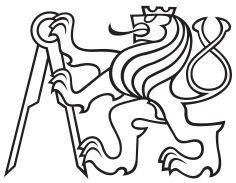
Master Thesis



Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Computer Graphics and Interaction

Rendering Night Cities

Tereza Hlavová

Supervisor: doc. Ing. Jiří Bittner, Ph.D.
May 2025

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Hlavová Tereza** Personal ID number: **492278**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Computer Graphics and Interaction**

Study program: **Open Informatics**

Specialisation: **Computer Graphics**

II. Master’s thesis details

Master’s thesis title in English:

Rendering Night Cities

Master’s thesis title in Czech:

Zobrazování nočních měst

Name and workplace of master’s thesis supervisor:

doc. Ing. Jiří Bittner, Ph.D. Department of Computer Graphics and Interaction

Name and workplace of second master’s thesis supervisor or consultant:

Date of master’s thesis assignment: **07.02.2025** Deadline for master's thesis submission: _____

Assignment valid until: **20.09.2026**

_____
Head of department’s signature

_____
prof. Mgr. Petr Páta, Ph.D.
Vice-dean´s signature on behalf of the Dean

III. Assignment receipt

The student acknowledges that the master’s thesis is an individual work.
The student must produce her thesis without the assistance of others, with the exception of provided consultations.
Within the master’s thesis, the author must state the names of consultants and include a list of references.

_____
Date of assignment receipt

_____
Student’s signature

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Hlavová Tereza** Personal ID number: **492278**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Computer Graphics and Interaction**

Study program: **Open Informatics**

Specialisation: **Computer Graphics**

II. Master’s thesis details

Master’s thesis title in English:

Rendering Night Cities

Master’s thesis title in Czech:

Zobrazování nočních měst

Guidelines:

Review methods suitable for displaying scenes with a large number of light sources. Focus on methods applicable to large scenes, such as city models. A suitable method should support the calculation of direct illumination from many static and dynamic light sources (public lighting, illumination from house windows, illumination from moving vehicles).
Implement a selected method using the Vulkan API. The implementation should support predefined illumination profiles of light sources and customizable scattering properties for roads, buildings, windows, and other model parts. Implement also visualization and exploration rendering modes that will be useful to visualize the illuminance and identify the most influential light sources in a selected part of the model.
Evaluate the implementation regarding visual quality (comparison with reference photographs) and rendering speed on at least three different city models that are freely available. Discuss the possibilities of indirect illumination calculation and simulation of the participating medium (fog, rain, clouds).

Bibliography / sources:

[1] Bitterli, B., Wyman, C., Pharr, M., Shirley, P., Lefohn, A. and Jarosz, W. (2020). Spatiotemporal reservoir resampling for real-time ray tracing with dynamic direct lighting. ACM Transactions on Graphics (TOG), 39(4), 148-1.
[2] Moreau, P., Pharr, M. and Clarberg, P. (2019). Dynamic Many-Light Sampling for Real-Time Ray Tracing. In Proceedings of High Performance Graphics (Short Papers), 21-26.
[3] Ouyang, Y., Liu, S., Kettunen, M., Pharr, M. and Pantaleoni, J. (2021). ReSTIR GI: Path resampling for real□time path tracing. In Computer Graphics Forum, 40(8), 17-29.
[4] Chandler, J., Yang, L. and Ren, L. (2015). Procedural window lighting effects for real-time city rendering. In Proceedings of the 19th Symposium on Interactive 3D Graphics and Games, 93-99.
[5] Conte, M. (2019). Real-time rendering of cities at night. MSc thesis, University of Montreal.
[6] Laurent, G., Delalandre, C., de La Rivière, G., Boubekeur, T. (2016). Forward Light Cuts: A Scalable Approach to Real□Time Global Illumination. In Computer Graphics Forum, 35(4), 79-88.

FAKULTA ELEKTROTECHNICKÁ
FACULTY OF ELECTRICAL ENGINEERING
Technická 2
166 27 Praha 6

DECLARATION

I, the undersigned

Student's surname, given name(s): Hlavová Tereza
Personal number: 492278
Programme name: Open Informatics

declare that I have elaborated the master's thesis entitled

Rendering Night Cities

independently, and have cited all information sources used in accordance with the Methodological Instruction on the Observance of Ethical Principles in the Preparation of University Theses and with the Framework Rules for the Use of Artificial Intelligence at CTU for Academic and Pedagogical Purposes in Bachelor's and Continuing Master's Programmes.

I declare that I used artificial intelligence tools during the preparation and writing of this thesis. I verified the generated content. I hereby confirm that I am aware of the fact that I am fully responsible for the contents of the thesis.

In Prague on 23.05.2025

Bc. Tereza Hlavová
.....
student's signature

Acknowledgements

I want to thank my family and friends for support during my studies and also to my supervisor doc. Ing. Jiří Bittner, Ph.D. for consultations and advices for this work.

Abstract

In recent years, the possibility of virtually browsing real-world locations has increased due to the progress of and wider usage of 3D reconstruction methods. Models of whole cities are even freely available in some cases. For the purposes of a simple visualization of city sceneries at nighttime to a simulation of light pollution, rendering would require to work with many light sources, all at realtime.

This work focuses mapping out the methods to render direct illumination of night scenes with many light sources efficiently, implements suitable one using Vulkan API with hardware raytracing support and showcases results on test scenes built from open data of city models.

Keywords: 3D scene, citygml, raytracing, Vulkan, ReSTIR, Light BVH

Supervisor: doc. Ing. Jiří Bittner, Ph.D.

Abstrakt

V poslední době je možné častěji virtuálně zkoumat reálné lokace kvůli pokroku a širšímu využití metod 3D rekonstrukce. Dokonce v některých případech jsou k dispozici modely celých měst. Pro účely ať už prosté vizualizace městských scenérií v nočních hodinách nebo až simulace světelného znečistění, vykreslování by vyžadovalo brát v potaz hodně světelných zdrojů, a to v reálném čase.

Tato práce se zaměřuje na zmapování metod pro vykreslování přímého osvětlení nočních scén s mnoha světelnými zdroji, vhodnou metodu implementuje pomocí Vulkan API s podporou hardwarového vrhání paprsků, a prezentuje výsledky na testovacích scénách vyrobených na základě otevřených dat modelů měst.

Klíčová slova: 3D scéna, citygml, vrhání paprsků, Vulkan, ReSTIR, Light BVH

Překlad názvu: Vykreslování nočních měst

Contents

Figures

Tables

Chapter 1

Introduction

In recent years, the possibility of virtually browsing real-world locations has increased due to the progress of and wider usage of 3D reconstruction methods. Models of whole cities are even freely available in some cases. Inspiration for this work came from an online application rendering the city of Prague [26]. It implements setting the day and time and positions the sun accordingly to present an immersive image. But at night-time, the area is left in ambient lighting only, as the application does not take real factors, like many street lamps, into account. For the purposes of a simple visualization of city sceneries at nighttime to a simulation of light pollution, rendering would require working with many light sources, preferably real-time. Using many light sources in the scene requires a lot of computation and it is time-consuming. To reduce the computation time for every frame without using complex acceleration structures, a state-of-art method called ReSTIR [3] is used. An implementation of a city renderer utilizing hardware-accelerated ray-tracing together with ReSTIR method is demonstrated on open geometric and geographic data.

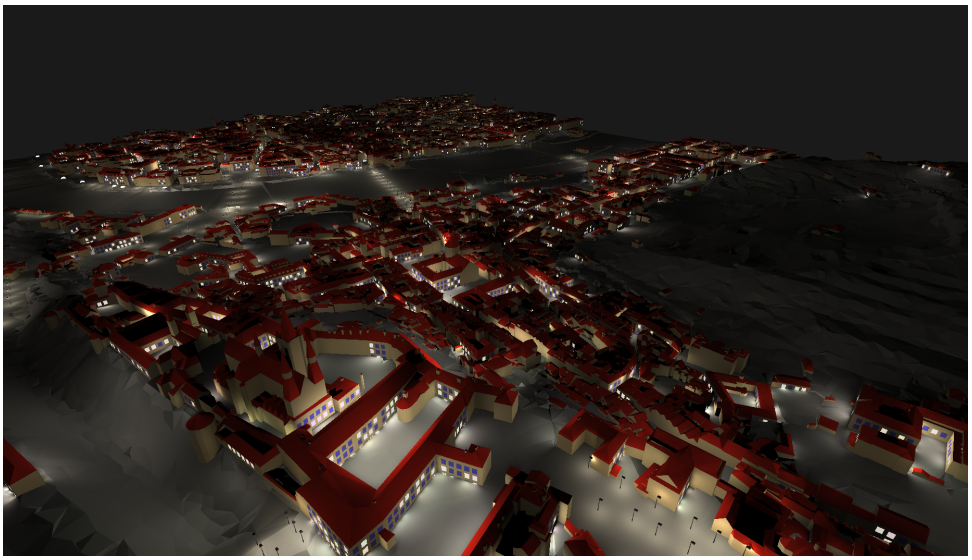


Figure 1.1: Rendered part of Prague.

This work is divided into 5 chapters. Chapter 2 introduces the problems of night city rendering, mainly the existing methods of rendering with many light sources. Used method for this work is overviewed in Chapter 3. Chapter 4 goes over the basics of Vulkan API and its hardware-accelerated ray-tracing features. Chapter 5 goes over the implementation of a night city renderer built for this work. Results are summarized in Chapter 6. And finally, Chapter 7 summarizes all work done.

In-process version of this work was presented and published at Central European Seminar Computer Graphics (CESCG) 2025 [14].

Chapter 2

Related Work

The main focus of this work will be the so-called many-lights problem introduced below. However, rendering at nighttime can be explored from multiple points of view, which are also quickly overviewed in the end of this chapter.

2.1 Introducing the problem

Rendering, both real-time and offline, has to deal with the problem of solving the rendering equation. Introduced by James Kajiya in 1986 [15], the rendering equation describes how light is reflected at surfaces by accounting for all possible incoming light directions and their contributions to the outgoing radiance.

Local light reflection at a point $\mathbf{x}$ gives outgoing radiance L_o in direction ω_o:

$$L_o(\mathbf{x}, \omega_o) = L_e(\mathbf{x}, \omega_o) + \int_\Omega L_i(\mathbf{x}, \omega_i) f(\omega_i, \mathbf{x}, \omega_o) \cos\theta_i d\omega_i, \tag{2.1}$$

where L_e is light emission illuminating point $\mathbf{x}$ from direction ω_i, integral is over the whole sphere Ω, describing both reflection and refraction of light, f is the bidirectional scattering distribution function (BSDF), and $\cos\theta_i$ is the dot product of normal vector n of the surface area around point $\mathbf{x}$ and direction of ω_i, with negative values clamped to zero.

Let us define a TRACE function, which returns the point of the closest intersection with scene geometry along a ray of given direction and origin. If participating media is omitted, causing the radiance to be constant along traced ray, the incoming radiance can be rewritten using outcoming radiance and the TRACE function as:

$$L(\mathbf{x}, \omega_o) = L_e(\mathbf{x}, \omega_o) + \int_\Omega L(r(\mathbf{x}, \omega_i), -\omega_i) f_r(\omega_i, \mathbf{x}, \omega_o) \cos\theta_i d\omega_i, \tag{2.2}$$

where, compared to the previous formulation, bidirection reflection distribution function (BRDF) f_r is used, TRACE function was substituted into the formula as:

$$L_i(\mathbf{x}, \omega_o) = L_o(r(\mathbf{x}, \omega_i), -\omega_i), \tag{2.3}$$

and indexes of incoming/outcoming indication are omitted.

This form models the steady-state distribution of radiance in a scene. It is integral, the unknown variable is both on the left side and right side, thus analytical solution is in most cases not possible. The equation is recursive in nature, because the computation of illumination at point x requires illumination computed at all visible points on the hemisphere, and again at visible points for them, as illustrated in Figure 2.2. Algorithms solving the global illumination in the scene look for such distribution of radiance, that fits the equation.

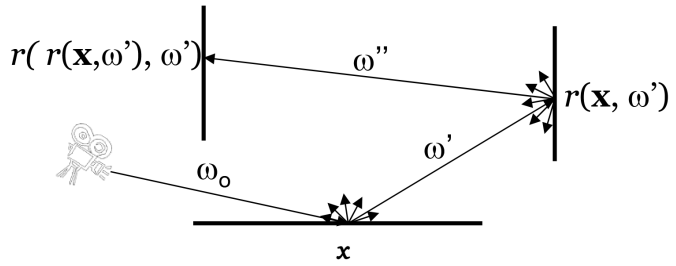


Figure 2.1: The recursive nature of lighting equation, it is the main feature that is used by path tracing [6].

This work focuses mainly on the direct illumination, so the recursive nature can be cut off. Because L_i includes only light coming directly from the light sources in the scene, let us introduce the rendering equation light surface area formulation for direct illumination as opposed to the previous ray direction formulation. Using the substitution

$$d\omega = dA \cdot \frac{\cos\theta}{r^2}, \tag{2.4}$$

where A is surface area of the light source and r is the distance between point x and the light source, as visualized in Figure **??**, the formula becomes:

$$L(\mathbf{x}, \omega_o) = L_e(\mathbf{x}, \omega_o) + \int_S L(r(\mathbf{y}, \mathbf{y} \to \mathbf{x}) f_r(\mathbf{y} \to \mathbf{x}, \mathbf{x}, \omega_o) G(\mathbf{x}, \mathbf{y}), V(\mathbf{x}, \mathbf{y}) dA, \tag{2.5}$$

where

$$\mathbf{y} \to \mathbf{x} = \frac{\mathbf{x} - \mathbf{y}}{||\mathbf{x} - \mathbf{y}||}, \tag{2.6}$$

is a normalized vector in direction from point $\mathbf{y}$ to point $\mathbf{x}$,

$$G(\mathbf{x}, \mathbf{y}) = \frac{\cos\theta_x \cdot \cos\theta_y}{||\mathbf{x} - \mathbf{y}||^2}, \tag{2.7}$$

is a geometric term, and $V(\mathbf{x}, \mathbf{y})$ is a visibility term between points $\mathbf{x}$ and $\mathbf{y}$, with value equal to 1 if no intersection can be found by tracing a ray from point $\mathbf{x}$ to point $\mathbf{y}$ (or vice versa), and equal to 0 otherwise. The visibility term $V(\mathbf{x}, \mathbf{y})$ is also the place in the computation where the TRACE function is usually used again to confirm no intersection between the two points has been found. This is considered a costly operation, its complexity rises with the scene's geometrical complexity.

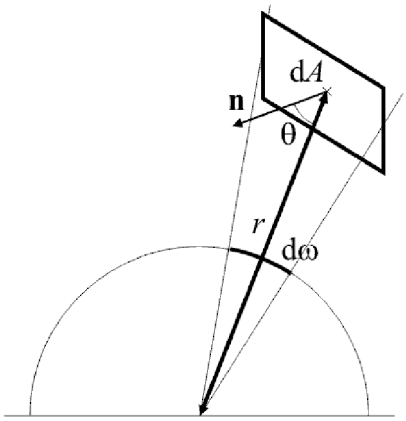


Figure 2.2: Solid angle illustration [6].

For brevity onwards, let us simplify the integral part of the equation 2.5 by dropping the viewing direction θ_o and visible point $\mathbf{x}$ as:

$$L = \int_S f(x)dx, \tag{2.8}$$

where

$$f(x) = L(x)f_r(x)G(x), V(x). \tag{2.9}$$

2.1.1 Importance Sampling (IS)

As analytical solution is usually not possible to obtain, the integral can be approximated using Monte Carlo Importance Sampling (IS). The traditionally used estimator has a form of

$$\langle L \rangle_{is}^N = \frac{1}{N} \sum_{i=1}^{N} \frac{f(x_i)}{p(x_i)} \approx L, \tag{2.10}$$

by sampling N independent samples (points across light sources) from the source probability density function $p(x)$. The result remains unbiased as long as $p(x)$ returns positive results whenever $f(x)$ is not zero. To reduce noise and converge to the result quicker, either the number of samples N has to be increased, which would noticeably add to the computational cost, or sample proportionally to $f(x)$, because the closer the $p(x)$ is to the integrand, the lower is the estimator error. However, that is usually infeasible in practice, because $f(x)$ is unknown until the time of computation and tracing, mainly because of the visibility term $V(x)$, but also due to the geometry term $G(x)$ and BRDF f_r.

