Lightcuts

KEDADRY Yannis

***Abstract*—This report presents an implementation of the Lightcuts framework [Wal+05b]; a scalable algorithm to approximate illumination from many lights at a sublinear cost. Our implementation is based on the modern Vulkan API and can handle both point lights and directional lights. Results are presented for simple and complex scenes to show how Lightcuts can approximate hundreds of point lights with a few dozen shadow rays.**

I. Introduction

Lights play an important role in making a scene look more realistic. Because the human brain is really good at spotting gradients, a scene without light interactions like shadows and reflections will then appear strange and unrealistic because it will lack these gradient effects produced by the light sources. The Rendering equation from J. Kajiya **[Kaj86]** models these interactions using integrals and is commonly approximated in any renderer using different algorithms. However, to represent accurately approximate the integrals from the Rendering equation; many discrete samples are required. Because complex scenes are made of thousands or millions of light sources, they also need as many more samples to render. In typical systems, rendering cost then increases linearly with the number of lights.

To reduce the time complexity, multiple solutions have been proposed in the literature. Shirley et al. **[SWZ96]** divided scenes into cells and split the lights into important and unimportant lists. Knowing in which of the lists a light is, they can adapt the samples and have more sparse rays for light in the second list. This technique and other Monte Carlo methods can perform well depending on the quality of the sampling probability functions. However, efficiently and robustly computing these functions for any scene with an arbitrary number of light sources is still challenging. In 2005 B. Walter et al. **[Wal+05b]** proposed a model based on cuts in a tree structure. The idea is to build a tree hierarchy of the lights in the scene. Each node in the tree stores a cluster representative light. They then used dynamic cuts in the tree to create cluster partitioning and only used representative lights of this partitioning for the rendering equation approximation. This method was further enhanced by firstly C. Yuksel **[Yuk19]** who combined the lighting approximation of Lightcuts with stochastic sampling and then by C. Yuksel and D. Lin **[LY20]** where they improved the tree builder and took advantage of the GPU to maximize the performance of light sampling.

In this work, we implement the Lightcuts algorithm using the modern Vulkan API. Our model can handle both point lights and directional lights. We tested our implementation on different scenes and compared the time performances with the brute force approach on a ray tracing pipeline. We also compared the visual aspect of the final scenes using different BRDF models from a Lambert diffusion model to a simpler version of the State of the Art Disney BRDF model [**[Bur12]**, **[Bur15]**] implemented from the T.Li course at UCSD **[Li24]**.

The following of this report is organized as follows: Section II dives into the details of our implementation. In Section III we compare the results of the Lightcuts model with the brute force approach in both the visuals and the complexities. Finally Section IV gives the conclusion and mentions the possible works to further improve the model. All our implementations are accessible through two GitHub pages for the Vulkan Engine and for the Lightcuts implementation based on the engine.

II. Methodology

Our implementation of the Lightcuts algorithm **[Wal+05b]** can be divided into two main components:

- The Lightcuts algorithm overview
- The code

A. Lightcuts

The rendering equation introduced by J. Kajiya **[Kaj86]** computes the radiance L given a surface point x and a view direction $\omega_o = (\theta_o, \phi_o)$.

$$L_k(x, \omega_o) = L_e(\omega_o) + \iint_\Omega L_i(\omega_i) f(\omega_i, \omega_o) \cos(\theta_i)\, d\theta_i d\phi_i \tag{1}$$

Where $\omega_i = (\theta_i, \phi_i)$ is the incoming light, $\Omega = [0, 2\pi] \times [0, \pi/2]$ is the domain, L_e is the emission, L_i is the incoming light, and f is the reflectance. The equation 1 needs to be approximate in a program. Given a set of light sources $\mathcal{S}$, the equation becomes:

$$L_{\mathcal{S}}(x, \omega_o) = \sum_{k \in \mathcal{S}} M_k(x, \omega_o) G_k(x) V_k(x) I_k \tag{2}$$

Where M_k, G_k, V_k, and I_k are respectively the material, the geometric, the visibility, and the intensity terms.

