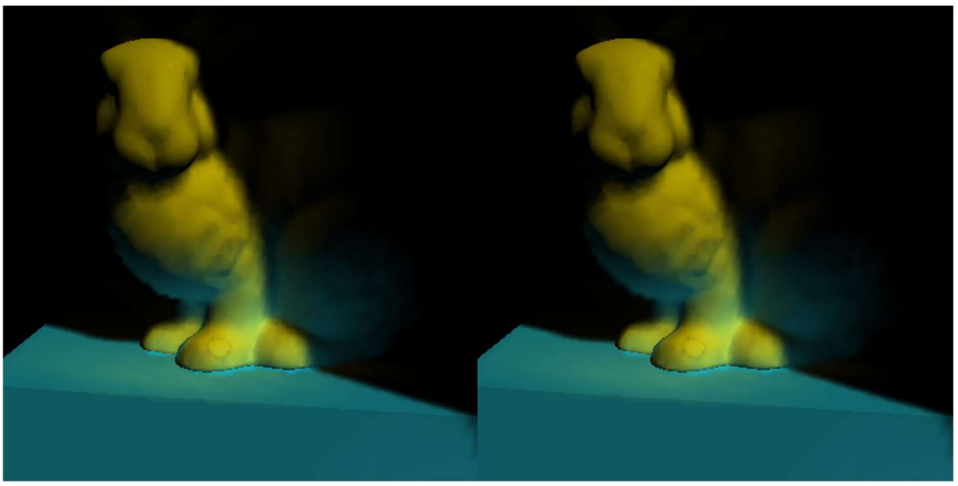


Figure 15. Comparison of the RSM result with upsampling & blur disabled (left) and enabled (right)

2.2.2 Compute shader of the main pass

Another optimization applied to RSM was moving the logic of the second/main pass from the original pixel shader to the compute shader. At the time when the original algorithm was invented, compute shaders did not exist which is why it was interesting to see the performance boost with them. The shader versions of that pass are identical functionally, except the fact that in the compute shader we execute the dispatched threads that are used for every pixel of the down-sampled texture. Due to the heavy logic of the algorithm that could not be changed (such as random sampling of a very big number of neighboring points) the differences between compute and pixel variations were not outstanding, although, on average, the total time was improved in the end *(Figure 16)*.

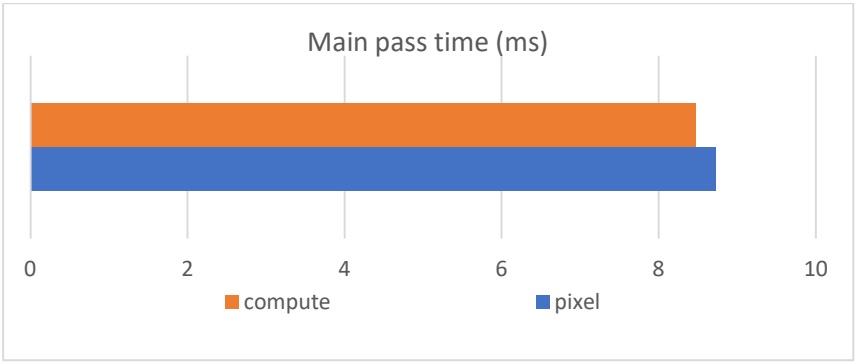
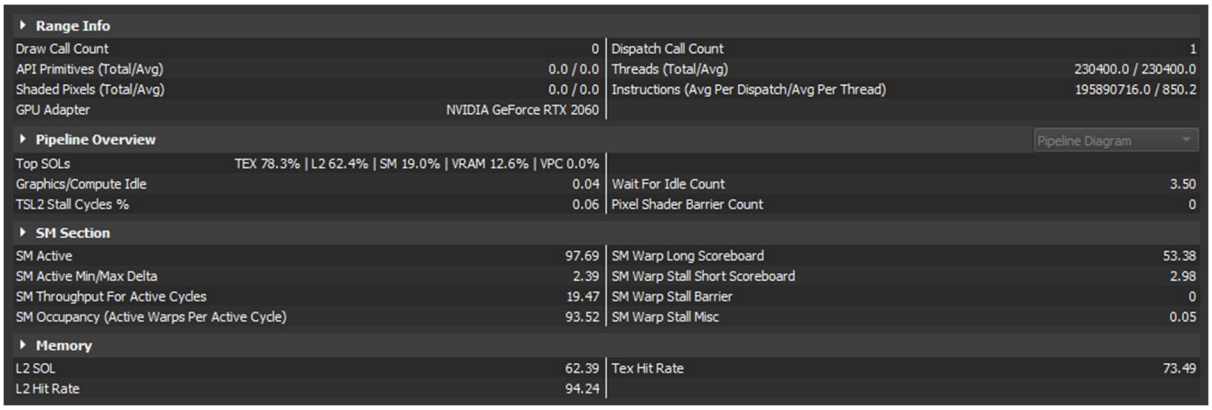


Figure 16. Comparison of compute and pixel shader versions of main RSM pass

After further tests of the optimizations, the algorithm remained very bandwidth-bound (many texture samples), although the planned performance was achieved for this project. The following screenshot

(Figure 17) shows the final statistics of the best-achieved result in the compute shader *(notice that SM's (streaming multiprocessor) maximum throughput is still very poor ~19% due to the heavy logic of sampling)*:



Range Info			
Draw Call Count	0	Dispatch Call Count	1
API Primitives (Total/Avg)	0.0 / 0.0	Threads (Total/Avg)	230400.0 / 230400.0
Shaded Pixels (Total/Avg)	0.0 / 0.0	Instructions (Avg Per Dispatch/Avg Per Thread)	195890716.0 / 850.2
GPU Adapter	NVIDIA GeForce RTX 2060		
Pipeline Overview			Pipeline Diagram
Top SOLs	TEX 78.3% | L2 62.4% | SM 19.0% | VRAM 12.6% | VPC 0.0%		
Graphics/Compute Idle	0.04	Wait For Idle Count	3.50
TSL2 Stall Cycles %	0.06	Pixel Shader Barrier Count	0
SM Section			
SM Active	97.69	SM Warp Long Scoreboard	53.38
SM Active Min/Max Delta	2.39	SM Warp Stall Short Scoreboard	2.98
SM Throughput For Active Cycles	19.47	SM Warp Stall Barrier	0
SM Occupancy (Active Warps Per Active Cycle)	93.52	SM Warp Stall Misc	0.05
Memory			
L2 SOL	62.39	Tex Hit Rate	73.49
L2 Hit Rate	94.24		

Figure 17. Statistics of the compute shader version of main RSM pass from "NVIDIA Nsight Graphics"

2.3 API optimizations – async compute

As we saw from the diagrams above, the RSM algorithm can put a severe hit on the GPU computation units. Additional optimizations in shaders would not be beneficial anymore due to the logic of the algorithm, which is why other alternative methods were researched. For instance, modern APIs, such as DirectX 12, allow the parallelization of different jobs on the GPU with multiple queues (graphics, async, copy) *(G. Thomas, A. Dunn, 2016).* This idea was promising in theory and had to be tested for this project as well. In particular, *asynchronous compute* was implemented in the engine. The goal was to run heavy RSM compute passes, such as the main RSM pass together with its' upsampling & blur, in parallel to other rendering passes, such as shadow mapping and RSM buffers generation.

The original rendering pipeline of a frame was presented earlier and was consecutive. That was changed to a parallel pipeline:

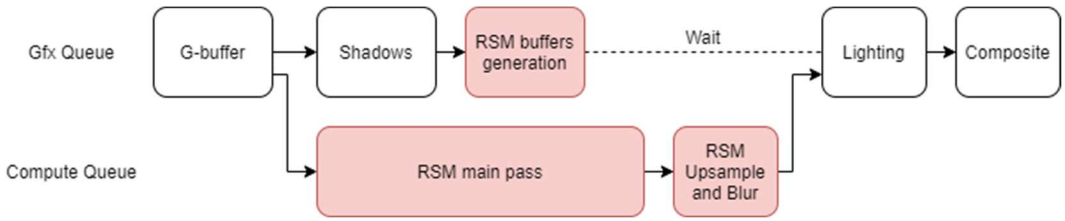


Figure 18. Rendering pipeline with asynchronous RSM; time order

A second queue (compute) was filled with compute-only tasks. As it can be seen from the diagram, the generation of RSM buffers also runs in parallel to the main pass of RSM which uses the result of that generation pass. A race condition occurs in that scenario on those resources – the aforementioned textures are being read from and written to at the same time. This can potentially create many artifacts at random moments which is unwanted in any multithreaded scenario. The problem was solved by working with a previous frame's result of the graphics queue or, in other words, copies of RSM buffers of a previous frame. The overhead was not significant and mainly increased the memory consumption a bit.

The potential performance upgrade, however, was not promising on NVIDIA GPU. The frame time of async and non-async versions stayed the same with some minor deviations. The application was profiled multiple times with "GPU trace" mode of "NVIDIA Nsight Graphics" and the results were the following:

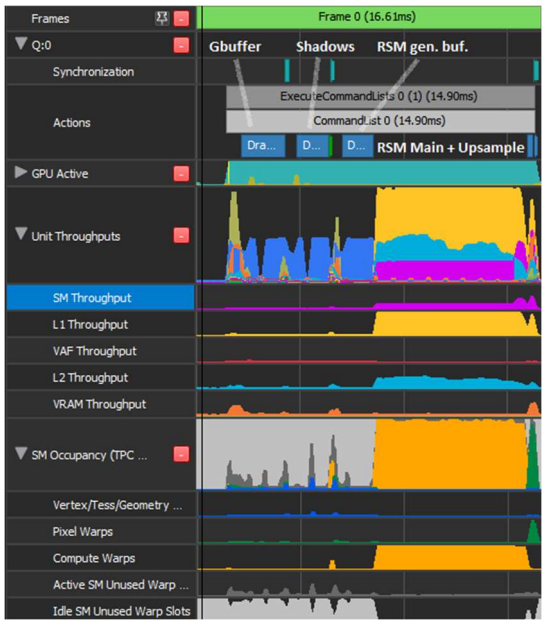


Figure 19. RSM original frame GPU statistics

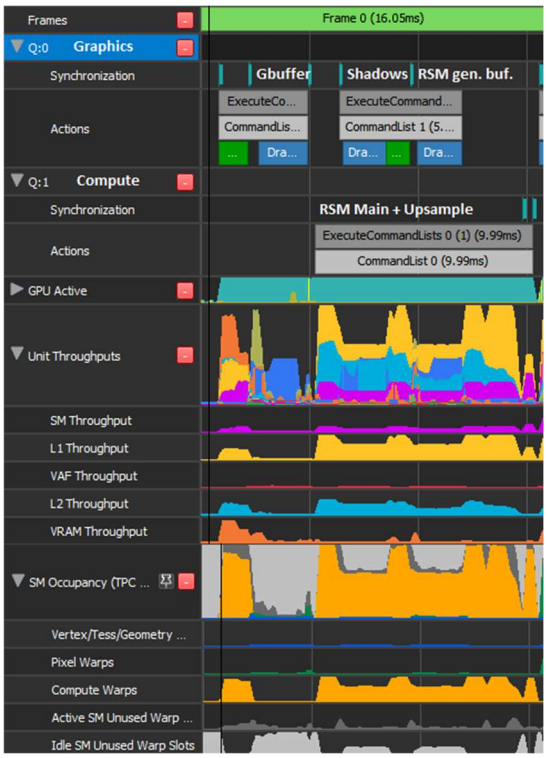


Figure 20. RSM async frame GPU statistics

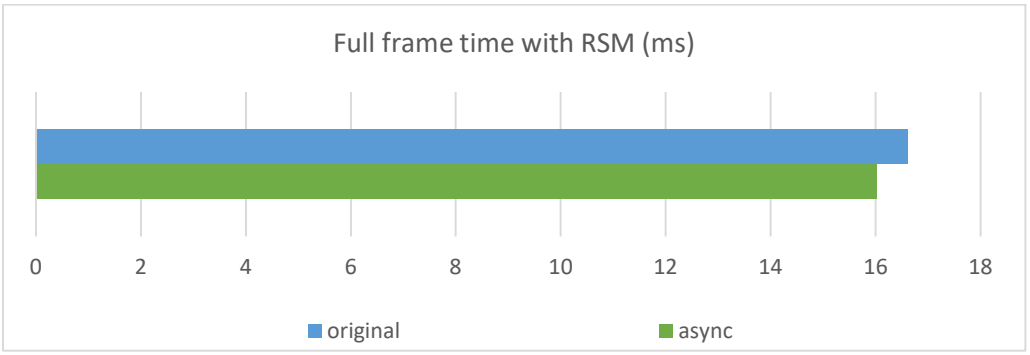


Figure 21. Comparison of frame times with async and non-async pipelines for RSM

As it can be seen from the screenshots above *(Figures 19, 20),* the duration of the frame time in both cases is around **16ms**. RSM main pass and upsampling take roughly **8-9ms** in the original non-async pipeline, whereas the async version executes the same compute tasks in **~10ms**. In some frames, it performs better than that, but on average, the boost is not noticeable. Yes, without a doubt, the compute tasks are being run together with the graphics ones, which is seen from the throughput and occupancy statistics, although the percental usage of the units for the compute task in async is lower than in the non-async version. For instance, SM, L1, and L2 throughputs become evenly distributed across the frame, which is good, but the compute tasks in async mode cannot use the maximum performance of

those units, since they share them with the graphics tasks. On the left image, however, it is seen that the compute tasks, which roughly start from the second half of the frame, utilize the most available throughput without any severe dips or spikes. The same can be said about the compute warps' utilization under the "SM Occupancy" tab.

The main RSM calculation pass still turned out to be very expensive and demanding in the current frame scenario on the tested GPU. The difference between the amount of work for vertex/pixel warps and compute warps together with the lack of any other heavy parallel tasks in the graphics queue might be the reason why the performance gain was not noticeable with an extra asynchronous queue. Another version of the pipeline was tried out of curiosity as well – doing compute tasks in parallel to the G-buffer, shadows, RSM buffers generation, and then rendering lighting with composite passes of a previous frame instead. There were some additional minor performance improvements, however, the visual results were unacceptable due to the G-buffer being generated for a previous frame and used in the current frame. In the end, it was proven with this experiment that one frame camera lag was more noticeable for the user than one frame light lag.

On the other hand, if we supply the graphics queue with some substantial work during the parallel compute passes of the RSM in the compute queue, the performance gain will be way more noticeable than in a non-async scenario. It was a concept that was tested with the technique from the next chapter, *Light Propagation Volume (Figure 22)*. The total frame's performance turned out to be **~5ms** better on some occasions than the consecutive implementation of RSM with LPV.

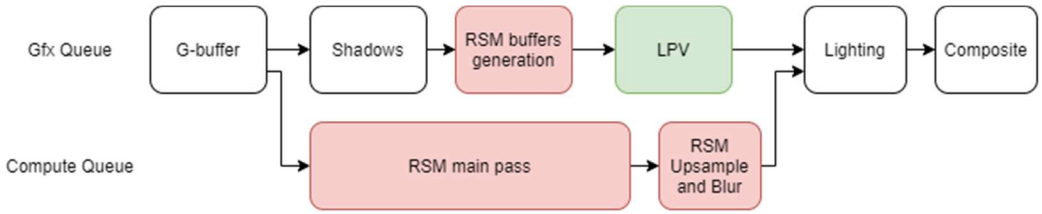


Figure 22. Rendering pipeline with async RSM; LPV is used to fill the stalls of the graphics queue.

2.4 Conclusions

The custom version of RSM from this project gives us a usable and optimized version for the production approximation of the indirect global illumination, although the technique itself has some considerate downsides.

