Implementing a Radiance Predicting Neural network in a real-time Graphics Engine*

1st Tumi Jourdan
Computer Science
University of the Witwatersrand
Johannesburg, South Africa
2180153@students.wits.ac.za

***Abstract*—Real-time rendering of volumetric clouds remains challenging due to complex light scattering behaviors. This paper presents an implementation of a Multi-Feature Radiance Predicting Neural Network (MRPNN) within the Unity game engine, addressing the integration of CUDA-based neural rendering with DirectX graphics. We evaluate optimization strategies through image quality metrics, demonstrating that volume resolution reduction offers significant memory savings while maintaining visual fidelity (PSNR 38dB, SSIM 0.99). Our main contributions include a practical framework for neural network integration in commercial game engines and detailed performance analysis of optimization strategies. The results show successful real-time cloud rendering while efficiently managing resources between the neural network and game engine.**

***Index Terms*—Compute Unified Device Architecture (CUDA), Graphical Processing Unit (GPU), Computer Processing Unit (CPU), Radiance Predicting Neural Network (RPNN),Multi-feature RPNN (MRPNN),**

I. Introduction

Real-time rendering of volumetric clouds is a persistent challenge in computer graphics due to the complex light interactions within participating media. Traditional methods often require extensive computational power to simulate the intricate scattering effects. Recent advancements in neural network-based rendering offer promising photorealistic outcomes, but integrating these techniques into real-time graphics engines remains a difficult task.

This paper explores the implementation of the Multi-Feature Radiance Predicting Neural Network (MRPNN) approach, as proposed by [1], within the Unity game engine. The work addresses two key challenges: establishing efficient communication between CUDA and DirectX graphics APIs, and optimizing resource usage while maintaining visual fidelity.

Our implementation leverages CUDA-DirectX interoperability to share graphics resources efficiently between the neural network and the game engine. We evaluate various performance optimization strategies, including resolution scaling, model complexity reduction, and ray marching parameter adjustments. Through comprehensive analysis using multiple image quality metrics (MSE, PSNR, and SSIM), we quantify the trade-offs between visual fidelity and computational performance.

The primary contributions of this work include:

- A practical framework for integrating CUDA-based neural rendering into a commercial game engine
- Detailed performance analysis of various optimization strategies and their impact on visual quality
- Implementation insights for managing graphics resource sharing between neural networks and real-time rendering pipelines

II. Background

A. Volumetric Point Data

Volumetric data is a type of n-dimensional data, where the first three dimensions describe positions in 3D spatial coordinates [2]. Each point in this dataset can represent multiple attributes beyond spatial location. These additional dimensions can represent physical or material properties such as density, temperature, opacity, or material characteristics. A collection of these data points is used to describe a volume comprehensively. In computer graphics, rendering high-resolution volumetric data poses significant computational challenges. To address this, the graphics industry has developed techniques that estimate a volume's surface representation. Typically, shape primitives—most commonly triangles—are used to approximate the volume's surface. These primitives are then processed through standard rasterization techniques to render the object efficiently. While this approach provides real-time rendering capabilities, it discards the interior volumetric point information, reducing the datas complexity to a mere surface representation.

B. Raymarching

Ray-marching involves taking incremental steps along a projected ray, a technique commonly used to efficiently identify intersections with geometry. In addition to collision detection, ray-marching is effective for sampling points within a volume. Once a ray penetrates a volumetric space, sampling the regions around each step can gather volumetric data along the ray's path. By repeatedly ray-marching through a volume, it's possible to accumulate data and create a comprehensive descriptor of the volume. These characteristics make ray-marching particularly suitable for cloud rendering, as highlighted and used in the paper [3] and demonstarted by industry giant Guerrilla Games in their presentation [4] . While it may not match the performance of shape primitives, ray-marching excels in delivering superior visual fidelity.

C. Clouds as light participating media

All participating media have a phase function that describes how the light scatters inside the medium. The water droplets inside the cloud have 2 unique properties. they are highly refractive with little absorption leading to a large amount of in-scattering. Secondly the size of the droplets is comparable to the wavelength of light in the cloud (droplet:5 microns, light:0.7microns) leading to a forward biased scattering of the light. The phase function of a cloud is most accurately described by the Lorenz-Mie function. This function simulates the unique properties of the cloud. Both Henyey-Greenstein and isotropic can be used as more efficient computations, although they do sacrifice accuracy.