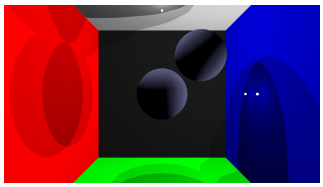
One key observation in equation 2 is its linearity in terms of the size of $\mathcal{S}$; the more lights in the scene, the more heavy the computation of the equation will be. The Lightcuts approach is a scalable sublinear method aiming at reducing this cost by approximating the contribution of a group of lights without having to evaluate each light individually. If we note $\mathcal{C} \subseteq \mathcal{S}$ a group of lights and $j \in \mathcal{C}$ a representative, together they form a cluster. To approximate the equation 2 using this cluster, we rewrite the latter as:

$$L_{\mathcal{C}}(x, \omega_o) = \sum_{k \in \mathcal{C}} M_k(x, \omega_o) G_k(x) V_k(x) I_k \tag{3}$$

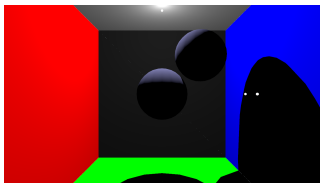
$$L_{\mathcal{C}}(x, \omega_o) \approx M_j(x, \omega_o) G_j(x) V_j(x) \sum_{k \in \mathcal{C}} I_k \tag{4}$$

By precomputing the cluster intensity $I_C = \sum_{k \in C} I_k$ and storing it alongside the cluster itself; the cost of a cluster approximation equals the cost of evaluating a single light.

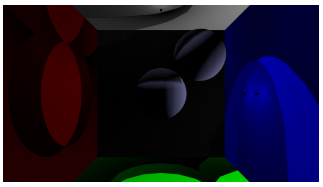
In Figures 1 and 2 we have four different point lights: two at the top and two on the right. We can see the results using different cuts in the light tree. As a basic example, we used the same cut for every point in the scene but for the actual algorithm, we can select different cuts for different points in the scene.



(a) Baseline using the 4 lights

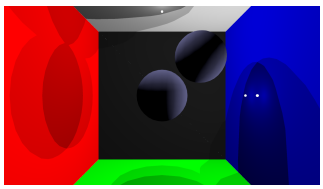


(b) Root as the global cut

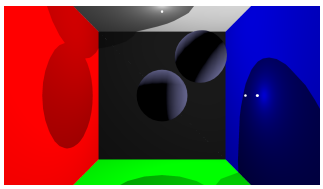


(c) Difference between the baseline and the Lightcuts results

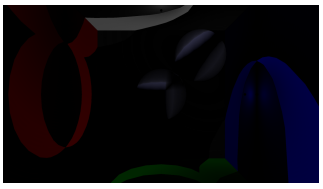
Fig. 1: Comparison between the baseline and cuts with a maximum size of 1



(a) Baseline using the 4 lights



(b) 2 clusters per cut



(c) Difference between the baseline and the Lightcuts results

Fig. 2: Comparison between the baseline and cuts with a maximum size of 2

1) LightTree: While simple in theory, the idea of clustering lights does not work well when really different lights are put in the same cluster. There is no single partitioning of the lights into clusters that is likely to work well on every point in the image. However, trying to dynamically create a cluster for each point would be very expensive. To tackle this issue, B. Walter et al proposed to build a global light tree once (or once per image) and dynamically find the best cut in the tree for every possible light. The light tree is a binary tree where leaves are individual lights. Interior nodes are the clusters representing all the lights in the subtree. The clusters contain its representative light, its total intensity I_C, its two children, an axis-aligned bounding box, and an orientation bounding cone (see Section **??** for details on the latter). Our implementation supports both point lights and directional lights. To create the bounding box for directional lights we use the same idea as presented in their paper which is to consider directional lights as points on the unit sphere. The tree is built from the bottom up. We combine a pair of lights or clusters that create the smallest error using the following metric:

