Rendering Night Cities

Tereza Hlavová*
Supervised by: doc. Ing. Jiří Bittner Ph.D.†

Department of Computer Graphics and Interaction
Czech Technical University in Prague
Prague / Czech Republic

Abstract

This paper focuses on efficient real-time rendering of cities at night, working with open geographic and geometric data of cities. Our goal is to find a technique we can reuse for a generalized night city renderer in the future. We present a solution that implements a reservoir-based spatiotemporal importance resampling method ReSTIR to deal with many light sources problem. Our implementation is written using the Vulkan API and its hardware ray tracing extensions. The implementation uses an open, standardized format CityGML to load city descriptions and geometry. Apart from using real-world street light placement, we explore the possibilities of generating windows and moving cars as light sources. Results regarding quality and performance are demonstrated on freely available city models.

Keywords: ReSTIR, Vulkan API, ray tracing, CityGML

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 [6]. 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 night-time 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, one solution could be to build acceleration data structures over the light sources to minimize the amount of shadow rays that need to be traced. However, considering both static and dynamic light sources, for example moving cars through the city, changing advertisement displays or lights from interiors through windows being shut on/off, would require these data structures to be rebuilt, again causing time-consuming computation. Instead, we use a state-of-art method called ReSTIR [1] that does not require maintaining a complicated structure. We demonstrate our effective implementation utilizing hardware-accelerated ray-tracing on open geometric and geographic data.

In Section 2, we introduce existing methods used for rendering with many lights in the scene. We then explain chosen method, ReSTIR, and how it is used in our work in greater detail in Section 3. In Section 4, we go over our implementation. We then showcase the results in Section 5 and discuss limitations and future work in Section 6.

2 Related work

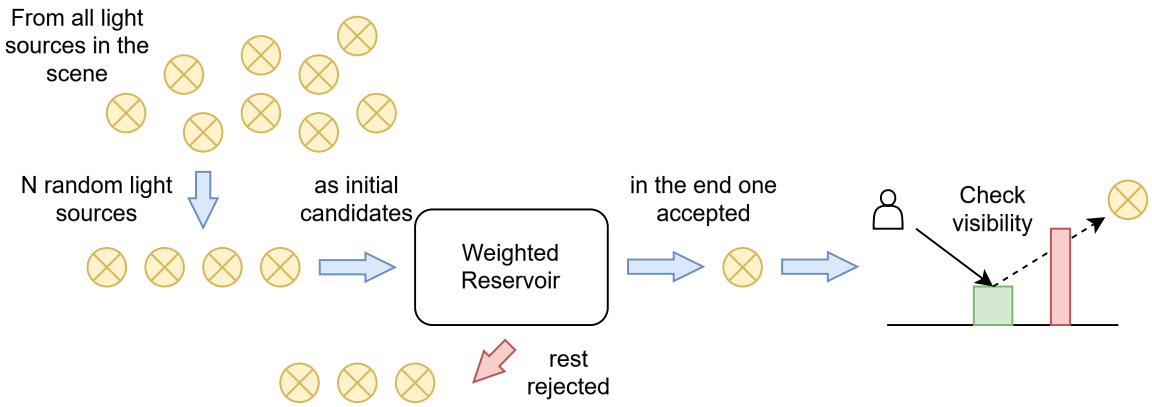
Rendering at nighttime can be explored from multiple points of view with different problems as the focus. An interesting problem to solve could be an efficient computation of natural illumination of objects by the moon, stars and atmosphere at night [8]. However, focusing on city scenes, our work is more focused on illuminating the scene mainly by a large number of human-made light sources.

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. 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.

For offline rendering, a notable ray tracing method used to solve the many-lights problem was introduced under the name LightCuts [15]. The method simplifies the light sampling by using clusters of light sources. They approximate all the light sources they 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, the so-called light tree, to form clusters of light sources by placing light sources into leaves and then

*hlavoter@fel.cvut.cz
†bittner@fel.cvut.cz

Figure 1: Choosing initial candidate with weighted reservoir sampling.



performing cuts of the tree.

Another prominent offline rendering technique is Matrix Row-Column Sampling [7]. It uses a transfer matrix of light sources and samples interactions, computing only a subset of rows and columns fully, reusing these calculations for the rest of the matrix.

