Real-time Night City Rendering

Caio José dos Santos Brito*

ABSTRACT

In a night city view, the light that leaves the windows from building interacts with the surrounding buildings and the participating media creating a glow that illuminates the buildings and attenuates the view. This work revisits ray marching algorithms using the DirectX Ray Tracing and proposes a real-time rendering algorithm to visualize the glow at a city at night. A city generator was developed in Unity 3D which uses Perlin noise to generate cities in a grid format filled by buildings with a box shape. The proposed method can render the participating media in real-time for 1920 x 1061 screen resolution, and it can render effects like the brighter glow near the window and indirect illumination.

1 INTRODUCTION

As light travels between surfaces, it interacts with a host media like air, water or clouds. In an indoor scene, the media influence is quite limited due to the visibility distance between walls, and a vacuum between surfaces can be considered, but for outdoor scenes, the visibility can change a lot due to the interaction between media and light.

Several rendering techniques were proposed to render participating media, like clouds [17] [20] and fire [15]. One aspect that was not heavily studied is the interaction of the lights leaving the windows of a building at a night atmosphere; that effect can be seen in Fig. 1. Langard and Haduin [7] proposed a simple image enhancement which adds the atmospheric color using a translucent sphere with a cubic attenuation at borders, this approach is cheap but does not give the proper lighting calculation.

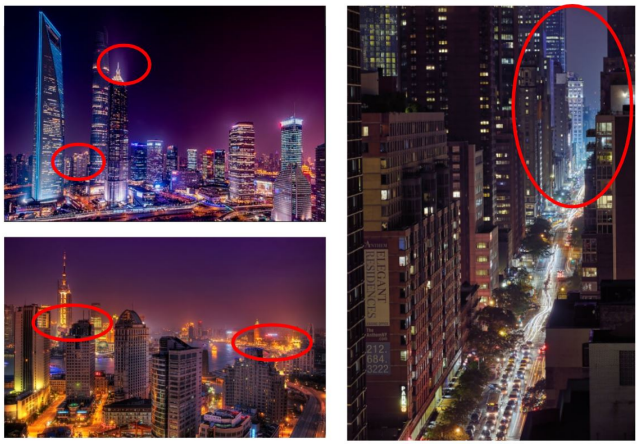

Figure 1: Real examples of light interaction from windows building.

This scenario has two main challenges, first is the big number of lights leaving the windows and interacting with the participating media around the buildings, even in a simple scene with just a single building, which is called many-lights problem. Also, how to render the image with a proper light transport calculation for a large scene with a large amount of media in real-time.

The main contributions of this paper are a city generator that can create simple city facade using Perlin noise, and a real-time rendering approach for nigh cities. The volumetric lighting is based on the work of Tóth and Umenhoffer [19], which is based on ray marching to handle the light scattering, and, to deal with the high number of lights in the scene, the lights information are precomputed on the scene creation and used to render the final image.

The next section discuss the state of the art works on participating media rendering. Then, 3.1 explains the rendering method that deals with participating media using ray marching. In Sect. 4 the algorithm used to generate the city landscape is explained. Sect. 5 explicits detail about the implementation of the city creator and rendering using Unity and DirectX ray tracing. The visual and performance results are presented in Sect. 6. Finally, in Sect. 7, the conclusions are discussed together with future works and enhancements.

2 STATE OF ART

Several works have been done to render participating media by calculating how light scatters on a media. This section will discuss some works on this subject.

To render the effect of light scattering because of the shadows in the atmosphere, Mitchell [9] proposed a post-processing method which can be done in real-time on a complex scene with multiple light sources via a GPU pixel shader.

Tóth and Umenhoffer [19] proposed a method to render light shafts by calculating single scattering in homogeneous media using ray marching from every point visible to the camera in the point light direction. The method can treat dynamic occluder objects and dynamic light sources at 42 fps in a 800 x 600 screen resolution and at 100 fps with interleaved sampling.

Looking for an approach that can be easily used in an application, like the fog from OpenGL, Sun et al. [16] proposed an analytic single scattering model for light transport for an isotropic point light source in a homogeneous participating media. The proposed solution is able to render at 20 fps a scene with 66454 triangle and 4 point light sources, and it can generate effects like a brighter glow around the lights, the effects of scattering on surface shading, environment maps, and precomputed light transport.

Wyman and Ramsey [22] proposed an algorithm that combines ray marching and shadow volumes to render volumetric shadows in homogeneous single scattering media. The work can render at 40 to 80 fps for scenes with complex objects and textured spotlights using a series of optimizations like blur, low resolution rendering, and mipmap hierarchy.