$$I_C(\alpha_C^2 + c^2(1 - cos\beta_C)) \tag{5}$$

In formula 5, α_C is the length of the cluster diagonal, β_C is the half-angle of the bounding cone and c is a constant controlling the relative scaling between spatial and directional similarity. The constant c is important for oriented lights, our system doesn't handle them yet but the barebone of their implementation is in the code for future integration. Another important characteristic of the light tree is that each cluster representative is the same as one of the cluster children's representatives. It is chosen randomly from the two children using their intensities as probabilities:

$$p_{left} = \frac{I_{left}}{I_{left} + I_{right}}$$
$$p_{right} = 1 - p_{left}$$

Choosing the same representative as one of the cluster children helps reduce the cost of cut search (see Section II-A2). However, this method leads to temporal instabilities in dynamic scenes. A solution to this problem is selecting the representative randomly from the lights in the cluster using other probability metrics. This approach is explained in more detail in **[Yuk19]**. However, while using the Vulkan API for our implementation, we did not implement a GPU-based Raytracer and our CPU version does not permit real-time dynamic scenes. For this reason, we focused on the original Lightcuts representative system.

2) Cuts: Once the tree is built we need to have a way to choose cuts for every point in the rendered image. To do so we need to compute reasonably cheap and tight upper bounds on the cluster errors. By cluster error we mean the difference between the exact 3 and approximation 4 of the cluster radiance at a given point and view direction. This error is set to zero when the cluster is a leaf, i.e. a single light. We then compute upper bounds on the material, geometric, and visibility terms and multiply them by the cluster intensity to get the cluster error.

Visibility Term. Because our implementation doesn't yet handle semi-transparent objects and alpha blending, the visibility of a light is either zero or one. However, bounding visibility in arbitrary scenes is a hard problem so we used the same upper bound of one as in the original paper. Visibility of one means that all lights are potentially visible.

Geometric Term. We bound the geometric terms differently depending on the light source (see Table I). Where y_i is the light position. Even if not presented in the original paper, we

Light Type	Point Light	Directional Light
$G_i(x) =$	$\frac{1}{\|y_i - x\|^2}$	1

TABLE I: Geometric bound depending on the type of light

used the closest point in the axis-aligned bounding box for y_i in the approximation.

Material Term. The material term M_i is the BRDF (Bidirectional Reflectance Distribution Function) times the cosine of the angle between the vector $y_i - x$, and the surface normal at x. We first start with the cosine bounding. We could have used a trivial bound of 1 however we decided to go with the tighter bound presented in the original paper.

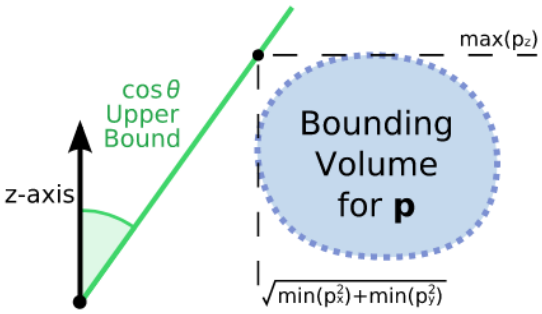


Fig. 3: Lower bound of the angle (upper bound of its cosine) for the bounding volume

Consider the example shown in Figure 3. For each point $p = [p_x, p_y, p_z]$ we have an angle θ between the origin-to-p direction and the z-axis. The cosine of that angle is:

$$cos(\theta) = \frac{p_z}{\sqrt{p_x^2 + p_y^2 + p_Z^2}}$$

To have an upper bound of this cosine, we maximize the nominator and minimize the denominator:

$$cos(\theta) \leq \begin{cases} \frac{max(p_z)}{\sqrt{min(p_x^2)+min(p_y^2)+(max(p_z))^2}} & \text{if } max(p_z) \geq 0 \\ \frac{max(p_z)}{\sqrt{max(p_x^2)+max(p_y^2)+(max(p_z))^2}} & \text{otherwise} \end{cases} \quad (6)$$