While sampling directly from $f(x)$ is not practically done, drawing samples proportionally to the individual terms in the integrand can be feasible. A basic sampling strategy is cosine hemisphere sampling, where directions for tracing are drawn according to the term $n \cdot \omega_i$, without using other knowledge

of the material or light sources. Material sampling strategy chooses directions with a probability proportional to $f_r(\omega_i, \mathbf{x}, \omega_o) \cdot (n \cdot \omega_i)$, the material's reflective properties. For glossy materials, this approach stands as highly effective, as most light sources that contribute to the shading of the visible point of that material are in directions close to the reflection direction, as illustrated in Figure 2.3. However, for fully diffuse materials, material sampling strategy is the cosine hemisphere sampling. In cases of small area/small spherical light sources in the scene, for diffuse surfaces, the material sampling strategy is inefficient, because the cone, in which the light source is visible from the surface point is thin, thus with a small probability of a direction to be selected in it.

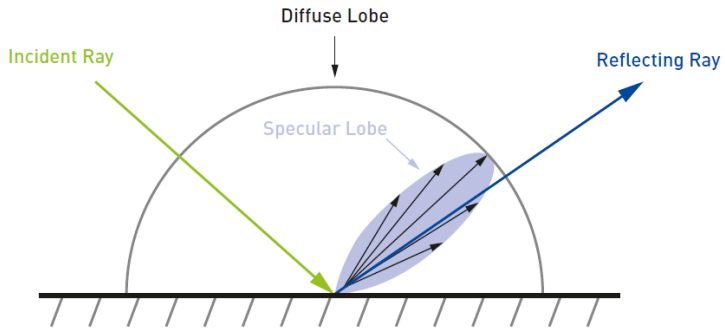


Figure 2.3: Importance sampling is effective for specular materials, as it samples mostly in the specular lobe's direction [23].

In these cases, light sampling strategy is more appropriate. Light sampling selects points on light sources in the scene directly instead of directions to trace for closest intersection.

2.1.2 Multiple Importance Sampling

With a technique called Multiple Importance Sampling (MIS), sampling with more strategies can be combined in a weighted estimator. For M strategies and N_s samples for individual strategy, the estimator has a form of:

$$\langle L \rangle_{mis}^{M,N} = \sum_{s=1}^{M} \frac{1}{N_s} \sum_{i=1}^{N_s} w_s^{mis} \frac{f(x_i)}{p(x_i)}. \tag{2.11}$$

The formula adds use of a weighting function w_s^{mis} for every strategy. The MIS estimator remains unbiased as long as the sum of the weights w_s of a sample equals to one: $\sum_{s=1}^{M} w_s^{mis}(x) = 1$ whenever $f(x)$ is non-zero, and $w_i^{mis}(x) = 0$ whenever $p(x) = 0$. While there can be other weighting functions used, a balance heuristic for non-negative weights is proven to not be much worse than any other possible heuristic.

$$w_s^{mis}(x) = \frac{N_s \cdot p_s(x)}{\sum_j N_j \cdot p_j(x)} \tag{2.12}$$

A widely used combination of strategies in MIS is light sampling and material sampling.

2.1.3 Resampled Importance Sampling

When dealing with light sampling of many light sources in the scene or complex BRDFs, especially useful is Resampled Importance Sampling (RIS). RIS is a method that draws samples from a different source distribution p, one that is easy to sample from and is sub-optimal to the complex one, for example $p \propto L_e$. It generates M candidate samples from a proposal distribution p, $\mathbf{x} = x_1, ..., x_M$. Then, importance weight is computed for each candidate sample $\mathbf{x}$:

$$w^{ris}(x) = \frac{\hat{p}(x)}{p(x)}, \tag{2.13}$$

where $\hat{p}(x_i)$ is the target complex PDF, that is hard to sample from. One index z of a sample from these candidates is then selected according to the discrete probabilities:

$$p(z \mid \mathbf{x}) = \frac{w^{ris}(x_z)}{\sum_{i=1}^{M} w^{ris}(x_i)}. \tag{2.14}$$

Drawing samples $y \equiv x_z$ multiple times and using it in a N-sample RIS estimator that corrects the approximation and averages the N samples:

$$\langle L \rangle_{ris}^{N,M} = \frac{1}{N} \sum_{i=1}^{N} \left(\frac{f(y_i)}{\hat{p}(y_i)} \right) \cdot \frac{1}{M} \sum_{i=1}^{M} w^{ris}(x_{ij}), \tag{2.15}$$

estimates the integral in the equation 2.8. If M and N are both greater than zero, and both the proposal distribution p and target $\hat{p}$ are greater than zero if f is non-zero. Assuming N=1 for simplicity, the pseudocode for RIS is shown in Algorithm 1.

RIS in this form requires computing cumulative distribution function (CDF) and storing all the candidates and their weights until the final sample is selected in the end. However, storing all past data would be inefficient if multiple sample candidates are to be evaluated per pixel.

2.2 Many-lights Rendering

While the MIS estimator can already produce promising results for certain scenes, scaling up the number of light sources in the scene either brings more noise and/or computational cost. Even if the scene consists only of point light sources and the ground truth image using direct illumination could be computed without noise, the visibility term in the integrand is still a costly computation.

Thus, the many-lights problem has posed an issue in the rendering scene for decades now. Naïve methods would just limit the number of light sources for a given scene or scene part and manually pick the most desirable ones.

Algorithm 1 Resampled Importance Sampling Pseudocode

Input: M number of candidates, q pixel
Output: y sample, $\sum_{i=1}^{M} w^{ris}(x_i)$ sum of RIS weights

```
x ← ∅
w^ris ← ∅
w^ris_sum ← 0
for i ← 1 to M do
    generate sample x_i ~ p
    x ← x ∪ x_i
    w^ris_i ← p̂(x_i)/p(x_i)
    w^ris_sum ← w^ris_sum + w^ris_i
    w^ris ← w^ris ∪ w^ris_i
end for
compute CDF C from w^ris
draw random index z ∈ [0, M) using C to sample ∝ w^ris_z
y ← x_z
return y, w^ris_sum
```

For offline rendering nowadays, given GPU rendering acceleration, we could theoretically use all of the light sources, at the cost of much more time spent evaluating light sources that contribute almost nothing to the result. With real-time rendering, however, this would not be acceptable at all. For the given reasons, methods trying to solve the many-lights problem were developed.

2.2.1 LightCuts

For offline rendering, a notable ray tracing method used to solve the many-lights problem was introduced under the name LightCuts [22] in 2005. The method simplifies the light sampling by using clusters of light sources. The so-called cluster representatives, together with precomputed sums of light source intensities, approximate all the light sources the clusters contain.

However, different parts of the image usually require different partitioning of the light sources into the clusters. For this, the method uses a binary tree, a global light tree, to form clusters of light sources by placing light sources into leaves and then performing cuts of the same tree for different views and positions in the scene. Every such cut is a set of nodes that every path in the tree down from root to leaf node must contain exactly one node from the cut set, resulting in a valid partitioning of the light tree into clusters of light sources.

Choosing correct cuts and light tree build is essential to minimize the error in the approximation of the light source cluster by its representative, to keep the error under a noticeable threshold. Selection is visualized in Figure 2.4. At each shading point, choosing the correct cut starts with a rough estimate and goes deeper towards more partitioning until the threshold is met. Because cluster's representative is also one of its children nodes, lighting computation

has to be done only one more time during the traversal down the tree, because the representative's computation can be directly reused for one of the children nodes.

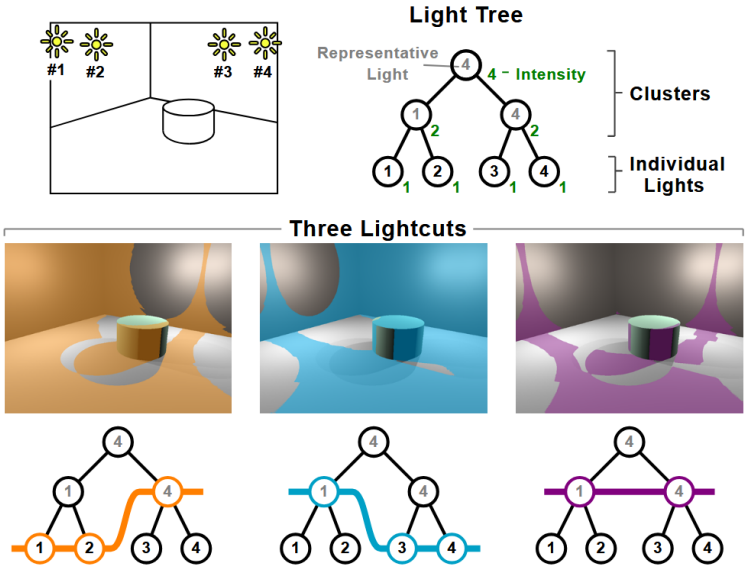


Figure 2.4: Different cuts visualization on the binary light tree. Colored areas corresponding to the cuts below highlight surfaces, where the error is indistiguishible[22].

Clusters, as nodes of the light tree, aim to group together similar light sources in geometric, material, and visibility terms - this can be approximated with spatial proximity and orientation similarity. Furthermore, in the case of more than one light source type, one light tree is built per type. The tree is built bottom-up, choosing to combine two clusters (or individual light sources) that would result in the smallest possible new cluster in terms of a metric computed as $I_{\mathbb{C}} \cdot (\alpha_{\mathbb{C}}^2 + c^2 \cdot (1 - \cos \beta_{\mathbb{C}})^2)$, where $I_{\mathbb{C}}$ is a total intensity of the cluster representative, $\alpha_{\mathbb{C}}$ is the length of cluster's bounding box diagonal, $\beta_{\mathbb{C}}$ is the half-angle of the cluster's bounding cone, which is chosen for directional type of light sources. The constant c controls proportional influence of spatial or orientation properties of the light sources. Each light source is its own representative and cluster representative is always one of its two children nodes. It is chosen randomly with a probability proportional to their total intensities.

For offline rendering, LightCuts method offers a sublinear performance and a possibility of global illumination approximation with virtual point light sources (VP). However, because the same tree is used for the entire image and the cut is chosen going from the root down, this repeated traversal results in a sampling correlation, because the nodes on top of the tree have higher probability of being chosen, which can lead to temporal instability.

Online Rasterized LightCuts

From real-time techniques, a rasterization method specialized for rendering cities at night, proposed by Conte [1] in 2018, integrates of LightCuts onto GPU for real-time use with rasterization. Due to the expected nature of cities, this method proposes using varying number of lights for nodes in light tree instead of two in the original binary tree. It is argued that this proposal was made to not only reduce the depth of the tree but also to better adapt to the city's layout - having building windows at the same depth, clustered into floors in a tree level above and floors into facades. The traversal is then executed on GPU in a fragment shader, solving visibility of light sources for close-by pixel groups using the light tree. This method is limited by its rasterization nature, so expanding the method for global illumination or area light sources would be complex.

Stochastic LightCuts

Modified variant of LightCuts, called Stochastic LightCuts [20], was first published in 2019. Apart from needing significantly fewer light source samples to achieve similar results as the original variant, it also successfully creates an unbiased lighting method.

To battle the sampling correlation, after the cut is found, the lighting estimation is not computed from the representatives, but by randomly selecting a light source in every cluster's subtree (of the cut) with a hierarchical sampling technique. Every light source is assigned a probability of being chosen in a given subtree. The light source in the subtree is not chosen directly, but by a traversal from root node of the subtree down to the leaf node, which is the selected light source in the end. Nodes are picked during the traversal with importance sampling.

Real-time Stochastic LightCuts

In 2020, a method to accommodate Stochastic LightCuts to the GPU was proposed [21]. It uses perfectly balanced binary trees, filling the structure with so-called bogus lights with zero intensities up to the power of two size. Furthermore, the light tree structure is changed into two-level structure, roots of bottom-level perfectly balanced binary trees are used as leaves in the top-level perfectly balanced binary tree. This allows for instancing and also faster rebuilds, as a partial rebuild is sufficient.

Cut selection is not performed for each individual pixel but rather for group of nearby pixels through Cut Sharing. This does not cause correlation, as individual pixels then execute the hierarchical sampling technique independently for the chosen shared cut.

2.2.2 Matrix Row-Column Sampling

A different prominent offline rendering technique is Matrix Row-Column Sampling [19], first published in 2007. It uses a transfer matrix, using light sources as rows and shaded visible points as columns, modelling their interactions. Instead of computing every shaded point interaction with every light source, it fully computes only a subset of rows and columns and then reuses these calculations for the rest of the matrix by approximation. This method can handle indirect light and area light sources, however, it is an offline method.

2.2.3 Light BVH

A dynamic many-light sampling real-time ray tracing method was proposed by Moreau et al. [2] in 2019. It works with a two-level bounding volume hierarchy (BVH). BVH stores the light sources, which it uses to estimate where the most important light sources for a given point on the to-be-shaded surface are and sample mostly those, thus limiting the number of shadow rays spatially.

The BVH is built on similarities in location and direction of the light sources in the scene. In offline rendering, a single BVH for all light sources suffices, but with dynamic scenes, the whole BVH would have to be rebuilt, causing a bottleneck for real-time rendering. The method groups light sources into separate bottom-level acceleration structures (BLAS), as shown in Figure 2.5. For emissive meshes, it is advised to use one BLAS per one emissive mesh and to split dynamic and static scene parts from one another. The BLASes are then grouped in a top-level acceleration structure (TLAS). With this division, a moving light source causes its BLAS and TLAS to be rebuilt, while other BLASes can stay untouched. This division also matches hardware ray-tracing API, making it suitable for use with the GPU-accelerated ray-tracing.

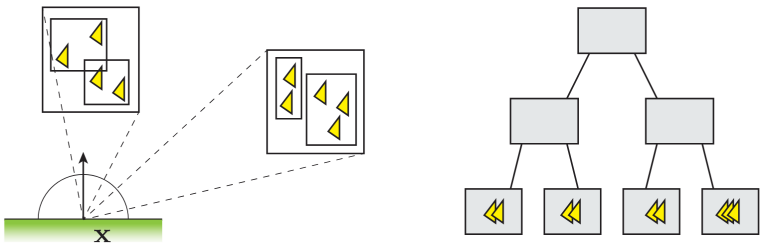


Figure 2.5: Light BVH build based on spatial similarity, light sources are stored in the leaves and tree is traversed stochastically from top to bottom [18].

BVH traversal is performed stochastically, first down the TLAS, evaluating an importance function. BLAS are traversed similarly.

2.3 Rendering the Night Sky

Rendering at nighttime can be explored from multiple points of view with different problems as the focus. An interesting problem to solve could be an efficient computation of natural illumination of objects by the moon, stars, as a full physically-based night sky model [8] delves into. Another interesting factor is the inclusion of participating media in the atmosphere.

2.3.1 Theory behind participating media

Rendering realistic imagery of the outside world often has to deal with participating media for effects such as smoke, rain, fog, or clouds. The problem at hand, however, becomes significantly more complex, as the physical simulation of light interaction is not only with the surface's material, but also with the volume the ray of light, or to be more precise - radiation, goes through.

In the absence of the participating media, the radiance L along the ray is presumed to be constant. From the point the ray enters the media till it exits, three kinds of interactions are usually modeled: scattering, absorption, and emission, as illustrated in Figure 2.6.

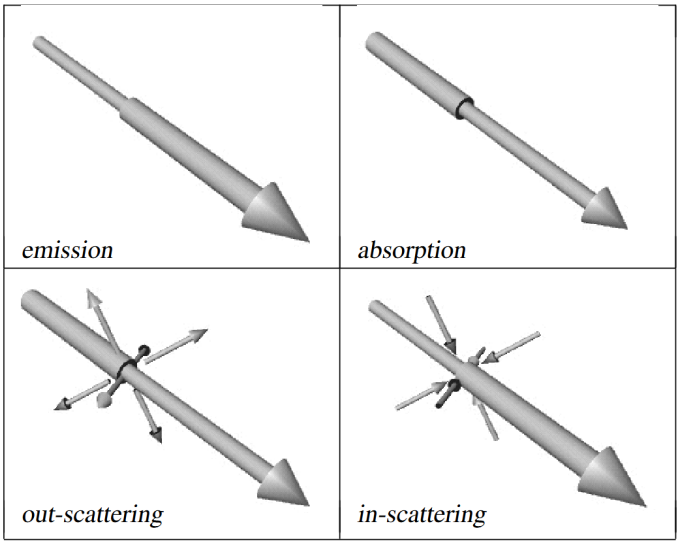


Figure 2.6: Ray interactions with participating media [7].

In point in space x along the ray in the direction ω, the participating medium is typically described with an absorption coefficient $\kappa_a(x)$, scattering coefficient $\kappa_s(x)$, extinction coefficient $\kappa_t(x)$, phase function $p(x, \omega_o, \omega_i)$, and scattering albedo $\Omega(x)$. This is a simplified model, as the factors mentioned above also vary for different wavelengths.

Reduction of radiance

The absorption phenomenon causes the loss of radiance transported by the ray as parts get transformed into different forms of energy, usually heat. The relative reduction of radiance is modeled with the previously mentioned absorption coefficient $\kappa_a(x)[m^{-1}]$ as:

$$dL(x,\omega) = -\kappa_a(x)L(x,\omega)dx, \tag{2.16}$$

for the differential distance dx along the ray.

