CA Report: Hardware Ray-Tracing Assisted Hybrid Rendering Pipeline for Games

Kim-Chan Tze Hui, Louiz
School of Computing, National University of Singapore, Singapore
e0176546@u.nus.edu

Table of Contents

ABSTRACT

This project creates a Rendering Pipeline that aims to combine rasterization and raytracing techniques to achieve better visual quality than traditional rasterization techniques at acceptable framerates. The technology stack and base code framework are described, followed by a literature review of algorithms used. This is followed by a report of the work that has been done, which includes the migration of the base raytracing framework, integration of a denoiser from the literature, new ease-of-use features, GBuffer optimization, as well as rasterized shadow passes and lighting passes. Finally, the work to be done in the upcoming semester is discussed.

CCS CONCEPTS

• Computer Methodologies • Computer Graphics • Rendering • Ray tracing • Rasterization

KEYWORDS

Visual Computing, Computer Graphics, Computer Games, Machine Learning

1 INTRODUCTION

This project aims to build a simple lightweight hybrid rendering pipeline. This will combine techniques from the traditional rasterization pipeline with hardware accelerated raytracing to achieve real-time rendering suitable for games, with visual quality improvements over the traditional rasterization pipeline.

1.1. Motivation

Rasterization is the traditionally used technique for the generation of computer graphics in real-time applications, while raytracing can generate more photorealistic images, but at a greater processing cost, and is thus often reserved for offline rendering use [13]. In recent years however, Nvidia's Turing GPU architecture, which sport RT Cores that accelerate bounding volume hierarchy traversal and ray-triangle intersection testing, has made it viable to utilize raytracing in real-time applications [12].

As part of NUS's raytracing research, novel techniques for the post-processing effects *motion blur* and *depth-of-field* (as well as other possible effects in the future), which make use of both hardware accelerated ray-tracing and rasterization techniques, are currently being developed/improved. The main purpose of *this* project is that of implementation - to build a framework that will be used to implement a rendering pipeline that makes use of various rasterization and raytracing techniques, along with NUS's novel post-processing effects.

For the scope of this project, I do not expect to "complete" the entire pipeline, but this project should continue to be built on over multiple years as part of NUS's raytracing research.

1.2. Technology Stack

1.2.1. Hardware and API

The premise of the project relies on hardware-accelerated raytracing. We are making use of Nvidia's Turing hardware (e.g. RTX series) which supports this. The implementation is built on Nvidia's Falcor framework, which is an abstraction layer on top of DirectX 12 and DirectX Raytracing (DXR). As DirectX is a low-level API, the use of Falcor greatly speeds up prototyping implementation. It provides numerous features, the most relevant of which include:

- DirectX context creation and management
- Window and input handling
- Shader program compilation and management

- Simple-to-use CPU-GPU data transfer
- 3D asset management
  - 3D scene file input/output
  - Construction of DXR acceleration structures
  - Convenient interface for toggling animation, camera movement, light/material properties
- Easy to use GUI framework

1.2.2. Base Code Framework

Even so, building a modular rendering pipeline, even on top of Falcor, would take a very long time. As such, our implementation takes its base code from the following DirectX Raytracing tutorial [4]. The tutorial makes use of a `RenderingPipeline` framework which was built on top of Falcor to provide a further abstraction layer. This framework's core engine comprises:

- `RenderingPipeline`
  - The "main" C++ Falcor program
  - Stores a series of `RenderPass`'s, and renders the selected 3D scene by executing the active passes in sequence
- `RenderPass`
  - Where rendering logic takes place
  - Retrieves scene and textures (textures form both inputs and rendering targets for passes) from `ResourceManager`, and executes shader programs to perform rendering tasks
- `ResourceManager`
  - Stores the state of the pipeline as a whole
  - Used by `RenderPass`'s to retrieve scene information and textures
    - Conceptually, it acts as a means of communication between `RenderPass`'s, as textures, which are the main rendering targets/inputs, are all managed by the `ResourceManager` Although these are modified heavily in my implementation, the structure still holds.

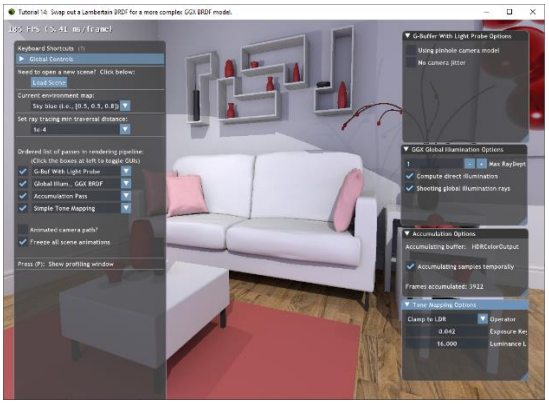

Figure 1: Original `RenderingPipeline` *framework. The C++ file running is the* `RenderingPipeline`*, where you can toggle/select the* `RenderPass`*'s from the options on the left (checkboxes and dropdowns). Each* `RenderPass` *has its own GUI window (the four pass's corresponding windows appear on the right).*

An alternative design choice would be to build on top of Falcor's native rendering pipeline framework, Mogwai. However, this pipeline would involve compiling each render pass as a separate C++ library, and pipelines are constructed using a Render Graph Editor which builds Python scripts to store the pipeline information. I felt that this

would be more unwieldy and was not as flexible. It also does not provide the same level of abstraction as the `RenderingPipeline` framework, which would mean that we either must work with more code repetition, or re-implement that abstraction.

Additionally, the previous NUS research work had already been built using the original `RenderingPipeline` framework (using the older version of Falcor, 3.1.0). Hence, it would be more convenient and lower the cost of integration to simply build on this framework.

2 LITERATURE REVIEW

In this section, we explore some of the inspiration for the chosen rendering passes used in our main rendering pipeline, as well as algorithms used.

2.1. Rendering Passes

The current selected passes in the rendering pipeline is heavily inspired by Unity's High-Definition Rendering (HDRP) pipeline [1,2]. To summarize, this is a deferred shading pipeline. A GBuffer that stores geometry, material, and various other information is created (with rasterization), then lighting effects such as directional/area shadows, ambient occlusion, reflections and indirect diffuse lighting are computed with raytracing. The lighting effects are combined to form the shaded image in the deferred lighting pass. After that, there are several more steps that are out of scope for my project currently.