Firstly, it only allows a one-bounce indirect diffuse without any effects such as reflections or ambient occlusion. This is very limiting and also not completely physically accurate due to the light leaks (the dynamic light source is also added for spotting them and other artifacts from 5.2). The potential production costs in a system with RSM will be spent on other additional solutions (for reflections, AO) and tweaks by the artists (this demo also features various sliders for the configuration that can help).

In addition, RSM requires several high-resolution RSM textures which should be loaded in the GPU memory that can result in a memory bottleneck on some low-end devices and can also be a reason for a poor bandwidth. Besides that, the main drawback is the sampling of those textures. As it could be conveyed from 2.1, the main shader pass of the algorithm is still very heavy even on a modern GPU, such as NVIDIA RTX2060. Unfortunately, additional optimizations from 2.2, 2.3 do not improve the situation significantly. Without a doubt, the number of samples can be reduced but with the loss of quality and with the appearance of extra graphical artifacts like shimmering. Modern AAA games are unlikely to be "satisfied" with that level of quality of the final GI, although mobile or handheld games might use it in some scenarios (i.e., as a fallback to another GI). The additional visual and technical comparisons with other techniques will be provided in the final sections of this research (5.1, 5.2, 5.3).

III. Light Propagation Volumes

3.1 Theory

The idea of the original algorithm is still based on the concept of virtual point lights of the scene that are stored in the *flux* texture after the RSM buffer generation pass. However, the VPLs are then *injected* into a uniform volumetric grid that covers the scene where each cell represents a surface element with position, flux, and normals encoded with spherical harmonics *(M. Seymour, 2013)*. This allows us to have the geometry of the scene subdivided into n^3 cells with each cell having a total irradiance of its' neighboring cells. The contribution of the irradiance can be collected or *propagated* iteratively from different layers of neighbors and then accumulated in one set of 3 (red, green, blue) 3D textures that contain the total contribution encoded with spherical harmonics which are then sampled when calculating the final outcoming radiance of the screen space point:

$$L_{in} \approx \mathrm{E}_{\mathrm{x}\rightarrow\mathrm{cell}} = \sum_{\mathrm{i}=0}^{\mathrm{P}} \sum_{\mathrm{j}=0}^{\mathrm{M}} \mathrm{E}_{\mathrm{cell_j}} \, ,$$

where M is the number of neighbors (6) of the cell and P is the number of iterative propagations (arbitrary number). Below is the visualization of the propagation of a simplified uniform grid:

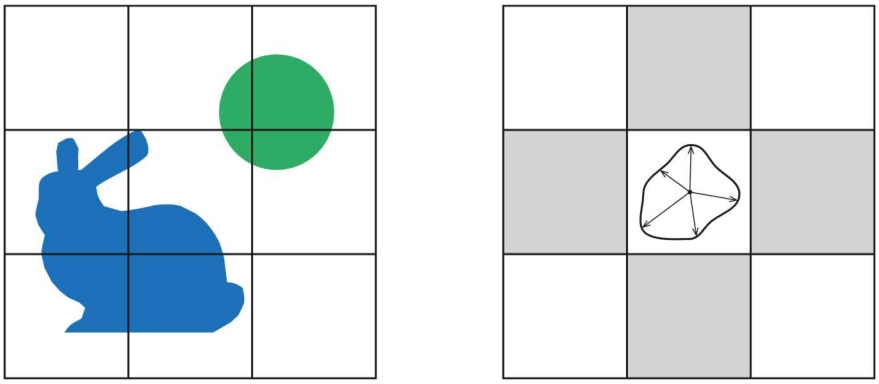


Figure 23. Scene's division into cells (left); scene's propagation of the neighbor cells (right)

Due to the time constraints of the project, additional details about the injection and the propagation passes together with the explanation behind why and how spherical harmonics are used in LPV for calculating E_{cell_j} are omitted, as the subject is quite big to cover in the scope of this paper. Instead, an interested reader might look into the final shaders `LPVInjection.hlsl`, `LPVPropagation.hlsl` *(Appendix I)* and some resources that are dedicated to the topic of spherical harmonics in computer graphics *(S. Green, 2003)*.

For a better understanding of the LPV pipeline for the next sections of this chapter, the diagram of LPV original implementation in the demo engine is presented below:

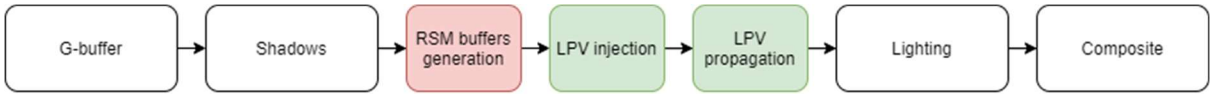


Figure 24. Rendering pipeline with the original LPV passes; time order

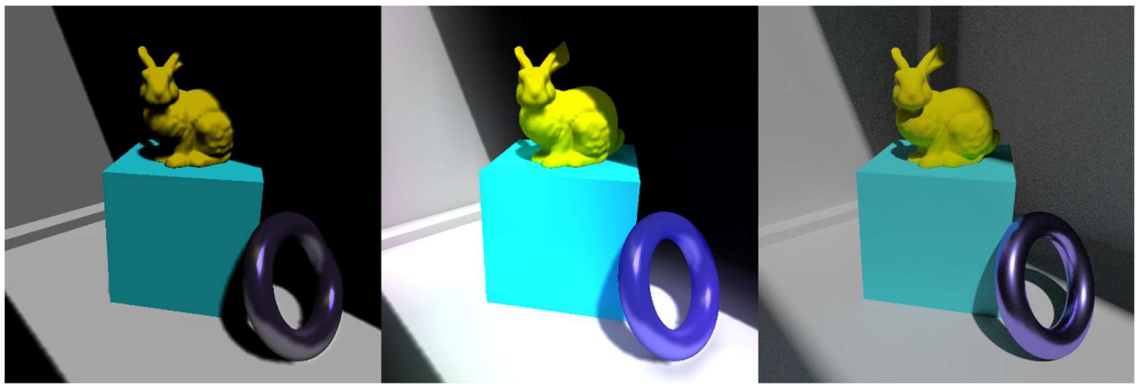


Figure 25. No GI (left), LPV (middle), path-traced (right)

3.2 Shader optimizations – downsampling of RSM buffers

As mentioned before, for gathering the radiance of surface elements the scene has to be approximated in a grid of cells. The amount of these cells is dependent on the resolutions of the provided RSM textures (flux/normal/world positions) - the higher the resolution, the higher the costs of the injection and propagation passes. The original RSM texture set from the previous section of this project was generated in 2048x2048 resolution which turned out to be quite expensive for generating a grid of 2048^3 surface elements. For this reason, a method to lower down the resolution had to be researched and applied.

Downsampling is the go-to way for such problems which is why it was used without a problem for this scenario as well. A custom `RSMDownsamplePS.hlsl` shader was written which calculates the average luminance of the RSM texels and then stores the neighbors' data of the texel in a cache for better access to them during the downsample. The pipeline of a frame changed to:

Figure 26. Rendering pipeline with RSM buffers downsample + LPV; time order

In general, the visual quality of the scene was acceptable with downsampling, and the performance gain was very evident. The computational complexity of the injection pass became:

$$O\left(\frac{rsmWidth}{k} \cdot \frac{rsmHeight}{k}\right),$$

where $k = 4$ is the downsampling factor of the RSM buffers in the current implementation and $rsmWidth$, $rsmHeight$ are the original dimensions of the RSM buffer. That factor was sufficient for the textures of the size $rsmWidth = rsmHeight = 2048$. The shader of the injection pass, which, in fact, generates the geometry of the point cloud of the scene from those RSM buffers, started running way faster after this optimization. In addition, the compute version of the shader was written to save a bit more on performance in the downsampling shader with the help of the dispatched threads. It did not change the visuals but improved the calculation times of the injection pass. The injection pass' performance and the downsampling pass' performance are presented in the following charts:

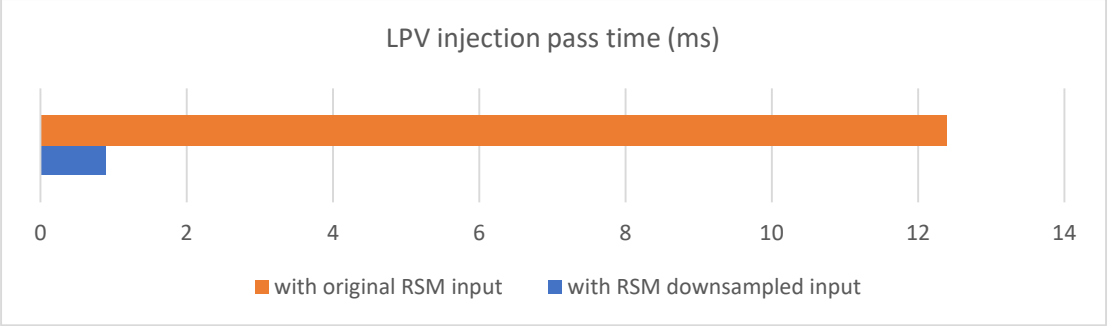


Figure 27. Performance time of the LPV injection pass with the downsampling of RSM buffers

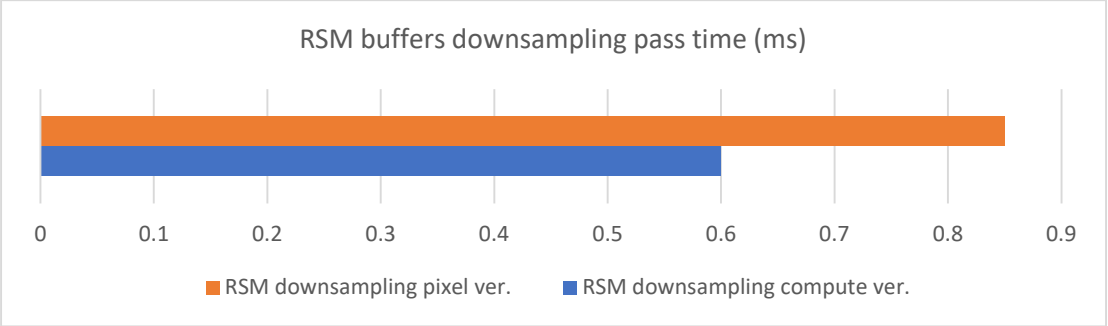


Figure 28. Performance time of the RSM buffers downsampling pass in pixel and compute shaders

Fortunately, the visual differences were not as noticeable as the performance of the final LPV. The algorithm produced a decent version of the indirect diffuse even with the downsampled input textures:

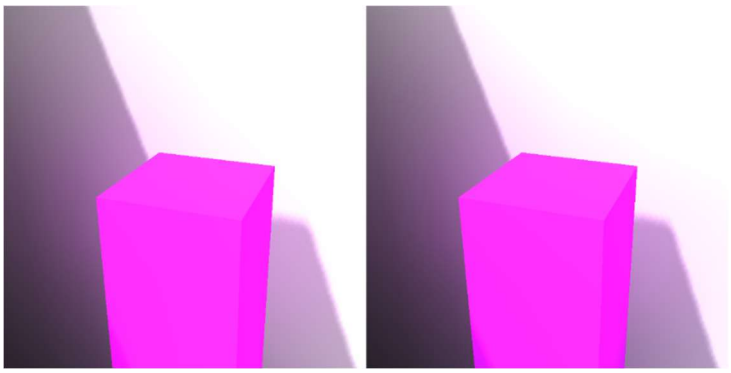


Figure 29. LPV result without downsampled input (left) and with downsampled input (right)

Due to the simplicity of the original shader of the propagation pass (~**0.03ms**), the GPU optimizations of it were not researched. Potentially, it is possible to move the logic from the pixel to the compute shader in the future. For extra performance, the number of propagation passes should be tweaked carefully which is why the application provides a slider for that via the options menu.

3.3 API optimizations - bundles

CPU submission of commands that have to be run on a GPU might be quite expensive *(G. Thomas, A. Dunn, 2016).* In the case of our LPV technique, propagation passes are separate, similar, and small draw-call commands that accumulate to the final set of 3D textures. The number of these commands might vary from 1 to 100 in the current demo with 50 being a default number. A way to group the submission of those commands was researched which also had to fit in the project's time for LPV so that no big code changes would be needed.

After reading the official documentation of DirectX 12 and some publications on how to optimize the submission on a CPU, one DirectX 12 feature came in handy – *bundles (G. Thomas, A. Dunn, 2016)*. In short, a bundle is a group of commands that are saved once and then re-executed when needed. A bundle will "reference" some useful information bound to it, such as buffers, and execute itself with an updated version of that data. This trick suited nicely for propagation passes and saved a workload on the CPU – dozens of commands (if bundles are on) are no longer re-added over and over again every frame *(Figures 30, 31).* The ability to turn the bundles on and off was also added to the cheat menu of the demo for testing purposes by the users.

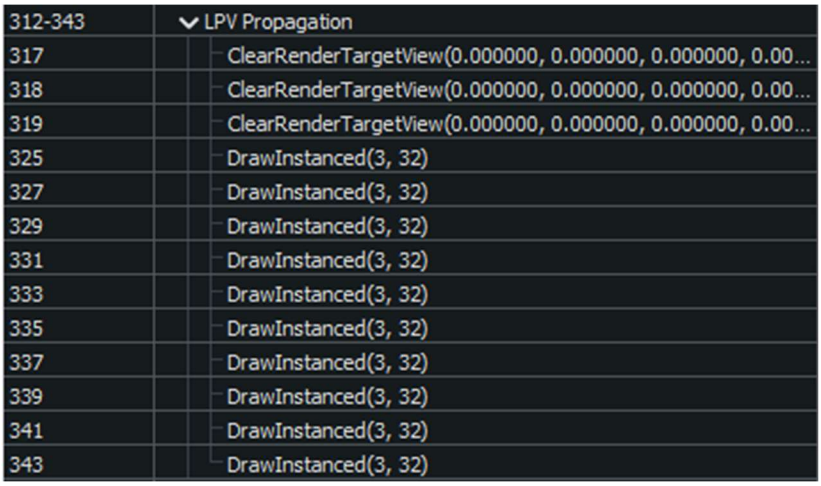


Figure 30. LPV propagations (10 steps) with original submission (Screenshot from "RenderDoc")

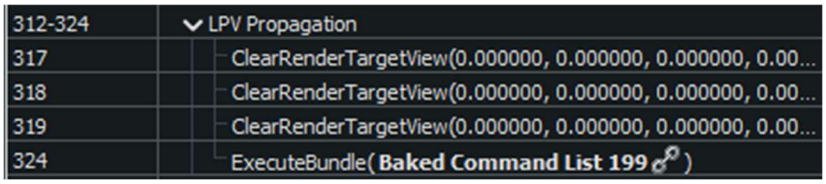


Figure 31. LPV propagations with DX12 bundle submission (Screenshot from "RenderDoc")

Since the usage of bundles is simply a CPU optimization, the performance difference on the tested CPU (Ryzen 3550H) with max. 100 propagation steps was not visible with the bundles. Yes, the C++ loop for propagations was not running inside the function that was rendering LPV, however, that number of loop iterations was very low and easy to handle on this CPU from the beginning. An additional "hardcore" test was made with 5000 propagation steps and the CPU performance difference of **5%** was achieved then. It is also possible that on lower CPU/GPU configurations the results could be better.

3.4 Conclusions

Overall, LPV proves to be a great technique for partial global illumination. The current version is relatively not expensive on modern GPU hardware and provides acceptable visual results for one bounce indirect diffuse. Due to the logic of spherical harmonics (the number of bands that is limited to 2 or 3 for real-time purposes), only low-frequency illumination is possible (i.e., diffuse) which makes indirect specular reflections unfeasible with the original implementation since they need the reconstruction of higher-frequency signal (more details). This is why, again, production costs might be spent on other solutions for the reflections.