To apply equation 6 in our situation, we first rotate the cluster bounding box by the angle θ between the normal at point x and the z-axis. Once we have the cosine bound, we need to compute the BRDF approximation. This part was not really detailed in the original paper so we implemented our own version similar to the bounding proposed in the stochastic Lightcuts paper **[Yuk19]**. We first bound the diffuse part of the BRDFs by the cluster total color times the albedo at point x. The cluster's total color is the sum of the light colors of every light in the cluster. We used this in our three BRDF implementations: Lambert BRDF, Microfacets BRDF, and Disney BRDF (see Section II-B3). For the Microfacets and Disney BRDF, we multiplied the cluster's total color by an attenuation factor. The attenuation factor is computed as:

$$attenuation = \frac{1}{min(dist, 1)^2}$$

Where $dist$ is the distance between the closest point of the cluster's bounding volume and x. However, more investigation can be done to have tighter bounds for both the Microfacets and the Disney BRDFs (see Section **??**).

B. Implementation

In this section, we dive into the concrete implementation of the algorithm and the renderers.

1) Cuts: To efficiently build the dynamic cuts we create a max heap in terms of cluster error for every shaded point using the C++ STL. We initiate the cut as the root of the light tree. Then, while the stopping condition is not met, we pop the cluster with the highest error from the heap and add its two children to the heap. There are two stopping conditions for the cut construction:

- Cut size limit: we used a hyperparameter to define the maximum number of clusters there can be in a cut
- Error threshold: we used another hyperparameter to define the minimum error of a cluster to be considered similar enough. If all the clusters in the cut satisfy this condition, we stop the loop.

To make the computation even faster, we use the fact that a cluster representative is the same as one of the cluster's children. When computing the cluster children errors after having popped the cluster from the heap, we reuse the material, geometric, and visibility terms of the popped cluster for the child with the same representative. The tree builder can be found in *src/engine/beCore/gameplay/be_lights* while the dynamic cut selector is available at *src/engine/beRenderer/renderingSubSystems/rayTracing/be_raytracer*. Our implementation also supports parallelism and multi-threading with OpenMP for both the raytracing and the dynamic cut selection. To allow parallelism with the latest, we used a bit more memory and stored for each element in the cut heap: a pointer to a cluster, its cluster estimate, and its cluster error. Because we are re-building the heap for every point in the scene we avoid concurrency issues.

2) Vulkan: For the rendering part of the project, we used the modern Vulkan API. Vulkan is a recent and portable API to abstract GPUs. Being developed by the Khronos group, it is the successor of OpenGL without the constraints from the 90's. With fewer abstractions and a more manual setup, Vulkan gives the user finer control over the target application making it the reference in the Video game industry. However, this comes at the cost of being less intuitive and the OpenGL API is often preferred as a learning tool. Because the Lightcuts algorithm is about performance and acceleration techniques we thought that it would be interesting to implement it using Vulkan and begin to learn the API in the meantime. To reuse this work on future projects we also built a simple game engine from scratch capable of handling different BRDF models (see Section II-B3) and rendering modes.

Our application handles both a rasterizer and a raytracing mode dynamically controlled by the user. The rasterizer offers

a preview of the scene in which the user can choose different BRDFs and shading modes. The Lightcuts algorithm is however handled in the path tracer (see Section **??**) to avoid the computation of building the tree every frame. In the rasterizer, we used three descriptor sets:

- set 0 contains the camera: Because the camera information is used by all the objects we are drawing, we put it in the set 0 since it is the least frequently updated
- set 1 contains the lights: To handle dynamic lights we decided to use set 1
- set 2 contains the material: This set contains all the loaded materials in the scene (we were first updating this set for every model to store its material directly but we realized that having dynamic user-controlled materials would not work with this approach and switched to an array of loaded materials instead)

We also used *push constants* to store both the model matrices and the object material's index to fetch it from set 2.