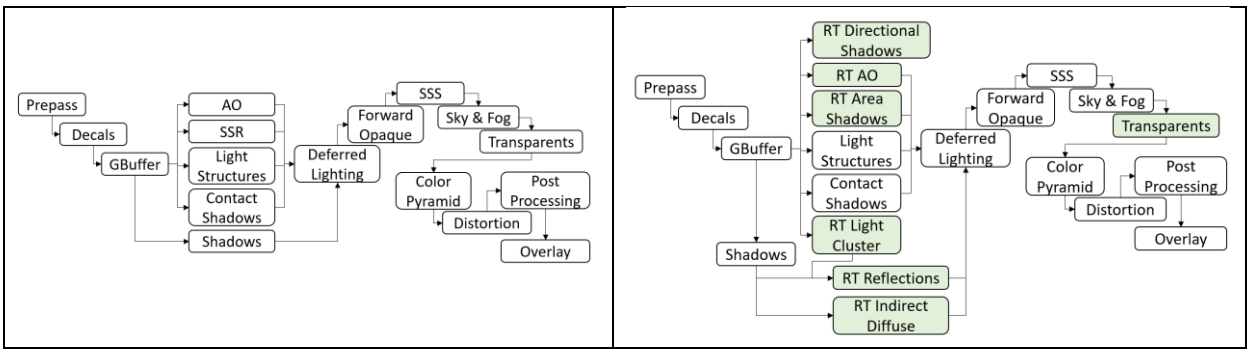

Figure 2: Unity's HDRP render pipeline [1]
Left: Simplified rasterization pipeline. Right: Hybrid pipeline, raytraced steps in green.

2.2. Denoiser

When rendering an image via raytracing, we typically require dozens to hundreds of samples per pixel to achieve noise-free results. To achieve framerates of 30Hz with a 1920×1080 viewport however, we are limited to just a few rays per pixel due to current hardware limitations [14]. As such, reconstruction filters which can denoise low samples-per-pixel frames are necessary to achieve non-noisy ray-traced renders with passable visual-quality in real-time, without which we can only get images similar to the top-right image in Figure 7.

For our main lighting computations (direct and indirect lighting), we make use of the Spatiotemporal Variance-Guided Filter (SVGF) by Schied et. al. [14]. In essence, this filter computes the incident light of each pixel as a weighted average across both space and time:

- Spatially, convolutions (with five passes of a 5×5 filter kernel) are performed with neighbouring pixels, such that each pixel rendered is a weighted average of its neighbours

    - The weight of each neighbour is a function of the difference in luminance, depth, and normal with respect to the pixel, normalized by the variance of that pixel's luminance variance. This makes it so that pixels that are similar in color and geometry are weighted more strongly in the average. This avoids smoothing across edges.
- Temporally, we render an exponential moving average of the current pixel's world position
    - This requires that we store "motion vectors" in our GBuffer, which tell us the previous pixel location of the pixel's current world position being rendered. If the depth, normal, or mesh ID is inconsistent, the temporal history is discarded, and the pixel is rendered as is.

For other effects, such as ambient occlusion (AO), we may use more lightweight denoisers. The example in [10] which demonstrates denoised AO uses a single pass 3×3 filter instead of a five pass 5×5 filter, which still works well since most pixels will have a high number of temporal samples per pixel. (I have yet to implement an AO pass for our rendering pipeline, but this will be kept in view)

2.3. Rasterized Lighting Passes

For the sake of proof-of-concept and comparisons with traditional rasterization techniques, in terms of visual quality comparison and performance profiling, traditional rasterization-based lighting algorithm implementations were investigated as well. It is possible that in the future, these passes will be used in tandem with the raytraced pipeline, but for now it is separate and only serves for comparison. For the lighting implementation, currently only directional lights and point lights are currently supported (e.g. no area lights).

2.3.1. Point Lights

To render point lights, it was necessary to implement omnidirectional shadow mapping. There are two shadow mapping methods that were considered – cube shadow mapping and dual-paraboloid shadow mapping (DPSM). For the implementation in the project, cube shadow mapping was chosen. The main reasons are that DPSM is more computationally expensive, as it depends on non-linear transformations, and the image quality is dependent on the occluding objects themselves and cannot match that of cube shadow maps [9].

The process for cube mapping is detailed in [11]. In short, for each point light in the scene, a cube shadow map is produced by rendering the scene with the light as the origin six times, one for each direction (±x, ±y, ±z). The shadow map only stores the squared distance of rasterized points' world position to the light's position, which saves on the square root operation. When performing the shading pass (after all cube shadow maps have been filled), at each point, we check for light contribution from each light by comparing the squared distance of the light to that point with the squared distance in the direction of the point (from the light's point of view) in the shadow map, and if it is greater than that in the shadow map, the object is in shadow, and the light does not illuminate that point.

A minor optimization is that instead of cube maps, one can use a single texture to store the 6 view directions. This increases the ease of filtering and is detailed in [7].

2.3.2. Directional Lights

For the implementation of directional lights, a single orthogographically projected shadow map is generally insufficient due to the jagged shadow artifacts. As such, we looked at cascaded shadow maps (CSM) as suggested in [8]. As the purpose of the rasterized lighting implementation is not to make a highly robust and customizable rasterization pipeline, but rather for comparison purposes with the hybrid raytracing pipeline, currently, the implementation makes use of a rather simple implementation based on the tutorial in [5]. Falcor itself comes with a CSM implementation for use with Mogwai but was too complex to integrate with our current implementation in the

given time, and hence the implementation of the simpler version. It should be noted that in our implementation, functions for computing frustum bounds were inspired by Falcor's implementation.

For the current implementation, in summary, far-plane distances are specified, such that at different distances from the camera, a different shadow map is used. At closest distances, the most high detailed shadow map is used (covering the smallest area, which provides the relative best resolution), while the furthest distances make use of the lowest detailed shadow maps (that cover the largest areas), since less detail is visible to begin with, so the loss in quality is not as observable.