RSM textures are needed for the preparation of the volume data during the injection phase. A way to compress these RSM textures in the future was suggested by the mentor and brought to the discussion at some point during the project: some textures, like a normal map, can be encoded in a smarter way, such as *"octahedron normal encoding" (K. Narkowicz, 2014),* which would save overall memory impact and bandwidth.

Finally, many uniform cells of the grid are empty and do not have any information which is often very wasteful for the propagation computation. A good experiment might be to research and change the original spatial structure with something more efficient, such as an octree *(J. D. Olovsson, M. Doggett, 2013).*

This demo features a version where propagation steps can be changed by the user together with some toggles for optimizations, such as downsampling of RSM or bundles. Some precision parameters for LPV are also added for better estimation of the needed visuals (LPV's "cutoff", intensity) that can be used in any other application.

IV. Voxel Cone Tracing

4.1 Theory

The last technique, *voxel cone tracing* (or VCT), of this research shares one logical similarity with LPV: the scene is presented in a set of volumetric structures – *voxels*. The voxels are more granular and smaller in size than the cells from LPV and they do not carry any information except the position and the current radiance of an element (direct lighting contribution, for example). Radiance of the geometry's surface is injected into a voxel stored in a 3D texture which is filtered/mipmapped *(B. Anuworakarn, 2019)* and processed for the raymarching *(S. Ashbery, 2019)*. Every world position in the screen space has contributions of different radiances around a hemisphere of that point which is approximated by several equal in size cones. Finally, for each cone, a set of ray-marching steps is applied that collect voxels on their way with different mipmap levels (LODs). The information about color and occlusion is accumulated in the front-to-back order during the tracing.

In addition, a similar trace of one extra cone can be used for some indirect specular reflections. The calculation of them is physically accurate and close to the real-world scenario, no artifacts are present on the contrary to the *screen space reflections* mentioned before (i.e., no missing objects in the reflections).

The final result of the total irradiance of the point can be calculated in the following way:

$$L_{in} \approx \sum_{\mathrm{i}=0}^{\mathrm{C}} \sum_{\mathrm{j}=0}^{\mathrm{S}} L_{\mathrm{dif.+ao}} + \sum_{\mathrm{j}=0}^{\mathrm{S}} L_{\mathrm{spec.}} \,,$$

where S is the amount of tracing steps per cone and C is the number of cones. (6 cones of 60° each are used in this project). Below are the diagrams of the cone tracing process and the outgoing radiance from cones around the point:

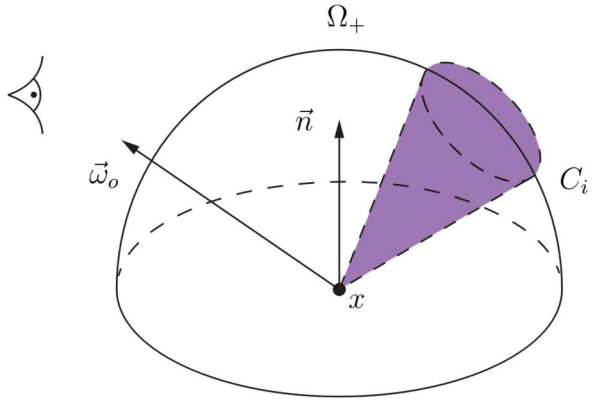


Figure 32. Approximated cone in the hemisphere

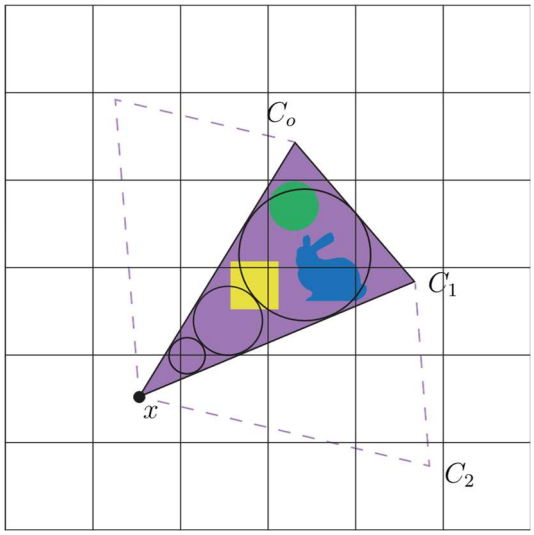


Figure 33. Cones in the voxelized scene

The rendering pipeline *(Figure 34)* is similar to the previous methods, although, it is completely independent of other GI solutions (i.e., LPV had to use RSM's buffers).

Figure 34. Rendering pipeline with the original VCT passes; time order

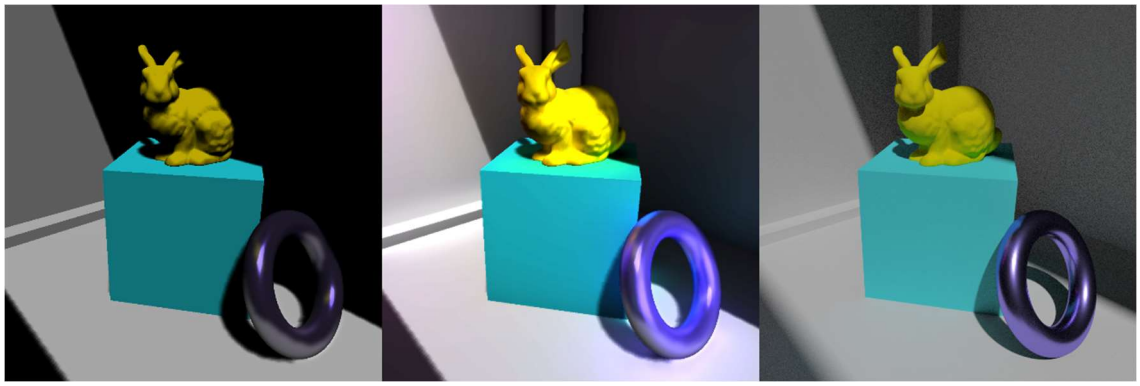


Figure 35. No GI (left), VCT (middle), path-traced (right)

4.2 Shader optimizations

VCT consists of three main passes, such as *voxelization* of the scene, *filtering* of the voxelized texture, and *cone tracing*.

The logic of the voxelization in `VoxelConeTracingVoxelization.hlsl` *(Appendix I)* can be improved greatly, but was not due to the time constraints, since better algorithms exist *(L. Zhang, W. Chen, D. S. Ebert, Q. Peng, 2007)* and the one which is implemented for this project is very naïve and suboptimal. In addition, as we voxelize the scene in a "brute-force" way, many empty spaces may end up being voxelized as well. They take extra memory on the GPU but do not contain any useful information for the final GI computation. For bigger scenes, solutions, such as an octree packing of the voxels *(C. Crassin, F. Neyret, M. Sainz, S. Green, E. Eisemann, 2011),* might be "tighter" and more efficient.

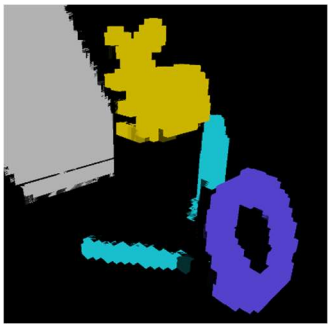


Figure 36. Current voxelization of the scene:
not "conservative", has empty voxels, and injects simplified radiance (albedo+shadows).

As for the filtering, DirectX 12 cannot perform any in-build mipmapping of the texture resource which is why a custom set of compute shaders that are based on the original paper had to be written for this project. These passes turned out to be very cheap and, thus, were not optimized.

Lastly, the main cone tracing pass is the heaviest part of the algorithm and was optimized in the project (4.2.1). It is quite demanding for GPU resources due to the logic which is similar to ray-marching – for every pixel of the screen virtual spheres are marched within a cone(s) in several steps as was shown in *Figure 33*. The computational shader complexity for a full-screen pass of an unoptimized version of this GI technique is the following:

$$O\big(screenWidth \cdot screenHeight \cdot S \cdot (C+1)\big),$$

where 1 is used for the extra specular cone.

4.2.1 Downsampling of the indirect illumination texture

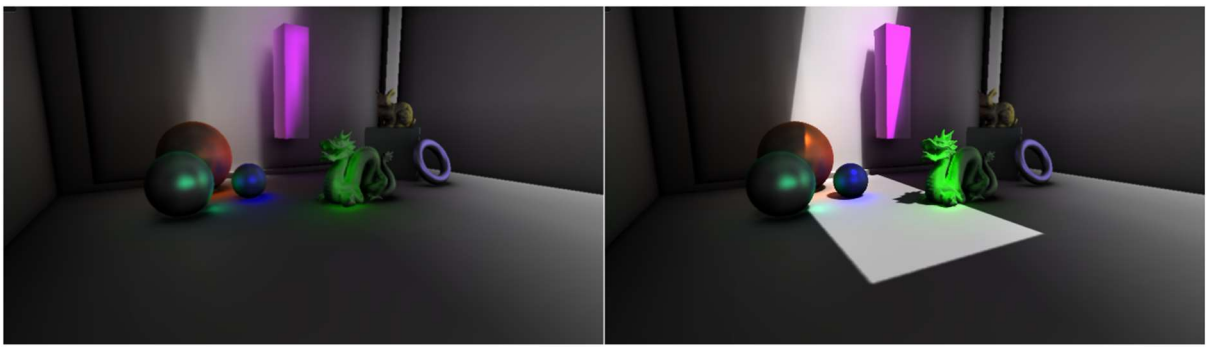


Figure 37. VCT's indirect illumination texture (left) and final output texture with direct & indirect lighting (right); VCT values are exaggerated for better visualization.

As it could be seen from the time complexity of the algorithm, the cone tracing part is dependent on the resolution of the screen. The same assumption from 2.2.1 was made that a downsampled version of the final VCT texture can be calculated instead and then upsampled and filtered afterward (`UpsampleBlurCS.hlsl` shader was again used as an upsampling/blurring solution). It worked well and gave a nice performance boost to the whole frame time as will be seen further in this section. The final version `VoxelConeTracingPS.hlsl` can be checked in *Appendix I*. As for the total pipeline of a frame, it was modified in the following way:

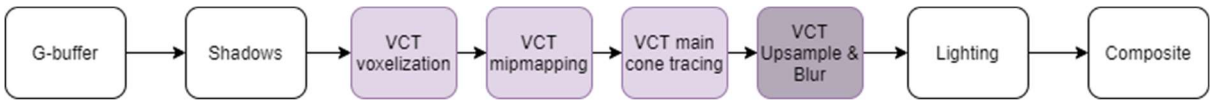


Figure 38. Rendering pipeline with VCT's indirect illumination texture upsampling and blur; time order

In the case of VCT, a downsampling factor of $k = 2$ gave the best visual results under very acceptable frame rates. The complexity has changed to:

$$O\left(\frac{screenWidth}{k} \cdot \frac{screenHeight}{k} \cdot S \cdot (C+1)\right).$$

In theory, it is also possible to run 2 separate cone tracing passes for indirect diffuse and indirect specular and use different downsampling factors and different texture resolutions for them. This approach can be nice to experiment with in the future if extra performance is needed.

The visual comparison of the final output with the original resolution (1920x1080) and with the downsampled version of it (960x540) can be seen on the image below:

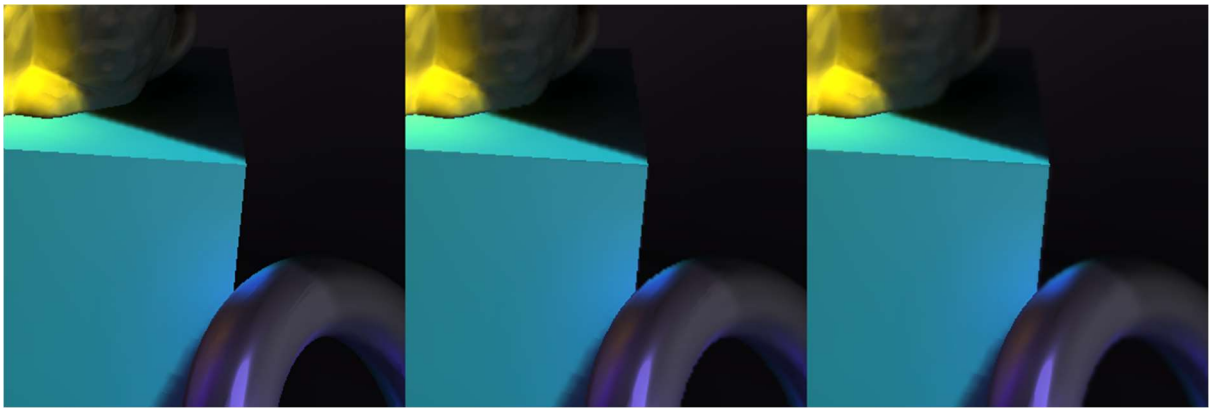


Figure 39. VCT result original (left), downsampled (middle), downsampled + upsampled + blurred (right)

The visual difference of the images above can be difficult to notice from the first glance, however, the middle image does indeed look more aliased on the borders of the objects where GI is present (i.e., on the upper parts of the torus) or in the zones which are in the shadow and only have some indirect diffuse illumination. Upsampling/blur helps to smooth out these cases for a very cheap cost (same **~0.46ms** as with RSM).

In the end, the performance improvement of this optimization, as expected, turns out to be proportional to the downsampling factor *(Figure 40)*.

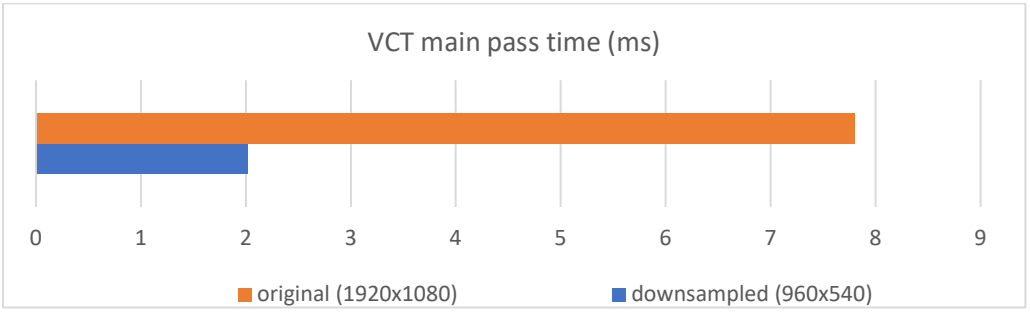
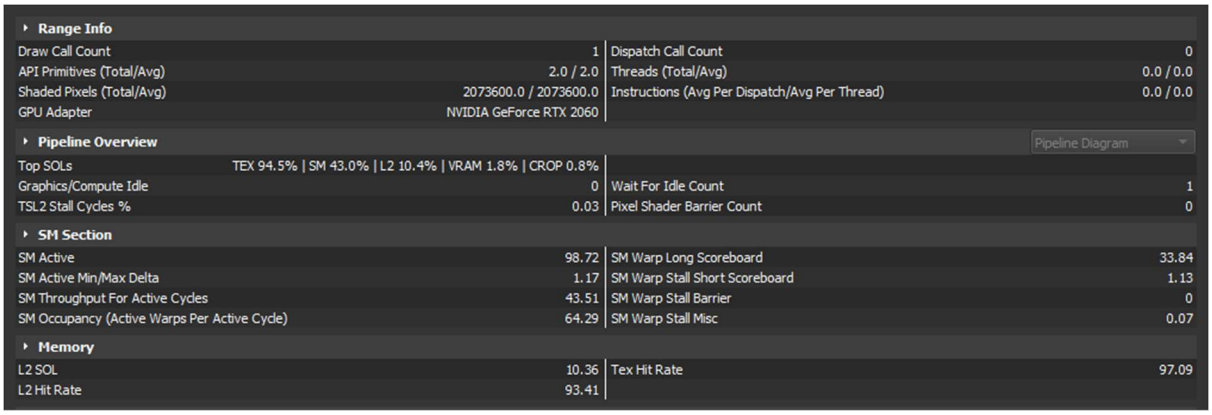