From real-time techniques, a rasterization method for rendering cities at night proposed by Conte [5] is a combination of Facade-Cluster visibility technique, Coherent Hierarchical Culling (CHC), and integration of LightCuts onto GPU for real-time use with rasterization. It is limited by its rasterization nature, so expanding the method for global illumination or area light sources would be complex.

A dynamic many-light sampling real-time ray tracing method was proposed by Moreau et al. [10], it is working with a two-level bounding volume hierarchy (BVH) storing 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. 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). They are 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.

Bitterli et al. [1] introduced 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. Later modifications demonstrated also a variant for global illumination [12], volumetric rendering [2], and path tracing [9]. For these reasons, we chose ReSTIR as a base for our implementation.

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 and together with it brought in methods that would take advantage of this feature. With ray-tracing, limiting the number of shadow rays is extremely important, as every ray-trace operation is costly in real-time application.

3 ReSTIR for Night City Rendering

ReSTIR works under the assumption that only a small subset of light sources in the scene will significantly influence the result for a given illuminated visible point. For this purpose, a structure called a weighted reservoir is used.

In a simple variant, a weighted reservoir structure carries its sample candidate y, total number of seen samples M, sum of their weights w_{sum} and a control weight W. In our case, samples will be chosen light sources and weighted reservoir would be set up for every pixel.

The reservoir structure has a defined update function, which is used to evaluate a new sample x_i with its weight w_i. The reservoir either discards it or takes it for its chosen candidate sample, discarding the previous one, with probability:

$$p = \frac{w_i}{w_{sum} + w_i}. \tag{1}$$

Struct contents and update function pseudocode is shown in Algorithm 1.

The weighted reservoir sampling technique chooses N random samples out of all M. These samples are used to update the weighted reservoir. The light source's estimated importance for the sample can be measured as the length of exit radiance from the shaded point illuminated by said light source. Weight w_i is then computed here as the light source's importance divided by the probability of choosing the light out of all of the M.

Algorithm 1 Weighted Reservoir

```
struct RESERVOIR
    y = 0
    w_sum = 0
    M = 0
    W = 0
    function UPDATE(x_i, w_i, m_i)
        w_sum += w_i
        M += m_i
        if rand() ¡ (w_i/w_sum) then
            y = x_i
        end if
    end
end
```

Control weight W is used for the final shading after the sample is truly chosen. It should be set to zero in cases of the visible point being shadowed from the light source sample. It is otherwise updated as:

$$W = \frac{w_{sum}}{importance_i \cdot M}. \tag{2}$$

Initial candidate selection and described weighted reservoir update can be seen in Figure 1.

While the use of weighted reservoirs already improves the candidate selection enormously, ReSTIR makes use of the already computed information to further polish the result. It uses a function to combine reservoirs into one, as shown in the pseudocode in Algorithm 2.

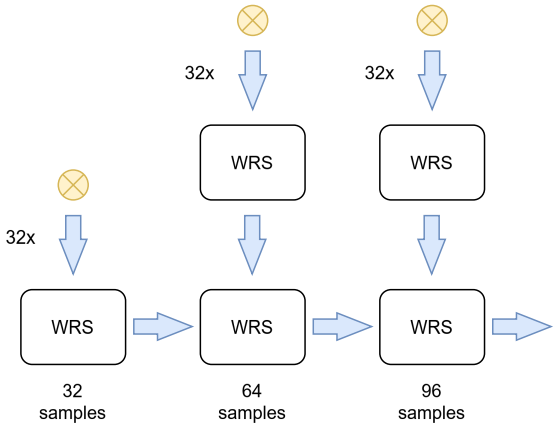
Algorithm 2 Combining Reservoirs

```
function COMBINERESERVOIRS(s,q)
    Reservoir r
    r.update(s.y, importance_s.y · s.W · s.M, s.M)
    r.update(q.y, importance_q.y · q.W · q.M, q.M)
    r.W = r.w_sum / (importance_r.y · r.M)
    return r
end
```

With this function, the idea is that the reservoirs of the same pixels can be reused in the next frame as can be seen in Figure 2, while also the reservoirs of the neighboring pixels will likely be illuminated by the same lights. Thus the full algorithm for a single frame does the following for all pixels with visible surface:

1. It generates N light source candidates and updates the pixel's reservoir with them, producing its chosen light source candidate.
2. It shoots a shadow ray towards the chosen light source candidate, updating the reservoir's control weight W accordingly.
3. It performs temporal reuse; the current reservoir gets combined with the last frame's one.
4. It performs spatial reuse, picks a number of neighboring pixels, and combines the current reservoir with theirs.
5. It computes the final pixel color by shooting a final shadow ray and computing radiance. Radiance is multiplied by control weight W and the pixel is shaded.