3 SYSTEM DESIGN

The system design that is being aimed towards is one similar to Unity's HDRP [1,2]. For now, the system is at its early stages and is not yet comparable, however. The current pipeline includes/intends to include:

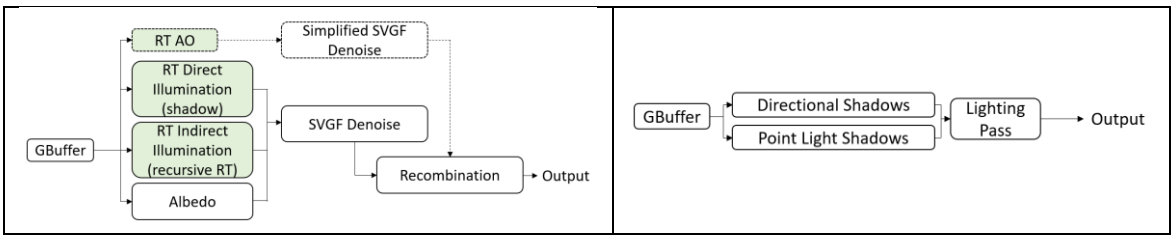

Figure 3: Rendering pipelines. Dotted boxes indicate that passes are yet to be implemented. Left: Hybrid render pipeline currently available (excluding dotted). Right: Current purely rasterized pipeline.

3.1. Hybrid Pipeline

For the hybrid rendering pipeline, we compute the albedo of each shaded point, as well as separate direct lighting and indirect lighting.

For direct lighting, we select a single random light from the scene (directional or point) to perform the computation by casting a shadow ray in the direction of the light (one ray per frame). This generates a very noisy image that must be denoised. For indirect lighting, we shoot a recursive ray in a random direction either in the GGX specular lobe or the diffuse lobe – the lobe selection is done with probability based on the specular value – and the contribution of this recursive ray is scaled by the probability of sampling this ray. This shading computation is based on the sample code from [4], and the sampling method/lighting equations have not been modified, despite code refactoring.

Integrating ambient occlusion into the hybrid pipeline has been looked into but it is not complete.

3.2. Rasterized Pipeline

For the implementation of the rasterized pipeline for comparison purposes/possible future integration with the hybrid pipeline, we simply have separate passes for the computation of shadow maps (for directional/point lights), and perform deferred shading with reference to those shadow maps. More details will be discussed in section 5.4.

4 DEVELOPMENT SUMMARY

The development of the pipeline thus far can be broken into stages as follow:

1. Migrate `RenderingPipeline` to latest version of Falcor
2. Migrate the rendering passes to integrate with the updated `RenderingPipeline`
3. Integration of SVGF denoiser
4. Creation of Rasterized Lighting Pass

In addition, ease-of-use features/optimizations were made:

- Preset Pipelines
- Automatic GUI alignment/“show all” option
- GBuffer optimization

5 PROGRESS AND ISSUES

5.1 Migration of `RenderingPipeline` Framework

As mentioned in 1.2.2, this project was implemented on top of a framework that was built on top of Falcor [4], which is also the framework used by previous NUS raytracing research. The first part of the project implementation involved migrating the framework from Falcor 3.1.0 to the then-latest version, Falcor 4.1.

In general, these were done without much issue (for detailed writeups, refer to PR #4, #2, #6). The main changes that needed to be made with the version change were:

- Animation handling changes
- Changes in Falcor interface names and virtual function arguments
- Combination of RTScene and (rasterized) Scene
- GUI API changes

5.1.1. Ease-of-Use: Preset Pipelines

Toggling between different pipeline arrangements for comparison purposes, which may involve simply swapping a single pass or swapping multiple passes, can take many clicks. As such, a presets feature was added to the `RenderingPipeline` framework that allows the user to specify a sequence of passes as a preset programmatically and select it from the preset dropdown menu. Details can be found in PR #17.

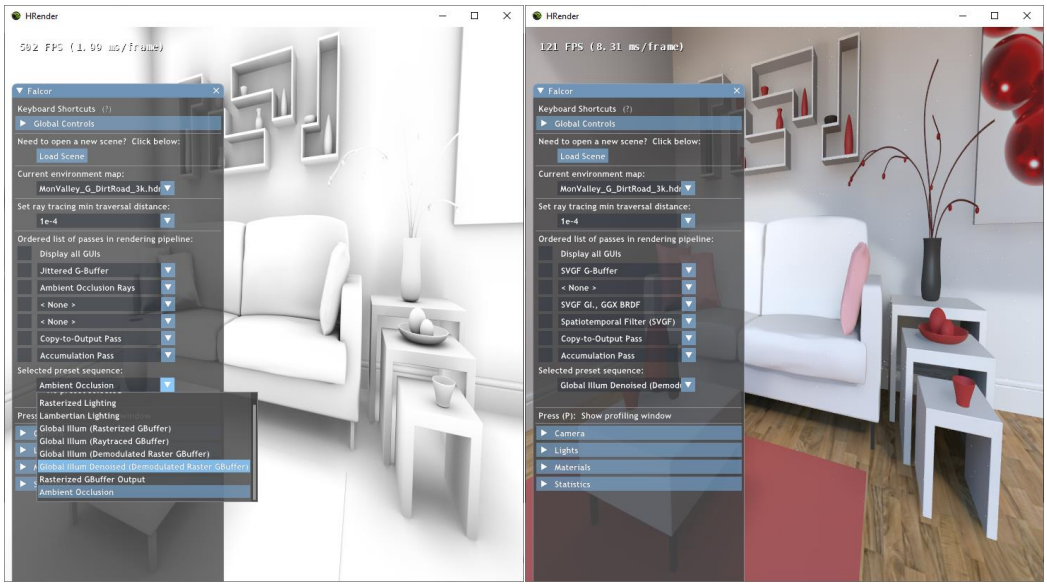

Figure 4: From left to right, a demonstration of preset pipeline selection.
Left: The dropdown menu option is clicked. Right: Multiple passes are selected for the `RenderingPipeline` at once automatically, based on the programmed preset.