The coefficient of scattering $\kappa_s(x)[m^{-1}]$ is used for the model of the so-called out-scattering. Out-scattering is a phenomenon during which the radiance gets also reduced because the radiant propagation changed direction. It is expressed similarly to the absorption phenomenon:

$$dL(x,\omega) = -\kappa_s(x)L(x,\omega)dx. \tag{2.17}$$

Together, the reduction of radiance is modeled as their sum:

$$\begin{aligned} dL(x,\omega) &= -\kappa_a(x)L(x,\omega)dx - \kappa_s(x)L(x,\omega)dx \\ &= -\kappa_t(x)L(x,\omega)dx, \end{aligned} \tag{2.18}$$

where $\kappa_t(x) = \kappa_a(x) + \kappa_s(x)$ is the extinction coefficient.

Increase of radiance

Emission within media causes the increase of the radiance by self-luminous properties, for example, black-body radiation. That is a thermal electromagnetic radiation, emitted by an idealized object called a black body — an object that absorbs all incident light and re-emits energy solely based on its temperature, not its material properties or color. The radiance increase due to emission is modeled fairly simply as:

$$dL(x,\omega) = \kappa_a(x)L_e(x,\omega)dx. \tag{2.19}$$

Scattering can also lead to the increase of radiance in the direction ω of the ray, in which case it is named in-scattering, caused by radiance incoming from all directions (on a sphere Sph) in the media to the point x that is scattered to the direction ω. The increase due to in-scattering is modeled as:

$$dL(x,\omega) = \frac{\kappa_s(x)}{4\pi} \cdot \int_{Sph} L(x,\omega_i) \cdot p(x,\omega,\omega_i)d\omega_i, \tag{2.20}$$

where the phase function $p(x,\omega,\omega_i)$ defines the probabilities of light scattering from an incoming direction ω_i into an outgoing direction ω. It can be modeled with many variants, however, all must obey $\int_{Sph} p(x,\omega,\omega_i)d\omega_i = 1$. The phase function is mostly modeled as fully rotationally symmetrical in regards to the ω_i direction, thus the used parameter is simpler than two directions ω and ω_i, but only cosine of an angle between them $\cos\theta = \omega \cdot \omega_i$. They also can vary a lot for different wavelengths.

The simplest phase function is equivalent to the diffuse BRDF material but with participating media, as it is a constant, isotropic function $p_{iso}(x, \omega, \omega_i) = \frac{1}{4\pi}$. The Rayleigh phase functions model how light is scattered by small particles (their radii about 10 times smaller than the wavelength of the light), such as smoke or gas molecules in the atmosphere. They are used for effects like the blue sky and reddish sunsets. They are also sometimes combined with Mie phase functions, which are used for scattering modeling for larger particles in atmospheric models, like aerosols, water droplets, dust, smoke, and cloud particles. The Mie functions are rather complex and can be approximated by the Henyey-Greenstein (HG) phase functions.

Light transport equation

Putting the differential equations for the increase and reduction of radiance along the ray together results in the following:

$$\frac{dL(x, \omega)}{dx} = \kappa_t(x)J(x, \omega) - \kappa_t(x)L(x, \omega). \tag{2.21}$$

where $J(x, \omega)$ represents so-called source radiance which models the increase of radiance previously described. This equation is vastly complex, especially when multiple in-scattering in the atmosphere is considered (modelling multiple bounces of light rays which then contribute to the increase of radiance). Methods usually iterate over the traced ray and compute increases and decreases of radiance in intervals, either fixed or adaptive.

Chapter 3
ReSTIR

Bitterli et al. [3] introduced a different approach that attempts to solve the many-lights problem. It is a reservoir-based spatiotemporal importance resampling method for direct illumination, ReSTIR. It does not rely on complex data structures that would require time-consuming rebuilding as with lights BVH in the previous method. It instead reuses once-computed sampling probabilities, both temporally and spatially between pixels. This method proved to be more efficient in terms of visual quality and speed of convergence, while also being scalable.

3.1 Weighted Reservoirs

ReSTIR can limit its use of shadow rays to only two for each pixel if performance is limited. It uses structures, so-called weighted reservoirs, to hold chosen light source candidates to sample in the current frame.

3.1.1 Weighted reservoir sampling

Reservoir sampling is a group of algorithms for randomly selecting N items from a stream of unknown or very large size M. It does so in a single pass and can be used for a dataset too large to fit into memory.

This form, which selects a random subset from the stream uniformly, is extended by weighted reservoir sampling (WRS). WRS is a technique used to sample a N-subset of elements from a stream of M elements with unequal probabilities, based on some weights assigned to each element as $p_i = w(x_i) / \sum_{j=1}^{M} w(x_j)$.

In a simple variant (N=1), a weighted reservoir structure carries one element candidate y, total number of seen elements M, sum of their weights w_{sum} and a control weight W. The reservoir structure has a defined update function, which is used to evaluate a new element x_i with its weight $w(x_i)$ in the stream. The reservoir element selection either discards the new element or accepts it, discarding the previously saved one. After the stream is gone through, the structure carries the selected element. Weighted reservoir structure's contents and update function pseudocode is shown in Algorithm 2.

Algorithm 2 Weighted Reservoir

struct RESERVOIR
  $y \leftarrow 0$
  $w_{sum} \leftarrow 0$
  $M \leftarrow 0$
  $W \leftarrow 0$
  function UPDATE(ELEMENT x_i, WEIGHT $w(x_i)$, COUNT m_i)
    $w_{sum} \leftarrow w_{sum} + w(x_i)$
    $M \leftarrow M + m_i$
    if rand() $< (w(x_i)/w_{sum})$ **then**
      $y \leftarrow x_i$
    end if
  end
end

3.1.2 Streaming resampled importance sampling

While RIS in Section 2.1.3 needed to store all M elements for final element selection, applying WRS to RIS transforms it into a streaming algorithm, streaming resampled importance sampling (SRIS), that can produce the result saving only one element at a time (in the case of N=1). Pseudocode is shown in Algoritm 3. Weights are computed according to Equation 2.13 as RIS weights $w(x) = \hat{p}(x)/p(x)$.

Algorithm 3 Streaming RIS using WRS

function SRIS(PIXEL q)
  Reservoir r
  for $i \leftarrow 1$ to M **do**
    generate sample $x_i \sim p$
    r.update$(x_i, w(x_i))$
  end for
  $r.W \leftarrow \frac{1}{\hat{p}_q(r.y)} \cdot \frac{r.w_{sum}}{r.M}$
  return r
end

Control weight W in the reservoir during SRIS is derived from Equation 2.15 under the assumption that N=1. It is used during the final pixel shading as shown in Algorithm 4. Weighted reservoir structure is used per image pixel and spatial complexity of this array of reservoirs is dependent on image resolution (and N, but we assume it is equal to 1), not on number of candidate elements M. However, time complexity is still linear in regards to M.

This process can be referred to as initial candidate selection. Elements to update into reservoirs are light sources in the scene.

Algorithm 4 Shading an image with SRIS

function SHADEPIXEL(RESERVOIR r, PIXEL q)
  return $f_q(r.y) \cdot r.W$
end
for all Pixel $q \in$ Image I **do**
  Image $I[q] \leftarrow$ shadePixel(SRIS(q),q)
end for

3.2 Temporal and Spatial Reuse

While the use of candidate selection through weighted reservoirs already decreases the variance enormously, ReSTIR makes use of the already computed information to further polish the result with non-complex computing operations, instead of increasing the number of candidates M, to which the time complexity is linear. After the initial candidate selection is executed on the image pixels, a function is used to combine reservoirs into one without the need to store or access all of their individual candidate elements. Each reservoir can be treated as a sample by itself and can thus be updated into another reservoir, as shown in Algorithm 5.

Algorithm 5 Combining Reservoirs

function COMBINERESERVOIRS(PIXEL q, RESERVOIR s,t)
  Reservoir r
  r.update($s.y$, $\hat{p}_q s.y \cdot s.W \cdot s.M$, $s.M$)
  r.update($t.y$, $\hat{p}_q t.y \cdot t.W \cdot t.M$, $t.M$)
  $r.W \leftarrow \frac{1}{\hat{p}_q(r.y)} \cdot \frac{r.w_{sum}}{r.M}$
  return r
end

3.2.1 Visibility reuse

Before the reuse of any pre-computed values takes place, it is advised to first perform visibility reuse. Because the initial candidate selection usually does not consider the visibility term for the PDF evaluation (for performance reasons as ray tracing for every candidate would be expensive in computation), larger scenes will have a problem with visibility noise as candidates might be selected with the path obstructed and thus the visible point effectively not illuminated with a possibility of propagating this candidate through reuses in the next steps.

To reduce this noise, a shadow ray should be traced right after the initial candidate selection. If the view of the selected candidate light source to the visible point is obstructed, the control weight W should be set to zero, eliminating its chance to be propagated further. Furthermore, zeroing out the control weight W gives the visible point in the pixel a higher chance of

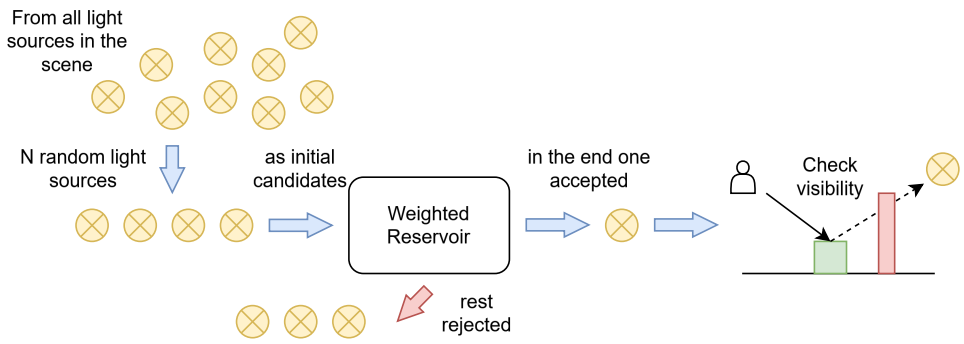


Figure 3.1: Choosing initial light source candidate with SRIS and weighted reservoirs, performing a subsequent visibility test.

selecting a different important light source propagated from reuses in the next steps instead. The run of the algorithm up until this moment, with SRIS and visibility reuse steps is shown in Figure 3.1.

3.2.2 Temporal reuse

Bitterli et al. [3] argue that computations executed in frames rendered prior to the current one can be reused because pixels are expected to have correlating shading properties and important light sources for illumination. While that holds true for a completely static scene, with an animated camera, motion vectors are to be utilized in a lookup for the position of a given pixel in the previous frame.

On a second rendered frame, combining the previous reservoir and the current reservoir (from the SRIS initial candidate selection) with Algorithm 5 results in a reservoir that examined double the amount of candidate elements (assuming the number of candidates in initial candidate selection per pixel per frame stays constant), while the time complexity did not double.

3.2.3 Spatial reuse

Just as visible points in the image are expected to somewhat share material and geometric properties, and important light sources, similar idea can be applied to neighboring pixels around the currently examined one with some limited radius. Algorithm 5 can be easily expanded for k number of reservoirs to combine, not just 2 as its traditionally used for temporal reuse. Furthermore, Bitterli et al. [3] showcase that combining k neighboring pixel's reservoirs can be done repeatedly to further polish the resulting image.

3.2.4 Considering bias

Spatiotemporal reuse as presented above brings in bias, because each pixel different integration domain and target distribution PDF, which breaks the

assumptions of RIS. Without compensating for this mismatch, the estimator becomes biased.

Origin of the bias

Biggest contributors are differences in normal orientation of visible points, their visibility of selected important light sources and also their material properties. For example, while two visible points might have similar material properties and might share normals, one point might be shadowed from a very close and powerful light source, which is almost guaranteed to be sampled by the other point. Combining these two reservoirs can propagate this light source to the shadowed point, which is shaded as not illuminated in return. For these reasons, bias typically results in an image that is darker than the ground truth even if computation and time resources are scaled up.

Bitterli et al. [3] demonstrate that bias stems from a RIS control weight $W(\mathbf{x}, z)$. Assuming N=1 in the RIS estimator 2.15, we can define function $W(\mathbf{x}, z)$ for $y \equiv x_z$ as:

$$\begin{aligned} \langle L \rangle_{ris}^{1,M} &= f(y) \cdot \left(\frac{1}{\hat{p}(y)} \cdot \frac{1}{M} \sum_{i=1}^{M} w(x_i)\right) = f(y) \cdot W(\mathbf{x}, z), \\ W(\mathbf{x}, z) &= \frac{1}{\hat{p}(x_z)} \cdot \frac{1}{M} \sum_{i=1}^{M} w(x_i). \end{aligned} \tag{3.1}$$

The expected value of $W(\mathbf{x}, z)$ is reciprocal of the sample PDF for the result to be unbiased. However, with varying PDF's it is equal to the following instead:

$$\mathbb{E}_{x_z=y}[W(\mathbf{x}, z)] = \frac{1}{p(y)} \cdot \frac{|\, Z(y) \,|}{M}, \tag{3.2}$$

where $|Z(y)|$ is the number of non-zero candidate PDFs whenever target PDF is non-zero. If $|Z(y)| = M$, the equation is reduced to $\mathbb{E}_{x_z=y}[W(\mathbf{x}, z)] = 1/p(y)$ and thus unbiased. But ff $|Z(y)| < M$, the $1/p(y)$ is further decreased by a multiplication with a number between 0 and 1. This is the cause of darker resulting images.

Unbiased RIS

According to Bitterli et al. [3], the bias can be eliminated by adjusting the control RIS weight $W(\mathbf{x}, z)$. The estimation:

$$\mathbb{E}_{x_z=y}[W(\mathbf{x}, z)] = \frac{1}{p(y)} \cdot \frac{1}{M} \sum_{i \in Z(y)} m(x_i), \tag{3.3}$$

for some function $m(x_i)$ while $\sum_{i \in Z(y)} m(x_i) = 1$ fulfills the requirements to be unbiased. The RIS weight $W(\mathbf{x}, z)$ is then modified to:

$$W(\mathbf{x}, z) = \frac{1}{\hat{p}(x_z)} \cdot m(x_z) \cdot \sum_{i=1}^{M} w(x_i). \tag{3.4}$$

The easiest valid $m(x_z)$ is proposed to be simply $1/|Z(x_z)|$, the number of so-far non-zero PDF light source candidates. Use of this unbiased reservoir combining can be seen in pseudocode in Algorithm 6.

Algorithm 6 Combining Reservoirs Unbiased with Uniform m(x)

function COMBINERESRVUNBIASED(PIXELS $q_1, q_2, ..., q_k$, RESERVOIRS $r_1, r_2, ..., r_k$)
  Reservoir s
  for all Reservoir $r \in$ Reservoirs $r_1, r_2, ..., r_k$ **do**
    s.update($r.y$, $\hat{p}_q r.y \cdot r.W \cdot r.M$, $r.M$)
  end for
  $Z \leftarrow 0$
  for $i \leftarrow 1$ to M **do**
    if $\hat{p}_{q_i}(s.y) > 0$ **then**
      $Z \leftarrow Z + r_i.M$
    end if
  end for
  $m \leftarrow 1/Z$
  $s.W \leftarrow \frac{1}{\hat{p}_q(s.y)} \cdot s.w_{sum} \cdot m$
  return s
end

However, it is argued that this approach, while valid, brings in a lot of noise. Combination with MIS balance weights is advised as:

$$m(x_z) = \frac{p_z(x_z)}{\sum_{i=1}^{M} p_i(x_z)}, \tag{3.5}$$

which requires evaluation of all PDFs for a given sample during the control weight W computation once. Unfortunately, this might involve shadow ray-tracing, as the visibility term might be crucial for PDF evaluation.

3.3 Algorithm overview

Thus the full algorithm, composing the previously gone through parts together, for a single frame does the following for all pixels with visible surface is shown in pseudocode, in Algorithm 7. For every pixel where a visible point in the scene is traced, initial candidate selection is executed through SRIS. After that, visibility reuse, one shadow ray cast towards the selected light source, is done. This ensures favoring different light source samples and no propagation of this light samples if obstructed. After temporal reuse and spatial reuse, a ray towards the final selected light source in the pixel's reservoir and the pixel is shaded.