Figure 2: Temporal reuse in ReSTIR, with each frame, pixel's reservoir has evaluated more and more samples to choose the most influential one. Spatial reuse is done similarly.



In our implementation, light sources can be both static and dynamic. And if a radius is set for the light sources, they are considered as light spheres. This currently does not affect the ReSTIR algorithm that much as parameters of chosen sample on the candidate light source are not preserved in the reservoir and are chosen randomly every time a computation is needed. This, as a preparation for area lights, is a topic for future work.

4 Implementation

This section will describe important points for the implementation of our solution. It is done with Vulkan API in C++ using NVIDIA nvpro-samples framework [11].

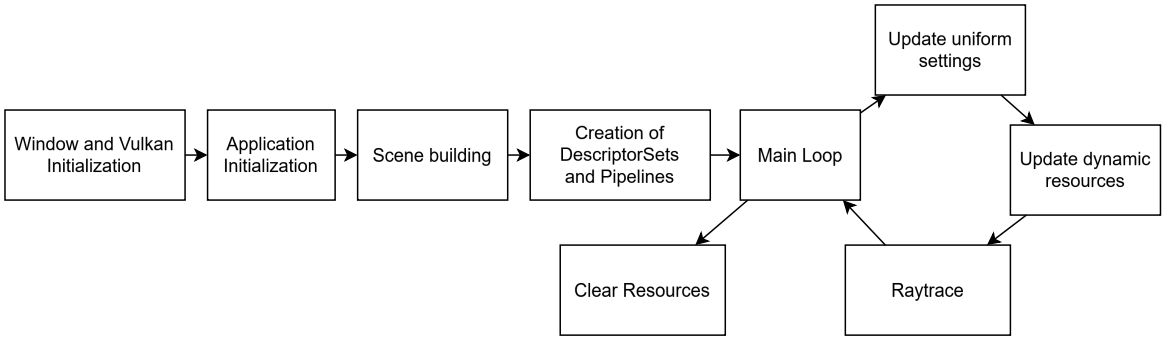
4.1 Resources

The most notable resources used by the application are buffers for: global uniforms (matrices for the camera, rendering settings), object descriptors and object data, reservoir structures, static and dynamic light sources (separate).

Figure 3: Weighted reservoir structure. 16B for each pixel.

int sample	4B
float w_sum	4B
float M	4B
float W	4B

Figure 4: Overview of application lifecycle.



The application stores the geometry of the models as a structure with buffers and their sizes. Then this geometry is referenced by object instances with an associated transform matrix for each one.

Reservoir structures are contained within a storage buffer. They are defined as a structure which can be seen in Figure 3.

Reservoir stores the light source candidate as a light source index to the light source buffers. Because the static and dynamic lights are kept in separate buffers, the index is recalculated as if the arrays were connected with dynamic light sources right after static light sources. Light sources are stored separately as static and dynamic in the application.

4.2 Main loop

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, main loop, and clearing of resources as showcased in Figure 4.

The main loop updates uniform buffers and all dynamic resources, ray traces 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. Every move of the camera resets the convergence values to avoid blur. The same happens upon a dynamic object being hit, though ghosting can be seen behind the object.

4.3 Vulkan API

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 previously widely used OpenGL.

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.

Vulkan API provides two-level acceleration structures for the ray traversal, the rest is 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. 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. For this purpose, the application keeps references to all TLAS.

With raytracing extension, new types of shaders are available. This application uses *ray generation* shader, *ray miss* shader, and *ray closest hit* 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 5.

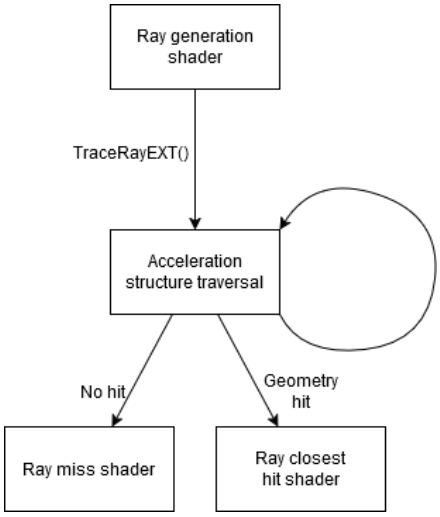
The closest hit shader acquires the hit object, the hit primitive and the hit point information, ray direction, pixel coordinates and transformation matrices for the instance both to world and to local coordinates. It returns computed color values back to the ray generation shader.