5.1.2. Ease-of-Use: GUI Auto-Toggle

Another, more minor, ease-of-use feature created was automatic rearrangement and toggling of GUI windows. When using the original `RenderingPipeline` framework, to open all GUI windows for toggling active passes’ options,

the user would have to manually check off each one individually. To save time, a checkbox that would toggle all windows at once was added. Over time, this saves a lot of clicks. When checked, all active passes' windows will open and be aligned. When unchecked, all windows will be hidden. This is shown in Figure 5. (Details: PR #17, #22)

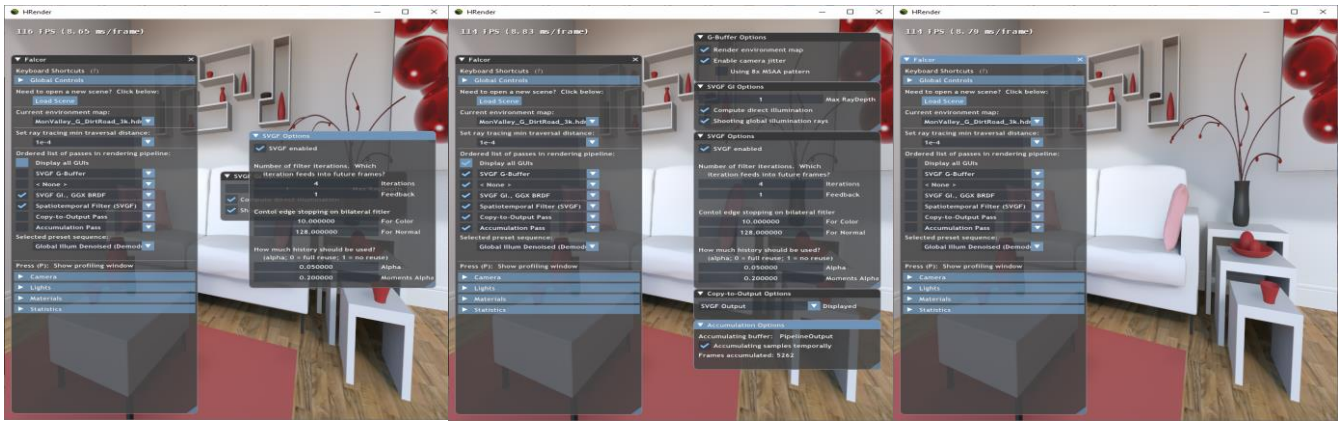

Figure 5: From left to right, a demonstration of the "Display all GUIs" toggle. Left to middle: Checkbox is checked, and all active passes' windows are opened and aligned. Middle to right: Checkbox is unchecked, and all windows are hidden.

5.2. Migration of Rendering Passes

The framework came with a few implementations of commonly used raytracing algorithms (not including denoising, so they are not useable for real-time rendering). A significant portion of time was put into this portion, migrating the passes' shaders to work with Falcor 4.1, and their C++ portions to work with Falcor 4.1 as well as the updated `RenderingPipeline` framework.

Details can be found in (PR #9, #10, #11, #12), but the main changes were:

- Changes to the wrappers around Falcor Shader program abstractions and scene rendering
- Changes to Falcor's shader libraries/implementation
- New steps to be performed on the shaders prior to scene rendering

In addition, changes were made such that features not available in some of the provided implementations were created, e.g. environment mapping in the regular rasterized GBuffer (version that does not support SVGF denoising) and emissive materials in GBuffer implementations other than the `LightProbeGBuffer` (the SVGF and non-SVGF rasterized GBuffers) (PR #22).

With this migration, we have passes that can compute effects for (without denoising):

- Rasterized/raycasted (non-recursive raytracing) GBuffer with camera jitter, and thin-lens distortion for raycasted GBuffer pass
- Simple temporal accumulation pass
- Raytraced ambient occlusion
- Simple raytraced local Lambertian (diffuse) shading (with shadows)
- Recursive raytraced global illumination (using GGX BRDF [3])