In order to accelerate the calculation for light scattering, Ali and Sunar [1] proposed to combine downsampling of ray marching and bilateral filtering, being able to achieve real-time performance of rendering soft light shafts and soft shadows.

To calculate multiple scattering in a real-time application, Billeter et al. [2] introduced a GPU based algorithm that uses the distribution of radiance from a single scattering light propagation as a source of diffuse light and uses as input for the multiple light propagation with ray marching. The work can improve the rendering quality in comparison to a single scatted model, and it is able to reach a performance of 30 fps on an HD screen resolution.

In order to render smoke under dynamic low-frequency environment lighting in real-time, Zhou et al. [23] proposed an algorithm

*e-mail: caio.jose.dos.santos.brito@umontreal.ca

entitled compensated ray marching which splits the smoke volume into a low frequency approximation and a residual field. The low frequency approximation is modeled by a set of radial basis functions (RBF), and the radiance is calculated at the RBF center, and a spherical harmonic exponentiation technique is used to solve the light scattering. The residual field is responsible for modeling fine-scale details, and it is calculated by a radiance integration that takes into account the extinction effects. The approach can render smoke with performance between 19 fps and 74 fps with interactive manipulation of the smoke characteristics, viewpoint, and lighting.

To reduce the artifacts on participating media with virtual point lights, Novák et al. [11] proposed a many-lights algorithm called Virtual Beam Lights which inflates light rays into beam lights with finite thickness to eliminated singularities from rendering complex indirect transport paths that travels thought the media. The technique generates high quality images without artifacts for scenes with complex lighting, anisotropic phase function, and heterogeneous media.

Kallweit et al. [6] proposed a combination of Monte Carlo integration and neural network to generate cloud images. The neural network learns the spatial and directional distribution of radiant flux for clouds. To render a scene, points of the cloud are sampled, and a descriptor is extracted from the samples, the descriptor is used as input to a deep neural network that predicts the radiance. The method can generate high quality images with multi-scattered illumination.

To improve the rendering of emissive heterogeneous volumes, Simon et al. [15] proposed a line integration estimator which accumulates emission for path segments and a forward next event estimation. The technique can generate results with lower variance with less sample per pixel, and the next event estimation is faster than previous works.

To improve the appearance of clouds, Bitterli et al. [3] new theory of volumetric light transport for media with non-exponential free-flight distributions which can be implemented with minor changes on previous rendering algorithms, and can provide a powerful tool for artists to design the shape of the attenuation profile.

A more detailed state of the art report on volumetric light transport can be found in the work of Novák et al. [10].

3 Participating media rendering

The change of radiance when a ray that travels through space and passes inside a media can be expressed by the radiative transport equation, as can be seen in Equation 1

$$\frac{dL(\overrightarrow{x}(s),\overrightarrow{\omega})}{ds} = -\tau L(\overrightarrow{x}(s),\overrightarrow{\omega}) + \tau a \int_{\Omega} L(\overrightarrow{x}(s),\overrightarrow{\omega'}) P(\overrightarrow{\omega}',\overrightarrow{\omega}) d\overrightarrow{\omega'} \tag{1}$$

where the ray is defined by $\overrightarrow{x}(s) = \overrightarrow{x}_0 + \overrightarrow{\omega}t$, with origin x_0, direction $\overrightarrow{\omega}$. τ is the the probability of collision in a unit distance, a is the albedo, and $P(\overrightarrow{\omega}',\overrightarrow{\omega})$ is the phase function which describes the light scattering in that media. In this work, the phase function is equal to $1/4\pi$ creates a perfect isotropic scattering.

This radiative transport equation can be solved by Monte Carlo method, but the method is quite time consuming and is not the best solution for real-time rendering. To reach a high time performance, Tóth and Umenhoffer [19] simplified the Equation 1 by ignoring the multiple scattering and assuming a single scattering only, which can be analytic as expressed in Equation 2.

$$L(\overrightarrow{x}(s),\overrightarrow{\omega}) = e^{-\tau s} L(\overrightarrow{x}_0,\overrightarrow{\omega}) + \int_0^s L_i(\overrightarrow{x}(l),\overrightarrow{\omega}) e^{-\tau(s-l)} dl \tag{2}$$

Finally, the integral is approximated with a finite Riemann summation, as expressed in Equation 3, and L_i can be calculated as Equation 4 for a single point light with power Φ and taking only one direction W into account.