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.

4.4 Data

Models in *Wavefront .obj* format can be loaded into the application with the framework's provided loader. Each model stores its vertices with position, normal, color, and texture coordinates (currently unused) and material.

Figure 5: Ray shader execution in Raytrace.



As for models actually used, because this work focuses on the rendering of cities, with the goal of planning a robust full city renderer, there was a need for a format that multiple real-world cities are available in and also that encapsulates more information than pure geometry. This is fulfilled by the CityGML format standard [4]. 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 describes five main LOD levels, for this work LOD 2 was important, as it describes the exterior of individual city buildings and their positioning inside a city. It is also important to note that geometry represented in this format is not always triangulated by default. Surfaces can also be described as polygons or polygon multisurfaces.

This implementation used an open source CityGML model loader library for C++, *libcitygml*. This library provides functions to load a city encoded in such format into a citygml::CityModel object, which contains a tree structure of different city objects. If a tessellator is provided, objects that contain geometric information get their geometry data triangulated and prepared for reading. Other metadata and labels are also saved for object nodes. Traversing child geometries and other child city objects, the whole city representation can be traversed. During such traversal, our 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 mainly to a terrain model). In the current implementation, for every model, one instance is created, later processed into one BLAS for ray-tracing. citygml::CityModel object's bounding box is used to center the geometry in the scene.

The city models sadly do not have to contain real-world latitude and longitude, although most sources do provide model bounds externally as model information upon download, together with scale ratio.

4.5 Light sources

For the test scenes, 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, but almost none have elevation data. For this work, a custom tool was used that, through OpenElevation API requests, fills in all nodes' elevation data into the XML file and is then ready for loading into the application with *xercerc* XML parser for C++. For positioning the light sources into the scene, their latitude and longitude is recomputed againts the centered terrain and city model.

With the current implementation, all loaded light sources are assigned the same 3D model representation, a sample old town lamp. The visualization model instance of the light source is masked in the shadow ray cast to not obscure the shadow ray into the light source but to still be visible. The light source can either be a point light or a spherical light of a common radius chosen by the user in the GUI during the application run.

Dynamic light sources are also supported, currently for testing purposes only, in a form of light spheres with generated trajectories. Their positions are reuploaded onto the GPU every frame, if turned on. This setting also results in TLAS rebuilding.

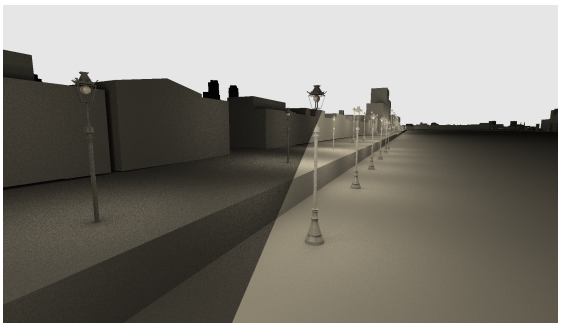
5 Results

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 ray-tracer sampling all light sources in every frame was also added as a reference. We will showcase results in two parts of cities: Prague and Rotterdam. We used FullHD resolution for rendering, meaning there are 2M temporal reservoirs maintained and at most 6M rays traced per frame (visible point, candidate testing, final shading).

In both cases, utilizing 32 candidates selection, temporal reuse, and convergence of the last 30 pixel samples, the results are still noisy, as expected. The result also seems to be biased, darker than the ground truth when all light sources are properly sampled, which is something the original ReSTIR paper also dives into. This can be seen in comparison with reference in Figure 6.

The candidates' usage in itself already makes a huge impact on the resulting image every time. The original paper states 32 candidates as the optimal number, and a similar outcome can be deduced out of this work's testing,

Figure 6: Comparison of ReSTIR implementation render frame on the left against reference on the right, in a scene of Rotterdam city. The result is biased, darker than the ground truth.



where using more candidates than 32 did not improve the result as much; rather, it took away from the performance a bit, which seems to also be the case here.