D. Radiative Transfer Equation (RTE)

The radiative transfer equation describes how light interfaces with the particles of the volume ccloud. The an RTE is derived in the frostbite paper [3], by combining the equations that describe the absorption, out-scatter, in-scatter and emission of the cloud. Equation 1

$$\begin{aligned} L(x, \overrightarrow{w}) = T_r(x, x_s)L_e(x_s, -\overrightarrow{w}) + \\ \int_0^S T_r(x, x_t)L_e(x_t, -\overrightarrow{w})\ \mathrm{d}t + \\ \int_0^S T_r(x, x_t)\sigma_s(x_t)L_i(x_t, -\overrightarrow{w})\ \mathrm{d}t \end{aligned} \tag{1}$$

- $L(x, \vec{w})$: This is the radiance at point x along a line traveling in the direction $\vec{w}$.
- $T_r(x, x')$: This is how much light is unobstructed between two points.
- $L(x_s, -\vec{w}$: This is the direct contribution of light from the light to point x_s
- $\int_0^S T_r(x, x_t)L_e(x_t, -\overrightarrow{w})\ \mathrm{d}t$ The first integral integrates the transmittance and the radiance contribution to all points along a ray traveling through the participating media.

In the [5] paper they describe the radiance as equation 2.

$$(\omega \cdot \nabla)L(x, \omega) = -\mu_t(x)L(x, \omega) + \mu_s(x) \int_{S^2} p(\omega \cdot \hat{\omega})L(x, \hat{\omega})d\hat{\omega} \tag{2}$$

These radiative transfer equations are important as the unique visual behaviours of clouds emerge from them. Such behaviors are their white colour, silver lining effect and asperitas.

E. Neural Networks

The most basic building block of a neural network is the perceptron. The perceptron has data fed to it. It multiplies the data by some weight, adds a bias , sums the results and then applies a function called the activation function to the result.

A neural network is constructed of multiple layers (columns) of perceptrons connected to each other. The Input data is fed to the first layer and propagates forward to achieve a result after the last layer. Based on a loss function between the desired output and the simulated output, the network can use an optimising algorithm to adjust its weights towards the correct true output. By repeating this the network progressively more accurate in predicting the correct result from some test data. This is essentially learned function approximation.

F. Computer Graphics Rendering Pipeline

The general structure of a graphics pipeline is well documented, the same cannot be said for the graphics pipeline from unity. As the product is proprietary, none of the documentation explicitly covers the full rendering process within unity. [6] dissects the graphics pipeline to the best of their abilities and the render process is described as below.
The CPU and the GPU work together to create an interactable real-time environment. The physics of the world and the input of the player are computed by the CPU. The CPU then passes these changes to the GPU. This is where the visual data of the video game lie.

The GPU then uses point vertices connected together to form polygons to visualize the world. It achieves this in the following steps (also shown in image**??**).

- Model and Camera transformation: The world geometry is transformed to the world co-ordinate system and a virtual camera is situated at the origin.
- Lighting: There are various lighitng models that each engine uses. Unity uses shadow casting to generate shadow volumes, physically-based rendering to determine how surfaces interact with the lighting and shading techniques.
- Projection: In video game engines, projecting a 3D world onto a 2D screen involves transforming the 3D coordinates of objects so they appear correctly from the camera's perspective. This is achieved by using a projection matrix, which adjusts the position of each vertex to simulate depth and perspective. The projection matrix creates a sense of depth by making objects farther from the camera appear smaller, effectively creating vanishing points. This transformation is crucial for achieving a realistic 3D effect on a flat screen.
- Clipping. Using a view frustum that defines the near and far plane, objects outside the frustum are discarded out of the rendering process to save rendering time.

After all of these operations are complete each triangle is then converted to pixel data in a process called rastorization. To account for overlapping geometry, a depth buffer is created from the camera. This depth buffer assigns a z value to each pixel. Using this and the pixels location we can determine which is closest to the camera and therefore which to render. This technique does fail in the case of transparent objects , and requires extra computations to accommodate.

G. DirectX

DirecX is one of the 3 graphics APIS that the Unity game engine uses for rendering[1].
DirectX serves as a suite of graphics APIs that provide an

[1]https://learn.microsoft.com/en-us/windows/win32/direct3d

interface for applications to use resources on Windows systems. Specifically the Direct3D drives provide low level access to the graphical compute and resources of the graphics card. DirectX provides a Software Development kit for developers to use when creating new applications.

III. Related Literature

A. Radiance Predicting Neural Networks

Kallweit [5] introduces a novel method in rendering clouds using a neural network. The network aims to estimate and predict the indirect in-scattered radiance. This scattering is described by the radiance transfer equation 2. It achieves this through a Multi-Layerd Neural Network. The descriptor fed into network is a set of point stencils that sample the cloud density along a ray. At each step in the ray, multiple stencils of increasing size are created , this is to capture the detail near the point but also the general shape of the cloud . Each stencil is then progressively introduced to the network at different layers. This process is visual described in figure 1.