$$L(x(s),\overrightarrow{\omega}) \approx e^{-\tau s} L(\overrightarrow{x}_0,\overrightarrow{\omega}) + \sum_{n=0}^{N} L_i(\overrightarrow{x}(l_n),\overrightarrow{\omega}) e^{-\tau(s-l_n)} \Delta l \tag{3}$$

$$L_i(x(s),\overrightarrow{\omega}) = \tau a \frac{\Phi C_i}{4\pi d^2} v(\overrightarrow{x}) e^{-\tau d} P(\overrightarrow{\omega_l},\overrightarrow{\omega}) \tag{4}$$

where $\Delta l = s/N$, s is the ray length, N is the number of sample points. L_i is the radiance for a single point light with color C, d is the distance between the considered point and the light source, $\overrightarrow{\omega_l}$ is the direction of the light source from the sample point, and v is the visibility factor.

3.1 Scattering calculation with ray marching

Ray marching is used to calculate the radiance expressed in Equation 3 based on the work of Tóth and Umenhoffer [19]. The algorithm for a single media is performed in a series of steps:

- Rays are cast from the camera in an orthogonal direction ω into the scene.
- If a ray hit the media in point P, another ray is cast from P with direction ω and reach the exit media point P'.
- A number of points are sampled along the vector PP'.
  - for each sample p the in-scattering term is computed for each visible light from p and multiplied by the absorption factor $e^{-\tau(s-l_n)}$.
  - The product is added to the radiance, as expressed in Equation 3.

For cases that a ray hits multiple media, the radiance of the first media is attenuated by the second ray and added to the second media radiance.

4 City creation

To generate the city landscape, a generator based on Perlin noise was developed to emulate a real city height distribution. The 2D Perlin noise was used, which creates a pseudo-random pattern of float values that gradually increase and decrease across the pattern, as can be seen in Fig. 2.

Figure 2: Perlin noise example on a gray scale image.

The Perlin noise is highly used to terrain generation [13] [5], and it was used to generate the city landscape because it creates blobs with similar values. This behavior is similar to a real city landscape where buildings with similar heights tend to stay close to each other, as visualized in the London landscape exhibited in Fig. 3.

The system can create buildings with a box shape with different sizes, windows in a grid distribution for each building face that

Figure 3: London landscape.

will be used as light source on the rendering and it can be turned on (white color) and of (black color), and the participating media between the building which are also built as a box. Examples of the cities generate can be seen in Fig. 4.

5 Implementation

5.1 City generator

The city generator was implemented on Unity 3D version 2018.3.8f1. The buildings and participating media were created as a Cube GameObject which is composed of 8 faces and 12 triangles. The windows are a Quad GameObject with 2 triangles, each building face is subdivided into a grid, and the windows are centered in the grid vertices outside the face border.

The cities are generated on a grid distribution, and the building height depends on the noise value. Also, the texture of the buildings can be modified on the script. The cities were generated with two different building templates, the small building has a square base with $100m^2$, $30m$ height, and 20 windows on each face, and the big building has the same base but with a $60m$ height, and 40 windows on each face with power of $50W$. The templates can be visualized in Fig. 5.

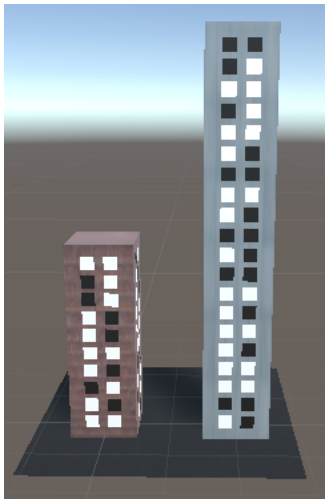

Figure 5: The building templates visualized on Unity 3D.

The space between building has $10m$ and it is filled with the media shape that has a square base with $100m^2$ and the height is the biggest height of the building around the media. The number of windows, the building texture, the building heights, and the street size can be modified on the script.

After the geometry generation, as the windows are considered light sources for the rendering, the ones that collide with a participating media are stored in a .txt file. The collision is computed using the bounding box of each game object, and, to avoid miscalculation, the bounding box is scaled by 1.2 for the media and 1.3 for the windows. Using this scale, the media boxes that are placed on the corner of the buildings do not hit any windows but there is light that leaves the windows and hits these medias indirectly, in order to get an approximation of that effect, the bounding box of corner medias are scaled by 1.5 to hit windows close to the building borders.