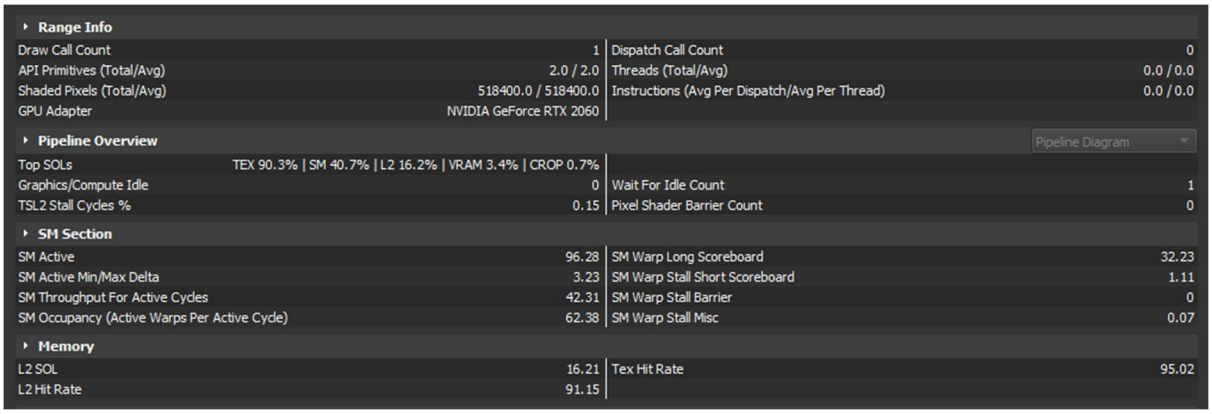
Figure 40. Shader execution times of the original main pass and the downsampled main pass

If we profile this event in "NVIDIA Nsight Graphics" in the original *(Figure 41)* and the downsampled *(Figure 42)* versions, we can notice that in the downsampled version the GPU's L2 cache's throughput increases from ~10% to ~16%, and the VRAM's throughput increases from ~1.8% to ~3.4%. As it can be seen from the statistics screenshots below, the cone tracing pass is still ALU-heavy (SM throughput is not high and only around 40% in both cases), although the amount of active SM units is very close to the maximum (90+ percent).



Range Info			
Draw Call Count	1	Dispatch Call Count	0
API Primitives (Total/Avg)	2.0 / 2.0	Threads (Total/Avg)	0.0 / 0.0
Shaded Pixels (Total/Avg)	2073600.0 / 2073600.0	Instructions (Avg Per Dispatch/Avg Per Thread)	0.0 / 0.0
GPU Adapter	NVIDIA GeForce RTX 2060		
Pipeline Overview			Pipeline Diagram
Top SOLs	TEX 94.5% | SM 43.0% | L2 10.4% | VRAM 1.8% | CROP 0.8%		
Graphics/Compute Idle	0	Wait For Idle Count	1
TSL2 Stall Cycles %	0.03	Pixel Shader Barrier Count	0
SM Section			
SM Active	98.72	SM Warp Long Scoreboard	33.84
SM Active Min/Max Delta	1.17	SM Warp Stall Short Scoreboard	1.13
SM Throughput For Active Cycles	43.51	SM Warp Stall Barrier	0
SM Occupancy (Active Warps Per Active Cycle)	64.29	SM Warp Stall Misc	0.07
Memory			
L2 SOL	10.36	Tex Hit Rate	97.09
L2 Hit Rate	93.41		

Figure 41. VCT's (original resolution) main pass statistics from "NVIDIA Nsight Graphics"



Range Info			
Draw Call Count	1	Dispatch Call Count	0
API Primitives (Total/Avg)	2.0 / 2.0	Threads (Total/Avg)	0.0 / 0.0
Shaded Pixels (Total/Avg)	518400.0 / 518400.0	Instructions (Avg Per Dispatch/Avg Per Thread)	0.0 / 0.0
GPU Adapter	NVIDIA GeForce RTX 2060		
Pipeline Overview			Pipeline Diagram
Top SOLs	TEX 90.3% | SM 40.7% | L2 16.2% | VRAM 3.4% | CROP 0.7%		
Graphics/Compute Idle	0	Wait For Idle Count	1
TSL2 Stall Cycles %	0.15	Pixel Shader Barrier Count	0
SM Section			
SM Active	96.28	SM Warp Long Scoreboard	32.23
SM Active Min/Max Delta	3.23	SM Warp Stall Short Scoreboard	1.11
SM Throughput For Active Cycles	42.31	SM Warp Stall Barrier	0
SM Occupancy (Active Warps Per Active Cycle)	62.38	SM Warp Stall Misc	0.07
Memory			
L2 SOL	16.21	Tex Hit Rate	95.02
L2 Hit Rate	91.15		

Figure 42. VCT's (downsampled resolution) main pass statistics from "NVIDIA Nsight Graphics"

4.3 API optimizations – async compute

Every pass in this technique except the voxelization is suitable for the execution in a compute shader version and was written in that way for the project. This fact made the current implementation possible to be transferred to a separate compute queue for parallel execution on a GPU.

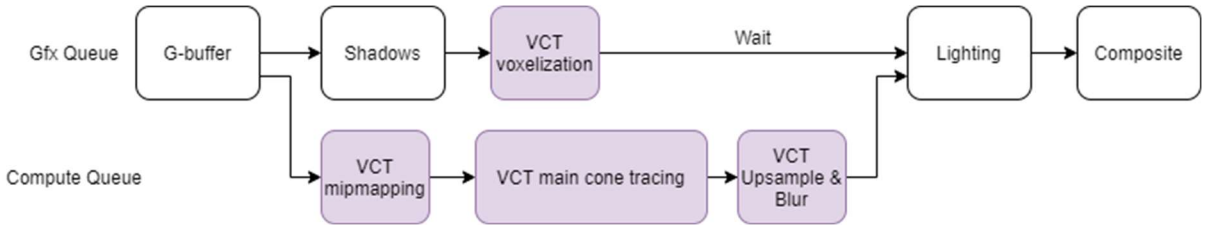


Figure 43. Rendering pipeline with VCT in the asynchronous queue; time order

The pipeline of VCT was modified in a custom way that was different from the original implementation *(Figure 43)*. In contrary to the RSM's case with the very expensive main pass, VCT has way cheaper passes in the async queue which in theory can make it more efficient for async and less detached from the performance difference of two queues. Since the performance of a non-async version after 4.2.2 was already high in the project's demo scene (all VCT passes except voxelization only took **~3.4ms**), the difference with async had to be checked again in "NVIDIA Nsight Graphics" with "GPU Trace" mode *(Figures 44, 45)*.

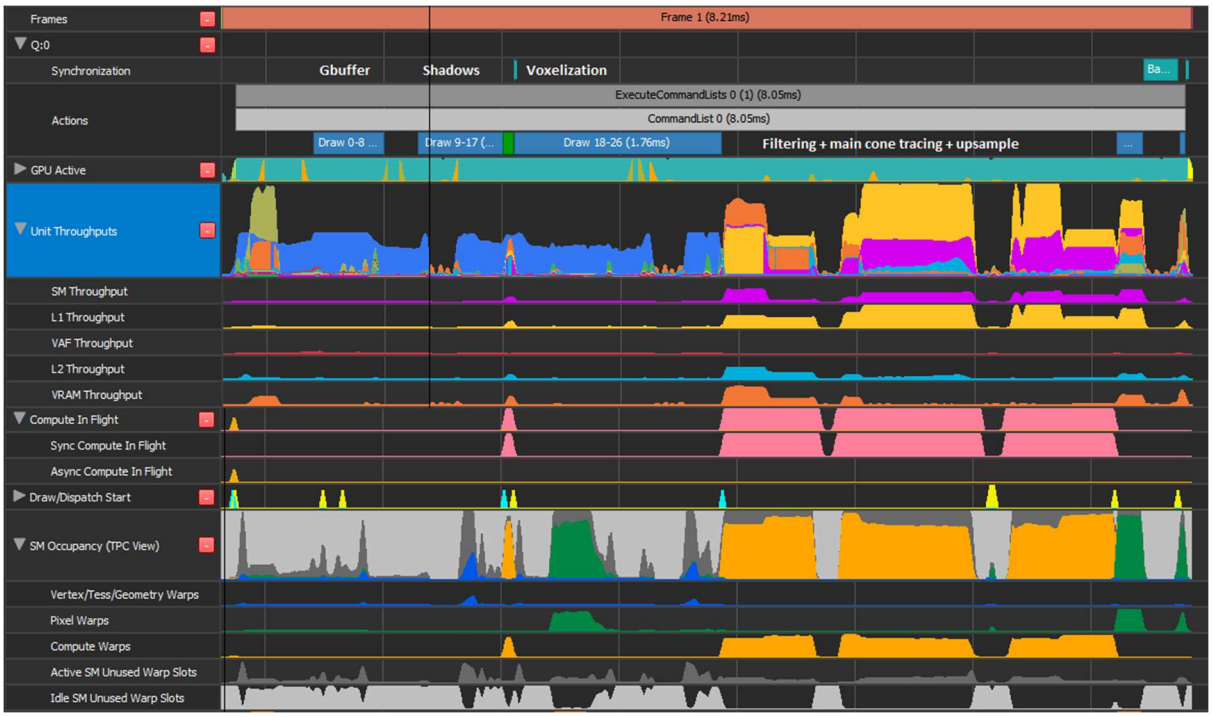


Figure 44. VCT original frame GPU statistics (synchronous pipeline)

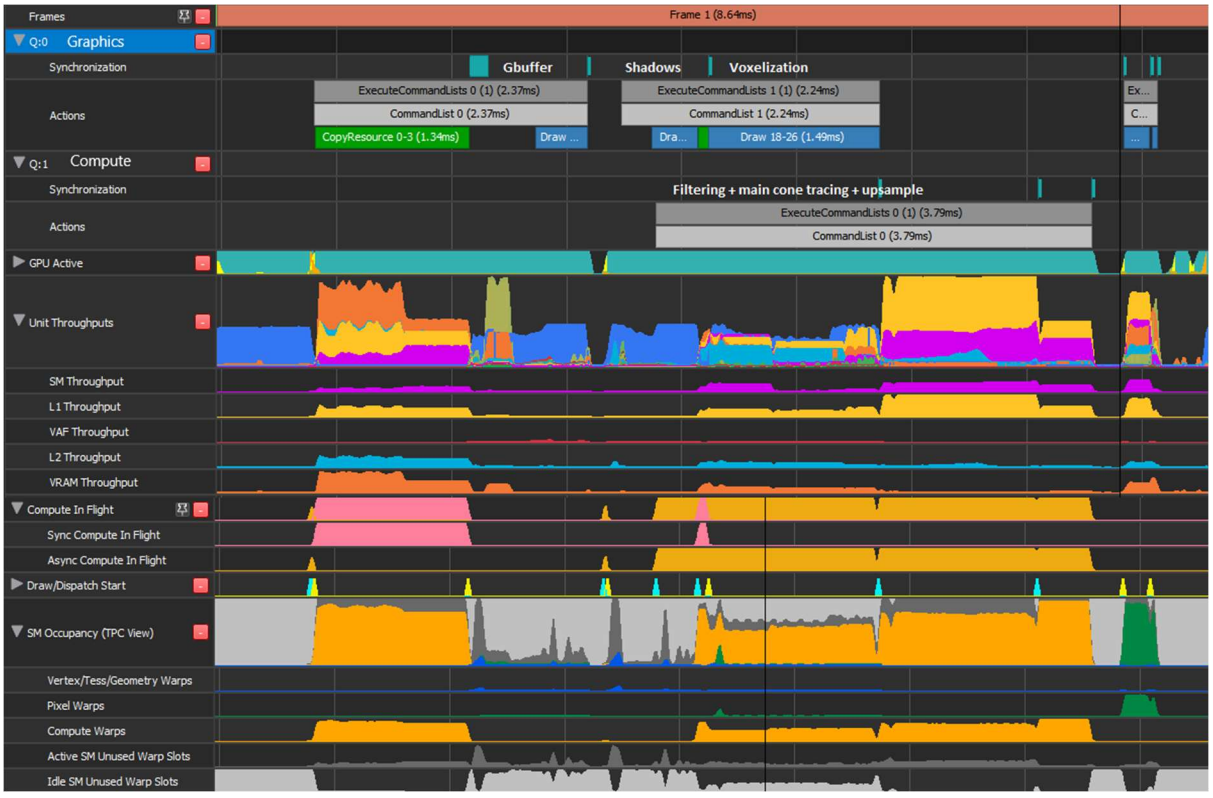


Figure 45. VCT parallel frame GPU statistics (asynchronous pipeline)

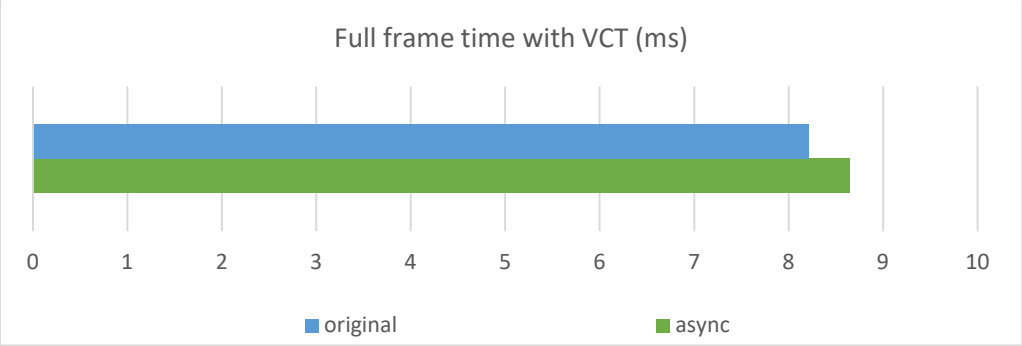


Figure 46. Comparison of frame times with async and non-async pipelines for VCT

Unfortunately, after multiple tests, the performance difference was not present *(Figure 46)*. Moreover, the original version was running slightly better in some frames. As it can be seen *(Figure 45)* the time required to execute VCT compute passes in the async mode is ~**3.79ms** (vs ~3.4ms in the original). Similar to the case with RSM, the pipelines do differ in the distribution of work and the throughputs of various GPU elements. For instance, in the original synchronous pipeline, SM and L2 throughputs are higher with shorter duration time during VCT compute tasks, whereas in the async they take more time and are less active. It is also clearly noticeable that the efficiency of vertex and pixel warps during VCT is low due to the low workload in the graphics queue (RSM suffered from the same issue because the same shadow pass was "easy-to-handle" on the tested GPU in comparison to the type and the amount of work in the async compute queue). On the other hand, the passes in the graphics queue, such as voxelization and shadow mapping, gained some performance with the help of async: from ~**1.76ms** to ~**1.49ms** for voxelization and from ~**0.73ms** to ~**0.4ms** for shadow mapping.

Overall, our results with async for both RSM and VCT could not drastically outperform their synchronous versions and in some scenarios, they were similar or even slightly lower in performance on the tested GPU. One may assume that the tested GPU architecture in this project (NVIDIA Turing) has less potential for async than the AMD architectures (i.e., GCN or RDNA) since async compute was originally advertised for the latter in the past *(J. Hruska, 2015)*. AMD could not be tested in the scope of this project but will be the next step for measuring async performance in the future. Another possible and most likely reason for the lack of any great boost with async might be the nature of the tasks we submitted to the asynchronous queue. In this project both graphics and compute queues were performing heavy texture reads and writes "at the same time" which resulted in bandwidth limitations. However, it is suggested *(G. Thomas, A. Dunn, 2016)* that similar workflows are considered "bad" and inefficient for async. For instance, a set of tasks such as culling or simulation in the async queue might be more effective for the final frame time since they do not work with GPU textures but perform heavy ALU operations (i.e., mathematical calculations). This type of work currently does not exist with the implemented GI techniques but might be researched for some parts in the future (i.e., spatial partitioning and/or cells culling in LPV or VCT).