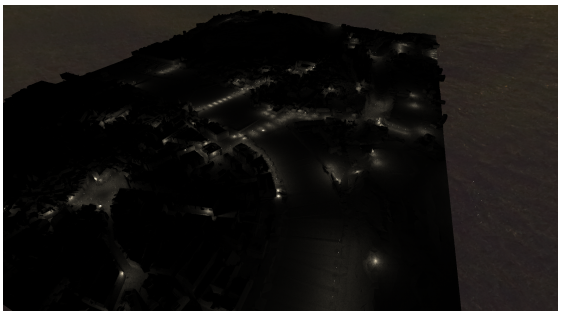
The reference raytracer easily falls down to 4-5 FPS, in case of a test (not included here as the reference was so unstable) with 2000 light sources even crashing. Meanwhile, ReSTIR running on 40 candidate samples and 30 neighbor spatial samples runs with stability in both our cases above 120 FPS.

Performance of ReSTIR is affected by ray traversal time (geometry and acceleration structure dependent), the number of light source candidates used, and how many neighbor reservoirs are reused. Following measurements are done from a single view in the test scenes.

5.1 Prague scene

The first testing scene used for this work was obtained from Prague's GeoPortal website. It consists of the model of part of Prague's buildings with terrain [13], and its light sources from OpenStreetMap, as can be seen in Figure 10a. This model together consists of over 300,000 triangles and is illuminated by 337 lights. A rendered frame of this scene using our implementation can be seen in Figure 7.

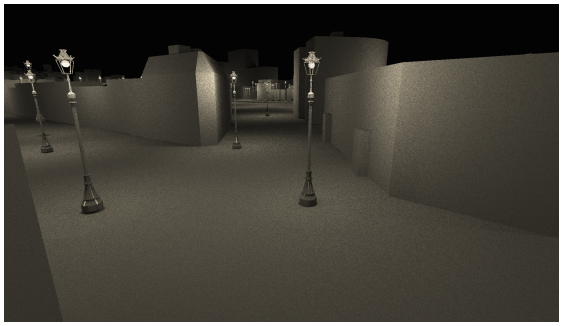
Figure 7: City of Prague rendered with the ReSTIR implementation, 337 light sources.



5.2 Rotterdam scene

The second testing scene was chosen to be from Rotterdam. The data is available through an online visualization application by selection and filtering of wanted content [14]. The model also consists of over 300,000 triangles, and it is illuminated by 913 lights. A rendered frame of this scene using our implementation can be seen in Figure 8. Furthermore, a visualization showcasing the outcoming computed radiance can be found in Figure 11.

Figure 8: City of Rotterdam rendered with the ReSTIR implementation, 913 light sources.



5.3 Measurements

Table 1: Performance on two testing scenes and their visual quality compared to the reference depending on the number of initial light source candidates for weighted reservoirs. Temporal and spatial reuse (5 neighbors). Performance in FPS/ms. For comparison, Prague's reference was around 7 FPS for Prague and 2 FPS for Rotterdam.

	Prague #triangles 304k		Rotterdam #triangles 311k	
#Cand.	perf	RMSE	perf	RMSE
1	279/3.58	10.68	371/2.69	23.32
2	278/3.60	9.60	371/2.70	21.53
4	275/3.64	8.35	369/2.71	19.33
8	270/3.70	7.14	362/2.76	16.76
16	254/3.94	6.17	334/2.99	14.05
24	229/4.37	5.84	304/3.29	12.57
32	201/4.98	5.69	271/3.69	11.62
40	176/5.68	5.59	245/4.08	10.96

5.4 Dynamic light sources

Dynamic light sources were tested in a simple scene. Their paths are randomly generated so far, for testing purposes only. This functionality is to be used when car traffic generation is done. These tests brought similar results as the static light sources tests, as Figure 9 shows.

However with rendering cities, it could be assumed that no objects would be moving as fast or be viewed as closely as in this simple testing scene to cause as much noise. Performance testing dynamic light sources found that TLAS

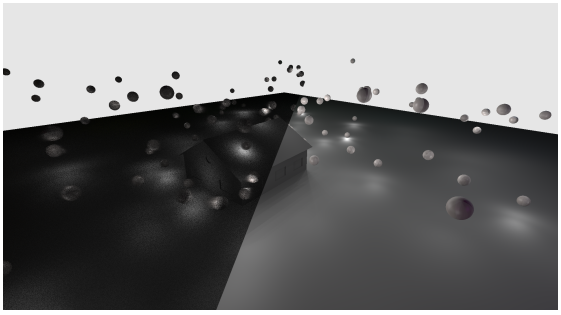


Figure 9: 50 dynamic light sources: ReSTIR 32 candidates left, reference right.

rebuild is probably the bottleneck there; rendering itself is not affected performance-wise.