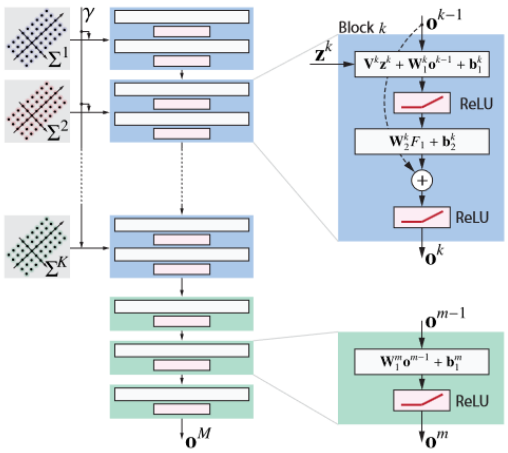


Fig. 1: Image describing perceptrons for neural network, Source: [1]

The paper then compares the RPNN solution to path tracing, a very expensive rendering technique. The result is that the RPNN is 24 times faster than the path tracing and takes seconds to render resulting in a render time of 1-3 seconds for one cloud. This improvement is not game ready, as 2ms render time is the desired speed. To achieve this speed the paper acknowledges the brute force calculation for the high frequency direct light (calculated using Monte-Carlo path tracing) as the primary bottleneck. [1] solves this.

B. Multi-Feature RPNN

Jinkai [1] design a two-stage network architecture. The first stage is fed the phase function and the transmittance fields, and its output is then passed to the second stage along with the albedo. Creating this two-stage network reduces the descriptor size and network complexity compared to the original RPNN from [5], while also allowing the albedo and alpha to be variable without requiring retraining. To address the Monte Carlo bottleneck from previous RPNN method, the first stage consists of two sub-networks - one for low-frequency indirect scattering using a spherical stencil descriptor, and another for high-frequency direct scattering using a cone-shaped stencil oriented towards the light source. This eliminates the need for expensive Monte Carlo path tracing to simulate the direct light transport. The paper provides implementation details on several key components:

1) Feature Extraction:

- Transmittance fields are computed using ray marching with uniformly-spaced samples towards the light aligned with the density maps.
- Density maps of the clouds are generated, which are later sampled along the stencil points.
- The Henyey-Greenstein phase function is used to describe how the light rays will scatter in the cloud, controlled by a parameter G.

2) Network Input: The descriptor is built from features describing a point by its density, scaled transmittance, and phase function values sampled at stencil points. This stencil and the albedo are what form the descriptor which is fed into the neural netowrk.

3) Architecture of Neural network: Squeeze-and-Excitation (SE) modules process the multi-feature inputs. The feature stage uses two sub-networks to calculate and join low and high frequency stencils In the albedo stage, SE is used to properly scale the input. To reduce computation time the dimensions are scaled down again, and then multiple fully connected layers calculate the output.

4) Training: Supervised training using an MSE loss. Log transforms squeeze the range of the albedo optimizing training speeds. 33 volumetric models are used to generate 5M training/validation samples, which is further sudbividing into groups with different lighting setups (angle and intensity).

By removing the Monte Carlo bottleneck, [1] achieve rendering times of 5ms for a single cloud, a 100x speedup over the original RPNN method, while also improving hard shadow accuracy for non-cloud, high-albedo media. This brings neural rendering much closer to real-time video game rates, though further speedups may be needed to render full cloudscapes interactively.

The major drawback is that the method overestimates the lighting of the objects, this is somewhat the opposite of the paper [7] which calculates the more absorptive process of sub surface scattering.

C. MRPNN implementation

The first few classes that are run are for initialization. The GUI class handles connections from CUDA resources to OpenGL. GLFW and GLEW are used to capture inputs from the user and output to a graphical window.
The Volume class is where the rendering begins. The class receives descriptors for the camera, light direction and alpha. It feeds this to the first Kernel program, Render Camera. The amount of threads for this is exactly the number of pixels,

similar to a shader in graphics engines. For each pixel it generates a direction vector from the cameras origin into its frustum. Each direction is peturbed by a small random number. The program determines whether the ray intersects with the box with the ray-box intersection algorithm [**?**]. If the ray does hit the box it enters either one of two sampling techniques.