Algorithm 7 ReSTIR

```
function ReSTIR(Reservoirs prev[ImageSize], Image I)
    Reservoirs reserv ← newArray[ImageSize]
    // Initial candidate selection
    for all Pixel q ∈ Image I do
        reserv[q] ← SRIS(q)
    end for
    // Visibility reuse
    for all Pixel q ∈ Image I do
        if shadowed(reserv[q].y) then
            reserv[q].W ← 0
        end if
    end for
    // Temporal reuse
    for all Pixel q ∈ Image I do
        q' ← getPrevPixel(q)
        reserv[q] ← combineReservoirs(q, reserv[q], prev[q'])
    end for
    // Spatial reuse
    for all Iteration i ← 1 to iters do
        for all Pixel q ∈ Image I do
            Q ← getNeighbors(q)
            R ← {reserv[q']|q' ∈ Q}
            reserv[q] ← combineReservoirs(q, reserv[q], R)
        end for
    end for
    // Shade
    for all Pixel q ∈ Image I do
        I[q] ← shadePixel(reserv[q], q)
    end for
    return reserv
end
```

3.4 ReSTIR Variants and Extensions

The ReSTIR algorithm comes in variants as it gives a lot of implementation freedom in terms of how probabilities and weights are computed, how light source candidates are selected, and how lights are sampled. The original ReSTIR for direct illumination by Bitterli et al. [3] has since been adapted also for global illumination as ReSTIR GI[4], improved as a generalized path tracer, ReSTIR PT[5] and expanded to interact with participating media as ReGIR.

3.4.1 ReSTIR GI

To apply ReSTIR's principles to the indirect illumination computation, indirect lighting paths (one-bounce, two-bounce) need to be represented and sampled. Reservoirs do need to sample directions, and need to be able to reuse them both temporarily and spatially to fully utilize the advantages of ReSTIR, as shown in Figure 3.2. Spatial reuse on different frame pixels, different visible points x_v in the scene, Ouyang et al. [4] represent the paths through visible points themselves.

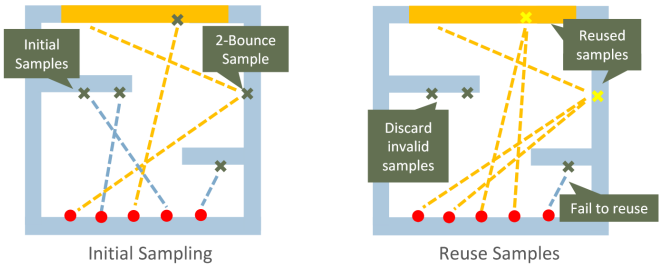


Figure 3.2: Sampling and reuse in ReSTIR GI, samples are generated by sampling random directions to find intersections and computing radiance at them by using path-tracing [4].

While initial candidates in original ReSTIR for direct illumination were produced by light sampling, ReSTIR GI's initial samples, named sample points x_s, are produced as the closest intersection of the traced ray of a randomly sampled direction from a visible point in the scene. Source PDF for the direction sampling can be uniform, a cosine-weight distribution, or based on BRDF/BSDF of the visible point's material. Outgoing radiance $\hat{L}_o$ at the sample point is then estimated.

This estimation at the sample point can either use emission and direct lighting only, making the algorithm compute a one-bounce global illumination, or path tracing can be leveraged, where n path-traced bounces result in $n + 1$ bounce global illumination computation. The sample is then saved in a struct representation, adding $\vec{n}_v$ as the normal of the surface at the visible point x_v and $\vec{n}_s$ as the normal of the surface at the sample point:

struct SAMPLEGI
    float3 x_v, $\vec{n}_v$
    float3 x_s, $\vec{n}_s$
    float3 $\hat{L}_o$
end

The algorithm uses three buffers with per-pixel stored info. Samples produced by the initial sample generation described above are stored in an initial sample buffer to be reused later. Same as with ReSTIR DI, temporal reservoir buffer contains a reservoir for every pixel from the previous frame and also a spatial reservoir buffer for spatial reuse.

Spatiotemporal reuse

Temporal reuse uses the initial sample buffer to update the temporal reservoir, similarly to ReSTIR DI. However, with spatial reuse, the differences between visible points' positions and surface normals need to be compensated for as the source PDFs are going to differ. The sample can be reused if the PDF is transformed to the different pixel's solid angle space. This is done as a division by the Jacobian determinant of this transformation [**?**]:

$$|J_{q\to r}| = \frac{\cos(\phi_s^r)}{\cos(\phi_s^q)} \cdot \frac{\| x_v^q - x_s^q \|^2}{\| x_v^r - x_s^q \|^2}, \tag{3.6}$$

where x_v are visible points (vertices) for pixels p and q, and ϕ_s are angles formed by the normal vector at the sample point x_s and vectors $x_v - x_s$.

3.4.2 ReSTIR PT

ReSTIR path tracing variant further extends ReSTIR GI, and compared to the former handles glossy surfaces, reflections, refractions, and overall convergence better. Because traditional RIS assumes independent, identically distributed samples, which is violated in ReSTIR due to sample reuse, spatial and temporal, Lin et al. [5] introduced generalized resampled importance sampling (GRIS) to allow reuse of paths with unknown or intractable PDFs while maintaining convergence and unbiasedness.

GRIS extends RIS to support correlated samples from multiple domains using so-called shift mappings. Given a function to integrate f with its domain Ω, and samples x_i drawn form domains Ω_i, those samples are then mapped with the shift mapping $y_i = T_i(x_i)$ to samples y_i in the domain Ω. Resampling weights w_i and control weights W_i under these new conditions are derived with a new technique, MIS resampling.

3.4.3 ReGIR

ReGIR variant uses a world-space grid of voxels. Each cell contains multiple reservoirs, so-called light slots, where one light sample is stored per light slot. The grid can be built in multiple ways, as a uniform grid that spans the entire scene (suitable for small scenes only), a grid in clipped range around the camera, a sparse hash-grid, or even an adaptive hash-grid with smaller cells near the camera. Original ReGIR proposal [12] used 16^3 cells and 512 light slots per cell, but both parameters depend on the scene's complexity.

Light selection is done in two steps - first, for every light slot in every grid cell, SRIS is executed to select initial light source candidates, sampled uniformly from all the light sources in the scene. It is advised to choose a very simple target PDF $\hat{p}(x)$ for this initial selection, like only light source intensity attenuated with the distance squared at the center of the grid cell.

Temporal reuse can also be used for grid-based approach for more stable results. It is to be used right after the first step of light selection. Every light

slot is merged with corresponding light slots from the previous frames (more than one can be used, default history of 8 frames back).

In the second step, a grid cell, where the visible point is located, is found. All the light slots in this grid cell are merged together to prepare for shading. However, a classic reservoir merge would be high for so many reservoirs (512), so SRIS is used again instead. This time samples are not drawn uniformly from the light slots, but it uses the target PDF $\hat{p}(x)$ from the previous step divided by an average reservoir weight of this grid cell as source PDF for this second step. Target PDF for current resampling is based on the visible point material's BRDF.

Finally, the visible point is shaded using BRDF, light intensity and corresponding RIS weight.

Chapter 4

Vulkan API

In September 2018, Nvidia introduced their GeForce RTX and Quadro RTX GPUs, bringing support for hardware ray-tracing. This opened the possibility for wide usage of real-time ray-tracing rendering. This feature can be used through Vulkan's ray-tracing extensions.

Vulkan is an open standard for 3D graphics and computing as of today developed by the Khronos Group. It is a low-level cross-platform API. It grants developers more control over the code's functionality on the GPU, thus making way for more efficient usage of GPU resources than the previously widely used OpenGL.

4.1 Resources

Similar to OpenGL, in the Vulkan API, buffers are essential GPU resources used to store arbitrary data such as vertex positions, indices, uniform data, or compute data. But unlike OpenGL, Vulkan provides and requires explicit control over buffer creation, memory allocation, and data transfer. While it enables high-performance graphics and compute workloads, it also places more responsibility on the developer.

Buffer up close, as a VkBuffer object, represents a region of memory that can store a sequence of bytes. The buffer does not include memory by default — memory must be allocated and bound manually. Buffers are usually used as: vertex buffers for vertices of chosen format, index buffers for vertex information reuse, uniform buffers for common data of all rendered vertices, storage buffers for larger data - mostly in compute shaders, and staging buffers for CPU → GPU data transfer. Their usage needs to be indicated during creation, together with memory type.

Upload of any buffer data from CPU to GPU is typically done through the use of a host-visible staging buffer. Data is first uploaded there and then copied to a device-local buffer through a command.

A different type of resource are the so-called push constants. They provide a highly efficient mechanism for passing small amounts of dynamic data from the CPU to any shader stage. Unlike uniform or storage buffers, push constants do not require separate buffer resources or memory allocations. They are a part of a command buffer, they are performant and are typically

used to update parameters of rendering.

4.2 Rendering Pipeline

The rendering pipeline in Vulkan is a configurable series of programmable and fixed-function stages. The pipeline is explicit and immutable, which means it must be created in advance with all configurations defined.

The rasterization process is done typically with the graphics pipeline type. It is run by a "draw" call and follows standard rasterization pipeline stages (input assembly, vertex shader, optional tesselation shaders, optional geometry shader, rasterization, fragment shader and blending). Resources for rendering need to be bound to the pipeline for access on the device. This is done through descriptor set objects.

4.3 Command buffer and synchronization

Unlike traditional graphics APIs where commands are executed directly, Vulkan requires that all rendering and compute commands be recorded into command buffers before execution. These command buffers, represented by a VkCommandBuffer handle, are submitted to a Vulkan queue for processing by the GPU.

Command buffers are allocated from so-called command pools. Once allocated, a command buffer enters the initial state. To record commands, vkBeginCommandBuffer() must first be used on the buffer, transitioning it into the recording state. During this phase, all desired rendering, compute, or transfer commands (starting with "VkCmd") are recorded. After recording is complete, the command buffer is ended with vkEndCommandBuffer(), making it executable. It can then be submitted to a queue via vkQueueSubmit() for execution by the GPU. After execution, the buffer may be reset using vkResetCommandBuffer() (or the entire pool can be reset), returning it to the initial state for reuse. This is illustrated in Figure 4.1

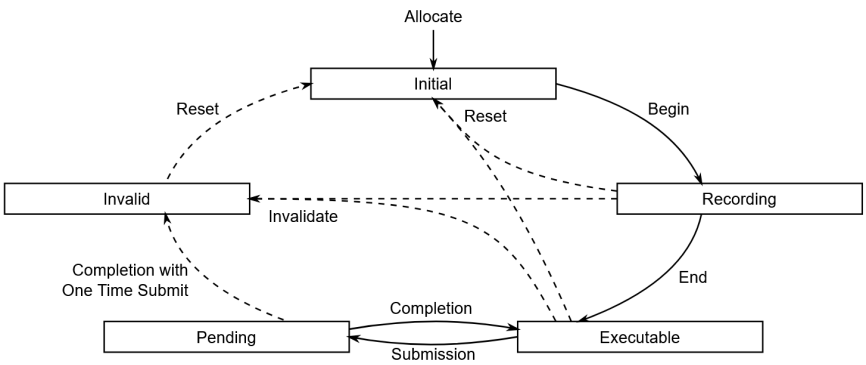


Figure 4.1: Vulkan command buffer lifecycle [18].

However, commands and resources must be manually managed to prevent

synchronization issues. Vulkan provides synchronization primitives: fences, semaphores, and events. "VkFence" is used for GPU-to-CPU synchronization, allowing the CPU to know when GPU operations have completed. "VkSemaphore" synchronizes operations between command operations on the GPU. "VkEvent" is used to signal and wait for specific points in the pipeline on the GPU without needing a full CPU-GPU synchronization.

For correct execution and memory access order across GPU operations, pipeline barriers and memory barriers are usually used. A pipeline barrier acts as a synchronization point that controls the execution order between commands and ensures that memory writes are visible to subsequent reads. Using vkCmdPipelineBarrier(), it is inserted into a command buffer, with specified source and destination pipeline stages, memory access types, and affected resources. Within a pipeline barrier, memory barriers are used to describe how memory dependencies should be handled between different pipeline stages. They ensure that operations like writing to an image or buffer in one stage are completed and made visible before reading from that same resource in a later stage. Three types of barriers are provided: "VkMemoryBarrier" for global memory dependencies, "VkBufferMemoryBarrier" for buffer-specific access, and "VkImageMemoryBarrier" for image layout transitions and access control.

4.4 Hardware accelerated raytracing

Vulkan currently offers acceleration structure and ray tracing pipeline extensions. They enable hardware-accelerated ray tracing on NVIDIA RTX graphics cards by supporting the use of a recursive ray tracing pipeline, acceleration structures, and ray tracing special shaders. Instead of a *VkCmdDraw()* calls, ray tracing pipeline uses *vkCmdTraceRaysKHR()* command.

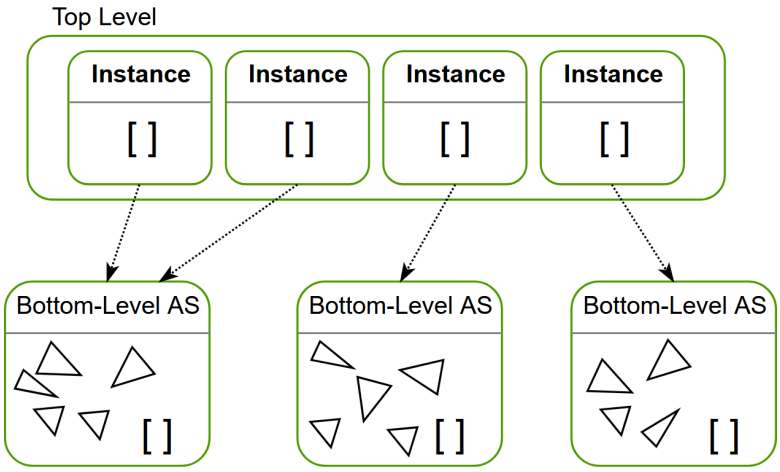


Figure 4.2: Acceleration structure schema [9].

4.4.1 Acceleration structures

Vulkan API provides two-level acceleration structures for efficient ray traversal and ray-scene intersection computations, which are managed hardware-wise. Bottom-level acceleration structures (BLAS) are for holding the actual geometry of models, and each one can encapsulate one or more buffers, as shown in Figure 4.2. Instances of the models and their transformation matrices are then provided to the top-level acceleration structure (TLAS). Dynamic scenes require TLAS to be rebuilt with rigid animations, both BLAS and TLAS require rebuild and update if the geometry itself is transformed.

4.4.2 Ray shaders

With raytracing extension, new types of shaders are available. This application uses *ray generation* shader, *ray miss* shader, and *ray closest hit* shader. Rest provided by Vulkan API are *any hit* shader and *intersection* shader.

Ray generation shader is run for every pixel and based on the camera setting casts a ray into the scene using *traceRayEXT()* function provided by Vulkan API and the extensions. This shader must always be implemented for the ray tracing pipeline to work.

Ray tracing through acceleration structure traversal is carried out and depending on traversal results, miss or hit shaders are run. Hit shader is then allowed to use the *traceRayEXT()* function once more for shadow rays. The progression of ray shaders is showcased in Figure 4.3.

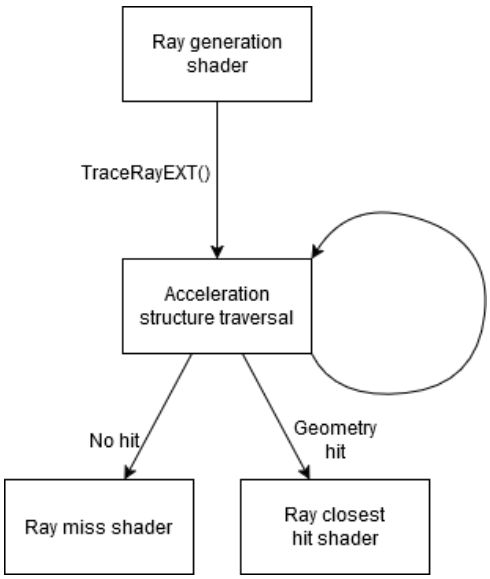


Figure 4.3: Ray shader execution in Raytrace.

Closest hit shader acquires hit object, hit primitive and hit point information, ray direction, pixel coordinates and transformation matrices for the instance both to world and to local coordinates. It returns computed color values back to ray generation shader.

Miss shader used directly from ray generation shader can return environment map texture values if turned on in the application, otherwise it returns clear color back to ray generation shader. The application then uses a second miss shader that confirms no hit for shadow rays.

Shader binding table

Shader binding table (SBT) is a structure that connects ray tracing pipeline to the appropriate acceleration structures. It is passed to the *vkCmdTraceRaysKHR()* command to determine which shader to invoke for each ray event. The SBT is divided into regions of a "ray generation table", which indicates an entry point ray generation shader for ray tracing pipeline, then a "miss table" for ray miss shaders, "hit group table" for closest hit, any hit, and intersection shaders and also optional "callable table", which is for general-purpose shader functions that can be invoked from other ray tracing shaders.

Chapter 5

Implementation

This section will describe important points for the implementation of this work. It is done with Vulkan API in C++ using NVIDIA nvpro-samples framework [9].