4.4 Conclusions

Voxel Cone Tracing is a relatively fast technique if we take its "built-in" support for indirect specular and ambient occlusion *(Figure 47)* into consideration. These additional features can save cost in a production of an application since everything fits in one shader and no special shaders are needed. VCT's custom implementation from this project gives an accurate approximation of indirect diffuse within a couple of milliseconds of a frame time and can be optimized more as was shown in 4.2. The technique is not difficult to set up and configure, especially if any voxelization solution already exists in the engine. One downside, however, is the memory consumption which is quite high and can be improved as was discussed in 4.2 with a spatial hierarchy. The issue with 3D storage of the scene is similar to the one LPV has, which is why some cascaded optimizations that process several layers of

voxel grids can also be used as a fix for bigger scenes. One such approach was successfully shipped in a PS4 title *"The Tomorrow Children"* by Q-Games *(J. McLaren, 2015).* A more detailed comparison with the previous techniques of this research is going to be presented in the next chapter.

This demo features various sliders for testing VCT's diffuse, specular, and AO. In addition, a voxelization mode *(Figure 36)* is provided for the users who would like to debug their custom voxelization method which can take extra costs in the production. These configurations can be easily transferred to another project without spending time on writing debug parameters and functions.

Figure 47. Ambient occlusion with Voxel Cone Tracing

V. Results

5.1 Technical comparison of the algorithms

As it was proven before, all the techniques have different performance impacts on the final frame time. The beginning of this section gives the results of the best-achieved performances with every technique and all the optimizations turned on. An interested reader might then compare the time and decide whether the technique can fit in their scenario.

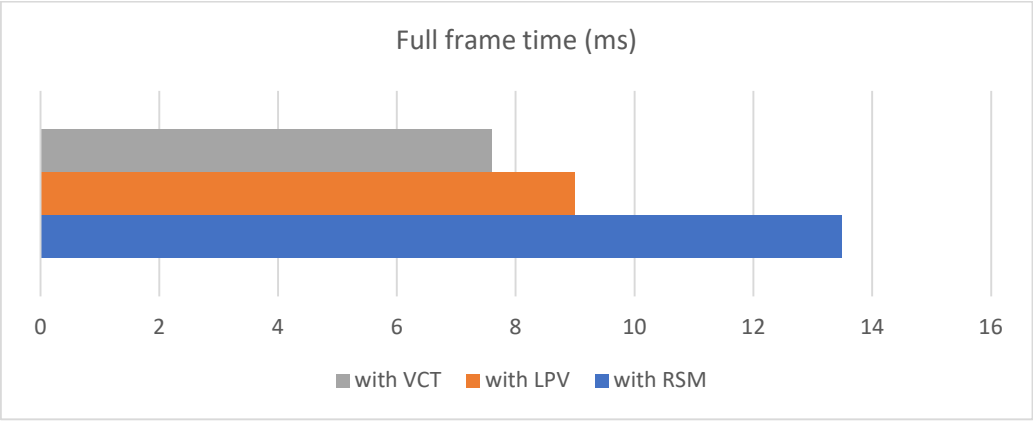


Figure 48. Full frame times with all three techniques optimized to the maximum (in the scope of this project).

It is clear that after the optimizations every technique managed to sustain less than 16ms (~60FPS) of a frame together with other passes, such as G-buffer, shadows, and lighting *(Figure 48)*. However, the tested demo environment was simple and not representative of a proper outdoor level in a AAA game. From personal experience, one-half (if not more) of a frame time is usually spent on G-buffer, shadows, and lighting, whereas the rest is left to additional passes, like post-processing. If any extra pass/technique is planned to be implemented in a pipeline, it is useful to consider its average percental usage from the full frame as well. For this reason, the figure below presents the percentages of all three optimized GI techniques:

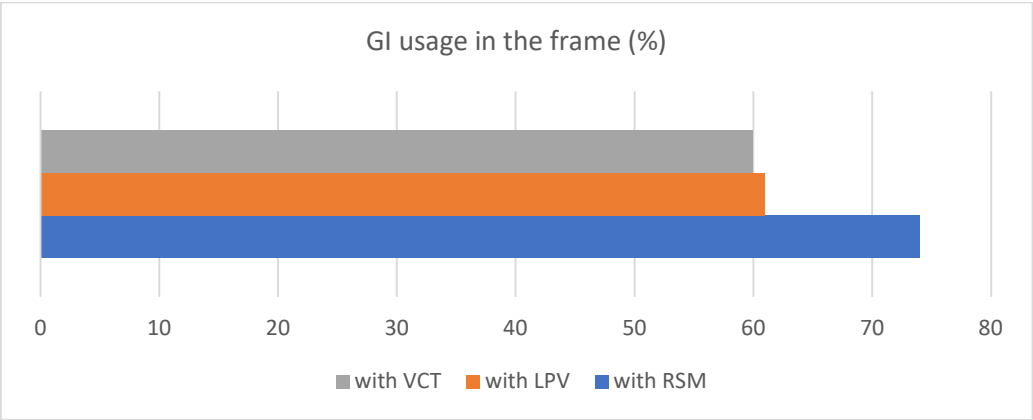


Figure 49. Percentages of every technique within one frame (100%).

The techniques are not cheap and take more than half of the frame even with the optimizations. Yes, the demo scene is geometrically simple, but the algorithms for GI computation from this research also depend on its complexity. For instance, both RSM and LPV need a flux buffer/texture which is dependent on the objects of the scene and VCT needs a voxelization of these objects. In general, other optimizations need to be taken for the rest of the pipeline (G-buffer, shadows, lighting) to achieve decent framerates on target hardware with these GI solutions. Those who would like to implement one or another GI technique must decide on the limitations and bottlenecks of their existing rendering pipeline, whether it is tight on GPU memory, compute units, or something else. The table below attempts to give an overview of the useful properties of every technique in an average scenario:

Table 1. Technical properties of the techniques

Name	Unoptimized time	Optimized time (max. achieved in this project)	GPU memory consumption (unoptimized, excluding constant buffers)	Implementational difficulty
RSM	**~61ms**	**~6.3ms**	2048 * 2048 * (32bits * 4 + 16bits * 4 + 8bits * 4) = 112MB (RSM buffers) 1920 * 1080 * 8bits * 4 = ~7.9MB (GI output texture) Total: **~120MB**	easy
LPV	**~6.6ms**	**~2.7ms**	2048 * 2048 * (32bits * 4 + 16bits * 4 + 8bits * 4) = 112MB (RSM buffers) 32 * 32 * 32 * 3channels * 8bit * 4 (SH RGB injection grid) + 32 * 32 * 32 * 3channels * 8bit * 4 (SH RGB accumulation grid) = 768KB Total: **~113MB**	medium
VCT	**~10.3ms**	**~4.5ms**	256 * 256 * 256 * 8bits * 4 + 6 mip directions * ((128 * 128 * 128 + 64 * 64 * 64 + 32 * 32 * 32 + 16 * 16 * 16 + 8 * 8 * 8 + 4 * 4 * 4) * 8bits * 4 = ~119MB (voxel texture with aniso-mipmapping) 1920 * 1080 * 8bits * 4 = ~7.9MB (GI output texture) Total: **~127MB**	medium

All the techniques are not difficult to implement into the engine and quite straightforward to integrate into the rendering pipeline. Any interested person can borrow the code from this project and reuse it for their scenario. The demo's source code uses a clean structure and clear coding conventions which make it easier to read and rewrite a custom solution for another graphics API. The techniques have only a couple of consecutive passes that could be seen on the rendering pipeline diagrams earlier in this document. The techniques can also be easily ported to compute shaders which is a nice addition and a confirmation that they are still relevant on the modern hardware.

As for the time measurements, RSM, LPV, and VCT, in their non-optimized versions, seem to be unfeasible for the scenarios that aim for 16ms of a frame time. This was mentioned earlier and explained that the reason for that is the existence of other essential non-GI rendering passes. Spending near or more than 8ms on the GI solution is unrealistic in an average modern real-time application with high-resolution geometry, materials, and lighting. LPV and VCT, however, can be run unoptimized if the target is 33ms (~30FPS), but RSM is way behind them and cannot be called "real-time" with its ~60ms requirement (~16FPS). The optimized versions of the techniques in this research allow the engine to run each one of them within 16ms. Moreover, some techniques can be combined for better GI if the performance allows that.

GPU memory consumption of these techniques in the current implementation is almost identical to each other before any optimizations. They all take around 120MB of GPU memory for the textures. These days consumer GPUs have more VRAM than at the time when the techniques were created (i.e., tested NVIDIA RTX2060 from 2019 has 6GB, whereas NVIDIA GTX750 from 2014 has only 1GB) which is why no extra memory optimizations were made in this research except the downsampling. Downsampling of RSM buffers can save a substantial amount of memory and bandwidth for RSM and LPV which makes VCT the most expensive GI solution from the memory perspective in this research. The resolution of the voxelized 3D texture in VCT cannot be very low because of the indirect reflections' quality (explained in 5.2) and the fact that the texture also needs to have mipmapped subtextures for the essential filtering. For this reason, VCT might not be suited for handheld/mobile devices with limited memory since they have graphics memory unified with RAM *(S. Ellis, 2013)*.

The demo application allows the user to enable or disable downsampling of various techniques in order to check the visual differences on the screen. It might be useful if a certain quality level (downsample ratio) is acceptable for an interested reader for their custom environment. Downsampling variables can also be easily configured via code and used after the recompilation of the project.

5.2 Visual comparison of the algorithms

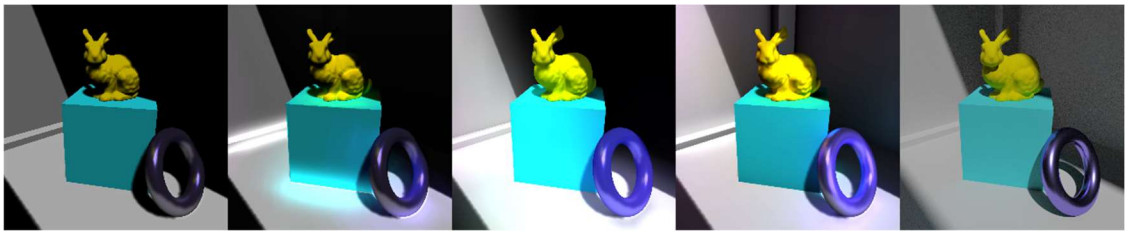


Figure 50. No GI, RSM, LPV, VCT, path-tracer (in order)

All three techniques are supposed to provide global illumination of the scene which aims for the quality of real-world lighting *(Figure 50)*. However, the techniques have unique algorithmic backgrounds and, thus, are visually quite different. This section can help the reader to get familiar with their differences and downsides together with some comparisons and potential improvements.

As it can be seen *(Figure 50)*, RSM, LPV, and VCT are still not able to replace the "ground truth" approach such as a path-tracer, although they make some close approximations to it. Most importantly, they try to approximate indirect diffuse global illumination of the scene *(Figure 51)*.

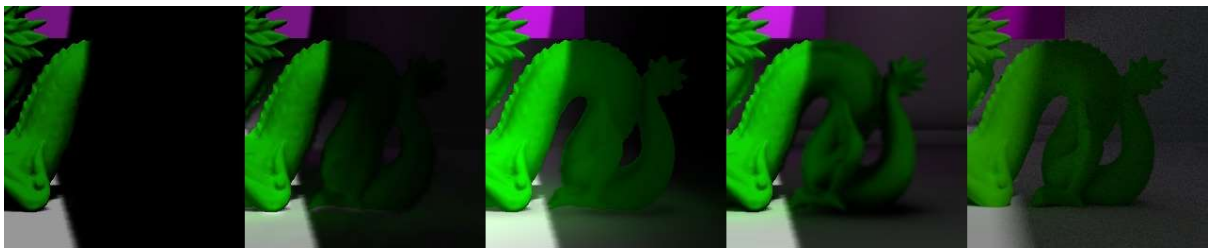


Figure 51. Indirect diffuse: missing (no GI), RSM, LPV, VCT, path-tracer (in order)

Both LPV and VCT can do it fairly well and without severe artifacts which cannot be said about RSM. RSM is the least visually appealing technique of all three since it is the simplest one so far. It does not utilize a third dimension of the scene in its calculation of GI which is a serious assumption that results in some physically inaccurate results. RSM is highly dependent on a random-sampling algorithm that also produces "sparkling" artifacts *(Figure 52)*. In the current demo, RSM also has a weird glow artifact under the objects likely due to the lack of any occlusion from the neighboring geometry (floor) *(Figure 52)*. Finally, it does not give any additional effects, such as ambient occlusion or reflections.

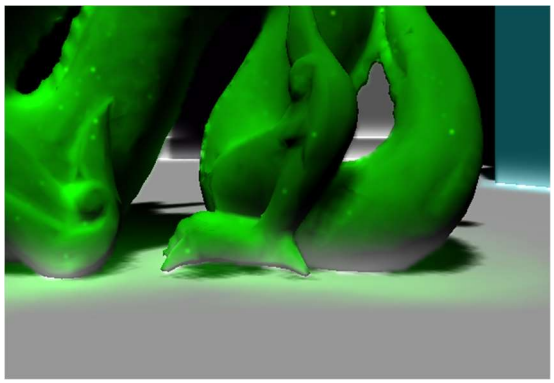


Figure 52. Artifacts of RSM: sparkling, white glow under the objects, light-leaking.

LPV and VCT take a step further from RSM and start to utilize the depth information of the scene. The results of indirect diffuse GI from these techniques look more accurate and natural, although noticeably different from each other. LPV produces smoother diffuse which is more scattered than VCT's diffuse due to the accumulated propagation steps that blend onto each other with every iteration. From this point of view, LPV's result looks more "natural" and similar to the path-traced offline result because darker areas of the scene (in shadow) are brighter than with VCT. It feels that the ambient effect is more pronounced with LPV, however, the technique does not provide any default ambient occlusion. The main drawback of LPV is the flickering of the final result when the light or the objects are moving. This happens because the grid of cells is being recalculated every frame and the radiances are being changed in the cells. It can be annoying for the viewer and might not suit the scenario with several dynamic light sources. This can be fixed by using snapping on the light camera (light view) when calculating RSM buffers: the camera should move by the amount of the size of one RSM texel in view space *(Lightness1024, 2013)*.