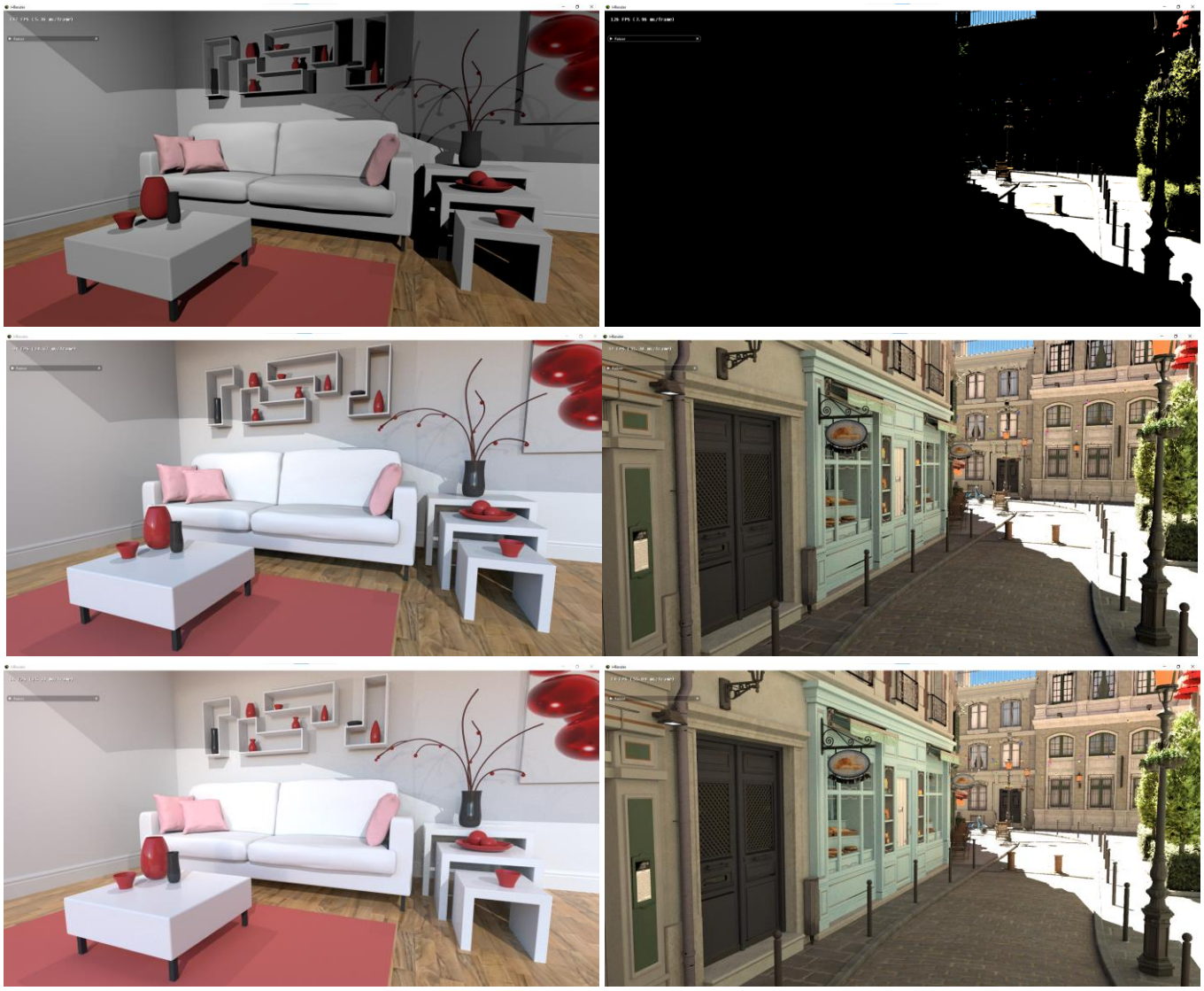

Figure 6: Sample Screenshots of the raytraced passes. Temporal averaging of hundreds/thousands of frames is used to reach convergence to a non-noisy image for the bottom two rows.
Top: Lambertian (diffuse, non-specular) shading pass (no global illumination). Middle: GGX (diffuse and specular) global illumination shading pass with 1 recursive ray (temporal average). Bottom: GGX global illumination shading pass with 2 recursive rays (temporal average).

5.2.1. GBuffer Compaction

One optimization made to the original implementation was compacting multiple of the GBuffer's output textures to a single texture. This is done to reduce memory cost and improve performance, and due to the limited maximum number of render targets allowed in a single pass (for RTX 2060, this is 8), which becomes a limiting factor when information such as motion vectors, normals, and derivatives etc. are needed for SVGF support.

Four `float4` textures (where the expected range is [0, 1]) were compacted to a single `float4` texture, where each original float is converted to a `uint8`, and four such `uint8`'s are stored in each 32-bit float of the `float4` texture. This way, we pack 16 numbers – diffuse color (RGB), specular color (RGB), emissive color (RGB), opacity, linear roughness, and a few other properties – into the float4 texture. Full details in PR #22.

5.3. Integration of SVGF Denoiser

For denoising of the final images as mentioned in 2.2, the sample code of the denoiser provided with the paper [14] (here) was integrated with the migrated `RenderingPipeline`. The implementation is not perfect – there is an issue that walls aligned with the y-axis (left/right walls) are noisier than expected. I am currently debugging this. Unfortunately, I cannot run the sample code (incompatible versioning issues with Visual Studio/Windows/DirectX, the paper's code is ~3 years old), so I cannot easily confirm if this is a pre-existing issue, or an issue with my migration.

Figure 7: Comparison of with/without SVGF Denoiser.
Top-left: SVGF Filtering. Top-right: No denoising. Bottom: No denoising, temporal average of ~1000 frames.

In addition to refactoring and using the new framework version's functions and abstractions, some modifications were made to the migrated lighting method:

- GBuffer pass (separate from the previously mentioned GBuffer passes, as this needs to support extra information for denoising)
  - Camera jitter (for anti-aliasing)
  - Toggle-able environment map
  - Emissive material support
- Shading pass
  - Allow for variable raytrace recursion depth

5.4. Rasterized Lighting Pass

As mentioned in 2.3, rasterized lighting algorithms for point lights and directional lights were implemented. A directional shadow map (currently, only one directional light per scene is supported) and cube shadow maps (for each point light) are created in shadow passes, then a deferred shading pass makes use of these shadow maps to shade the output frame for the scene.

5.4.1. Point Lights/Omnidirectional Shadow Mapping

The algorithm used for shading point lights was briefly mentioned in 2.3.1 and 3.2. Here we summarize the implementation, but for full details, see the implementation in PR #29, #30.

The shadow map texture's implementation is summarized as follows. Each point light has a single texture that stores its shadow map of width 1024px and height 6*1024px, which simply stores the values of $d^2 = \|\mathbf{posW} - \mathbf{lightPos}\|^2$ at each rasterized pixel (posW is the world coordinate of the pixel). The shadow map performs this rasterization of the scene 6 times (one per axis-aligned direction) for each light, into a viewport of (startX = 0, startY = i*1024, width = 1024, height = 1024) (with i from 0 to 5) on its shadow map.

For the (deferred) shading pass, at each shaded point, we consider each light source. We compute for that light source $\boldsymbol{v} = \mathbf{lightPos} - \mathbf{posW}$, and use $\boldsymbol{v}$ to sample our shadow map as described in [7], to get the shadow map's d^2 value in that direction. If the squared distance, $\boldsymbol{v} \cdot \boldsymbol{v}$, of the shaded point is less than or equal to d^2 (plus a small bias offset factor), we will add the shading contribution of that light source to that pixel (using Lambertian shading for now). We do this for each light source, so we can simulate multiple shadows.

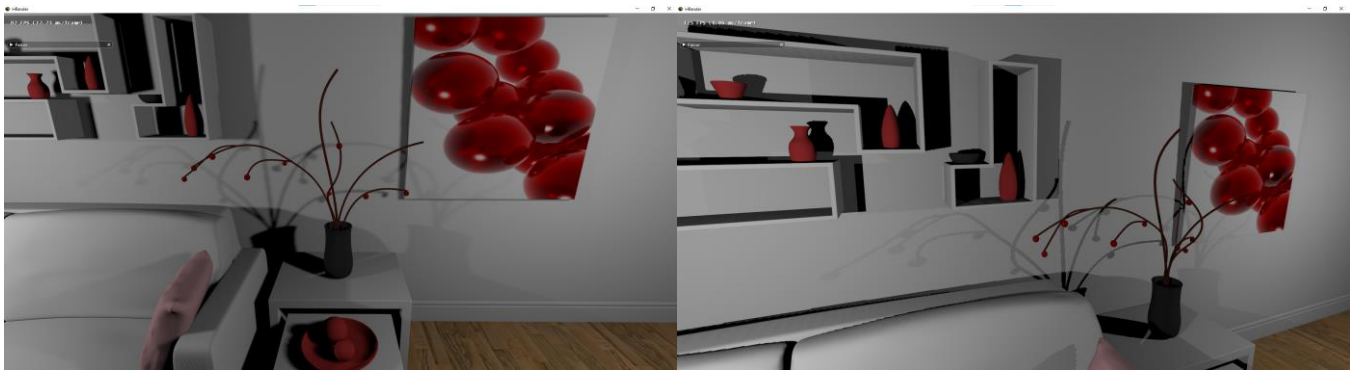

Figure 8: Scene with two point lights.
Left: PCF is used. Right: No PCF is used, jaggies are slightly visible.