The .txt file stores the number of windows that emits light on a media and the center of the windows which will be used as point light source. This information will be passed to the shaders which store these values on a constant buffer (cbuffer) which are optimized to constant-variable usage, which is our case since the windows that emit light do not turn the lights off.

The geometry is exported as a .fbx file which was done by the FBX Exporter [18] asset from Unity Asset Store. As the fbx reader from the rendering application merges geometries with similar material, to avoid that and force each media to have a different id on the rendering, each media has a unique emissive color. Also, the fbx exporter scale is 100 times bigger than the rendering scale, to match the scales between exporter and rendering, the whole geometry is scaled by 0.01 factor before being exported.

5.2 Rendering

The rendering technique was implemented using DirectX Raytracing (DXR) [12] which is an extension of DirectX 12 that includes a fully integrates ray tracing into DirectX and it can be used with the graphics card rasterization.

The ray tracing shaders are dispatched as grids which leads to high parallel processing on the GPU, similar to a rasterizing shader. Also, resources like textures, buffers, and constants are shared among shaders.

The rendering was programmed on the Falcor framework which is an open-source real-time rendering framework that supports DXR and implements abstraction layers on top of DirectX 12 that allow the graphics programmer to focus on the algorithm implementation, instead of focusing on the DirectX backend programming. Several works were done using Falcor, like the work of Chaitanya et al. [4], and the work of Schied et al. [14].

Last year, Wyman presented the course entitled "Introduction to DirectX Raytracing" [21] and the course code is available. The code uses the Falcor framework and the implementation of the participating media rendering was built on top of it.

The precomputed information is set on the shaders via the SceneRenderer::setPerModelData method which is coded in the SceneRendered.cpp of the Falcor framework. The part of the code that sets the values of the cbuffer which contains the list of lights on for each object can be visualized in Listing 1. There is a difference between the coordinate system between Falcor and Unity 3D, to correct that difference, the x-coordinate of the precomputed information is multiply by -1.

Listing 1: SceneRenderer::setPerModelData code part.

```
1  //iterates over the array that have the list of lights
       on for each object
2  for (int i = 0; i < maxObj*maxLights; i++) {
3     // get the variable name on the shader
4     std::string varP = "lightPos1D[" + std::to_string(i
          ) + "]";
5
6     //get the offset of this variable
7     size_t offsetP = pCB->getVariableOffset(varP);
8
9     //set the variable value
10    pCB->setVariable(offsetP, lightPositions.at(i));
11 }
```

Due to the integration between DXR and rastering shaders, instead of ray tracing from the camera to the scene, the first step is

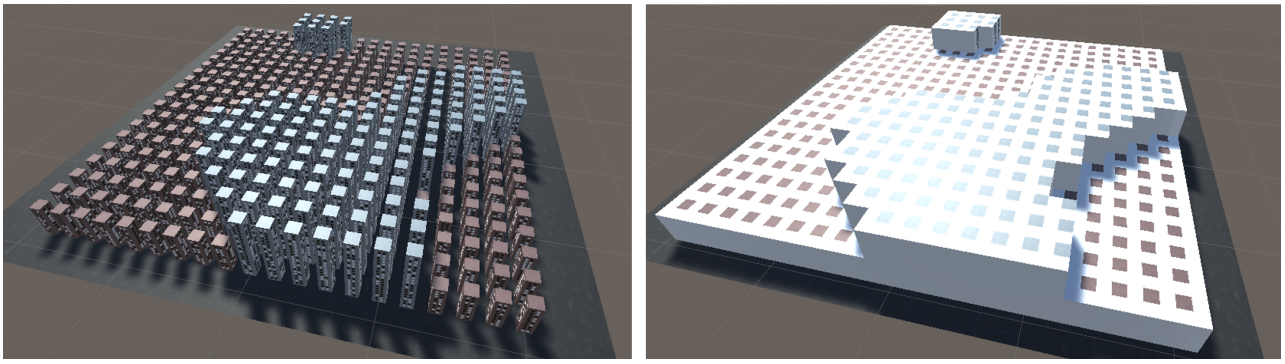

Figure 4: Examples of generated cities visualized on Unity 3D. The building blocks can be seen in the left image, and the blocks with the media around (white boxes) can be seen in the right image.

calculated via rasterization which returns the depth map, the normal map, the color map, and the map with the material ID. Then, if the pixel is filled by a non-media object, the object is rendered as a Lambertian surface. And for pixels that are filled with media object, a ray is cast on the camera orthogonal direction and the pixel value is calculated as described in Sect. 3.1 with the visibility factor equal to one as there is no occlusion between the windows and the media.