1) Scattering - once the ray enters the box, the light steps through the volume until it has sampled a denisty above some threshold. Once this incident point is found. A new point and direction are then generated from this point using the Henyey scattering. This returns a sample point.
2) Ray Marching - based on the entry and exit points of the ray box, the algorithm determines a stepping distance based on number of steps and distance in the box. It then steps along the ray. At each step of the ray it samples the density of the cloud. This sampled density then contributes to updating the transmittance of the ray. When the sample density exceeds some threshold, extra-calculations are made for refraction and reflection which alter the direction of the ray. Once transmittance drops below some threshold, the max distance is exceeded or the steps exceed a threshold the loop exits. This returns the last sample point. A supplementary class diagram at 2.

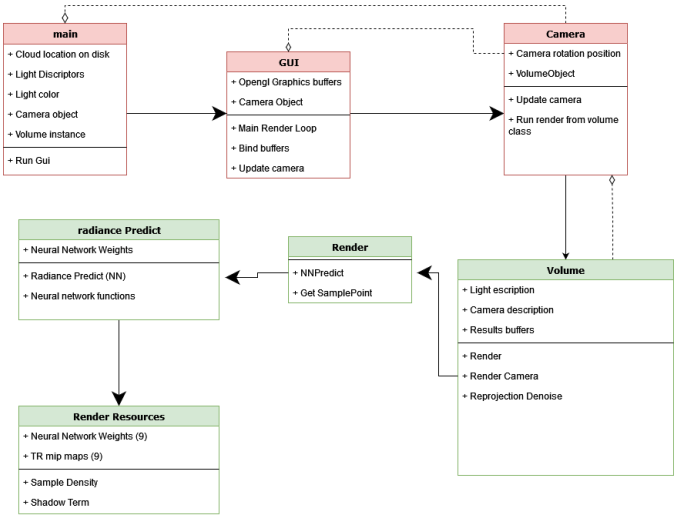


Fig. 2: Class diagram of MRPNN

After the sampling techniques the code feeds the point, light direction, alpha, and scatter rate into the neural network as stencil data. The output of the neural network is then mixed with the lightcolor, transmittance, scatter rate and background color (HDRI). This result is then stored in a buffer (target1) whos size is described as the flattened dimensions of the total screen size, so each element corresponds to a pixel.

1) Convergence: Using a histogram buffer and a second buffer (target2), the program collects current predictions for each pixel from target1 and blends them with the prvious resultant colours stored in the histogram. This creates an accumulation process of all previous frames pertubated rays for a given pixel. The pixel data then converges overtime , effectively de-noising the noise the random pertubations produce.

IV. Methodology

A. Experimental Design Choices

1) Code pipeline: Two approaches were considered for integrating a CUDA-based neural network into a game engine. The first option was to wrap the CUDA code into a dynamic-link library (DLL) that the engine could query. This would minimize the computation time, as the engine could leverage the shared graphical resources. However, the downside of this approach is the additional layer of abstraction between the engine and the network. The second approach was to reinterpret the network into the native graphics code for the engine, such as using compute shaders in Unity. This would remove the abstraction layer, but would require significant effort to convert the CUDA-specific thread management techniques into a format compatible with the engine. Given the short time-frame for the project, this conversion was deemed too time-consuming. Consequently, the decision was made to integrate the network as a native plugin DLL. This would provide a balance between development time and performance.

2) Game Engine: When evaluating game engines, two options were considered: Unity and Unreal Engine. Unreal Engine's C++ code base allows for easier integration with the CUDA DLL. However, Unreal's project environments are bloated with many unwanted default features. In contrast, Unity's Scriptable Render Pipeline offers a more lightweight and flexible solution. Given this the Unity Game engine is used as the developement game engine for this research.

B. Unity Implementation

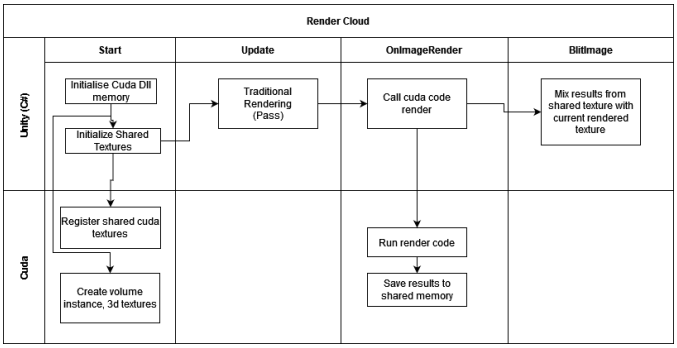


Fig. 3: Flow diagram of how code moves through unitys game loop. Update, On render Image and BlitImage are the main loop functions. Start occurs once on program startup.