Currently, for smoothening of jaggies (and softer shadow edges), we use a rather naïve implementation of PCF mentioned in this tutorial [6]. For each shading point's shadow map sampling, we simply average the result of shadow sampling after offsetting the direction vector (of the point to the light source) by 20 hard-coded offset values. This is not performant and will need to be changed to a more efficient method in a future iteration.

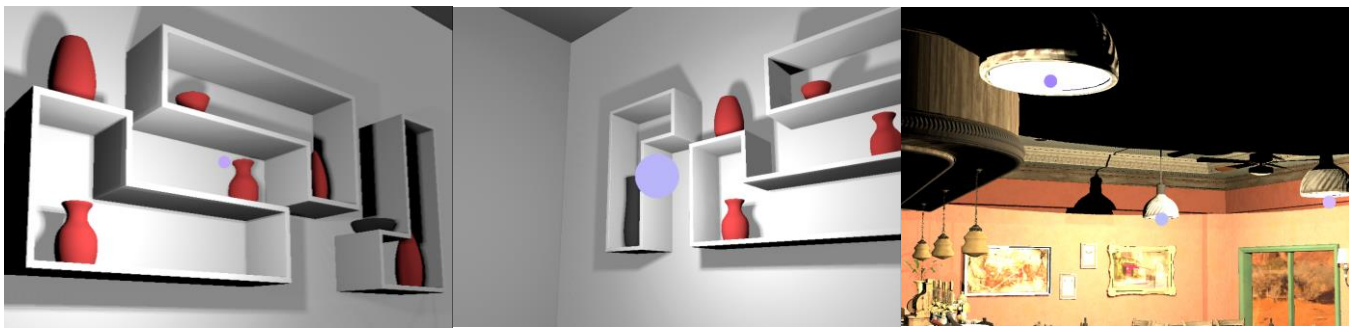

Figure 9: Debug Light Drawing.
Left, middle: The same light/scene from different angles, with different drawn size. Right: Multiple debug lights in the scene.

5.4.1.1. Debug Light Drawing

To aid with debugging of the shadow mapping implementation initially, a simple feature to draw all point lights' positions on the screen was added. It is not made to be performant, so it should be disabled for actual rendering.

For the drawing of debug lights, the GUI lets the user:

- Toggle drawing of debug lights
- Adjust size (radius) of debug lights drawn, l_r

Figure 10: Debug light GUI options.

Since the lights do not have an actual mesh, the rendering is done in the shader by checking the screen space coordinates of the current pixel, and if it is within the l_r of any of the visible point light's screen space positions, we draw a color instead of the original shaded pixel. Screen space positions of the scene's visible point lights are computed on the CPU and sent to the GPU before the frame rendering of this pass, and are updated whenever the scene is updated (e.g. camera moves or light positions are changed). This computation is done by transforming the world space coordinates of each point light to screen space coordinates with the camera's view-projection matrix, and simulating perspective divide and clipping calculations. If the light should be clipped, its location is not recorded, and this light is not counted in the variable for $n_{visibleLights}$ sent to the shader. For full details, refer to PR #29.

5.4.2. Directional Lights/Cascaded Shadow Mapping

For directional lights, a simple CSM implementation was made as described in section 2.3.2. Here we summarize the implementation, but for full details, see the implementation in PR #30.

For now, we only support one directional light per scene. For the scene's directional light (we will not do this if it does not have one), we have a texture array with three (number of cascades is arbitrary, and can be easily modified in the C++ code, but it is not changeable during program execution currently) shadow maps for three "cascades". Each cascade will have the active shadow map for up to the user-specified distances (changeable during execution).

For each cascade, we compute the eye-space coordinates of the view frustum (with near and far plane distances specified per-cascade by the user) by multiplying the clip space view-frustum bounds with the inverse camera view-projection matrix. We create an orthographic view-projection matrix, M_{vp}, facing in the light's direction, centred on the average of the view frustum coordinates, $\boldsymbol{c}_{ave}$, with radius $r = \max(|\boldsymbol{c}_i - \boldsymbol{c}_{ave}|)$ assigned as the width/height/depth of the viewing volume (where $\boldsymbol{c}_i$ takes the value of frustum coordinate $i \in [0,7]$). This way, we compute the three cascade centres, the three cascade radii, as well as three view-projection matrices for the cascades.

For the shadow pass, we rasterize the scene once for each cascade (using their individual view-projection matrices) and store the depth values at each pixel in the shadow map. For the deferred shading pass, for each shaded point (at world-space coordinate posW), we check for the first cascade (in near to far order) that the point is in, i.e. the distance of the shaded point from the cascade center is less than that cascade's radius. For now, blending of cascades is not done, we simply choose the closest cascade. We get a shadow coordinate, $\boldsymbol{c}_{shadow} = M_{vp} * \text{posW}$, and compare the depth of $\boldsymbol{c}_{shadow}$ to the depth value on the shadow map at $\boldsymbol{c}_{shadow}.xy$, and if it is less than or equal to the shadow map's depth value (plus a bias offset), the directional light contributes to that point's shading.

Some miscellaneous functions can be seen in Figure 11 and 12. The user can toggle to preview the cascades for debugging and update them in real-time. The shadow can be 100% dark, or partial. We also have a basic naïve PCF implementation (Gaussian blurring) that can be toggled (a better implementation is to-be-done).