As the media fills the whole volume between buildings, z-fighting occurs between the media face and a building face. A first attempt to solve this issue was done by setting the emissive texture of the participating media, if a ray hits a surface point with black emissive color, this hit is ignored. The texture in Fig. 6 (a) was used for z-fighting with only the floor, Fig. 6 (b) is used to avoid z-fighting with the bottom, front, and back building faces, and Fig. 6 (c) avoid z-fighting with bottom, left, and right building faces.

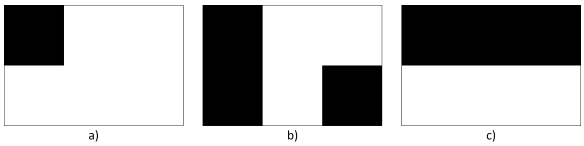

Figure 6: Emissive texture used to avoid z-fighting.

The emissive texture color covers the whole building face which solves the z-fighting for media and building the share the same whole face, but for cases that the building only shares part of the media face, a hit is ignored. An example can be visualized in Fig. 7.

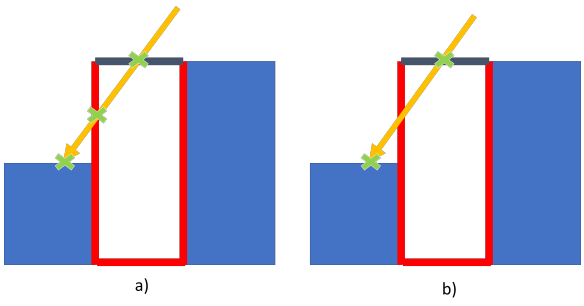

Figure 7: Hit miscalculation due the emissive texture. The building are the blue boxes, red lines are the ignored faces, and black lines are the media faces that are not ignored. The correct calculation should results in 3 hits (a), but the emissive texture lead to ignore the second hit (b).

Table 1: Performance given the number of ray subdivisions.

# subdivisions	5	25	40	100
fps	197	109	75	36

To correctly solve the z-fighting, a first ray is cast which only hits an object with the same id material as the one from the starting point, in that case, a ray hits the other side of the media, and the participating media is calculated.The material id comparison is made on the *Any Hit* shader, which is called if a ray hits a non-opaque object, and hits with different material id are ignored by the *IgnoreHit()* command.

Latter, a second ray is cast that only hits an object with a different id material of the media, if the next hit is another media, the radiance is calculated as explained previously, and the process continues until it reaches a non-media object. As it is very difficult to notice the variance of the Lambertian shading at a night view, the non-media objects are shaded as a simple attenuation of the object color by a factor of 0.1.

6 Results

This section will discuss the visual results and time performance of the rendering approach. The tests were performed on a computer with an Intel® Core™ i7-8700K CPU @ 3.7 GHz with 32 GB of installed RAM, Windows 10 64-bit operating system (x64), and with a GPU NVIDIA GeForce RTX 2080 with 8 GB of RAM. The rendering was done in 1920 x 1061 screen resolution.

6.1 Visual Results

A ray that passes through a participating media is subdivided to calculate the radiance using the ray marching technique. As the number of subdivision increases, the results should be more precise, and it must converge.

Several tests were conducted to find the best compromise between visual quality and time performance (fps). As can be seen in Fig. 8 a), a low number of subdivisions lead to a miscalculation of the brighter glow near the windows, the rendering results converge as the number of subdivisions increases but the performance in frames per second gets far from real-time. The performance results are found in Fig. 6.1.

As this work aims to get high visual quality results with real-time performance, 25 was used as the standard number of subdivisions, which can create visual results close to the converge image and with real-time performance.

There is some major real life visual characteristic that this work wishes to reproduce. The first one is that the media attenuates the building visualization, and also creates a bloom effect on the