Diagram 3 demonstrates the general flow of the code.

- Start: This function is called once when the unity application begins execution.
- Update : The update function is called at the start of every frame, before any render operations are called, this is also where the game logic is calculated.
- OnRenderImage : This function is called after the scene is rendered and is where postprocessing effects can be added.
- BlitImage: This operation is called in OnRenderImage, it applies the post processing effects to the camera.

The CUDA code is wrapped in a dynamically linked library (DLL). A new interface class is used to defined functions that act as entry points for for the DLL. Unity can then use these entry points to call the interface functions/entry points in CUDA.

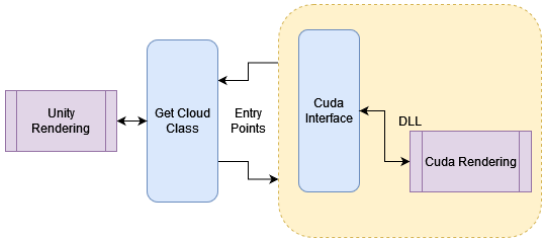


Fig. 4: Unity interfacing with CUDA DLL

This interface class contains global persistent variables (shared memory pointers, CUDA accumulation buffers), as much of the code is independent of the current frame. Additionally it contains functions for memory initialization and the main render function that approximates the cloud lighting.

Within each step in the rendering structure of unity, the code executes functions that initialize and run the CUDA neural network. The following is an in-depth breakdown of each of these steps:

1) Start: Starting in unity, 2 textures are created. The 4 channel 4 bit texture is used to receive color data describing the cloud from CUDA. The 2 channel 32 bit texture is used to receive transmittance and depth data from CUDA. The transmittance and depth texture requires 32 bits as it requires a higher accuracy than the standard color data textures hold.
The pointers to the location of each texture in GPU memory are passed to CUDA through a function call. CUDA then uses DirectX interop to register these pointers as graphics resources. Unity also informs CUDA to initialize memory. Here CUDA initializes buffers for results and accumulation, generates a volume class instance which holds the neural network in GPU memory and initializes default parameters for this volume instance.

2) Update: General game logic and then standard rendering of Unitys pipeline.

3) OnRenderImage: There are several environmental descriptors that are extracted from the scene and passed to CUDA to contextualize how CUDA will perform its rendering. The light direction,light color, camera origin, camera right, up and forward vectors and an adjustable cloud alpha.

Any vector data that is to be passed to the DLL first needs to be converted from Unitys co-ordinate system to CUDAs co ordinate system. Relative to Unity, CUDAs z-axis and right vectors are inverted.
The graphics resource pointers (which were previously registered) are then mapped. The mapping ensures that CUDA will be the sole program writing to the resource.
After the mapping is complete a surface objet is generated. This surface object is a CUDA class that facilitates a pipeline to write to texture resources.
The main CUDA render function is then run. During ray-marching through the volume (per pixel), the transmittance is calculated through sampling the density of the cloud at each ray march step. This result is written to the first channel of the 2 channel texture. In the calculation of the transmittance, the insident distance is also determined and written to the second channel of the texture.
Once the raw colour results are calculated from the neural network for each pixel, they are accumulated for the final colour in the Reprojection Denoise function. This final result is written to the 4 channel color texture.
On completion of the render function, the surface object is destroyed and the resources are unmapped. Importantly the registered texture resources persist across frames and are not de-registered at the end of the frame.

4) BlitImage: Back in Unity, a shader is used to read the pixel data of 3 textures: shared 4 channel color texture, shared 2 channel transmittance & depth texture and the rendered texture from the scene. The shared colour texture and rendered texture colour data are blended through linear interpolation (lerp) scaled by the transmittance. The lerp function is given as :

$$\begin{pmatrix} r \\ g \\ b \end{pmatrix} = \begin{pmatrix} r_0 \\ g_0 \\ b_0 \end{pmatrix} + \left(\begin{pmatrix} r_1 \\ g_1 \\ b_1 \end{pmatrix} - \begin{pmatrix} r_0 \\ g_0 \\ b_0 \end{pmatrix} \right) t$$

where t is the transmittance of the cloud. This lerping transitions each pixel between the colour of the unity scene to the colour of the cloud. This is not equivelant to how the paper mixes colours. MRPNN lerps the measured value, and accumulates the result. While our technique accumulates the measured results, then lerps in unity.
To account for geometry overlapping the cloud the incident distance in the second channel of the 2 channel texture is compared to the linearised pixel value of the the Unity Camera depth texture. If the distance of the cloud incident point is greater than the linear distance from the depth texture the cloud pixel is occluded.