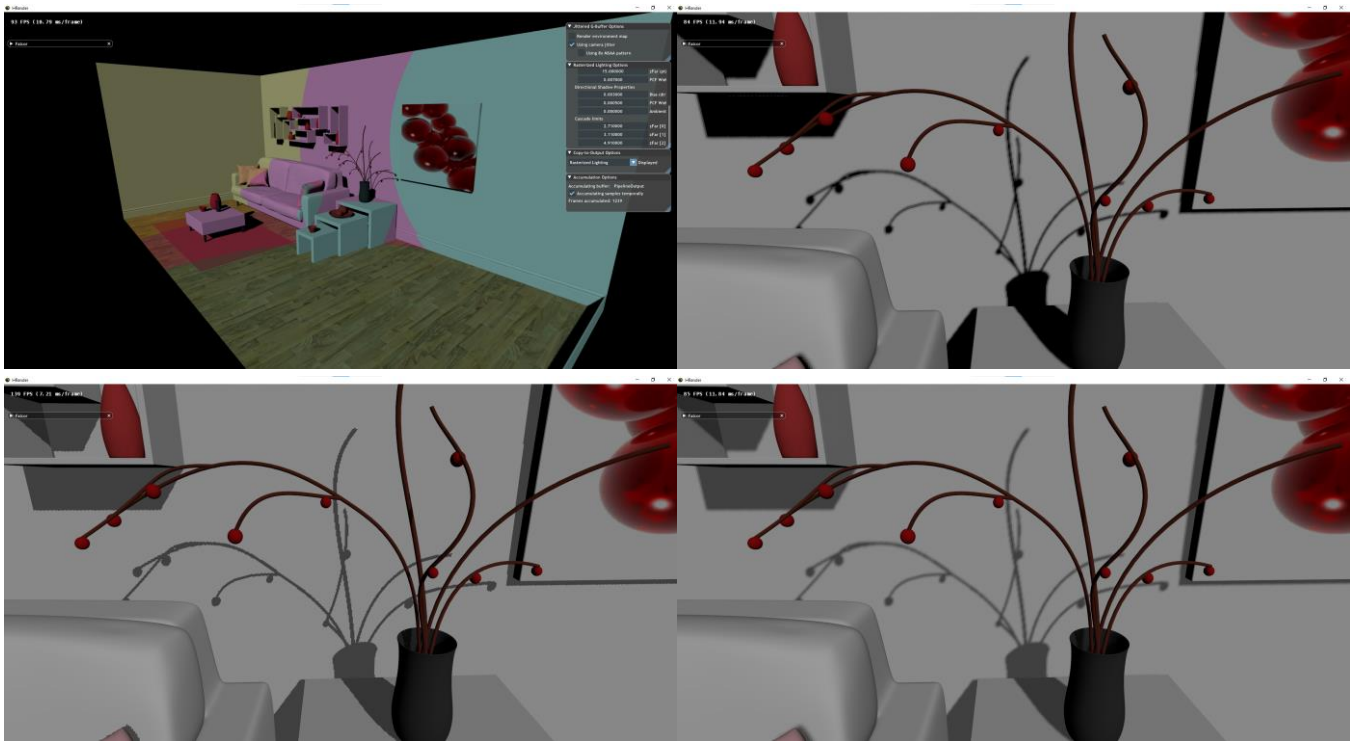

Figure 11: Several screenshots of CSM in action.

Top-left: The user can preview cascade volumes and modify the cascade far-plane distances in real-time. Top-right: Close up of a 100% dark shadow. Bottom-left: The same view as top-right, but with a semi-dark shadow setting. Bottom-right: The same view with PCF turned on – jaggies are less prominent.

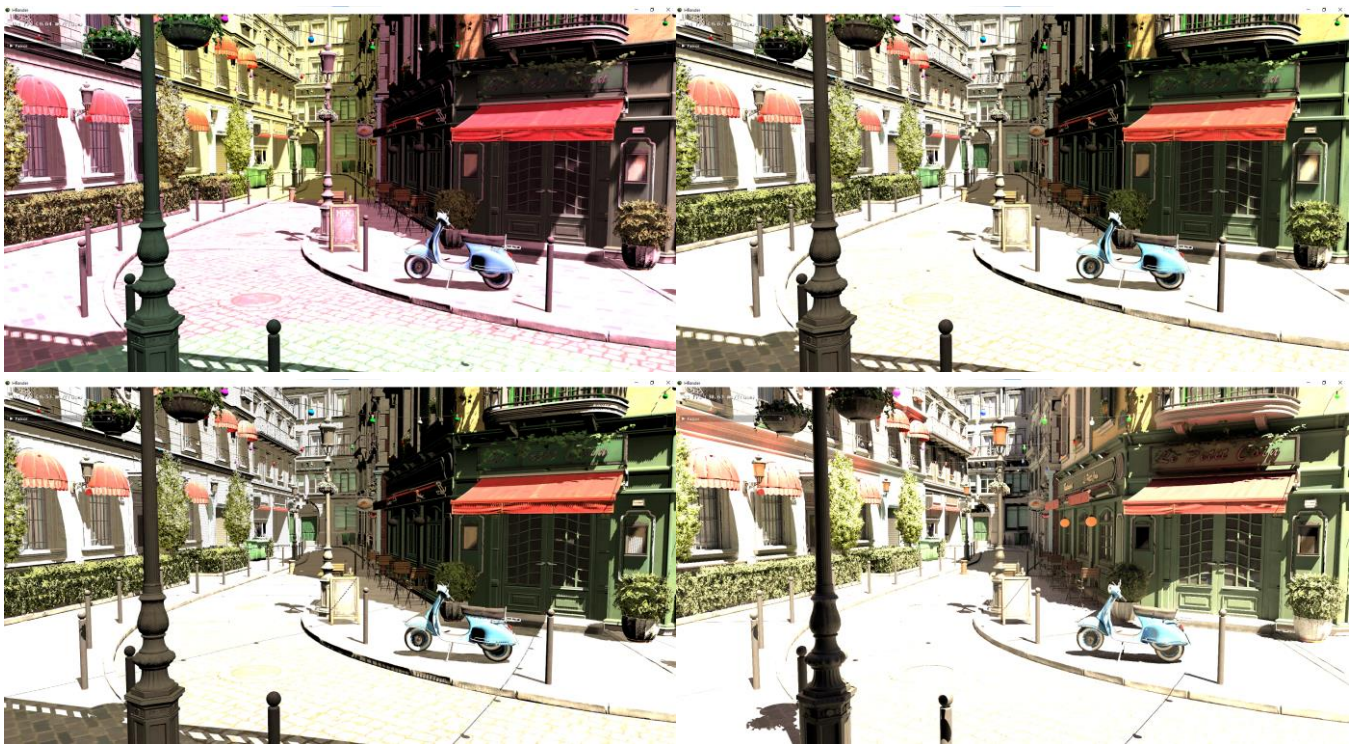

Figure 12: A larger scene with CSM in action.