Even if Vulkan can handle GPU raytracing, we decided to implement a CPU path tracer to have a better understanding of the algorithm. Before running, the path tracer builds a Bounding Volume Hierarchy (it also handles Bounding Sphere Hierarchy) and, if the Lightcuts mode is on, builds the light tree. The path tracer hyperparameters are configurable through an ImGui window in the scene. To render the output image of the path tracer, we fill a texture that we bind to a rectangle covering the entire viewport.

As this project was our first hand on Vulkan, the implementation might not be optimal; for example, to create the rasterizer wireframe modes we simply duplicated all the different graphics pipelines and changed the cull mode into *VK_CULL_MODE_NONE*.

3) BRDF: Our Lightcuts implementation handles three types of BRDF: the Lambert BRDF, the Microfacets BRDF, and a simplified Disney BRDF. In fact, our rasterizer handles three more shaders for debug purposes: a shader displaying the color of the objects, one displaying their normals, and a Blinn-Phong BRDF. The main difference between our Microfacets BRDF and our Disney BRDF is that the Microfacets model only handles a diffuse and a Metallic specular reflection. On the other hand, the Disney BRDF adds Sheen and Clearcoat reflections. To build the Disney BRDF we followed the homework guidelines from TM. Li **[Li24]**, however, because our systems do not handle transmittance we did not add the Glass effect and we are only emitting rays in a hemisphere when finding a hit in the pathtracer. Moreover, we are not building vertex tangents and bitangents so we can't take into account the anisotropy in the Metallic component. The different shaders are accessible in the *src/engine/shaders* folder and their CPU versions for the path tracer are available at *src/engine/beRenderer/pbr*. Shaders are also preprocessed using a custom Python script to allow the *#include* directive before being compiled in SPIR-V using the *glslc* compiler.

III. Results

In the following, we present some of the results obtained with our implementation.

In Figure 4 we compare the results of our brute-force path tracer with the Lightcuts model on a complex scene. This scene rendered with the Disney BRDF is made of 399 point lights and was built using the path tracer with 4 rays per pixel, 1 bounce, and 32 samples per bounce. The brute force approach (Figure 4a) was built in 814 minutes. The Lightcuts image (Figure 4b was built in 319 minutes using a maximum cut size of 100 and an error threshold of 2%. In Figure 5c we can see the differences between the two images. We can see that the bottom shadows in the baseline scene are more blurred than in the Lightcuts version.

In Figure 5 we compare the results of different cut maximum sizes. The scene is again rendered with the Disney BRDF. It is this time made of 856 lights with 4 rays per pixel but no bounces. We used a maximum cut size of 100 for Figure 5a and a maximum of 10 for Figure 5b. The brute-force approach (Figure 6a) built the image in 328 minutes, while the image was built in 55 minutes with maximum cuts of size 100 and in 7 minutes for cuts of size at most 10. The differences between Figures 5b and 5a are negligible meaning that using the Lightcuts approach with a few dozens of lights already gives sufficient results.

IV. Conclusion

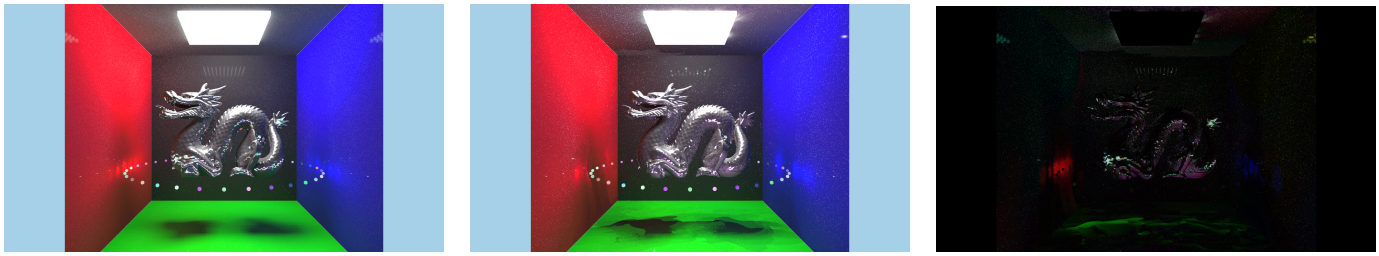
Our Lightcuts implementation gives promising results in different scenes. However, some points would need more investigation.