To enable hardware ray tracing support, extensions:

```
VK_KHR_ACCELERATION_STRUCTURE_EXTENSION_NAME,
VK_KHR_RAY_TRACING_PIPELINE_EXTENSION_NAME,
VK_KHR_DEFERRED_HOST_OPERATIONS_EXTENSION_NAME
```

are needed. With them, use of a recursive ray tracing pipeline, acceleration structures, and ray tracing special shaders is made possible.

The application has the following lifecycle: window and Vulkan initialization, application class initialization, scene loading and building, creation of resources for GPU - buffers, descriptor sets, and pipelines, then a main loop, and clearing of resources as showcased in Figure 5.1.

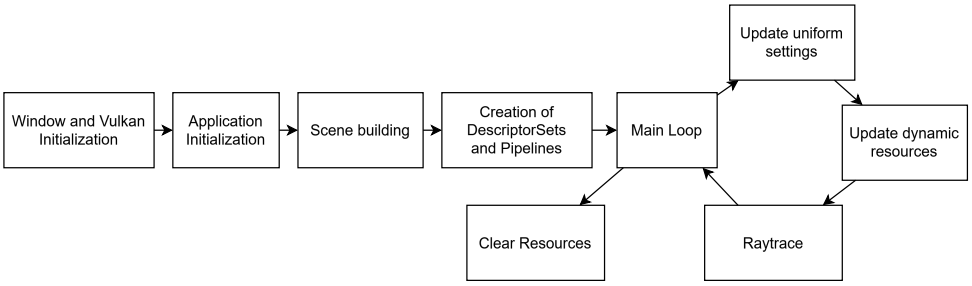


Figure 5.1: Overview of application lifecycle.

Main loop updates uniform buffers and all dynamic resources, raytraces scene and adds GUI. The implementation allows for the result to converge, and the number of pixel values used can be set in the GUI.

5.1 Scene Building

Because this work focuses mainly on the rendering of cities, there was a need for a format that multiple real-world cities are available in and also that

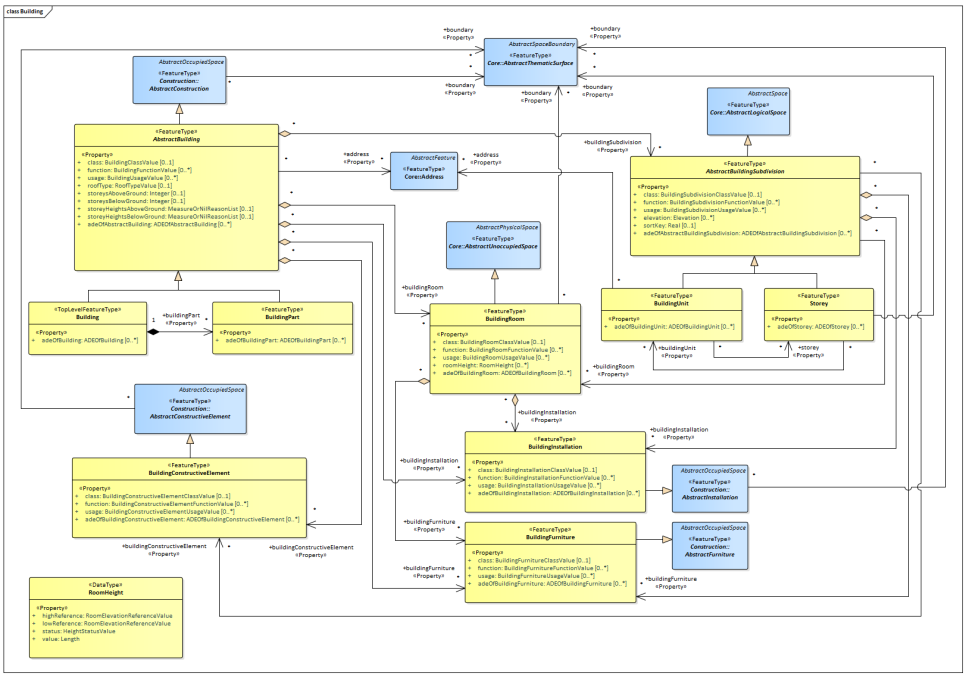


Figure 5.2: Example of CityGML class diagram - building [13].

encapsulates more semantic information than pure geometry. This is fulfilled by the CityGML format standard [10].

5.1.1 CityGML

CityGML is an open standard used for describing 3D models of landscapes and cities developed by the Open Geospatial Consortium. It is an implementation of Geography Markup Language (GML), an XML encoding for geographical data, ISO standardized.

The format supports geometrical 3D representations of city objects spatially grouped into recursive hierarchies, for example, walls with windows can be grouped under a building floor [13]. Features can carry topological information between its subgeometries. Every object can be labeled under a specific type, for example vegetation, city furniture, building, bridge, terrain, water body etc. Example of class diagram for building is shown in Figure 5.2.

Every object in the conceptual model is either of type feature, top-level feature, or geometry. Geometries, features, and top-level features all carry a featureID, a unique identifier in the entire dataset. Furthermore, features and top-level features can also contain an identifier, which would be the same for all features that describe the same object - as each object can have multiple versions within the same dataset. The featureID can be used for referencing within the dataset, typically for the use of geometries within features, or materials and textures for them. Thus topological relationships for shared geometry, or object instantiation within a scene graph are realized.

Spatial representation

The format describes five main LOD levels; LOD0 is typically the terrain model, LOD1 approximates city buildings as blocks, LOD2 roughly describes the exterior of individual city buildings and LOD3 provides detailed architecture models, as can be seen in Figure 5.3 LOD4 was intended for the addition of interiors; it was later, however, removed from the official conceptual model.

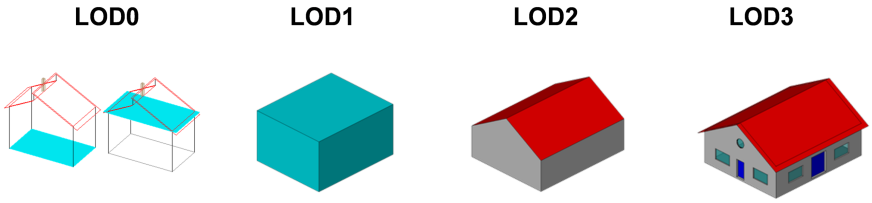


Figure 5.3: LODs in cityGML format [13].

Cities are represented under a root node CityModel, with CityObject nodes as children. CityObject nodes can represent and contain various things, but it is mostly terrains as TINRelief nodes and Building nodes. Building node is an aggregation of building parts, usually thematic surfaces like ground, roof, wall surface, etc., building installations like balconies and stairs, or for more detailed models even openings like doors and windows, individual rooms and furniture.

Main element of city object's geometry is usually a polygon. Polygons in GML are required to be planar surfaces or almost-planar, allowing small bends and inaccuracies within a given small threshold. Polygons are defined with one or more boundaries, LinearRing nodes. LinearRing node contains an ordered sequence of points in 3D space, that forms a closed non-self-crossing or -self-touching boundary. Polygon always has one as an ExteriorRing boundary, but can also include representation of holes as InteriorRing boundaries. Polygon has a defined normal vector. It is also important to note that geometry represented in this format is usually not triangulated by default, although a special variant of polygon does exist in GML as a Triangle element, a polygon bounded by four points with no InteriorRings.

If two polygons share an edge and their normals are in the same direction but they otherwise do not overlap or cross boundaries, they can be grouped into a CompositeSurface. Polygons can also be grouped together much less conservatively into an unstructured set called MultiSurface. MultiSurface geometry does not need to be connected, oriented a certain way or non-overlaping. In contrast a so-called Solid is a set of polygon boundaries, where each two polygons either do not intersect or share points and edges in-between, normal vectors of the defined surfaces point to the outside of the object that the Solid represents and all polygons are connected. A special type of CompositeSurface is TriangulatedSurface that consists of Triangle polygons only. If two (or transitively more) Solids (partly) share a surface and otherwise do not intersect, they can be further grouped into a CompositeSolid element.

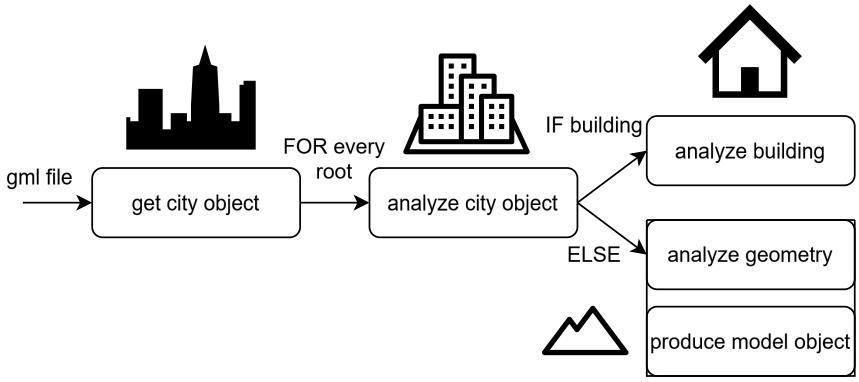


Figure 5.4: Parsing of the city object during scene building.

5.1.2 Internal representation

The application stores vectors of model references and instance references in the following format:

```
struct ObjModel
{
    uint32_t     nbIndices{0};
    uint32_t     nbVertices{0};
    nvvk::Buffer vertexBuffer;
    nvvk::Buffer indexBuffer;
    nvvk::Buffer matColorBuffer;
    nvvk::Buffer matIndexBuffer;
};
struct ObjInstance
{
    glm::mat4 transform;
    uint32_t  objIndex{0};
};
```

Each model stores its vertices with position, normal, color, and texture coordinates (not used in this implementation). Material contains values for phong shading. Simpler models (for cars and lamps) in *Wavefront .obj* are loaded into the application with the framework's provided loader.

For CityGML models, this implementation used an open source CityGML model loader for C++, *libcitygml*. It provides a functionality to parse city models of this format into a C++ instance of a citygml::CityModel object. By recursively requesting geometries and child objects, the whole city model is traversed as diagrams showcase in Figure 5.4 and 5.5.

5.1.3 City object traversal

During the traversal, the application stores every object labeled as a Building as one model together with all its child nodes' geometries. All other geometry types are stored as one model each, not grouped together (this is applied

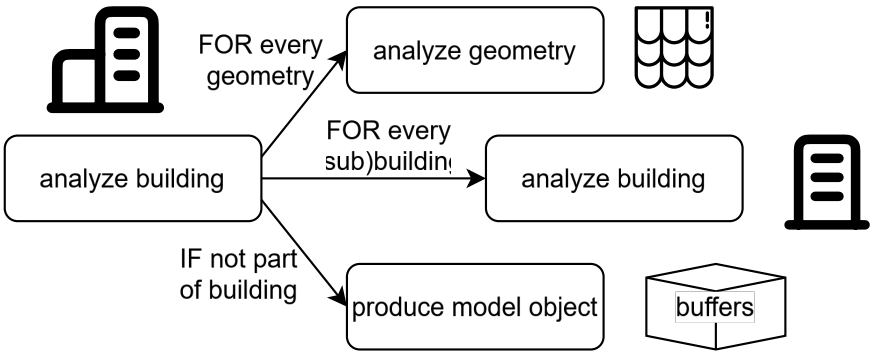


Figure 5.5: City gml structure traversal during scene building.

mainly to a terrain model). In the current implementation, for every model, one instance is created.

Every geometry node is first pre-scanned for sizes of buffers that need to be allocated. As mentioned previously, surfaces in cityGML format do not have to be triangulated, and typically aren't. This implementation uses a build of the libcitygml library that provides a tessellator for needed geometry triangulation. Every polygon in the geometry nodes contains lists of vertices and their indices. This is also the first instance where the library's code had to be adjusted for this implementation, as the tessellation did not preserve normal vectors, so with few small changes, the normal vector is correctly preserved in the Polygon object itself.

Because every vertex needs to be processed and transformed to a different format for this implementation anyway, differences between cityGML and Vulkan coordinate systems are solved during this processing (y and z axis coordinate switch, *up* flip) and not by applying transformations real-time.

Temporary vertex and index buffers are filled either as a single geometry for non-buildings or as a union of geometry and sub-geometries for building nodes. After the sub-tree is processed, model object can be produced. Model's buffers on the device are allocated and data submitted, an object descriptor for access in shaders is created. An instance of the object is also saved in the scene, together with transform if provided. Temporary vertex and index buffers are cleared. Geometry processing is showcased in Figure 5.6.

5.1.4 Materials

During the geometry processing, shared predefined materials are also assigned to individual primitives. Material properties can be adjusted at runtime inside the GUI for the terrain, roofs, windows, walls and other. These groups are differentiated based on the cityGML semantic labelling as surfaces and geometry contain this information under a "type" property. However, this is a place for a second adjustment in the libcitygml library that needed to be done. Neither geometry nodes nor the object nodes contained this information most of the time, as geometry nodes are typically predefined in the gml format files and later only referenced in the building node, which got lost in the

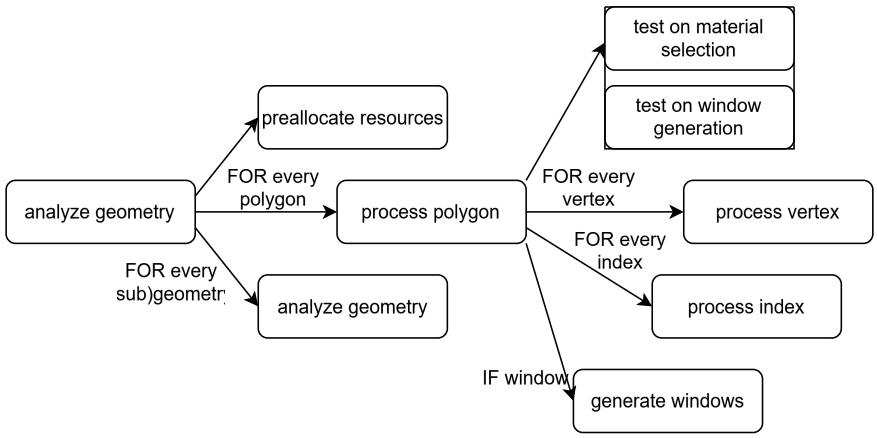


Figure 5.6: City gml geometry processing.

final structure production. The code was appropriately changed to retain type information in individual polygons for easier information access. Every primitive is then assigned a material index corresponding to the type of polygon it came from.

5.1.5 Window Generation

As wall surfaces can be recognized during the polygon inspection, this implementation provides a very simple window generation algorithm. Firstly, the normal vector of the polygon is evaluated. If it is perpendicular (within a threshold) to the scene's up vector, it is tested further. If bounding rectangle of the polygon is of given minimal surface area, width and height, it is deemed eligible for window generation.

The area of the bounding rectangle is divided into fixed size window areas and centered, With a non-zero margin, new rectangles are created from two triangles on every window area, shifted a small threshold in the direction of the polygon's normal vector forward, to in front of the wall surface. Wall partition is shown in Figure 5.7. Triangles are assigned a window material index and vertices get light color with random tint. Generated windows' buffers are stored separately from the rest of the geometry and they are also kept track of as separate light sources. Generated windows in the application are shown in Figure 5.8.

Through the GUI, ratio of windows lit up and not lit up is given. On change of the value, windows in the scene are iterated through and assigned a status of "on" or "off" taking the ratio as the probability. The appropriate material is then chosen at run-time in the shader according to this property.

5.1.6 Light Sources

The implementation supports light sources as point lights, sphere lights (by adding radius to the point light position) and emissive triangles. In the city scenes, street lamps, windows and car lights pose as light sources. While

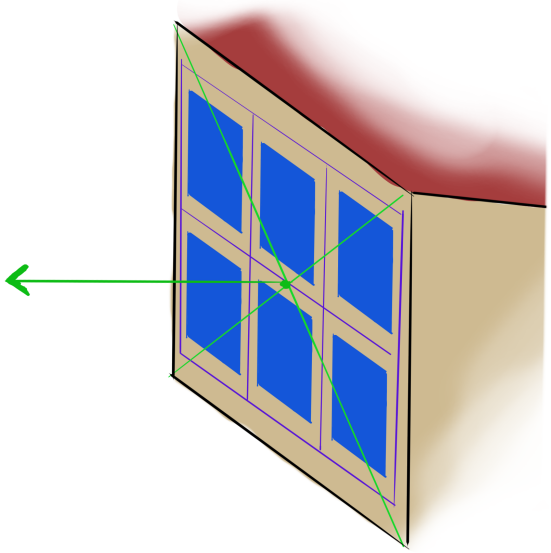


Figure 5.7: Wall partition for window generation. Window areas form a rectangular grid of fixed-size cells the centered inside the wall rectangle. Each window area holds a window - two triangles of new geometry each. Green is the normal vector of the wall surface

street lamps and car lights can (in the case of point lights use) already contain position and color information, for windows, the position needs to be uniformly generated on the surface and color is drawn from window vertices. Meanwhile, window area lights need to retain index information to appropriate buffers about model instance and vertices. Based on these requirements, light source instance is represented as a unified struct:

```
struct LightSource
{
  vec3 pos;
  vec3 col;
  uint type;
  uint midx;
  uint vidx;
  uint on;
  uint idx;
}
```

Intensity and attenuation is controlled through global parameters. Indexing is also unified, shaders assume fixed number of the static lights and the dynamic lights, thus the light sources can be ordered and indexed as: the static lights first, then the dynamic ones (if the scene is built as dynamic) and the area lights (windows), whose number varies depending on the chosen lit-up ratio.