Top-left: Cascade volume preview. Top-right: same view, without the debugging volume preview, PCF on. Bottom-left: The same shadow as top-right, but PCF off – shadows are harder and more jagged. Bottom-right: Same scene using raytravced global illumination for comparison.

5.4.3. Miscellaneous Points/Difficulties Faced

- A minor optimization to save computation cost is that we only recompute the shadow maps (i.e. perform the shadow pass) when the scene is modified.
- Although an attempt was made for a slight optimization by storing the omnidirectional shadow maps in a single texture each instead of a texture array, this optimization does not cause any significant difference in performance.
- Shadow Peter Panning is difficult to overcome. One method to overcome it is to enable front-face culling for the shadow passes [8], but I have had difficulty with doing so with Falcor/the current framework.
- The texture creation and usage for render target purposes required functionality not supported by the `ResourceManager`, and the `ResourceManager` was generally bypassed. If time permits, it may be good to extend the `ResourceManager`'s functionality to support these.

6 FUTURE WORK

As mentioned throughout the report, there are a fair bit of places that require further work/tweaking. These include:

- Better PCF algorithms for rasterized shadows (directional and omnidirectional)
- SVGF Denoiser debugging (noisy y-plane aligned walls)
- Front face culling for shadow mapping to improve Peter Panning
- Geometry shader
    - Instead of rendering the scene 6 times from the CPU side for omnidirectional shadow maps, try to do so using a geometry shader. Profile the performance differences and choose the more performant one.
- Denoised Ambient Occlusion pass

There are a lot of other potential areas to explore, but I would like to prioritize the improvements and debugging first. I then want to implement and integrate a denoised ambient occlusion pass for the pipeline.

After that, I would like to look into increasing the level of control the user has over our GGX lighting pass, e.g. customizable number of rays per pixel (currently, we always shoot a single shadow ray and a single recursive ray), which will build towards adaptively varying the number of rays needed depending on the rendered image's properties.

Something that eventually needs to be added, but may be too much for my current project, is support for the other kinds of lights (e.g. area lights and spotlights), and support for transparent objects. Eventually, we may also like to investigate other optimizations such as baking.

7 REFERENCES

[1] Anis Benyoub. 2019. Leveraging Real-Time Ray Tracing To Build A Hybrid Game Engine. Retrieved October 30, 2020 from http://advances.realtimerendering.com/s2019/Benyoub-DXR%20Ray%20tracing-%20SIGGRAPH2019-final.pdf

[2] Anis Benyoub. 2020. High Definition Render Pipeline real-time ray tracing is now in Preview. Retrieved October 30, 2020 from https://blogs.unity3d.com/2020/03/06/high-definition-render-pipeline-real-time-ray-tracing-is-now-in-preview/

[3] Bruce Walter, Stephen R. Marschner, Hongsong Li, and Kenneth E. Torrance. 2007. Microfacet models for refraction through rough surfaces. In Proceedings of the 18th Eurographics conference on Rendering Techniques (EGSR'07). Eurographics Association, Goslar, DEU, 195–206.

[4] Chris Wyman. 2019. A Gentle Introduction to DirectX Raytracing. Retrieved October 30, 2020 from http://cwyman.org/code/dxrTutors/dxr_tutors.md.html

[5] Etay Meiri. n.d. Tutorial 49: Cascaded Shadow Mapping. Retrieved October 30, 2020 from http://ogldev.atspace.co.uk/www/tutorial49/tutorial49.html

[6] Joey de Vries. n.d. LearnOpenGL – Point Shadows. Retrieved November 3, 2020 from https://learnopengl.com/Advanced-Lighting/Shadows/Point-Shadows

[7] Kosmonaut's Blog. 2017. Shadow Filtering for Pointlights. Retrieved October 30, 2020 from https://kosmonautblog.wordpress.com/2017/03/25/shadow-filtering-for-pointlights/

[8] Microsoft Docs. 2020. DirectX Technical Articles - Common Techniques to Improve Shadow Depth Maps. Retrieved November 3, 2020 from https://docs.microsoft.com/en-us/windows/win32/dxtecharts/common-techniques-to-improve-shadow-depth-maps

[9] Navrátil, Jan & Kobrtek, Jozef & Zemcík, Pavel. (2012). A Survey on Methods for Omnidirectional Shadow Rendering. 20.

[10] Peter Kristof. 2019. D3D12 Raytracing Real-Time Denoised Ambient Occlusion sample. Retrieved October 30, 2020 from https://github.com/microsoft/DirectX-Graphics-Samples/tree/master/Samples/Desktop/D3D12Raytracing/src/D3D12RaytracingRealTimeDenoisedAmbientOcclusion

[11] Philipp S. Gerasimov. 2004. GPU Gems – Chapter 12. Omnidirectional Shadow Mapping. Retrieved October 30, 2020 from https://developer.download.nvidia.com/books/HTML/gpugems/gpugems_ch12.html

[12] Rick Champagne. 2020. Real-Time Ray Tracing Realized: RTX Brings the Future of Graphics to Millions. Retrieved October 30, 2020 from https://blogs.nvidia.com/blog/2020/08/25/rtx-real-time-ray-tracing/

[13] Sabino T.L., Andrade P., Gonzales Clua E.W., Montenegro A., Pagliosa P. (2012) A Hybrid GPU Rasterized and Ray Traced Rendering Pipeline for Real Time Rendering of Per Pixel Effects. In: Herrlich M., Malaka R., Masuch M. (eds) Entertainment Computing - ICEC 2012. ICEC 2012. Lecture Notes in Computer Science, vol 7522. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-33542-6_25

[14] Schied, Christoph & Salvi, Marco & Kaplanyan, Anton & Wyman, Chris & Patney, Anjul & Chaitanya, Chakravarty & Burgess, John & Liu, Shiqiu & Dachsbacher, Carsten & Lefohn, Aaron. (2017). Spatiotemporal variance-guided filtering: real-time reconstruction for path-traced global illumination. 1-12. 10.1145/3105762.3105770.