VCT is the only technique that accurately approximates the illumination and is mostly based on the actual theory of the problem, however, lots of extra tuning is needed. It uses cones in the hemisphere similar to light bounce directions which help to get better visuals and additional effects by default. One possible drawback of this approach is light-leaking due to the space gaps between cones (adding more cones is a possible solution). As it was shown before, the technique produces ambient occlusion without any extra cost and specular reflections *(Figure 53)* for the cost of one extra cone trace. The latter is physically accurate, but, unfortunately, not detailed due to the voxelization which is supposed to represent the scene and its geometry. Visually, it is the main drawback because the quality of the result depends on the number of voxels and their size ratios to the world coordinates of the scene. The voxelization should also be as accurate as possible to cover smaller details that, otherwise, might be skipped due to precision issues. The problem is not evident in indirect diffuse or ambient occlusion since these effects look "scattered" on the surfaces and are of lower frequency, however, the reflections can be "mirror-like" and can require high-frequency reconstruction of the scene that is determined by the roughness of the surfaces *(M. Migliore, L. Gandel, A. Boucaud, J. Pouderoux, M. Westphal, 2019)*. As an example, a modern real-time DXR approach is implemented in the engine with the help of a new feature set of DirectX12 *(D3D Team, 2018)* and is presented below side-by-side to the reflections of VCT and the path-tracer:

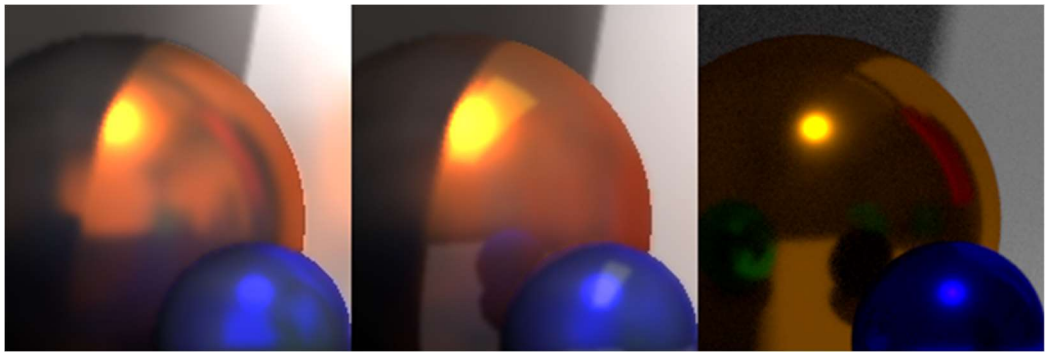


Figure 53. Reflections of VCT, DXR, path-tracer (in order);
Glossy BRDF in the path-tracer is different from the one in the engine's demo.

To finalize the visual properties of all three techniques, a table is given *(Table 2)* that might come in handy in case one or another technique is going to be implemented in a different environment. The comparison is not only useful for graphics programmers but also for artists and game designers since GI is a very important aspect of any real-time application which aims for realism.

Table 2. Visual properties of the techniques

Name	Quality of GI compared to offline render	Default additional effects	Light leaking	Additional artifacts
RSM	Low	No	Yes	"Sparkling", white glow under objects, flickering
LPV	Medium-high	No	Yes	Flickering
VCT	Medium-high	Reflections, AO	Yes	Flickering, "blockiness" in reflections

5.3 Conclusion

RSM, LPV and VCT are decent alternatives for global illumination approximation in real-time scenarios. RSM can be called a bit "outdated" visually from the rest and overly expensive in higher resolutions, although suitable for mobile or handheld platforms where low-resolution render targets are used. It is also easy to implement as it requires only one shader and the extension of the shadow shader. RSM is useful when the production costs are set low for the GI solution. LPV takes more time, is better visually, and has low memory consumption (if optimized). It can be a great solution for the cases when additional effects, such as reflections or AO, are not needed as a part of GI and are made with other methods. Finally, VCT can be considered a winner from all three both in performance and in visuals. The technique provides additional complex effects for little cost and can be run efficiently with minimum optimizations. However, both LPV and VCT might need some extra production costs for complex tuning and special optimizations (i.e., memory).

Modern graphics software allows us to implement all three techniques more efficiently with the help of compute shaders and better parallelism. However, special care should be taken of the more complex optimizations, as they might not work in a "brute-force" way. A clear example of that is "asynchronous compute" which is great in theory but a bit misleading in practice. The optimizations should suit the scene's complexity, the number of light sources, and many other factors such as bandwidth and throughputs of various GPU components.

Modern graphics hardware, such as NVIDIA Turing, can not only help to increase the performance of these algorithms by using extra compute units but also add some additional effects, like ray-traced reflections. Other effects, such as ray-traced ambient occlusion *(B. Nemire, 2018),* were not researched in this project but can be explored in the future.

References

Anuworakarn B. (2019) - *"Why you really should be using mipmapping in your graphics applications".* Retrieved from Imagination's website: https://www.imaginationtech.com/blog/why-you-really-should-be-using-mipmapping-in-your-graphics-applications/

Ashbery S. (2019) - *"Raymarching".* Retrieved from Medium: https://si-ashbery.medium.com/raymarching-3cdf86c637ba

Bavoil L. (2019) - *"The Peak-Performance-Percentage Analysis Method for Optimizing Any GPU Workload".* Retrieved from NVIDIA's website: https://developer.nvidia.com/blog/the-peak-performance-analysis-method-for-optimizing-any-gpu-workload/

Carlsson J. (2010) - *"Realistic Ambient Occlusion in Real-Time".* Retrieved from the website of Chalmers University of Technology: http://www.cse.chalmers.se/~uffe/xjobb/Joakim%20Carlsson-Realistic%20Ambient%20Occlusion%20In%20Real-Time.pdf

Caulfield B. (2018) - *"What's the Difference Between Ray Tracing and Rasterization?".* Retrieved from NVIDIA's website: https://blogs.nvidia.com/blog/2018/03/19/whats-difference-between-ray-tracing-rasterization/

Crassin C., Neyret F., Sainz M., Green S., Eisemann E. (2011) - *"Interactive Indirect Illumination Using Voxel Cone Tracing".* Retrieved from NVIDIA's website: https://research.nvidia.com/sites/default/files/publications/GIVoxels-pg2011-authors.pdf

Cupisz K., Alexander Franke T. (2018) - *"Progressive Lightmapper: An introduction to lightmapping in Unity".* In Unite Berlin course 2018.

D3D Team (2018) - *"Announcing Microsoft DirectX Raytracing!".* Retrieved from Microsoft's developer blog website: https://devblogs.microsoft.com/directx/announcing-microsoft-directx-raytracing/

Dachsbacher C. (2005) - *"Reflective Shadow Mapping".* Retrieved from KlayGE's website: http://www.klayge.org/material/3_12/GI/rsm.pdf

Dimitrov R. (2007) - *"Cascaded Shadow Maps".* Retrieved from NVIDIA's website: https://developer.download.nvidia.com/SDK/10.5/opengl/src/cascaded_shadow_maps/doc/cascaded_shadow_maps.pdf

Ellis S. (2013) - *"Memory Management on Embedded Graphics Processors".* Retrieved from ARM's website: https://community.arm.com/developer/tools-software/graphics/b/blog/posts/memory-management-on-embedded-graphics-processors

Green S. (2003) - *"Spherical Harmonic Lighting: The Gritty Details".* Retrieved from the website of Chalmers University of Technology: http://www.cse.chalmers.se/~uffe/xjobb/Readings/GlobalIllumination/Spherical%20Harmonic%20Lighting%20-%20the%20gritty%20details.pdf

Hruska J. (2015) - *"DirectX 12, LiquidVR may breathe fresh life into AMD GPUs thanks to asynchronous shading".* Retrieved from ExtremeTech: https://www.extremetech.com/gaming/202407-directx-12-liquidvr-may-breathe-fresh-life-into-amd-gpus-thanks-to-asynchronous-compute

Kajiya J. (1986) - *"The Rendering Equation"*. Retrieved from the website of Chalmers University of Technology: http://www.cse.chalmers.se/edu/year/2016/course/TDA361/rend_eq.pdf

Kaplanyan A. (2010) - *"Cascaded Light Propagation Volumes for Real-Time Indirect Illumination"*. Retrieved from the website of Karlsruhe Institute of Technology: http://cg.ivd.kit.edu/publications/p2010/CLPVFRII_Kaplanyan_2010/CLPVFRII_Kaplanyan_2010.pdf

Koskivuori V. (2021) - *"C-Ray"*. Retrieved from GitHub: https://github.com/vkoskiv/c-ray

Lightness1024 (2013) - *"Light Propagation Volumes Flickering"*. Retrieved from the thread on GameDev.net forum: https://www.gamedev.net/forums/topic/637694-light-propagation-volumes-flickering/

McLaren J. (2015) - *"The Technology Of The Tomorrow Children"*. Retrieved from GDC's website: https://www.gdcvault.com/play/1022428/The-Technology-of-The-Tomorrow

Migliore M., Gandel L., Boucaud A., Pouderoux J., Westphal M. (2019) - *"Introducing Physically Based Rendering with VTK"*. Retrieved from Kitware: https://blog.kitware.com/vtk-pbr/

Narkowicz K. (2014) - *"Octahedron normal vector encoding"*. Retrieved from the blog of Krzysztof Narkowicz: https://knarkowicz.wordpress.com/2014/04/16/octahedron-normal-vector-encoding/

Nemire B. (2018) - *"RTX Coffee Break: Ray Traced Ambient Occlusion"*. Retrieved from NVIDIA's website: https://developer.nvidia.com/blog/rtx-coffee-break-ray-traced-ambient-occlusion-417-minutes/

Olovsson J. D., Doggett M. (2013) - *"Octree Light Propagation Volumes"*. Retrieved from the website of Linköping University: https://ep.liu.se/ecp/094/004/ecp13094004.pdf

Peng B., Poulin P. (2019) - *"Fast Screen Space Global Illumination"*. Retrieved from the website of Université de Montréal: https://www-ens.iro.umontreal.ca/~pengbowe/ift3150/res/IFT3150_E19_Fast_Screen_Space_Global_Illumination.pdf

Pharr M., Jakob W., Humphreys G. (2016) - *"1.2 Photorealistic Rendering and the Ray-Tracing Algorithm"*. In *"Physically Based Rendering: From Theory to Implementation. 3rd edition"*. Retrieved from the official website of the book: https://www.pbr-book.org/3ed-2018/Introduction/Photorealistic_Rendering_and_the_Ray-Tracing_Algorithm.html

Pharr M., Jakob W., Humphreys G. (2016) - *"14.5 Path Tracing"*. In *"Physically Based Rendering: From Theory to Implementation. 3rd edition"*. Retrieved from the official website of the book: https://www.pbr-book.org/3ed-2018/Light_Transport_I_Surface_Reflection/Path_Tracing.html

Pranckevičius A. (2018) - *"Pathtracer 13: GPU threadgroup memory is useful"*. Retrieved from Aras' blog: https://aras-p.info/blog/2018/05/28/Pathtracer-13-GPU-threadgroup-memory-is-useful/

Ritschel T., Dachsbacher C., Grosch T., Kautz J. (2012) - *"The State of the Art in Interactive Global Illumination".* Retrieved from the website of TU Clausthal: https://www.in.tu-clausthal.de/fileadmin/homes/CG/data_pub/paper/GISTAR.pdf

Seymour M. (2013) - *"The science of spherical harmonics at Weta Digital".* Retrieved from FXGuide: https://www.fxguide.com/fxfeatured/the-science-of-spherical-harmonics-at-weta-digital/

Sharda A. (2021) - *"Image Filters: Gaussian Blur".* Retrieved from Medium: https://aryamansharda.medium.com/image-filters-gaussian-blur-eb36db6781b1

Stachowiak T. (2015) - *"Stochastic Screen-Space Reflections".* In the ACM SIGGRAPH "Advances in Real-Time Rendering" course 2015.

Stefanov N. (2016) - *"Global Illumination in Tom Clancy's The Division".* Retrieved from GDC's website: http://mrakobes.com/Nikolay.Stefanov.GDC.2016.pdf

Thomas G., Dunn A. (2016) - *"Practical DirectX 12".* Retrieved from NVIDIA's website: https://developer.nvidia.com/sites/default/files/akamai/gameworks/blog/GDC16/GDC16_gthomas_adunn_Practical_DX12.pdf

Williams L. (1978) - *"Casting curved shadows on curved surfaces".* Retrieved from the website of University of Sand Diego: http://cseweb.ucsd.edu/~ravir/274/15/papers/p270-williams.pdf

Yudintsev A. (2019) - *"Scalable Real-time Global Illumination for Large Scenes".* Retrieved from GDC's website: https://www.gdcvault.com/play/1026469/Scalable-Real-Time-Global-Illumination

Zhang L., Chen W. Ebert D. S., Peng Q. (2007) - *"Conservative voxelization".* In "The Visual Computer" (23, pp. 783-792).

Appendix I - Shaders

ReflectiveShadowMappingPS.hlsl

```
#include "Common.hlsl"
#define RSM_SAMPLES_COUNT 512

Texture2D<float4> worldPosLSBuffer : register(t0);   // light space
Texture2D<float4> normalLSBuffer : register(t1);     // light space
Texture2D<float4> fluxLSBuffer : register(t2);       // light space

Texture2D<float4> worldPosWSBuffer : register(t3);   // world space
Texture2D<float4> normalWSBuffer : register(t4);     // world space

SamplerState RSMSampler : register(s0);

cbuffer RSMConstantBuffer : register(b0)
{
    float4x4 ShadowViewProjection;
    float RSMIntensity;
    float RSMRMax;
    float2 UpsampleRatio;
};
cbuffer RSMConstantBuffer2 : register(b1)
{
    float4 xi[RSM_SAMPLES_COUNT];
}

struct VSInput
{
    float4 position : POSITION;
    float2 uv : TEXCOORD;
};

struct PSInput
{
    float2 uv : TEXCOORD;
    float4 position : SV_POSITION;
};

struct PSOutput
{
    float4 rsm : SV_Target0;
};

float3 CalculateRSM(float3 pos, float3 normal)
{
    float4 texSpacePos = mul(ShadowViewProjection, float4(pos, 1.0f));
    texSpacePos.rgb /= texSpacePos.w;
    texSpacePos.rg = texSpacePos.rg * float2(0.5f, -0.5f) + float2(0.5f,
0.5f);

    float3 indirectIllumination = float3(0.0, 0.0, 0.0);

    uint width = 0;
    uint height = 0;
    uint nol = 0;
    normalLSBuffer.GetDimensions(0, width, height, nol);

    for (int i = 0; i < RSM_SAMPLES_COUNT; i++)
    {
```

```
        float2 coord = texSpacePos.rg + RSMRMax * float2(xi[i].x *
sin(2.0f * PI * xi[i].y), xi[i].x * cos(2.0f * PI * xi[i].y));

        float2 texcoord = coord * float2(width, height);
        float3 vplPosWS = worldPosLSBuffer.Load(uint3(uint2(texcoord),
0)).rgb;
        float3 vplNormalWS = normalLSBuffer.Sample(RSMSampler,
coord).rgb;
        float3 flux = fluxLSBuffer.Load(uint3(uint2(texcoord), 0)).rgb;

        float3 vplPosDir = (pos - vplPosWS);

        float3 res = flux * ((max(0.0, dot(vplNormalWS, vplPosDir)) *
max(0.0, dot(normal, -vplPosDir))) / (dot(vplPosDir, vplPosDir) *
dot(vplPosDir, vplPosDir)));
        res *= xi[i].x * xi[i].x;

        indirectIllumination += res;
    }

    return indirectIllumination;
}

PSInput VSMain(VSInput input)
{
    PSInput result;

    result.position = float4(input.position.xyz, 1);
    result.uv = input.uv;

    return result;
}

PSOutput PSMain(PSInput input)
{
    PSOutput output = (PSOutput) 0;

    float4 normalWS = normalWSBuffer[input.position.xy * UpsampleRatio];
    float4 worldPosWS = worldPosWSBuffer[input.position.xy *
UpsampleRatio];
    output.rsm = saturate(float4(CalculateRSM(worldPosWS.rgb,
normalWS.rgb), 1.0f));

    return output;
}
```