C. Communication Avenues

These inputs are passed from unity through the DLL to CUDA. This has a marginal performance hit as the bus size for CPU to GPU can go up-to 80 Gbps in a 32-lane configuration. This means that transferring a KB would take around 0.1 micro

seconds, a negligible amount of time.
The only data that cannot be moved through these calls are the textures. Copying the data to and from GPU and CPU is incredibly inefficient. Hence the shared memory is used. By registering the Unity textures in CUDA, CUDA is able to write directly to the textures using a mapping and the surf graphics object. The impact on performance is microseconds, which is acceptable.

V. Experiments

When evaluating this code there are 3 broad features to test. Performance, visual fidelity and resource usage. Each of these are effected by a few hyperparameters from the model.

A. Performance

The performance of the program can be measured in render time of each render call in the OnRenderImage, it can also be related to the frame rate of Unity Program. By changing the hyperperameters we can measure and graph these metrics to gauge how much the performance is influenced.

B. Visual fidelity

To quantify the visual fidelity of the image 3 image processing techniques are deployed:
Mean Squared Error (MSE): MSE measures the average of the squared differences between the pixel values of the reference and test images. It provides an overall indication of the magnitude of differences, with lower values indicating higher similarity.
Peak Signal-to-Noise Ratio (PSNR): PSNR is a metric that compares the signal power to the noise power of an image. Results are measured in dB and y above 30dB is considered acceptable. The paper [8] evaluates the effectiveness of the method in the context of image content.
Structural Simularity Index Measure (SSIM) compares the local structures of an image to quantify the visibility of errors [9]. Some components of these structures are contrast and luminance.

C. Resource use

Majority of the programs and data reside on the GPU. There are 2 main measures, the GPU compute usage (in %) and the GPU dedictated ram usage.

D. Testing Hyperperameters

There are 3 main hyperperameters that can be adjusted to possibly improve performance : Screen resolution, model resolution, and ray marching step count.

1) Screen Resolution: By running the model at different resolutions we directly increase the amount of rays projected from the screen (1 per pixel). Using a geometric step of 1.125, 10 different resolutions are generated ranging from 512 to 1440. 1440 is an important square resolution as it has the same amount of pixels as the standard resolution of 1920*1080 pixels.

2) Model resolution: The models volumetric data impacts the memory usage of the program, as the size is cubic in memory consumption (3 dimensions). Using a sampling method we can move the model to lower resolutions to save memory. We test the resolutions from 1024 to 128 using a cubic step function.

3) Ray Marching Steps: When the ray incidents with the cloud, it steps through the volume. The amount of steps is controlled by a maximum step count, maximum distance (if the ray exits the bounding box) and lastly a minimum transmittance. To test the performance hit of this parameter, different maximum step counts are used (1024, 500,400,300,200,100,50,25,10).

E. Testing environment

All of the tests are executed on a Lenovo ideapad Gaming 3. It has an RTX 3060 6gb, i7-12650H processor and 32 Gigabytes of . CUDA is version 11.8 and Unity is 2022.3.4.f1 memory.

VI. Results and discussions

A. Qualitative Results

TABLE I: PSNR, SSIM, MSE increasing resolutions

Resolutions	512	576
psnr	inf	38.92
ssim	1.0	0.99
mse	0.0	3.29

Fig. 5: MRPNN Paper

Fig. 6: Unity

Fig. 7: MRPNN vs Unity results

Comparing the output images from our Unity implementation with the original MRPNN implementation [1] reveals a remarkable degree of visual similarity that approaches indistinguishability. The side-by-side comparison (Fig. 7) demonstrates that our implementation successfully reproduces the complex volumetric characteristics of the original work, including subtle details in cloud density gradients, light scattering effects, and overall cloud morphology. The quantitative metrics strongly support this visual assessment. As shown in Table I, the Peak Signal-to-Noise Ratio (PSNR) is exceptionally high at 38.92dB , exceeding the typical threshold of 30 dB for high-quality image reproduction. More tellingly,

Fig. 8: Performance impact of different resolutions

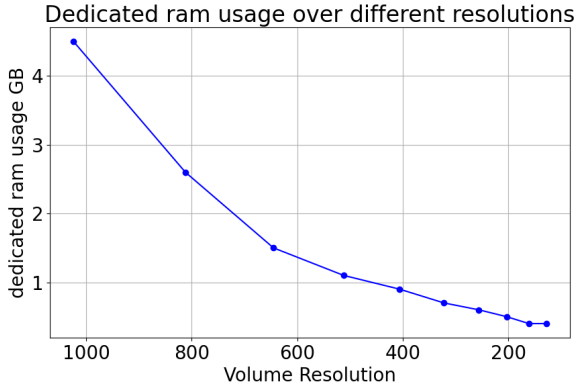


Fig. 9: Dedicated ram usage against volume resolution

the Structural Similarity Index Measure (SSIM) scores a 0.99 indicating near-perfect structural similarity between the implementations.