5.5 Discussion

The performance of the implementation turned out to be quite view-dependent, future work should focus on stabilizing FPS throughout the model, probably by organizing the acceleration structures better. Also, the darkness of the images should be taken into consideration, especially their unpredictability. Both these factors could be seen in the results - while the Prague scene worked with fewer light sources, it performed worse than the Rotterdam scene with more than twice the light sources, but around 100 more FPS. On the other hand, the Rotterdam scene turned out much noisier and darker than the Prague scene. Moreover, adding in dynamic objects further darkens the image. A lot of the noise also comes from the usage of spherical lights instead of point lights. Preserving parameters from the sample selection on spherical/future area lights will probably be needed for better results.

6 Limitations and future work

The current implementation has many flaws - inefficient convergence computation, shader algorithms not modularized, and the test scene adjusted for only a few models. Also, the whole nvpro-samples framework is not needed for one standalone application, but it was easier to use for this initial testing. The application is currently being rewritten with the Daxa Vulkan framework with the goal of abstraction, modularization, and generalization. We want to allow seamless switching of city models and resources during the application run and also offer more control over the variant of the rendering algorithm used, additionally with optional visualizations.

Apart from that, for our future work, we see as our goal a full application, a general city renderer capable of handling big scenes with many light sources of various kinds. We also want to focus on the visualization of light pollution. ReSTIR could be modified and extended towards the generalized path tracer variant. Effects such as smoke, fog, clouds, and rain could also be added as participating media; they can also be considered with ray-tracing. The existing method that works with it is ReGIR, where reservoirs are grid-based.

The addition of generated windows [3] to building walls could also be possible because during the model load, if the model is of the CityGML format, the model parts labeling could determine where to generate them. The same goes for the placing of car traffic - if streets are marked as such, or optionally, roads could be exported from OpenStreetMaps.

7 Conclusion

For this work, we have introduced existing methods used to solve the many-lights rendering problem, as we aim to render city sceneries during nighttime, illuminated by street lamps, traffic, building windows, and various other light sources. We then focused on the ReSTIR method, explaining its core principles.

We went over the implementation's architecture, functionality, data loading, and scene building. Rendering techniques and details were laid out. Limitations of the current implementation were also highlighted.

The implementation is now able to load given city scenes, together with lights, and render them using the ReSTIR method. Light sources are point lights by default but can also work as spherical light sources. The implementation is also capable of rendering dynamic lights. Our work tested the method and let us decide to continue on this project, building a more complex and general renderer.

Acknowledgments

This work was supported by the Grant Agency of the Czech Technical University in Prague, No SGS25/150/OHK3/3T/13.

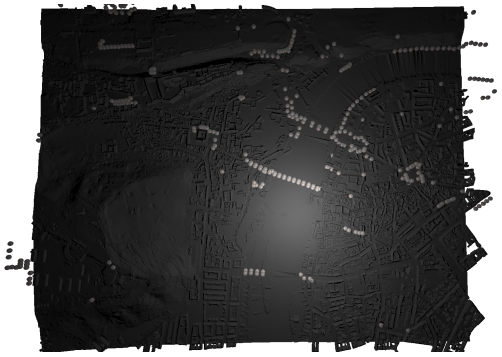
References

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

[2] J. Boksansky, P. Jukarainen, Ch. Wyman, and NVidia. *Rendering many lights with grid-based reservoirs.* Apress, 2021. Chapter 23.

[3] J. Chandler, L. Yang, and L. Ren. Procedural window lighting effects for real-time city rendering. *Proceedings of the 19th Symposium on Interactive 3D Graphics and Games*, pages 93–99, 2015.

(a) Model of part of Prague, with light source positions visualized.



(b) Part of Prague with inserted lamps.