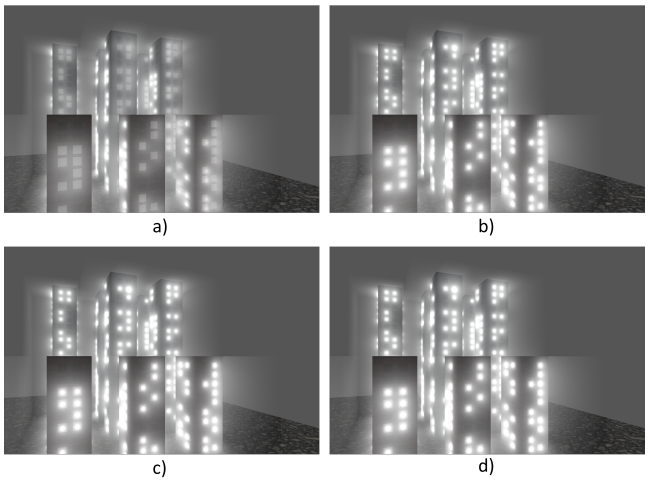

Figure 8: Night city rendering for different number of ray subdivisions: 5 subdivisions (a), 25 subdivisions (b), 40 subdivisions (c), and 100 subdivision (d).

Table 2: Performance profile given a grid size.

% of the screen filled with the scene	fps
25	300
50	150
75	75
100	50

media. Additionally, the windows create a brighter glow around it in comparison with the global participating media radiance. The method creates a circular brighter glow which, in comparison with real night city images, is suitable for long distance light sources that are approximated to a point light, but, windows close to the camera, it is expected that the glow has a shape similar to the window. This effects can be seen in Fig. 9.

Another visual effect that was aimed in this project is that the camera can visualize indirect illumination from light sources that are not visible to it. That effect can be seen in Fig. 10 which is possible to visualize the glow from the light sources on the side of the building, and it can even visualize the brighter glow close to light sources. Also, the indirect illumination gets brighter where the lights are on and where the lights are close to the camera.

6.2 Performance

Two tests were done to examine the performance of the rendering approach. The first one was to analyze the time performance was also analyzed given the percentage of screen height that is filled with the buildings and media: 25%, 50%, 75%, and 100%, approximately. Each test was done with 16 small buildings in a grid formation (4x4) and 1280 media blocks, this scenario has 3532 triangles, and all the windows were considered as point light sources.

As the percentage increases, the performance in fps drops because of the number per pixel of rays cast and radiance calculation increases, as can be seen in Sect. 6.2, but, even when the whole screen is filled with media and building, the rendering can reach real-time performance.

The time performance was also tested given a different number of buildings. In this test, a grid of small buildings was constructed, and every single building has 80 windows which are used as point light sources, and it is surrounded by 8 participating media blocks. The performance results can be seen in Table 6.2, and each test

Table 3: Performance profile given a grid size.

Grid Size	# Buildings	# Windows (lights)	# Media Blocks	# Triangles	fps
1x1	1	80	8	268	250
2x2	4	320	21	940	140
3x3	9	720	40	2028	90
4x4	16	1280	65	3532	75

was performed with 75% of the screen height filled with media and buildings.

7 Conclusion

Participating media like air, water, fog interacts with the light and may affect the radiance along the ray. The air interaction with the light leaving the windows in a night city view creates a glow between buildings that illuminates the surrounding and attenuate the building color, this effect was done previously in games by a post-processing effect that is cheap but does not treat the light scattering. This work proposed a real-time rendering approach to visualized the glow on a night city view due to the windows illumination that is done by calculating the light scattering with ray marching.

The city generator was developed on Unity that creates simple buildings cities in a grid shape and the media around them; the geometry is exported as a .fbx file and the center of the windows that illuminates the media are stored in a file. The rendering solution was implemented using the Falcor framework which allows rapid prototyping using DirectX ray tracing feature. The solution reads the file that keeps all the lights on that is inside a media and uses that information as point lights source, the rendering approach can reproduce the glow between buildings in real-time for a 1920 x 1061 screen resolution, and it was able to render effects as brighter glow on close to the windows and indirect illumination.

7.1 Limitation and future work

The rendering solution was tested in up 16 building with to 4x4 grid with 1280 light sources and 65 media blocks. Each block can have a maximum of 40 lights that illuminates the block, and this information is stored into a single buffer which has a limit size. The next step is to use multiple cbuffers in our solution and be able to render a large scale city landscape.

One effect that was not accomplished in this work is that the participating media also influence the illumination on the top of the building and can create a glow on that region, this can be done by a creating media blocks around the whole building. Another limitation of this solution is that the city generator is only able to create building and media in a box shape, future work may be to generate different shape buildings and be able to create city landscape close to real cities.