RSMDownsamplePS.hlsl

```
#include "Common.hlsl"
#define DOWNSAMPLE_SIZE 4

Texture2D<float4> worldPosLSBuffer : register(t0); // light space
Texture2D<float4> normalLSBuffer : register(t1); // light space
Texture2D<float4> fluxLSBuffer : register(t2); // light space

struct RSMTexel
{
    float3 flux;
    float3 normalWS;
    float3 positionWS;
};

cbuffer RSMDownsampleCB : register(b0)
{
    float4 LightDir;
    int ScaleSize;
};

struct VS_INPUT
{
    float4 position : POSITION;
};

struct VS_OUTPUT
{
    float4 position : SV_POSITION;
};

struct PS_OUTPUT
{
    float4 worldPosDownsampled : SV_Target0;
    float4 normalDownsampled : SV_Target1;
    float4 fluxDownsampled : SV_Target2;
};

VS_OUTPUT VSMain(VS_INPUT input)
{
    VS_OUTPUT result;
    result.position = float4(input.position.xyz, 1);
    return result;
}

RSMTexel GetRSMTexel(uint2 coords)
{
    RSMTexel texel = (RSMTexel) 0;
    texel.positionWS = worldPosLSBuffer.Load(int3(coords, 0)).xyz;
    texel.normalWS = normalLSBuffer.Load(int3(coords, 0)).xyz;
    texel.flux = fluxLSBuffer.Load(int3(coords, 0)).xyz;
    return texel;
}

float GetLuminance(RSMTexel texel)
{
    return (texel.flux.r * 0.299f + texel.flux.g * 0.587f + texel.flux.b
* 0.114f) + max(0.0f, dot(texel.normalWS, -LightDir.rgb));
}
```

```
PS_OUTPUT PSMain(VS_OUTPUT input)
{
    PS_OUTPUT output = (PS_OUTPUT) 0;

    RSMTexel groupTexels[DOWNSAMPLE_SIZE * DOWNSAMPLE_SIZE];
    uint2 rsmCoords = uint2(input.position.x, input.position.y);

    RSMTexel resultTexel = (RSMTexel) 0;
    uint numSamples = 0;
    float maxLuminance = 0.0f;
    int3 brightestCellPos = int3(0, 0, 0);

    for (int y = 0; y < DOWNSAMPLE_SIZE; y++)
    {
        for (int x = 0; x < DOWNSAMPLE_SIZE; x++)
        {
            int2 texIdx = rsmCoords.xy * DOWNSAMPLE_SIZE + int2(x, y);
            groupTexels[y * DOWNSAMPLE_SIZE + x] = GetRSMTexel(texIdx);
            float texLum = GetLuminance(groupTexels[y * DOWNSAMPLE_SIZE +
x]);
            if (texLum > maxLuminance)
            {
                brightestCellPos = int3(groupTexels[y * DOWNSAMPLE_SIZE +
x].positionWS * LPV_SCALE + float3(LPV_DIM_HALF, LPV_DIM_HALF,
LPV_DIM_HALF));
                maxLuminance = texLum;
            }
        }
    }

    for (int g = 0; g < DOWNSAMPLE_SIZE * DOWNSAMPLE_SIZE; g++)
    {
        int3 texelPos = int3(groupTexels[g].positionWS * LPV_SCALE +
float3(LPV_DIM_HALF, LPV_DIM_HALF, LPV_DIM_HALF));
        int3 deltaPos = texelPos - brightestCellPos;
        if (dot(deltaPos, deltaPos) < 10)
        {
            resultTexel.flux += groupTexels[g].flux;
            resultTexel.positionWS += groupTexels[g].positionWS;
            resultTexel.normalWS += groupTexels[g].normalWS;
            numSamples++;
        }
    }

    // normalize
    if (numSamples > 0)
    {
        resultTexel.positionWS /= numSamples;
        resultTexel.normalWS /= numSamples;
        resultTexel.normalWS = normalize(resultTexel.normalWS);
        resultTexel.flux /= numSamples;
    }

    output.worldPosDownsampled = float4(resultTexel.positionWS, 1.0f);
    output.normalDownsampled = float4(resultTexel.normalWS, 1.0f);
    output.fluxDownsampled = float4(resultTexel.flux, 1.0f);
    return output;
}
```

LPVInjection.hlsl

```
#include "Common.hlsl"

struct RSMTexel
{
    float3 flux;
    float3 normalWS;
    float3 positionWS;
};

Texture2D<float4> worldPosLSBuffer : register(t0); // light space
Texture2D<float4> normalLSBuffer : register(t1); // light space
Texture2D<float4> fluxLSBuffer : register(t2); // light space

cbuffer LPVConstantBuffer : register(b0)
{
    float4x4 worldToLPV;
    float LPVCutoff;
    float LPVPower;
    float LPVAttenuation;
};

struct VS_IN
{
    uint vertexID : SV_VertexID;
};

struct GS_IN
{
    float4 cellPos : SV_POSITION;
    float3 normal : NORMAL;
    float3 flux : FLUX;
};

struct PS_IN
{
    float4 screenPos : SV_POSITION;
    float3 normal : NORMAL;
    float3 flux : FLUX;
    uint layerID : SV_RenderTargetArrayIndex;
};

struct PS_OUT
{
    float4 redSH : SV_Target0;
    float4 greenSH : SV_Target1;
    float4 blueSH : SV_Target2;
};

RSMTexel GetRSMTexel(uint2 coords)
{
    RSMTexel texel = (RSMTexel)0;
    texel.positionWS = worldPosLSBuffer.Load(int3(coords, 0)).xyz;
    texel.normalWS = normalLSBuffer.Load(int3(coords, 0)).xyz;
    texel.flux = fluxLSBuffer.Load(int3(coords, 0)).xyz;
    return texel;
}

GS_IN VSMain(VS_IN input)
{
```

```
    GS_IN output = (GS_IN)0;

    uint2 RSMsize;
    worldPosLSBuffer.GetDimensions(RSMsize.x, RSMsize.y);
    uint2 rsmCoords = uint2(input.vertexID % RSMsize.x, input.vertexID /
RSMsize.x);

    RSMTexel texel = GetRSMTexel(rsmCoords.xy);

    float3 pos = texel.positionWS;
    output.cellPos = float4(int3(pos * LPV_SCALE + float3(LPV_DIM_HALF,
LPV_DIM_HALF, LPV_DIM_HALF) + 0.5f * texel.normalWS), 1.0f);
    output.normal = texel.normalWS;
    output.flux = texel.flux;

    return output;
}

[maxvertexcount(1)]
void GSMain(point GS_IN input[1], inout PointStream<PS_IN> OutputStream)
{
    PS_IN output = (PS_IN)0;
    output.layerID = floor(input[0].cellPos.z);

    output.screenPos = float4((input[0].cellPos.xy + 0.5f) *
LPV_DIM_INVERSE * 2.0f - 1.0f, 0.0f, 1.0f);
    output.screenPos.y = -output.screenPos.y;

    output.normal = input[0].normal;
    output.flux = input[0].flux;

    OutputStream.Append(output);
}

PS_OUT PSMain(PS_IN input)
{
    PS_OUT output = (PS_OUT) 0;

    float4 SH_coef = DirCosLobeToSH(input.normal) / PI;
    output.redSH = SH_coef * input.flux.r;
    output.greenSH = SH_coef * input.flux.g;
    output.blueSH = SH_coef * input.flux.b;

    return output;
}
```

LPVPropagation.hlsl

```
#include "Common.hlsl"

Texture3D<float4> redSH : register(t0);
Texture3D<float4> greenSH : register(t1);
Texture3D<float4> blueSH : register(t2);

static const float3 cellDirections[6] =
{
    float3(0, 0, 1),
    float3(1, 0, 0),
    float3(0, 0,-1),
    float3(-1,0, 0),
    float3(0, 1, 0),
    float3(0,-1, 0)
};

struct VS_IN
{
    uint vertex : SV_VertexID;
    uint depthIndex : SV_InstanceID;
};

struct VS_OUT
{
    float4 screenPos : SV_Position;
    uint depthIndex : DEPTHINDEX;
};

struct GS_OUT
{
    float4 screenPos : SV_Position;
    uint depthIndex : SV_RenderTargetArrayIndex;
};

struct PS_OUT
{
    float4 redSH : SV_Target0;
    float4 greenSH : SV_Target1;
    float4 blueSH : SV_Target2;
    float4 acc_redSH : SV_Target3;
    float4 acc_greenSH : SV_Target4;
    float4 acc_blueSH : SV_Target5;
};

//rendering a "quad" but with 3 vertices only
VS_OUT VSMain(VS_IN input)
{
    VS_OUT output = (VS_OUT) 0;

    if (input.vertex == 0)
        output.screenPos.xy = float2(-1.0, 1.0);
    else if (input.vertex == 1)
        output.screenPos.xy = float2(3.0, 1.0);
    else
        output.screenPos.xy = float2(-1.0, -3.0);

    output.screenPos.zw = float2(0.0, 1.0);
    output.depthIndex = input.depthIndex;
```

```
    return output;
}

[maxvertexcount(3)]
void GSMain(triangle VS_OUT input[3], inout TriangleStream<GS_OUT>
OutputStream)
{
    for (int i = 0; i < 3; i++)
    {
        GS_OUT output = (GS_OUT)0;
        output.depthIndex = input[i].depthIndex;
        output.screenPos = input[i].screenPos;

        OutputStream.Append(output);
    }
}

struct SHContribution
{
    float4 red, green, blue;
};

static const float2 cellsides[4] = { float2(1.0, 0.0), float2(0.0, 1.0),
float2(-1.0, 0.0), float2(0.0, -1.0) };
static const float directFaceSubtendedSolidAngle = 0.4006696846f / PI;
static const float sideFaceSubtendedSolidAngle = 0.4234413544f / PI;

float3 getEvalSideDirection(int index, int3 orientation)
{
    const float smallComponent = 0.4472135; // 1 / sqrt(5)
    const float bigComponent = 0.894427; // 2 / sqrt(5)

    const int2 side = cellsides[index];
    float3 tmp = float3(side.x * smallComponent, side.y * smallComponent,
bigComponent);
    return float3(orientation.x * tmp.x, orientation.y * tmp.y,
orientation.z * tmp.z);
}

float3 getReprojSideDirection(int index, int3 orientation)
{
    const int2 side = cellsides[index];
    return float3(orientation.x * side.x, orientation.y * side.y, 0);
}

SHContribution GetSHGatheringContribution(int4 cellIndex)
{
    SHContribution result = (SHContribution) 0;

    for (int neighbourCell = 0; neighbourCell < 6; neighbourCell++)
    {
        int4 neighbourPos = cellIndex -
int4(cellDirections[neighbourCell], 0);

        SHContribution neighbourContribution = (SHContribution) 0;
        neighbourContribution.red = redSH.Load(neighbourPos);
        neighbourContribution.green = greenSH.Load(neighbourPos);
        neighbourContribution.blue = blueSH.Load(neighbourPos);

        // add contribution from main direction
```

```
        float4 directionCosLobeSH =
DirCosLobeToSH(cellDirections[neighbourCell]);
        float4 directionSH = DirToSH(cellDirections[neighbourCell]);
        result.red += directFaceSubtendedSolidAngle *
dot(neighbourContribution.red, directionSH) * directionCosLobeSH;
        result.green += directFaceSubtendedSolidAngle *
dot(neighbourContribution.green, directionSH) * directionCosLobeSH;
        result.blue += directFaceSubtendedSolidAngle *
dot(neighbourContribution.blue, directionSH) * directionCosLobeSH;

        // contributions from side direction
        for (int face = 0; face < 4; face++)
        {
            float3 evaluatedSideDir = getEvalSideDirection(face,
cellDirections[face]);
            float3 reproSideDir = getReprojSideDirection(face,
cellDirections[face]);

            float4 evalSideDirSH = DirToSH(evaluatedSideDir);
            float4 reproSideDirCosLobeSH = DirCosLobeToSH(reproSideDir);

            result.red += sideFaceSubtendedSolidAngle *
dot(neighbourContribution.red, evalSideDirSH) * reproSideDirCosLobeSH;
            result.green += sideFaceSubtendedSolidAngle *
dot(neighbourContribution.green, evalSideDirSH) * reproSideDirCosLobeSH;
            result.blue += sideFaceSubtendedSolidAngle *
dot(neighbourContribution.blue, evalSideDirSH) * reproSideDirCosLobeSH;
        }
    }

    return result;
}

PS_OUT PSMain(GS_OUT input)
{
    PS_OUT output = (PS_OUT) 0;

    int4 cellIndex = int4(input.screenPos.xy - 0.5f, input.depthIndex,
0);
    SHContribution resultContribution =
GetSHGatheringContribution(cellIndex);

    output.redSH = resultContribution.red;
    output.greenSH = resultContribution.green;
    output.blueSH = resultContribution.blue;

    output.acc_redSH = resultContribution.red;
    output.acc_greenSH = resultContribution.green;
    output.acc_blueSH = resultContribution.blue;

    return output;
}
```

VoxelConeTracingVoxelization.hlsl

```
cbuffer VoxelizationCB : register(b0)
{
    float4x4 WorldVoxelCube;
    float4x4 ViewProjection;
    float4x4 ShadowViewProjection;
    float WorldVoxelScale;
};

cbuffer perModelInstanceCB : register(b1)
{
    float4x4 World;
    float4 DiffuseColor;
};

struct VS_IN
{
    float3 position : POSITION;
    float3 normal : NORMAL;
};
struct GS_IN
{
    float4 position : SV_POSITION;
    float3 normal : NORMAL;
};
struct PS_IN
{
    float4 position : SV_POSITION;
    float3 voxelPos : VOXEL_POSITION;
};

RWTexture3D<float4> outputTexture : register(u0);
Texture2D<float> shadowBuffer : register(t0);

SamplerComparisonState PcfShadowMapSampler : register(s0);

GS_IN VSMain(VS_IN input)
{
    GS_IN output = (GS_IN) 0;

    output.position = mul(World, float4(input.position.xyz, 1));
    return output;
}

[maxvertexcount(3)]
void GSMain(triangle GS_IN input[3], inout TriangleStream<PS_IN>
OutputStream)
{
    PS_IN output[3];
    output[0] = (PS_IN) 0;
    output[1] = (PS_IN) 0;
    output[2] = (PS_IN) 0;

    float3 p1 = input[1].position.rgb - input[0].position.rgb;
    float3 p2 = input[2].position.rgb - input[0].position.rgb;
    float3 n = abs(normalize(cross(p1, p2)));

      float axis = max(n.x, max(n.y, n.z));

    [unroll]
```

```
    for (uint i = 0; i < 3; i++)
    {
        output[0].voxelPos = input[i].position.xyz / WorldVoxelScale *
2.0f;
        output[1].voxelPos = input[i].position.xyz / WorldVoxelScale *
2.0f;
        output[2].voxelPos = input[i].position.xyz / WorldVoxelScale *
2.0f;

        if (axis == n.z)
            output[i].position = float4(output[i].voxelPos.x,
output[i].voxelPos.y, 0, 1);
        else if (axis == n.x)
            output[i].position = float4(output[i].voxelPos.y,
output[i].voxelPos.z, 0, 1);
        else
            output[i].position = float4(output[i].voxelPos.x,
output[i].voxelPos.z, 0, 1);

        //output[i].normal = input[i].normal;
        OutputStream.Append(output[i]);
    }
    OutputStream.RestartStrip();
}

float3 VoxelToWorld(float3 pos)
{
    float3 result = pos;
    result *= WorldVoxelScale;

    return result * 0.5f;
}
void PSMain(PS_IN input)
{
    uint width;
    uint height;
    uint depth;

    outputTexture.GetDimensions(width, height, depth);

    float3 voxelPos = input.voxelPos.rgb;
    voxelPos.y = -voxelPos.y;

    int3 finalVoxelPos = width * float3(0.5f * voxelPos + float3(0.5f,
0.5f, 0.5f));
    float4 colorRes = float4(DiffuseColor.rgb, 1.0f);
    voxelPos.y = -voxelPos.y;

    float4 worldPos = float4(VoxelToWorld(voxelPos), 1.0f);
    float4 lightSpacePos = mul(ShadowViewProjection, worldPos);
    float4 shadowcoord = lightSpacePos / lightSpacePos.w;
    shadowcoord.rg = shadowcoord.rg * float2(0.5f, -0.5f) + float2(0.5f,
0.5f);
    float shadow = shadowBuffer.SampleCmpLevelZero(PcfShadowMapSampler,
shadowcoord.xy, shadowcoord.z);

    outputTexture[finalVoxelPos] = colorRes * float4(shadow, shadow,
shadow, 1.0f);
}
```