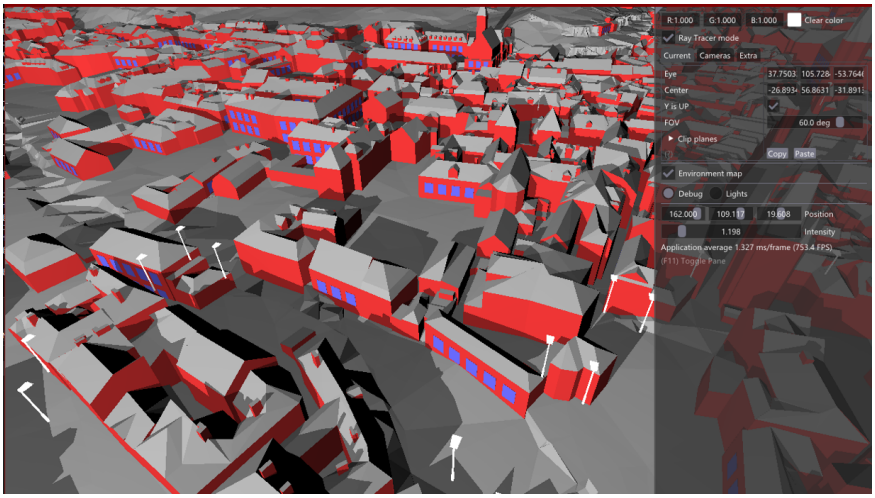


Figure 5.8: Showcase of generated windows(blue) on found wall surfaces(red) in Prague scene. They add #60,550 triangles to the scene, 2 per window.

Street lamps

Positions of real-world light sources were extracted as XML file from *OpenStreeMap*, an open, community-maintained map database. Light sources obtained are mainly of types "street_lamp", "lantern" and "floodlight". All nodes include latitude and longitude information, are loaded into the application with xercerc XML parser for C++.

Light sources from XML are parsed into StreetLamp instances.

```
class StreetLamp
{
  size_t     idx;
  glm::vec3  pos;
  double     lat;
  double     lon;
}
```

Then position is computed from latitude and longitude. For the Prague scene, the precise conversion to Křovák's projection [17], that the scene uses, was implemented; for the rest of the cities, the computation is not as accurate and relies on manually defining latitude and longitude bounds on the city model.

As the street lamps mostly do not contain information about elevation, they get later, together with car way points if added, grounded by a separate ray tracing pipeline. It simply finds the intersections with the ground in the down direction from the sky and the data is read back from the GPU to process.

The light source is not only added as a point light (spherical light), but also a model instance of a light source outer object, in the case of test scenes for this work - a lamp, and a visualization model instance of the light source

itself, which is masked in the shadow ray cast to not obscure the shadow ray into the light source, but still be visible.

Dynamic light sources

Windows are treated as emissive triangle light sources, while also being dynamic, as they can be made active/inactive at run-time. Fully dynamic light sources are also lights of moving cars through the scene. These have manually predefined paths in the individual scenes, made out of line segments, that instances of car models drive through in an animation loop. While the cars cycle through all the paths, the paths are not cyclic. The format of loaded path points is very simple XML of latitudes, longitudes and end parameters for individual points. End set to true signals, that said point is the end of the track and that the animation should not be interpolated to the next one, rather just jump. Dynamic light sources are kept in separate buffers for updating from the CPU.

5.1.7 BLAS and TLAS

BLAS are created only once, as the city scenes do not contain deformations of geometry. After all models in the scene are loaded and windows generated, BLAS is built for each registered model by transforming model's geometry into VkAccelerationStructureGeometryKHR geometry information representation and built using nvpro-samples helper functions.

TLAS is built afterwards, from all model instances in the scene with references to their built BLASes. However, as there are rigid animations in the scenes, TLAS needs to be rebuilt every time some object is animated, changing the instance's transformation matrix.

5.2 ReSTIR

For ReSTIR implementation, reservoirs had to be implemented according to the schema in Algorithm 2 as:

```
struct Reservoir
{
  Sample s;
  float w_sum;
  float M;
  float W;
};
```

Reservoir stores light source candidate as a Sample structure. This sample contains all the information needed for shading and reuse: vec3 position, vec3 normal, vec3 color, float pdf, a previously mentioned unified index of the light source, and a light type.

5.2.1 Candidate selection

The implementation follows the algorithm steps presented in Chapter 3. PDF is computed as suggested by Bitterli et al. [3] by $\hat{p} = \rho \cdot L_e \cdot G$, where ρ is a very simple computation of the Phong model, L_e is the light source's emission (length of its color vector multiplied by the intensity parameter), and G is a geometric term of the cosine of the angle between the light direction vector and the surface normal vector, divided by the squared distance of the visible point to the light source. This PDF is unshadowed for faster computation. Candidates are sampled uniformly from all active light sources, and also uniformly on their surface (for emissive triangles and spherical lights) and their number depends on global setting anywhere between one to forty. SRIS is performed on a temporary Reservoir structure.

5.2.2 Reuse

After candidate is selected through SRIS as outlined in Algorithm 3, a shadow ray is traced towards the candidate light source sample to check the occlusion. If the sample is occluded, reservoir's control weight is zeroed out.

For temporal reuse, pixel's position is firstly reprojected to the previous frame with the camera matrix from the previous frame and the visible point's world position, as illustrated in Figure 5.9. The computation directly mirrors rasterization pipeline approach, ending with the visible point's position in framebuffer space in the previous frame. Reservoir structure produced by candidate selection is then merged with a reservoir from this reprojected position taken from a storage buffer of past frame reservoirs.

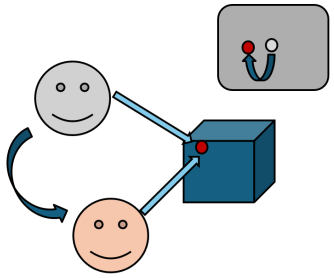


Figure 5.9: Temporal reprojection, camera position and orientation changed from the past frame, framebuffer space projection with past frame's camera matrix of the currently visible point is needed.

Afterwards, produced reservoir is placed into a storage buffer. Hit point and ray info is also stored, because memory barrier must be inserted into the execution to read correct values during the spatial reuse. Spatial reuse is thus executed in a ray generation shader after the execution is let through the barrier. Number of neighbors to merge and the radius to choose them in during spatial reuse is directed by a global parameter from the GUI. To lessen bias, the Algorithm 6 is used. Also the difference in direction of surface normals in the pixels is limited by a threshold, same for the differences in

depth. However, to eliminate bias completely, full, shadowed PDF would need to be computed, as occluded surfaces which are close to the light are otherwise much darker than ground truth. As this is a costly operation that requires cast of the shadow ray, this setting is optional in the GUI.

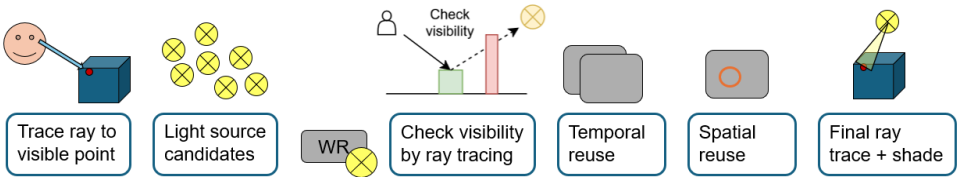
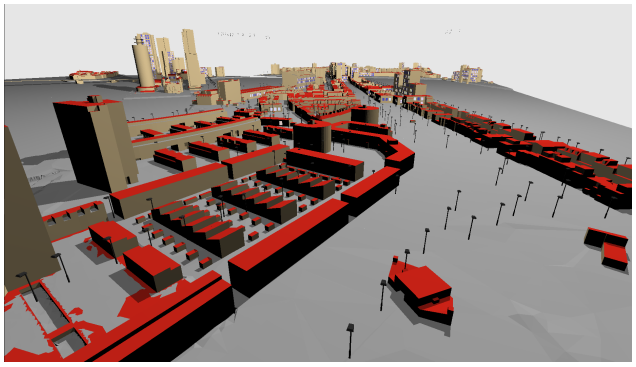


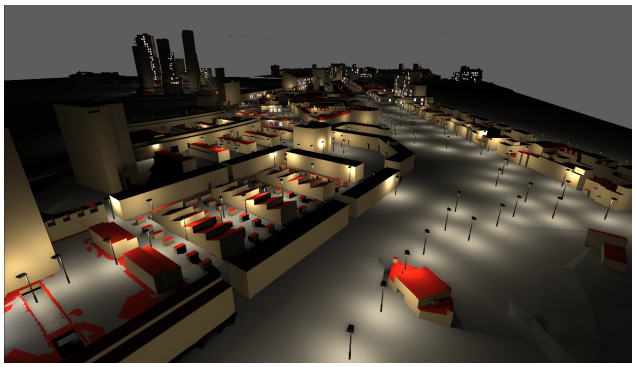
Figure 5.10: ReSTIR implementation overview. From visible point, to candidate selection in SRIS, to visibility reuse, temporal reuse, spatial reuse and final shading.

5.3 Exploration and Visualization Mode

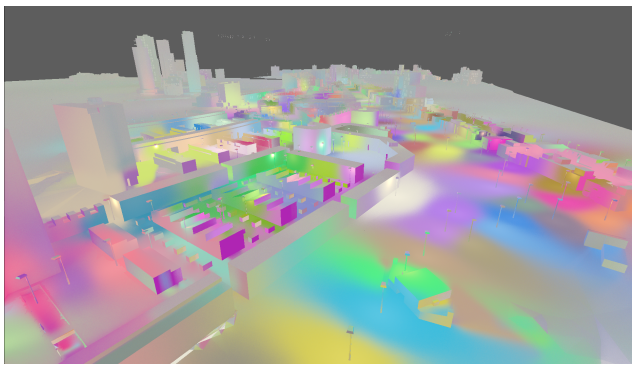
After the whole process, visualized in Figure 5.10, a final shadow ray is traced and weight zeroed out, if occluded. Then, for normal exploration mode, the visible point gets shaded and optionally mixed with past results to converge. Or, a visualization mode can be switched to instead. There are two visualization modes available, first one shows which light source is the most influential as the output color is set according to the chosen light source's index, and then outgoing radiance visualization, with values mapped to a color gradient. These different modes in a scene with city model of Rotterdam, can be seen in Figure 5.11.



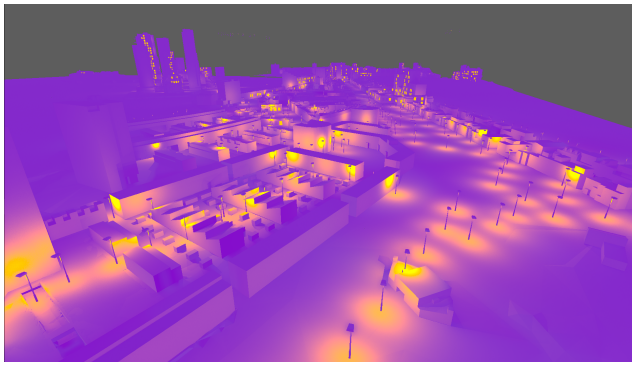
(a) : Model loaded, simple illumination.



(b) : Model illuminated with ReSTIR method.



(c) : Final sample selection visualization.



(d) : Outgoing radiance visualization.

Figure 5.11: Different implementation modes, shown on the city model of Rotterdam.

Chapter 6

Results

This chapter showcases results on three city model scenes - Prague, Rotterdam and Montreal. They are compared in terms of both visual quality and performance.

Testing was done on a desktop computer with NVidia GeForce RTX 3060 16GB graphics card, AMD Ryzen 5 5600G CPU, 32 GB of memory, and Windows 10 OS. For comparison with this work's efforts, a raytracer sampling all light sources across few frames was also added. This work used FullHD resolution for rendering, meaning there are 2M temporal reservoirs maintained and at about 6M rays traced per frame (visible point, candidate testing, final shading - with spatial visibility testing, it is even more).

6.1 Testing scenes

The city scenes, of Prague, Rotterdam and Montreal, were chosen based on their data availability in the cityGML format and street lamps registered in OSM. They differ vastly both in the scene complexity and in the light source count. These differences are introduced in Table 6.1.

Scene	#triangles	#generated	#windows	#lamps	#cars
Prague	1,081,353	60,550	30,275	337	22
Rotterdam	978,240	18,952	9,476	913	24
Montreal	1,587,696	2,524	1,262	115	-

Table 6.1: Comparison of scene characteristics: number of triangles, of those how many are generated (windows), how many windows there are, lamps and cars count. Lamps and cars are instanced (lamp model #72 triangles, car model #60 triangles), but added all into these statistics.

6.1.1 Prague

The first testing scene used for this work was obtained from Prague's Geo-Portal website. It consists of the model of part of Prague's buildings with terrain [26], as can be seen in Figure 6.1.

The scene uses light sources exported from OpenStreetMap shown in Figure 6.2a, and can also be loaded with a dynamic option to load tracks for

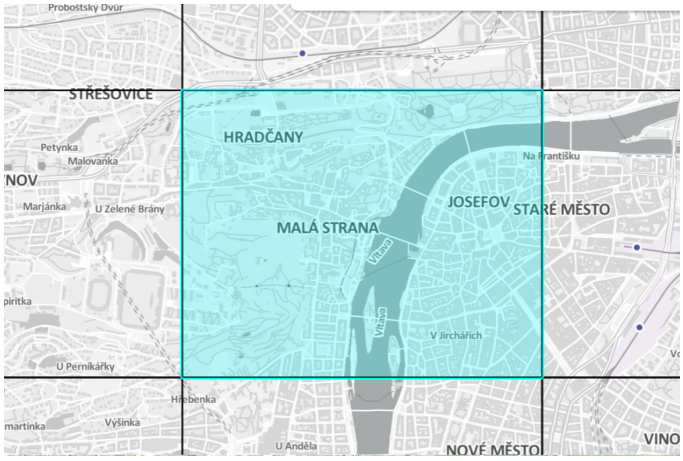
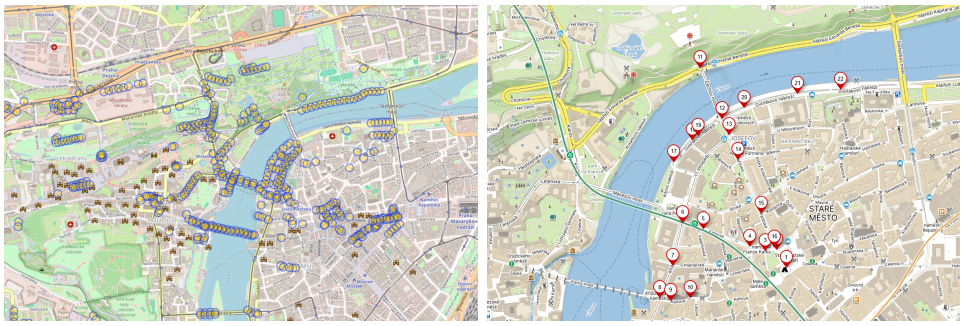


Figure 6.1: Prague scene area.

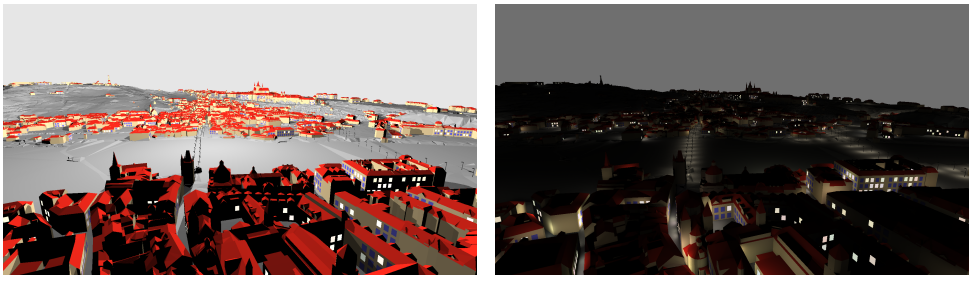
cars. The waypoints are shown in Figure 6.2b, they are three sets representing three different tracks. City scene after loading and building can be seen in Figure 6.3a. Render with ReSTIR method is shown in Figure 6.3b.