Media blocks that are on the corner of the building do not get direct illumination from the windows, but lights from indirect sources may hit that media. To get the indirect illumination, the bounding box that is used to determine which light is inside of a media block was increased on corner blocks. This solution gave a nice overall illumination but missed the radiance contribution of lights sources close to the media which are outside the bounding box. A better approach could be using a global illumination like path tracing or photon mapping to calculate the indirect illumination, but these solutions are quite expensive for real-time rendering. One way of solving this issue is to pre-compute the indirect illumination and stores it in a light probe [8] which will be used in real time to recover the illumination of a region without computing the whole light path.

Future works might take the direction of treating light transport more precisely with multiple scattering. Billeter et al. [2] proposed a solution that runs on a GPU at 30 fps and it uses ray marching to capture low frequency effects, now with the DXR this performance

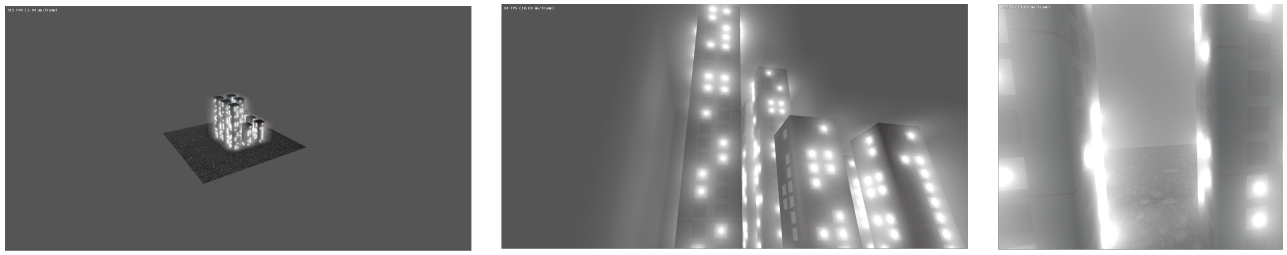

Figure 9: Night city rendering results with long distance view (a), close view (b), and inside a participating media (c).

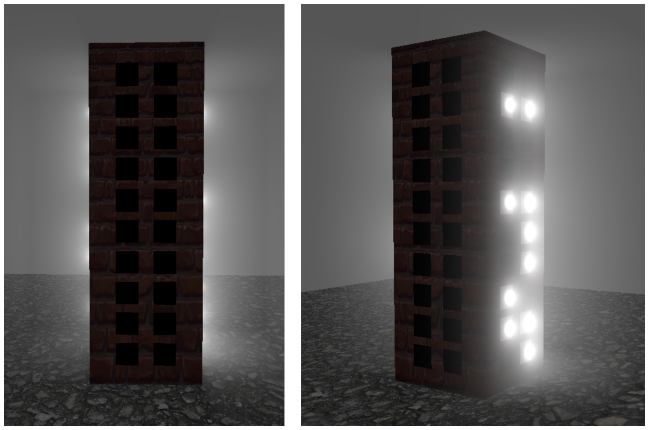

Figure 10: Window glow example.

can be improved too. Also, the proposed approach only uses a single point light for each window which is able to capture the brighter glow next to it but it has a circular shape which is enough for far view of the small windows but it for close views or big light emitter, for instance, a large window or a billboard, the brighter glow should have the shape of the emitter.

REFERENCES

[1] H. H. Ali, M. S. Sunar, and H. Kolivand. Realistic real-time rendering of light shafts using blur filter: considering the effect of shadow maps. *Multimedia Tools and Applications*, 77(13):17007–17022, Jul 2018. doi: 10.1007/s11042-017-5267-8

[2] M. Billeter, E. Sintorn, and U. Assarsson. Real-time multiple scattering using light propagation volumes. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '12, pp. 119–126. ACM, New York, NY, USA, 2012. doi: 10.1145/2159616.2159636

[3] B. Bitterli, S. Ravichandran, T. Müller, M. Wrenninge, J. Novák, S. Marschner, and W. Jarosz. A radiative transfer framework for non-exponential media. *ACM Trans. Graph.*, 37(6):225:1–225:17, Dec. 2018. doi: 10.1145/3272127.3275103

[4] C. R. A. Chaitanya, A. S. Kaplanyan, C. Schied, M. Salvi, A. Lefohn, D. Nowrouzezahrai, and T. Aila. Interactive reconstruction of monte carlo image sequences using a recurrent denoising autoencoder. *ACM Trans. Graph.*, 36(4):98:1–98:12, July 2017. doi: 10.1145/3072959.3073601

[5] A. Frasson, T. A. Engel, and C. T. Pozzer. Improving terrain visualization through procedural generation and hardware tessellation. *Sociedade Brasileira de Computação (SBC)-Proceedings of Simpósio Brasileiro de Games e Entretenimento Digital (SBGames)*, 2016.