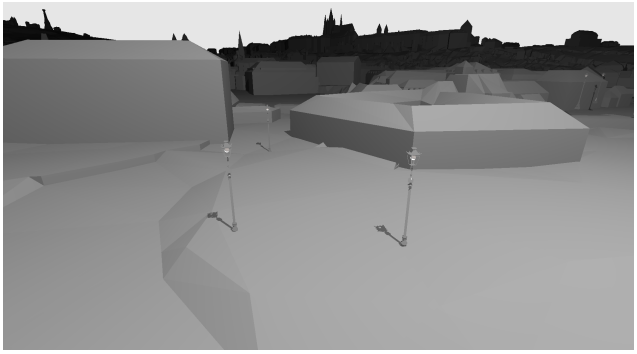
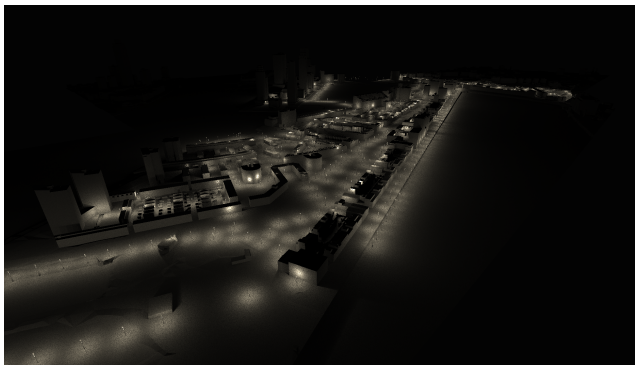


Figure 10: Testing scene of Prague showcase. 300,000 triangles, 337 light sources.

(a) Rotterdam scene from above render frame.



(b) Rotterdam scene from above outgoing radiance visualization.

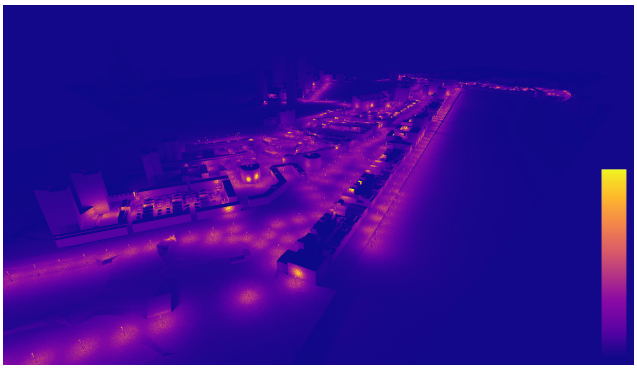


Figure 11: Rotterdam scene - further visualization. 300,000 triangles, 913 light sources.

[4] Open Geospatial Consortium. *CityGML Standard*, 2019. https://www.ogc.org/publications/standard/citygml/.

[5] M. Conte. Real-time rendering of cities at night. Master's thesis, University of Montreal, 2019.

[6] Esri, IPR Praha, and ČÚZK. 3d model of prague, online application. https://app.iprpraha.cz/apl/app/model3d/. Accessed: 2025-03-06.

[7] M. Hašan, F. Pellacini, and K. Bala. Matrix row-column sampling for the many-light problem. *SIGGRAPH*, 2007.

[8] H.W. Jensen, S. Premoze, P. Shirley, W. Thompson, J. Ferwerda, and M. Stark. Night rendering. Technical Report UUCS-00-016, Computer Science Department, University of Utah, 2000.

[9] D. Lin, M. Kettunen, B. Bitterli, J. Pantaleoni, C. Yuksel, and C. Wyman. Generalized resampled importance sampling: Foundations of restir. *ACM Transactions on Graphics (SIGGRAPH 2022)*, 2022.

[10] P. Moreau, M. Pharr, and P. Clarberg. Dynamic many-light sampling for real-time ray tracing. *High Performance Graphics (Short Papers)*, pages 21–26, 2019.

[11] NVIDIA. Nvidia designworks samples. https://github.com/nvpro-samples. Accessed: 2025-04-01.

[12] Y. Ouyang, S. Liu, M. Kettunen, M. Pharr, and J. Pantaleoni. Restir gi: Path resampling for real-time path tracing. *Computer Graphics Forum*, 40(8):17–29, 2021.

[13] Geoportal Praha. 3d buildings of prague, open data. https://geoportalpraha.cz/data-a-sluzby/7e6316e95cfe4f36ae06bbfb687bf34b. Accessed: 2025-03-06.

[14] Gemeente Rotterdam. 3d model of rotterdam - buildings and terrain, open data. https://www.3drotterdam.nl/. Accessed: 2025-03-06.

[15] B. Walter, S. Fernandez, A. Arbree, K. Bala, M. Donikian, and D. P. Greenberg. Lightcuts: A scalable approach to illumination. *SIGGRAPH*, 24(3):1098–1107, 2005.