In the Lightcuts version of the scene presented in Figure 4, we can see that the light colors reflected on the dragon have a more purple tint than in the baseline. This observation indicates that we might need a better approach to compute the color of representative lights. One idea could be to use a weighted sum of the light colors in the cluster depending on their distance to the representative for example.

Another possible improvement might be to dynamically update the maximum size of a cut depending on the depth of the sampled point. In Figure 5, we saw that the difference between images generated by different cut maximum sizes can be rather small. Instead of manually tuning this parameter, we could use the depth of each pixel in view space to automatically control the maximum size of the pixel's cut.

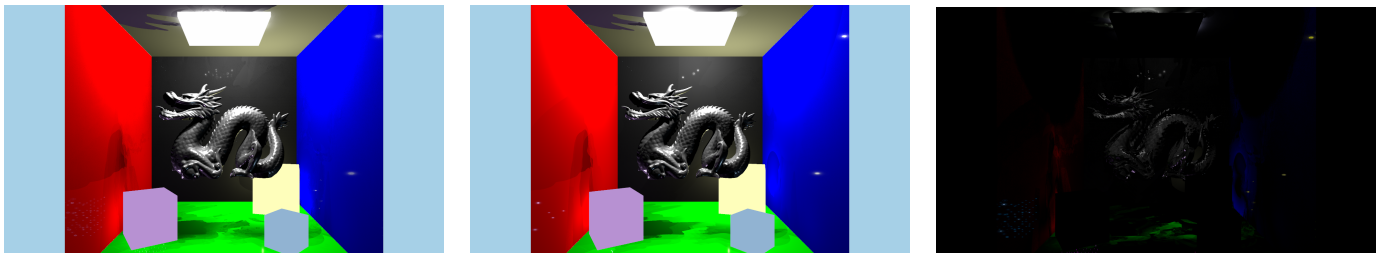
To improve computation time we can also implement the dynamic cut selection in a compute shader. Indeed, the light tree being a binary tree of a known size, it can easily be stored in an array of size $2n - 1$, n being the number of lights in the scene. Having this known size makes the array easily transferable inside a GPU buffer, hence permitting the implementation of the cut selection in a compute shader.

Finally, we could have better visual results using tighter bounds for the Microfacets and Disney BRDFs. For now, we are only using the diffuse, the attenuation, and the angle with the normals as bounds for our error estimation. If we could bound the metallic, sheen, and clearcoat terms with a tighter bound than just 1 we would probably increase the quality of the produced images.



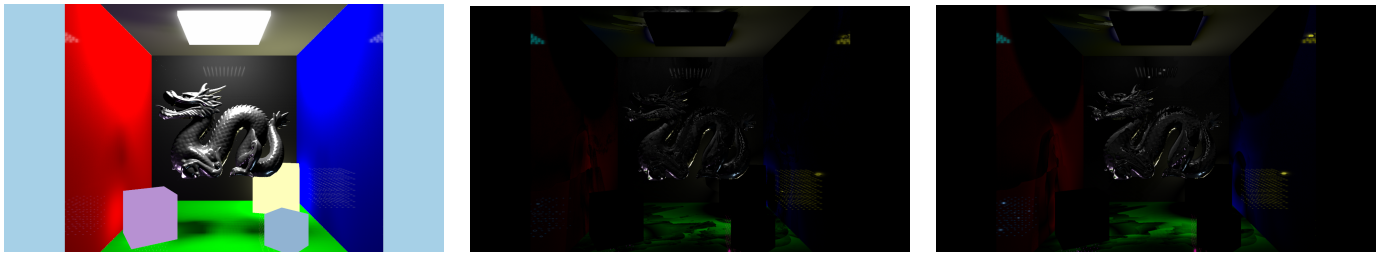
(a) Baseline - built in 814 min (b) Lightcuts - built in 319 min (c) Difference between the two