[6] S. Kallweit, T. Müller, B. Mcwilliams, M. Gross, and J. Novák. Deep scattering: Rendering atmospheric clouds with radiance-predicting neural networks. *ACM Trans. Graph.*, 36(6):231:1–231:11, Nov. 2017. doi: 10.1145/3130800.3130880

[7] S. Lagarde and L. Harduin. The art and rendering of remember me. *GDC Talk*, 2013.

[8] M. McGuire, M. Mara, D. Nowrouzezahrai, and D. Luebke. Real-time global illumination using precomputed light field probes. In *Proceedings of the 21st ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '17, pp. 2:1–2:11. ACM, New York, NY, USA, 2017. doi: 10.1145/3023368.3023378

[9] K. Mitchell. Volumetric light scattering as a post-process. *GPU Gems*, 3:275–285, 2007.

[10] J. Novák, I. Georgiev, J. Hanika, and W. Jarosz. Monte carlo methods for volumetric light transport simulation. In *Computer Graphics Forum*, vol. 37, pp. 551–576. Wiley Online Library, 2018.

[11] J. Novák, D. Nowrouzezahrai, C. Dachsbacher, and W. Jarosz. Progressive virtual beam lights. In *Computer Graphics Forum*, vol. 31, pp. 1407–1413. Wiley Online Library, 2012.

[12] NVIDIA. RTX™ platform. `https://developer.nvidia.com/rtx/`, 2019. Online; accessed 02-june-2019.

[13] I. Parberry. Modeling real-world terrain with exponentially distributed noise. *Journal of Computer Graphics Techniques (JCGT)*, 4(2):1–9, May 2015.

[14] C. Schied, A. Kaplanyan, C. Wyman, A. Patney, C. R. A. Chaitanya, J. Burgess, S. Liu, C. Dachsbacher, A. Lefohn, and M. Salvi. Spatiotemporal variance-guided filtering: real-time reconstruction for path-traced global illumination. In *Proceedings of High Performance Graphics*, p. 2. ACM, 2017.

[15] F. Simon, J. Hanika, T. Zirr, and C. Dachsbacher. Line integration for rendering heterogeneous emissive volumes. *Computer Graphics Forum*, 36(4):101–110, 2017. doi: 10.1111/cgf.13228

[16] B. Sun, R. Ramamoorthi, S. G. Narasimhan, and S. K. Nayar. A practical analytic single scattering model for real time rendering. In *ACM SIGGRAPH 2005 Papers*, SIGGRAPH '05, pp. 1040–1049. ACM, New York, NY, USA, 2005. doi: 10.1145/1186822.1073309

[17] L. Szirmay-Kalos, I. Georgiev, M. Magdics, B. Molnár, and D. Légrády. Unbiased light transport estimators for inhomogeneous participating media. *Computer Graphics Forum*, 36(2):9–19, 2017. doi: 10.1111/cgf.13102

[18] U. TECHNOLOGIES. FBX Exporter. `https://assetstore.unity.com/packages/essentials/fbx-exporter-101408`, 2019. Online; accessed 02-june-2019.

[19] B. Toth and T. Umenhoffer. Real-time Volumetric Lighting in Participating Media. In P. Alliez and M. Magnor, eds., *Eurographics 2009 - Short Papers*. The Eurographics Association, 2009. doi: 10.2312/egs.20091048

[20] R. Villemin, M. Wrenninge, J. Fong, and P. A. Studios. Efficient unbiased rendering of thin participating media. *Journal of Computer Graphics Techniques Vol*, 7(3), 2018.

[21] C. Wyman, S. Hargreaves, P. Shirley, and C. Barré-Brisebois. Introduction to directx raytracing. In *ACM SIGGRAPH 2018 Courses*, August 2018.

[22] C. Wyman and S. Ramsey. Interactive volumetric shadows in participating media with single-scattering. In *2008 IEEE Symposium on Interactive Ray Tracing*, pp. 87–92, Aug 2008. doi: 10.1109/RT.2008.

4634627

[23] K. Zhou, Z. Ren, S. Lin, H. Bao, B. Guo, and H.-Y. Shum. Real-time smoke rendering using compensated ray marching. In *ACM SIGGRAPH 2008 Papers*, SIGGRAPH '08, pp. 36:1–36:12. ACM, New York, NY, USA, 2008. doi: 10.1145/1399504.1360635