B. Increasing image Resolution

1) Frame rate: The performance (measured in frames) linearly drops in 8 until the the 1100 mark where it drastically drops. The linear trend is expected as more rays means more compute. The sudden drop in performance is because the GPUs compute reached 96 %, effectively bottle-necking the performance.

2) Image quality: In table II SSIM we see promising results, all of the images score above 30dB. There is a slow downward trend but all images are in a suitable range. The same is true for the PSNR .
MSE has given conflicting results. This is because it is very sensitive to the resolution instead of visual fidelity, invalidating these results.

C. Reducing Volumetric Resolution

The memory required for volumetric cloud data at full resolution is 4GB, which nearly fills the RTX 3060's 6GB memory, limiting resources for other graphics. To address this, a down-sampling method was developed to reduce texture size . Down-sampling greatly reduces memory size as the 3d

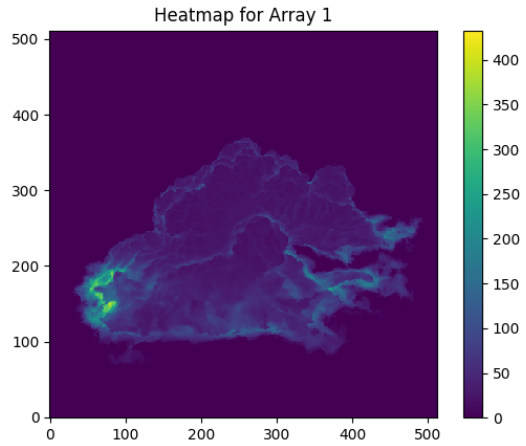


Fig. 10: Number of steps per ray

texture is cubic in scaling memory consumption, and there are 9 3d mipmap textures saved in memory. To test the effectiveness of the downsampling we ran the program with 10 samples, using cubic steps from 1024 to 128. We can then graph at what point does the image quality degrade and how much ram is reduced by.
Figure 9 shows a memory usage from 4.3 down to 0.4. The first 3 resolutions show a massive improvment which slows and converges towads 0.4 in the last few resolutions.
In table III we see:

- SSIM : shows little change. This is because the image is structurally not changing, mostly becoming more blurry
- PSNR Shows a more linear decline in the DB, but only at the resolution of 161 reaches a sub 30 dB measure. Indicating that even though there is a linear trend, there is space to improve the models memory efficiency without a major drop in visual fidelity.
- MSE : reaffirms the trend displayed in PSNR.

For a good compromise between image quality and memory usage, values between 500 and 600 are optimal. Future values show less of an improvement in performance and any reduction is directly proportional to image degradation.

D. Reducing Ray-marching

As seen in Figure 10 the number of ray-marched steps is already much lower than their default 1024. This is due to early termination parameters. The max is 400 steps and most pixels sit around 200-300 ray marched steps.

In testing each max step amount, table IV clearly shows a steep decline in visual fidelity around 50 steps (the lowest PSNR,SSIM and the highest MSE of all other tests). This drastic decrease in visual fidelity bears little fruit as figure 11 demonstrates a performance change of 1ms at the lowest step count. This marginal change to performance along side a large decrease in fidelity suggests this avenue of optimization is redundant.

TABLE II: PSNR, SSIM, MSE increasing resolutions

Resolutions	512	576	648	729	820	923	1038	1168	1314	1440
psnr	40.75	41.42	42.08	42.81	43.56	44.33	44.95	45.61	46.15	inf
ssim	0.99	0.99	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
mse	1.33	1.	1.16	1.08	1.0	0.93	0.87	0.74	0.66	0.0

TABLE III: PSNR, SSIM, MSE decreasing volume resolution

Resolutions	1024	812	645	512	406	322	256	203	161	128
psnr	inf	43.43	42.21	40.09	38.54	36.36	34.51	32.46	30.52	28.7
ssim	1.0	0.99	0.99	0.99	0.99	0.98	0.98	0.97	0.96	0.96
mse	0.0	1.35	1.56	1.92	2.33	2.9	3.47	4.27	5.02	6.08