Fig. 4: Comparison between the baseline and the Lightcuts on a scene with 399 lights



(a) Max cut size 100 - built in 55 min (b) Max cut size 10 - built in 7 min (c) Difference between the two

Fig. 5: Comparison between the Lightcuts with a maximum cut size of 100 and 10 on a scene with 856 lights



(a) Baseline - built in 328 min (b) Difference with cut size of 100 (c) Difference with cut size of 10

Fig. 6: Comparison between the baseline and the Lightcuts on a scene with 856 lights

References

[Kaj86] James T. Kajiya. "The rendering equation". In: 20.4 (1986). ISSN: 0097-8930. DOI: 10.1145/15886.15902. URL: https://doi.org/10.1145/15886.15902.

[SWZ96] Peter Shirley, Changyaw Wang, and Kurt Zimmerman. "Monte Carlo techniques for direct lighting calculations". In: *ACM Trans. Graph.* 15.1 (Jan. 1996), pp. 1–36. ISSN: 0730-0301.

[Wal05] Bruce Walter. "on the Ward BRDF". In: 2005. URL: https://api.semanticscholar.org/CorpusID:16165762.

[Wal+05a] Bruce Walter et al. "Implementing lightcuts". In: *International Conference on Computer Graphics and Interactive Techniques*. 2005. URL: https://api.semanticscholar.org/CorpusID:8075784.

[Wal+05b] Bruce Walter et al. "Lightcuts: a scalable approach to illumination". In: *ACM Trans. Graph.* 24.3 (July 2005), pp. 1098–1107. ISSN: 0730-0301.

[Bur12] Brent Burley. "Physically-Based Shading at Disney". In: 2012. URL: https://api.semanticscholar.org/CorpusID:7260137.

[Bur15] Brent Burley. "Extending the Disney BRDF to a BSDF with Integrated Subsurface Scattering". In: 2015. URL: https://api.semanticscholar.org/CorpusID:208625014.

[Yuk19] Cem Yuksel. "Stochastic Lightcuts". In: *High-Performance Graphics (HPG 2019)*. Strasbourg, France: The Eurographics Association, 2019, pp. 27–32. ISBN: 978-3-03868-092-5. DOI: 10.2312/hpg.20191192.

[Bla20] Blanco, Victor. *Vulkan Guide*. https://vkguide.dev/. Distributed by a MIT license. 2020.

[Gal20] Galea, Brendan. *Vulkan Game Engine Tutorials*. https://www.youtube.com/playlist?list=PL8327DO66nu9qYVKLDmdLW_84-yE4auCR. YouTube playlist. 2020.

[LY20] Daqi Lin and Cem Yuksel. "Real-Time Stochastic Lightcuts". In: *Proc. ACM Comput. Graph. Interact. Tech. (Proceedings of I3D 2020)* 3.1 (2020).

DOI: 10.1145/3384543. URL: http://doi.acm.org/10.1145/3384543.

[Unt21] Unterguggenberger, Johannes. *Vulkan Lecture Series*. https://www.youtube.com/playlist?list=PLmIqTlJ6KsE1Jx5HV4sd2jOe3V1KMHHgn. YouTube playlist. 2021.

[Yer21] Yerli, Alexander. *Vulkan Tutorial*. https://vulkan-tutorial.com/. Distributed under the CC BY-NC-SA 4.0 license. 2021.

[Li24] Tzu-Mao Li. *Homework 1: Disney Principled BSDF*. UCSD CSE 272: Advanced Image Synthesis (Winter 2024). Jan. 2024.