(a) : Light sources from OSM through OverpassTurbo.

(b) : Car waypoints in Prague, exported from mapy.cz and processed.

Figure 6.2: Prague scene resources



(a) : City scene after load and build.

(b) : City scene rendered with ReSTIR.

Figure 6.3: Prague city scene loaded, built and rendered.

6.1.2 Rotterdam

Rotterdam scene was exported from an online application [24] by a selection of three different regions, as Figure 6.4 shows.

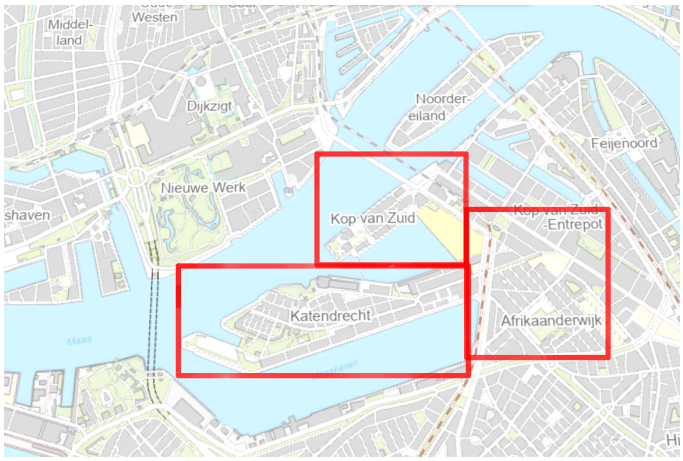
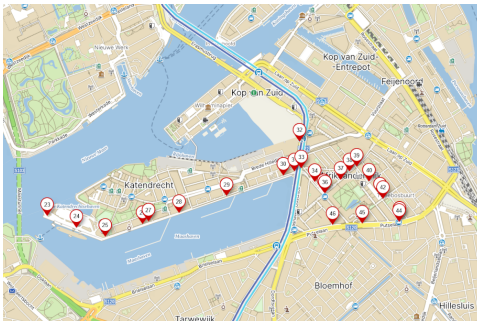


Figure 6.4: Rotterdam scene area (approximate).

Similarly to the Prague scene, street lamps were also exported from OSM and tracks from mapy.cz. This is showcased in Figure 6.5. As the conversion of the GPS coordinates into Rotterdam model's system (probably EPSG:25832, but it is not specified in the portal) is not as precise as Prague's, car waypoints are visibly off in the renders.

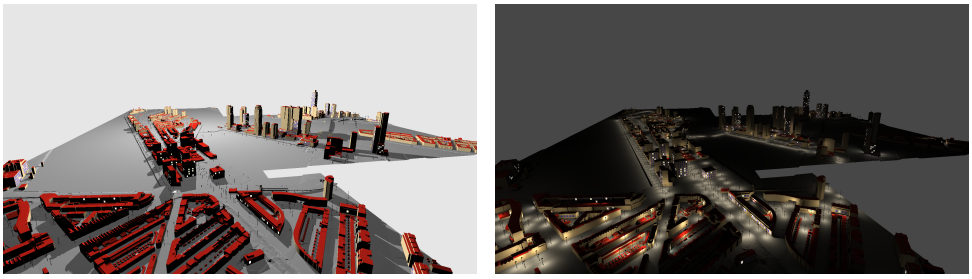
(a) : Light sources from OSM through OverpassTurbo.



(b) : Car waypoints in Rotterdam, exported from mapy.cz and processed.

Figure 6.5: Rotterdam scene resources.

Loaded and built Rotterdam city scene is shown in Figure 6.6a. Figure 6.6b then shows the render using the ReSTIR method. The window generation in this scene produced far fewer windows than in the Prague scene. While the wall surfaces were still found appropriately and the test on wall width passed, many building models here are short in height and thus the test on wall surface did not pass.

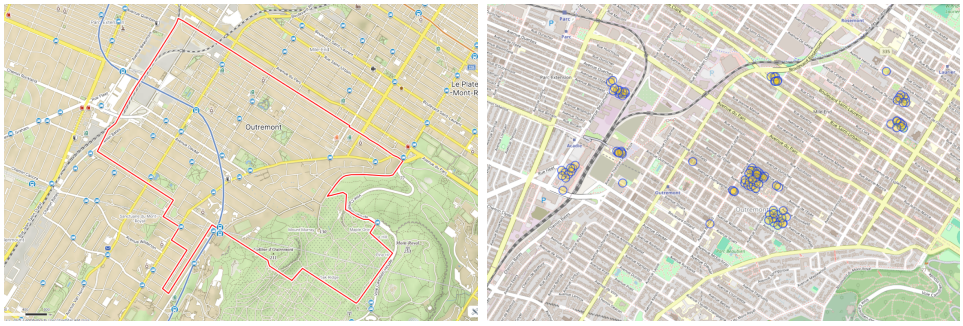


(a) : City scene after load and build. **(b) :** City scene rendered with ReSTIR.

Figure 6.6: Rotterdam city scene loaded, built and rendered.

6.1.3 Montreal

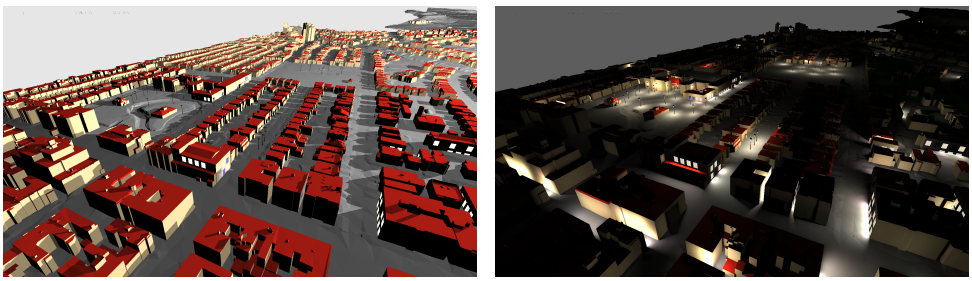
3D data of the city Montreal are available directly at the city's website [25]. The resources are shown in Figure 6.7. The coordinate system used, NAD83 SCRS (98) Projection MTM-08, is very different from the implementation's general GPS latitude/longitude converter and thus no car waypoints were added as none actually projected onto streets.



(a) : Montreal scene area. **(b) :** Light sources from OSM through OverpassTurbo.

Figure 6.7: Montreal scene model and resources.

Loaded and built scene for the Montreal city can be seen in Figure 6.8a. Render using the ReSTIR method is showcased in Figure 6.8b.



(a) : City scene after load and build. **(b) :** City scene rendered with ReSTIR.

Figure 6.8: Montreal city scene loaded, built and rendered.

The Montreal scene generated the least window instances out of the three models. The scaling of the provided models differs, the model is smaller after import compared to the others. And thus less windows generate, because the wall surface area is smaller.

6.2 Visual Quality

Even without reuse, just candidate selection through SRIS, immediately illuminates the scene considerably well, for Prague scene using the street lights, effect of the number of SRIS samples is shown in Figure 6.9.

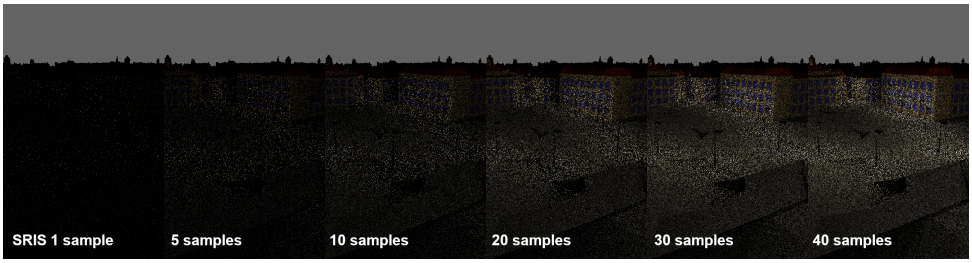


Figure 6.9: From left to right the number of samples used for initial candidate selection in SRIS increases and so does the visual quality. It can be noted that above 30-40 samples, the quality does not increase as much anymore, but the performance decreases.

However, as Figure 6.10 demonstrates, selected light samples for the scene are still considerably noisy, but with the addition of temporal and spatial(bisased/unbiased), the noise decreases dramatically. Biased version produces black noise around corners, as visibility is not tested in spatial reuse.

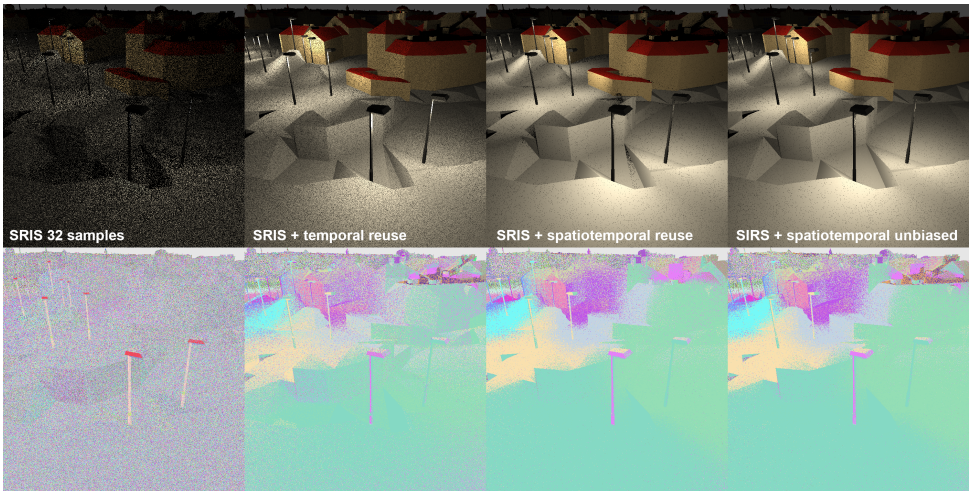
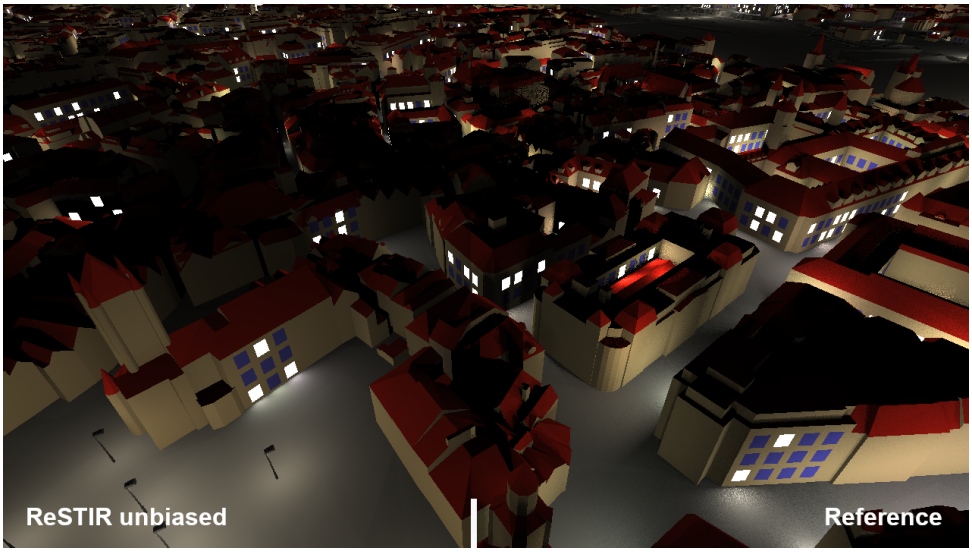
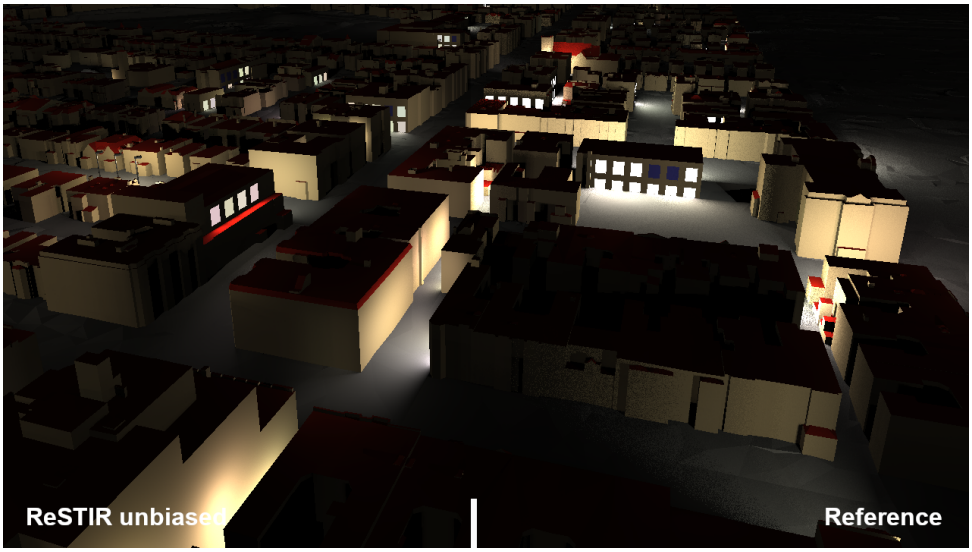


Figure 6.10: From left to right, full ReSTIR method is pieced together, ending with the unbiased variant, which traces shadow rays during spatial reuse - note the dissapearance of black noise around shape corners. Bottom row showcases the selected samples in the visualization mode.

Compared to the reference (by sampling all the lights in the scene), the unbiased variant of ReSTIR produces a close to identical render image. Such comparison can be seen in Figure 6.11.



(a) : Prague scene, root mean square error (RMSE) of 12.89.



(b) : Montreal scene, root mean square error (RMSE) of 13.63.

Figure 6.11: Unbiased ReSTIR produces an image very close to the reference image.

Comparison of converging first few frames of ReSTIR to the reference is shown in Figure 6.12. Associated root mean square error values are presented in a graph in Figure 6.13. The error stabilizes at around 11.0 for this shot.

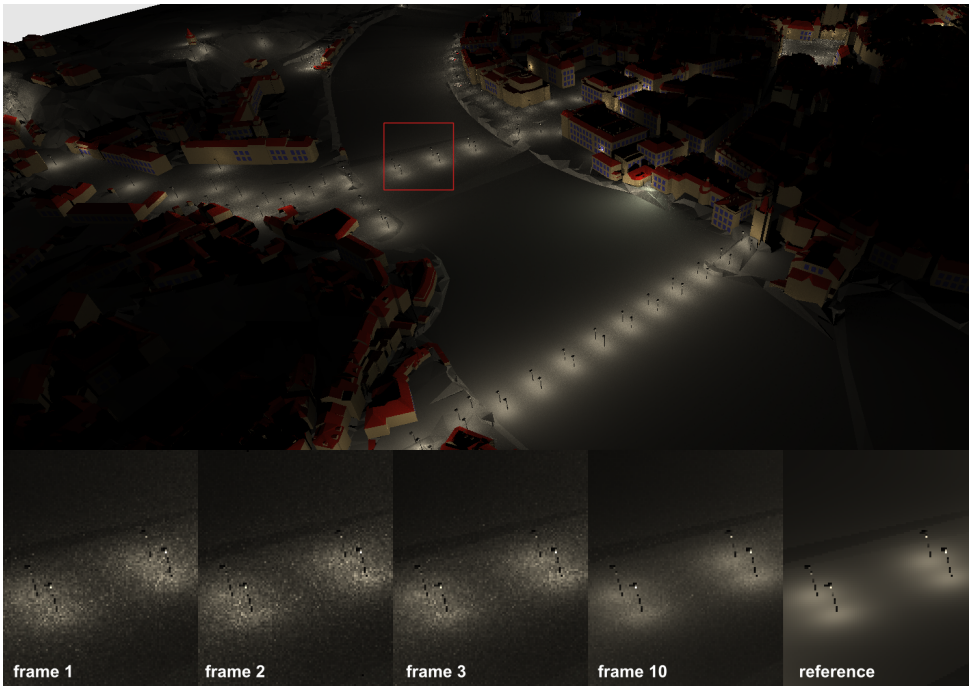


Figure 6.12: ReSTIR convergence frame by frame compared to the reference. Prague scene used.

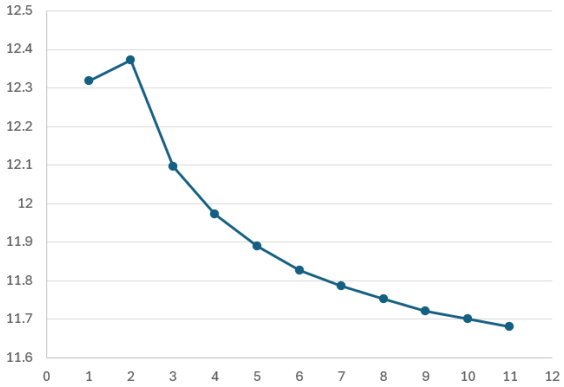


Figure 6.13: RMSE for the first 10 frames produced by the ReSTIR method. Prague scene used.