VoxelConeTracingPS.hlsl

```
#include "Common.hlsl"

#define NUM_CONES 6

static const float coneAperture = 0.577f; // 6 cones, 60deg each,
tan(30deg) = aperture
static const float3 diffuseConeDirections[] =
{
    float3(0.0f, 1.0f, 0.0f),
    float3(0.0f, 0.5f, 0.866025f),
    float3(0.823639f, 0.5f, 0.267617f),
    float3(0.509037f, 0.5f, -0.7006629f),
    float3(-0.50937f, 0.5f, -0.7006629f),
    float3(-0.823639f, 0.5f, 0.267617f)
};
static const float diffuseConeWeights[] =
{
    0.25, 0.15, 0.15, 0.15, 0.15, 0.15
};

static const float specularOneDegree = 0.0174533f; //in radians
static const int specularMaxDegreesCount = 5;

Texture2D<float4> albedoBuffer : register(t0);
Texture2D<float4> normalBuffer : register(t1);
Texture2D<float4> worldPosBuffer : register(t2);

Texture3D<float4> voxelTexturePosX : register(t3);
Texture3D<float4> voxelTextureNegX : register(t4);
Texture3D<float4> voxelTexturePosY : register(t5);
Texture3D<float4> voxelTextureNegY : register(t6);
Texture3D<float4> voxelTexturePosZ : register(t7);
Texture3D<float4> voxelTextureNegZ : register(t8);
Texture3D<float4> voxelTexture : register(t9);

SamplerState LinearSampler : register(s0);

cbuffer VoxelizationCB : register(b0)
{
    float4x4 WorldVoxelCube;
    float4x4 ViewProjection;
    float4x4 ShadowViewProjection;
    float WorldVoxelScale;
};

cbuffer VCTMainCB : register(b1)
{
    float4 CameraPos;
    float2 UpsampleRatio;
    float IndirectDiffuseStrength;
    float IndirectSpecularStrength;
    float MaxConeTraceDistance;
    float AOFalloff;
    float SamplingFactor;
    float VoxelSampleOffset;
};

struct VS_IN
{
```

```
    float4 position : POSITION;
    float2 uv : TEXCOORD;
};

struct PS_IN
{
    float4 position : SV_POSITION;
    float2 uv : TEXCOORD;
};

struct PS_OUT
{
    float4 result : SV_Target0;
};

PS_IN VSMain(VS_IN input)
{
    PS_IN result = (PS_IN)0;

    result.position = float4(input.position.xyz, 1);
    result.uv = input.uv;

    return result;
}

float4 GetAnisotropicSample(float3 uv, float3 weight, float lod, bool
posX, bool posY, bool posZ)
{
    int anisoLevel = max(lod - 1.0f, 0.0f);

    uint width;
    uint height;
    uint depth;
    voxelTexturePosX.GetDimensions(width, height, depth);

    width >>= anisoLevel;
    height >>= anisoLevel;
    depth >>= anisoLevel;

    float4 anisoSample =
    weight.x * ((posX) ? voxelTexturePosX.SampleLevel(LinearSampler, uv,
anisoLevel) : voxelTextureNegX.SampleLevel(LinearSampler, uv,
anisoLevel)) +
    weight.y * ((posY) ? voxelTexturePosY.SampleLevel(LinearSampler, uv,
anisoLevel) : voxelTextureNegY.SampleLevel(LinearSampler, uv,
anisoLevel)) +
    weight.z * ((posZ) ? voxelTexturePosZ.SampleLevel(LinearSampler, uv,
anisoLevel) : voxelTextureNegZ.SampleLevel(LinearSampler, uv,
anisoLevel));

    if (lod < 1.0f)
    {
        float4 baseColor = voxelTexture.SampleLevel(LinearSampler, uv,
0);
        anisoSample = lerp(baseColor, anisoSample, clamp(lod, 0.0f,
1.0f));
    }

    return anisoSample;
}
```

```
float4 GetVoxel(float3 worldPosition, float3 weight, float lod, bool
posX, bool posY, bool posZ)
{
    float3 offset = float3(VoxelSampleOffset, VoxelSampleOffset,
VoxelSampleOffset);
    float3 voxelTextureUV = worldPosition / WorldVoxelScale * 2.0f;
    voxelTextureUV.y = -voxelTextureUV.y;
    voxelTextureUV = voxelTextureUV * 0.5f + 0.5f + offset;

    return GetAnisotropicSample(voxelTextureUV, weight, lod, posX, posY,
posZ);
}

float4 TraceCone(float3 pos, float3 normal, float3 direction, float
aperture, out float occlusion, bool calculateAO, uint voxelResolution)
{
    float lod = 0.0;
    float4 color = float4(0.0f, 0.0f, 0.0f, 0.0f);

    occlusion = 0.0f;
    float voxelWorldSize = WorldVoxelScale / voxelResolution;
    float dist = voxelWorldSize;
    float3 startPos = pos + normal * dist;

    float3 weight = direction * direction;

    while (dist < MaxConeTraceDistance && color.a < 0.9f)
    {
        float diameter = 2.0f * aperture * dist;
        float lodLevel = log2(diameter / voxelWorldSize);
        float4 voxelColor = GetVoxel(startPos + dist * direction, weight,
lodLevel, direction.x > 0.0, direction.y > 0.0, direction.z > 0.0);

        // front-to-back
        color += (1.0 - color.a) * voxelColor;
        if (occlusion < 1.0f && calculateAO)
            occlusion += ((1.0f - occlusion) * voxelColor.a) / (1.0f +
AOFalloff * diameter);

        dist += diameter * SamplingFactor;
    }

    return color;
}

float4 CalculateIndirectSpecular(float3 worldPos, float3 normal, float4
specular, uint voxelResolution)
{
    float4 result;
    float3 viewDirection = normalize(CameraPos.rgb - worldPos);
    float3 coneDirection = normalize(reflect(-viewDirection, normal));

    float aperture = clamp(tan(PI * 0.5f * (1.0f - specular.a)),
specularOneDegree * specularMaxDegreesCount, PI);

    float ao = -1.0f;
    result = TraceCone(worldPos, normal, coneDirection, aperture, ao,
false, voxelResolution);

    return IndirectSpecularStrength * result * float4(specular.rgb, 1.0f)
* specular.a;
```

```
}

float4 CalculateIndirectDiffuse(float3 worldPos, float3 normal, out float
ao, uint voxelResolution)
{
    float4 result;
    float3 coneDirection;

    float3 upDir = float3(0.0f, 1.0f, 0.0f);
    if (abs(dot(normal, upDir)) == 1.0f)
        upDir = float3(0.0f, 0.0f, 1.0f);

    float3 right = normalize(upDir - dot(normal, upDir) * normal);
    float3 up = cross(right, normal);

    float finalAo = 0.0f;
    float tempAo = 0.0f;

    for (int i = 0; i < NUM_CONES; i++)
    {
        coneDirection = normal;
        coneDirection += diffuseConeDirections[i].x * right +
diffuseConeDirections[i].z * up;
        coneDirection = normalize(coneDirection);

        result += TraceCone(worldPos, normal, coneDirection,
coneAperture, tempAo, true, voxelResolution) * diffuseConeWeights[i];
        finalAo += tempAo * diffuseConeWeights[i];
    }

    ao = finalAo;

    return IndirectDiffuseStrength * result;
}

PS_OUT PSMain(PS_IN input)
{
    PS_OUT output = (PS_OUT) 0;
    float2 inPos = input.position.xy;

    float3 normal = normalize(normalBuffer[inPos * UpsampleRatio].rgb);
    float4 worldPos = worldPosBuffer[inPos * UpsampleRatio];
    float4 albedo = albedoBuffer[inPos * UpsampleRatio];

    uint width;
    uint height;
    uint depth;
    voxelTexture.GetDimensions(width, height, depth);

    float ao = 0.0f;
    float4 indirectDiffuse = CalculateIndirectDiffuse(worldPos.rgb,
normal.rgb, ao, width);
    float4 indirectSpecular = CalculateIndirectSpecular(worldPos.rgb,
normal.rgb, albedo, width);

    output.result = saturate(float4(indirectDiffuse.rgb * albedo.rgb +
indirectSpecular.rgb, ao));
    return output;
}
```

UpsampleBlurCS.hlsl

```
// Generic upsampling and blurring CS shader
//
// Modified version of https://github.com/microsoft/DirectX-Graphics-
Samples/blob/master/MiniEngine/Core/Shaders/UpsampleAndBlurCS.hlsl

cbuffer UpsampleAndBlurCbuffer : register(b0)
{
    bool Upsample = true;
};

Texture2D<float4> Input : register(t0);
RWTexture2D<float4> Output : register(u0);

SamplerState BilinearSampler : register(s0);

// The guassian blur weights (derived from Pascal's triangle)
static const float Weights5[3] = { 6.0f / 16.0f, 4.0f / 16.0f, 1.0f /
16.0f };
static const float Weights7[4] = { 20.0f / 64.0f, 15.0f / 64.0f, 6.0f /
64.0f, 1.0f / 64.0f };
static const float Weights9[5] = { 70.0f / 256.0f, 56.0f / 256.0f, 28.0f
/ 256.0f, 8.0f / 256.0f, 1.0f / 256.0f };

float4 Blur5(float4 a, float4 b, float4 c, float4 d, float4 e, float4 f,
float4 g, float4 h, float4 i)
{
    return Weights5[0] * e + Weights5[1] * (d + f) + Weights5[2] * (c +
g);
}
float4 Blur7(float4 a, float4 b, float4 c, float4 d, float4 e, float4 f,
float4 g, float4 h, float4 i)
{
    return Weights7[0] * e + Weights7[1] * (d + f) + Weights7[2] * (c +
g) + Weights7[3] * (b + h);
}
float4 Blur9(float4 a, float4 b, float4 c, float4 d, float4 e, float4 f,
float4 g, float4 h, float4 i)
{
    return Weights9[0] * e + Weights9[1] * (d + f) + Weights9[2] * (c +
g) + Weights9[3] * (b + h) + Weights9[4] * (a + i);
}
#define BlurPixels Blur9

// 16x16 pixels with an 8x8 center that we will be blurring writing out.
Each uint is two color channels packed together
groupshared uint CacheR[128];
groupshared uint CacheG[128];
groupshared uint CacheB[128];
groupshared uint CacheA[128];

void Store2Pixels(uint index, float4 pixel1, float4 pixel2)
{
    CacheR[index] = f32tof16(pixel1.r) | f32tof16(pixel2.r) << 16;
    CacheG[index] = f32tof16(pixel1.g) | f32tof16(pixel2.g) << 16;
    CacheB[index] = f32tof16(pixel1.b) | f32tof16(pixel2.b) << 16;
    CacheA[index] = f32tof16(pixel1.a) | f32tof16(pixel2.a) << 16;
}

void Load2Pixels(uint index, out float4 pixel1, out float4 pixel2)
```

```
{
    uint rr = CacheR[index];
    uint gg = CacheG[index];
    uint bb = CacheB[index];
    uint aa = CacheA[index];
    pixel1 = float4(f16tof32(rr), f16tof32(gg), f16tof32(bb),
f16tof32(aa));
    pixel2 = float4(f16tof32(rr >> 16), f16tof32(gg >> 16), f16tof32(bb
>> 16), f16tof32(aa >> 16));
}

void Store1Pixel(uint index, float4 pixel)
{
    CacheR[index] = asuint(pixel.r);
    CacheG[index] = asuint(pixel.g);
    CacheB[index] = asuint(pixel.b);
    CacheA[index] = asuint(pixel.a);
}

void Load1Pixel(uint index, out float4 pixel)
{
    pixel = asfloat(uint4(CacheR[index], CacheG[index], CacheB[index],
CacheA[index]));
}

// Blur two pixels horizontally.  This reduces LDS reads and pixel
unpacking.
void BlurHorizontally(uint outIndex, uint leftMostIndex)
{
    float4 s0, s1, s2, s3, s4, s5, s6, s7, s8, s9;
    Load2Pixels(leftMostIndex + 0, s0, s1);
    Load2Pixels(leftMostIndex + 1, s2, s3);
    Load2Pixels(leftMostIndex + 2, s4, s5);
    Load2Pixels(leftMostIndex + 3, s6, s7);
    Load2Pixels(leftMostIndex + 4, s8, s9);

    Store1Pixel(outIndex, BlurPixels(s0, s1, s2, s3, s4, s5, s6, s7,
s8));
    Store1Pixel(outIndex + 1, BlurPixels(s1, s2, s3, s4, s5, s6, s7, s8,
s9));
}

void BlurVertically(uint2 pixelCoord, uint topMostIndex)
{
    float4 s0, s1, s2, s3, s4, s5, s6, s7, s8;
    Load1Pixel(topMostIndex, s0);
    Load1Pixel(topMostIndex + 8, s1);
    Load1Pixel(topMostIndex + 16, s2);
    Load1Pixel(topMostIndex + 24, s3);
    Load1Pixel(topMostIndex + 32, s4);
    Load1Pixel(topMostIndex + 40, s5);
    Load1Pixel(topMostIndex + 48, s6);
    Load1Pixel(topMostIndex + 56, s7);
    Load1Pixel(topMostIndex + 64, s8);

    Output[pixelCoord] = BlurPixels(s0, s1, s2, s3, s4, s5, s6, s7, s8);
}

[numthreads(8, 8, 1)]
void CSMain(uint3 Gid : SV_GroupID, uint3 GTid : SV_GroupThreadID, uint3
DTid : SV_DispatchThreadID)
```

```
{
    uint inputWidth = 0;
    uint inputHeight = 0;
    Output.GetDimensions(inputWidth, inputHeight);

    // Load 4 pixels per thread into LDS
    int2 GroupUL = (Gid.xy << 3) - 4; // Upper-left pixel coordinate of
group read location
    int2 ThreadUL = (GTid.xy << 1) + GroupUL; // Upper-left pixel
coordinate of quad that this thread will read
    // Store 4 unblurred pixels in LDS
    int destIdx = GTid.x + (GTid.y << 4);

    if (Upsample)
    {
        float2 uvUL = (float2(ThreadUL) + 0.5) * float2(1.0f /
inputWidth, 1.0f / inputHeight);
        float2 uvLR = uvUL + float2(1.0f / inputWidth, 1.0f /
inputHeight);
        float2 uvUR = float2(uvLR.x, uvUL.y);
        float2 uvLL = float2(uvUL.x, uvLR.y);

        Store2Pixels(destIdx + 0, Input.SampleLevel(BilinearSampler,
uvUL, 0.0f), Input.SampleLevel(BilinearSampler, uvUR, 0.0f));
        Store2Pixels(destIdx + 8, Input.SampleLevel(BilinearSampler,
uvLL, 0.0f), Input.SampleLevel(BilinearSampler, uvLR, 0.0f));
    }
    else
    {
        Store2Pixels(destIdx + 0, Input[ThreadUL + uint2(0, 0)],
Input[ThreadUL + uint2(1, 0)]);
        Store2Pixels(destIdx + 8, Input[ThreadUL + uint2(0, 1)],
Input[ThreadUL + uint2(1, 1)]);
    }

    GroupMemoryBarrierWithGroupSync();

    uint row = GTid.y << 4;
    BlurHorizontally(row + (GTid.x << 1), row + GTid.x + (GTid.x & 4));
    GroupMemoryBarrierWithGroupSync();
    BlurVertically(DTid.xy, (GTid.y << 3) + GTid.x);
}
```