Dynamic light sources are present in Prague and Rotterdam scenes, for example of render with cars in Rotterdam, see Figure 6.14. With dynamic light sources in the scene, it is better to not let the result converge, because it will bring in a lot of ghosting, as full TAA is not implemented yet. This, however, allows for a lot of noise.

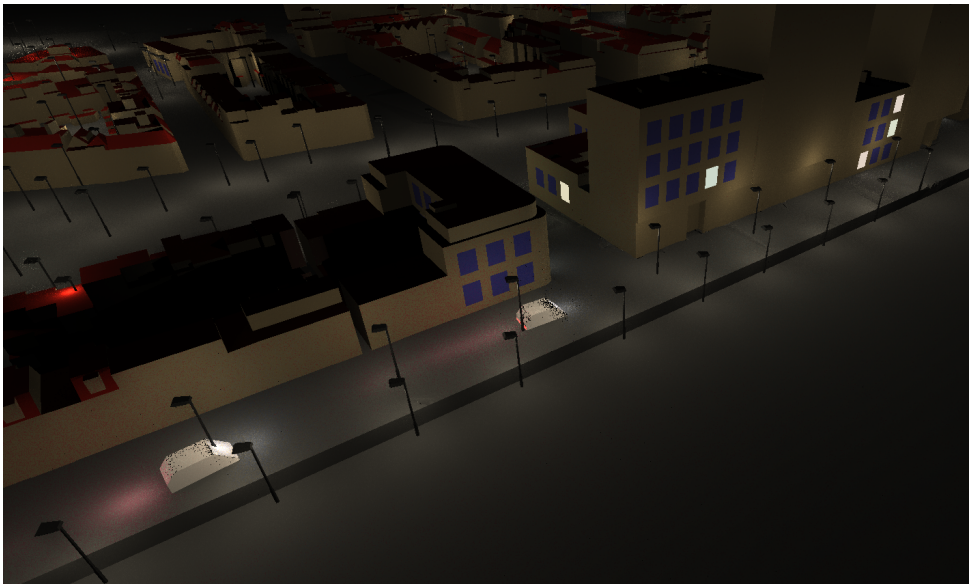


Figure 6.14: Cars as dynamic light sources in the Rotterdam scene. The car in the middle of the image is moving relatively fast compared to the on the left, and notice light selection ghosting on the ground from the car in the center. The reservoirs of these illuminated points still hold light sample from the car and it takes a few frames to lower its importance.

6.3 Performance

Testing showed that in terms of performance, the most influential are the ReSTIR settings and memory access. Testing with only street lamps gives the measurements in Table 6.2

Setup	Prague Scene	Rotterdam scene	Montreal scene
SRIS#1	144/6.94	204/4.90	173/5.78
SRIS#5	141/7.09	202/4.95	172/5.81
SRIS#10	139/7.19	199/5.02	173/5.78
SRIS#20	139/7.19	196/5.10	172/5.81
SRIS#30	137/7.30	190/5.26	165/6.06
SRIS#40	130/7.69	188/5.32	156/6.41
TEMP	123/8.13	173/5.78	150/6.67
ReSTIR	99/10.10	140/7.14	121/8.26
Unbiased	85/11.76	125/8.00	98/10.20

Table 6.2: The table tracks FPS/(ms) for every city scene under candidate selection only for the first six rows, with number representing the number of candidates, then with temporal reuse, then with spatial reuse (making it full ReSTIR) and then with shadow ray casting, making it unbiased. Illumination is by street lamp light sources only.

However, as Table 6.3 shows, the performance goes drastically down once window lights get sampled too. Testing various ratios of the lit-up windows,

this seems to be an issue with very inefficient access to storage buffers with resources for the window area lights. That is also proved by the fact that given smaller resolution, when fewer threads access the same data, the performance increases. Also as ReSTIR itself does not depend on the number of lights, as its number of candidates and shadow rays is fixed based on settings, this is the only explanation. It would also explain why the Prague scene suffers from this performance decrease the most, as its number of generated windows is the highest.

Setup	Prague Scene	Rotterdam scene	Montreal scene
SRIS#1	137/7.30	174/5.75	169/5.92
SRIS#5	86/11.63	154/6.49	144/6.94
SRIS#10	59/16.95	137/7.30	125/8.00
SRIS#20	37/27.03	108/9.26	99/10.10
SRIS#30	30/33.33	88/11.36	83/12.05
SRIS#40	21/47.61	79/12.66	74/13.51
TEMP	21/47.61	72/13.89	69/14.49
ReSTIR	21/47.61	62/16.13	62/16.13
Unbiased	21/47.61	55/18.18	55/18.18

Table 6.3: The table tracks FPS/(ms) for every city scene under candidate selection only for the first six rows, with number representing the number of candidates, then with temporal reuse, then with spatial reuse (making it full ReSTIR) and then with shadow ray casting, making it unbiased. Illumination is by street lamp lights and the window lights.

Performance also depends on the number of neighbors used for spatial reuse, as demonstrated in Table 6.4. Although these performance tests showcased high settings, the visual quality stays consistent for the optimal setting of 32 initial candidates and 5 neighbors, which does not decrease performance as much. Dynamic light sources decrease performance by about 3 FPS, given that TLAS needs to be rebuilt and it is not done asynchronously in this work.

	Prague		Rotterdam		Montreal	
Setup	bias	unbias	bias	unbias	bias	unbias
#1	123/8.13	92/10.87	163/6.13	163/6.13	142/7.04	140/7.14
#5	98/10.20	83/12.05	139/7.19	125/8.00	120/8.33	99/10.10
#10	83/12.05	60/16.67	135/7.41	95/10.53	102/9.80	71/14.08
#20	64/15.63	39/25.64	112/8.93	63/15.87	81/12.35	45/22.22

Table 6.4: The table tracks FPS/(ms) for every city scene under full ReSTIR, both biased and unbiased, with 40 initial candidates, depending on set number of neighbors for spatial reuse in individual rows. Street lamp illumination only.

6.4 Discussion and Future Work

The main limitation of this work is the performance. The performance of ReSTIR should remain unaffected by the number of light sources present in the scene, only add noise. However, as the number of generated light

windows increases, the performance degrades significantly. This suggests that the current implementation does not scale efficiently with scene complexity. A crucial direction for future work will be the optimization of memory access patterns and synchronization mechanisms, which currently represent significant bottlenecks.

Another critical limitation is the observable noisiness in rendered images, which can largely be attributed to the inefficient light candidate selection strategy. For this work, candidate selection was performed uniformly, without accounting for the relative contribution of each light source. A better sampling strategy could be used, such as power-based selection, which would potentially improve rendering quality. However, considering that the city scenes are usually characterized by a high number of similar low-intensity "minor" light sources, the benefits of power-based sampling may not be as high. Additionally, integrating a denoising step could further help minimize visual noise and enhance output quality.

A broader goal for future development is the visualization of light pollution. This could be achieved by extending ReSTIR toward a generalized path tracer and incorporating participating media like smoke, fog, clouds, and rain into the rendering pipeline. This could be achieved through ReGIR grid sampling in conjunction with ray tracing to accurately model light interactions within these media.

Furthermore, an important point to note is the need for enhancement of code modularity. The current implementation is not very flexible for scalability and integration of new methods and their variants.

Chapter 7

Conclusion

Firstly, I described the task at hand and mapped out existing methods of rendering scenes with many light sources and possibly for light pollution and participating media effects. Out of the methods mentioned, the ReSTIR method seemed to surpass the others mentioned in terms of both quality, performance, scalability, and flexibility.

I outlined features of Vulkan API which is used for this work. Then I described the implementation's architecture, functionality, data loading, scene building, and rendering principles and details that were laid out. The implementation is able to load and build the city test scenes, together with lights and car tracks, and render them using the ReSTIR method, as I showcased on three different city scenes made from open city data. This work is limited in performance and lack of modularity, both of these issues, together with focus on rendering with participating media, is a topic for future work.

Bibliography

[1] Conte, M. *Real-time rendering of cities at night.* MSc thesis, University of Montreal, 2019.

[2] Moreau, P., Pharr, M. and Clarberg, P. *Dynamic Many-Light Sampling for Real-Time Ray Tracing.* In High Performance Graphics (Short Papers) (pp. 21-26). 2019, July.

[3] Bitterli, B., Wyman, C., Pharr, M., Shirley, P., Lefohn, A. and Jarosz, W. *Spatiotemporal reservoir resampling for real-time ray tracing with dynamic direct lighting.* ACM Transactions on Graphics (TOG), 39(4), pp.148-1. 2020.

[4] Ouyang, Y., Liu, S., Kettunen, M., Pharr, M. and Pantaleoni, J. *ReSTIR GI: Path resampling for real-time path tracing.*In Computer Graphics Forum (Vol. 40, No. 8, pp. 17-29). 2021.

[5] Lin, D., Kettunen, M., Bitterli, B., Pantaleoni, J., Yuksel, C., Wyman, C.*Generalized Resampled Importance Sampling: Foundations of ReSTIR*In ACM Transactions on Graphics (SIGGRAPH 2022). 2022, July.

[6] Křivánek, J. *Rendering equation and its solution.* Computer graphics III at MFF UK. `https://cgg.mff.cuni.cz/~jaroslav/teaching/2015-npgr010/slides/07%20-%20npgr010-2015%20-%20rendering%20equation.pdf`

[7] Cerezo, E., Perez, F., Pueyo, X., Serón, F., Sillion, F. *A Survey on Participating Media Rendering Techniques.* The Visual Computer. 21.10.1007/s00371-005-0287-1. 2005.

[8] Jensen, H. W., Durand, F., Stark, M. M., Premože, S., Dorsey, J. Shirley, P. *A Physically-Based Night Sky Model.* SIGGRAPH 2001.

[9] NVIDIA. *NVIDIA DesignWorks Samples.* `https://github.com/nvpro-samples`

[10] Open Geospatial Consortium. *CityGML Standard.* `https://www.ogc.org/publications/standard/citygml/`

[11] ISO/TC 211, Open Geospatial Consortium. *Geography Markup Language.* https://www.iso.org/standard/75676.html

[12] Boksansky, J., Jukarainen, P., and Wyman, Ch., NVIDIA *Rendering many lights with grid-based reservoirs.* Ray Tracing Gems II, Chapter 23. 2021.

[13] Open Geospatial Consortium. *CityGML Standard Conceptual Model.* https://docs.ogc.org/is/20-010/20-010.html#toc0

[14] Hlavová, T. *Rendering Night Cities.* CESCG 2025, April. https://cescg.org/cescg_submission/rendering-night-cities/

[15] Kajiya, J. T. *The Rendering Equation.* In Proceedings of the 13th annual conference on Computer graphics and interactive techniques. pp. 143–150 (SIGGRAPH 1986). 1986.

[16] Esri and IPR Praha and ČÚZK. *3D model of Prague, online application.* https://app.iprpraha.cz/apl/app/model3d/

[17] jonez1 *Projection conversion algorithm.* https://gist.github.com/jonez1/2e64c1fd186dadc6a3367663c9cee30b

[18] Khronos Group. *Vulkan documentation.* https://docs.vulkan.org/spec

[19] Hašan, M., Pellacini, F., and Bala, K. *Matrix Row-Column Sampling for the Many-Light Problem.* SIGGRAPH, 2007.

[20] Yuksel, C. *Stochastic Lightcuts for Sampling Many Lights.* IEEE Transactions on Visualization and Computer Graphics, 2020.

[21] Yuksel, C., Lin, D. *Real-Time Stochastic Lightcuts.* Proc. ACM Comput. Graph. Interact. Tech. (Proceedings of I3D 2020), 3, 1, 2020.

[22] Walter, B., Fernandez, S., Arbree, A., Bala, K., Donikian, M., and Greenberg, D. P. *Lightcuts: A Scalable Approach to Illumination.* SIGGRAPH, 2005.

[23] NVIDIA, Boksansky, J., Marrs, A. *The Reference Path Tracer* Ray Tracing Gems II, Chapter 14. 2021.

[24] Gemeente Rotterdam. *3D model of Rotterdam - buildings and terrain, open data.* https://www.3drotterdam.nl/

[25] Ville de Montréal. *3D model of Montreal - buildings and terrain, open data.* https://donnees.montreal.ca/dataset/maquette-numerique-plateau-mont-royal-batiments-lod2-avec-textures

[26] Geoportal Praha. *3D model of Prague - building and terrain, open data.* https://geoportalpraha.cz/data-a-sluzby/7e6316e95cfe4f36ae06bbfb687bf34b

Appendix A

Abbreviations

3D 3-dimensional

BLAS bottom-level acceleration structure

BRDF bidirectional reflectance distribution function

BVH bounding volume hierarchy

CHC coherent hierarchical culling

FPS frames per second

GML Geography Markup Language

GUI graphic user interface

IS importance sampling

ISO International Organization for Standardization

MIS multiple importance sampling

RIS resampled importance sampling

SRIS streaming resampled importance sampling

TLAS top-level acceleration structure

UI user interface

URL uniform resource locator

WRS weighted reservoir sampling

Appendix B

Attached files index

```
/
├── img
├── src
├── bin
└── "README"
```

- **img** - screenshots
- **src** - source code

Appendix C

User Manual

C.1 Build

Application is dependent on xercesc, libcitygml and nvpro_core libraries. Precompiled xerces and libcitygml binaries are prepared for x64 Windows architecture in the /src/bin folder.

Either use them or build xercesc, then use it to build libcitygml included in src/libcitygml (forked and modified!!).

These DLLs are expected next to the application binary on Windows architecture.

Library nvpro_core should be inserted as a folder into /src.

C.2 Run

Application needs city and terrain model, light positions, and optionally car track points to run as:

```
/src/media/city.gml
/src/media/terrain.gml
/src/media/lights.xml
/src/media/tracks.xml
```

Prepared lights and tracks are already included for Prague tile 57-1 from GeoPortal(as shown in the Prague scene description), models are not. Terrain and city models are of cityGML format and are expected to be in Křovák's projection.

Given the appropriate media files, the application can be run from the root folder. If tracks are available the application accepts an additional parameter $-dyn$ which initiates loading and preparing of cars in the scenes.

C.3 Controls

Figure C.1 illustrates the usage of this application.

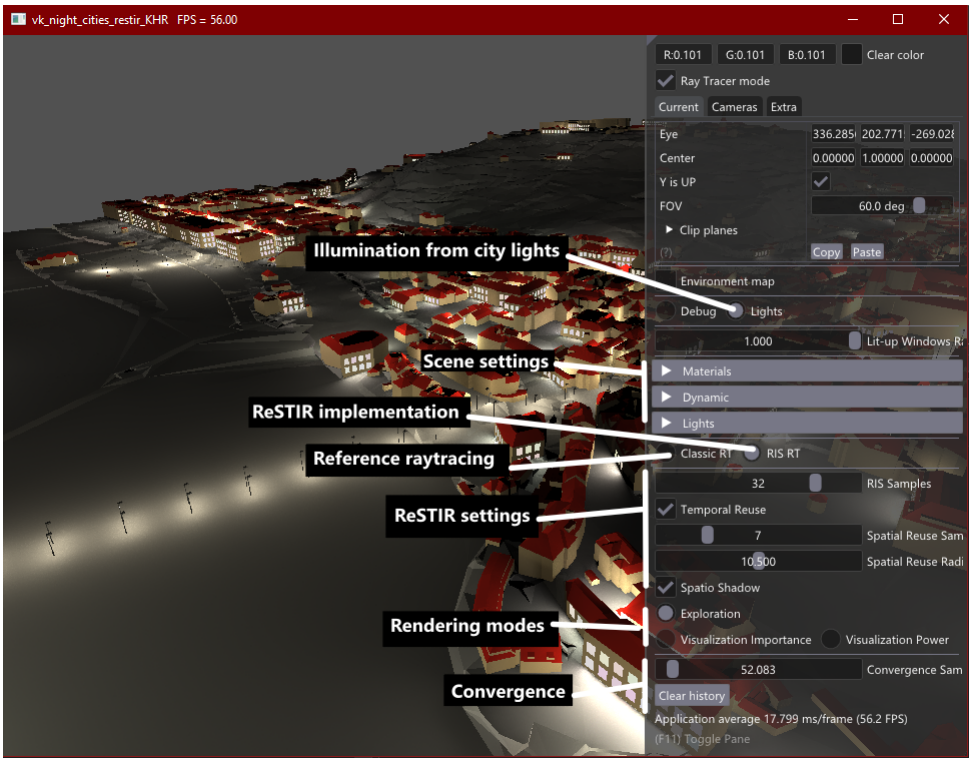


Figure C.1: Application controls.