TABLE IV: PSNR, SSIM, MSE decreasing ray-marching

Resolutions	1024	500	400	300	200	100	50	25	10
psnr	inf	39.46	39.3	39.5	39.51	38.18	34.92	30.39	25.39
ssim	1.0	1.0	1.0	1.0	1.0	0.99	0.98	0.97	0.95
mse	0.0	0.87	0.88	0.88	0.93	1.33	2.8	5.08	8.37

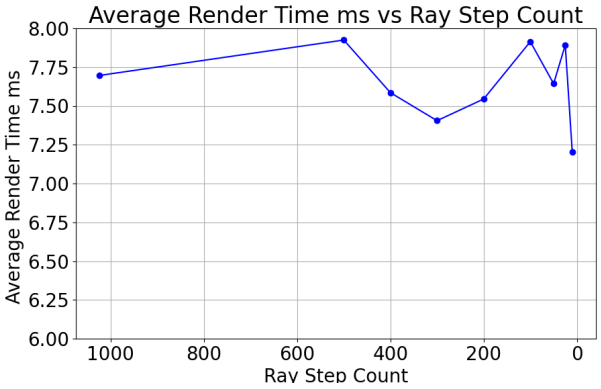


Fig. 11: Performance of ray step count against render-time

VII. Conclusion

This research successfully demonstrates the integration of an MRPNN-based cloud rendering system within the Unity game engine, achieving real-time performance through efficient resource sharing between CUDA and DirectX. Our implementation maintains visual fidelity while managing the computational demands of neural network inference and volumetric rendering.

Key findings from our optimization studies reveal that screen resolution has the most significant impact on performance, showing a linear relationship with frame rates until GPU utilization reaches 96%. While reducing ray marching steps offered minimal performance benefits, our analysis shows that model resolution can be effectively decreased to optimize memory usage without severely compromising visual quality, as evidenced by SSIM scores consistently above 30dB.

However, several limitations and opportunities for future work emerge from this study. The inference time of the neural network remains a significant performance bottleneck, suggesting that architectural improvements or alternative inference strategies could bring further optimization. Additionally, comparative analysis against industry-standard cloud rendering techniques would provide valuable context for assessing the practical viability of this approach.

Future research directions could explore:

- Devolping a more relaxed version of the neural network,
- Implementing adaptive sampling strategies based on viewing conditions
- Conducting comprehensive comparisons with existing cloud rendering solutions

The successful integration of neural rendering techniques into commercial game engines represents an important step toward making advanced volumetric effects accessible in real-time applications. As graphics hardware and neural network architectures continue to evolve, the approach demonstrated in this paper provides a foundation for further development of efficient, high-quality cloud rendering solutions.

References

[1] J. H. C. Y. H. L. L. Y. Y. Wu and X. Jin, ". deep real-time volumetric rendering using multi-feature fusion," in *Special Interest Group on Computer Graphics and Interactive Techniques Conference Conference Proceedings (SIGGRAPH '23 Conference Proceedings)*, Aug. 2023.

[2] C. D. Hansen and C. R. Johnson, *The Visualization Handbook*. Burlington, MA, USA: Elsevier Butterworth–Heinemann, 2005, ch. Overview of Volume Rendering, pp. 127–164.

[3] R. Högfeldt, "Convincing cloud rendering–an implementation of real-time dynamic volumetric clouds in frostbite," 2016.

[4] G. Games, "Nubis: Authoring real-time volumetric cloudscapes with the decima engine," Advances in Real-Time Rendering Course, 2015, July 2017, accessed: 2024-11-18. [Online]. Available: https://www.guerrilla-games.com/presentation-url

[5] T. Müller, B. McWilliams, M. Gross, and J. Novák, "Deep scattering: Rendering atmospheric clouds with radiance-predicting neural networks," in *ACM Transactions on Graphics (TOG)*, vol. 36, no. 4, November 2017.

[6] F. Messaoudi, G. Simon, and A. Ksentini, "Dissecting games engines: The case of unity3d," in *2015 International Workshop on Network and Systems Support for Games (NetGames)*, 2015, pp. 1–6.

[7] L. Ge, B. Wang, L. Wang, X. Meng, and N. Holzschuch, "Interactive simulation of scattering effects in participating media using a neural network model," *IEEE Transactions on Visualization and Computer Graphics*, vol. 27, no. 7, pp. 3123–3134, 2019.

[8] J. Korhonen and J. You, "Peak signal-to-noise ratio revisited: Is simple beautiful?" in *2012 Fourth International Workshop on Quality of Multimedia Experience*, 2012, pp. 37–38.
[9] W. Zhou, "Image quality assessment: from error measurement to structural similarity," *IEEE transactions on image processing*, vol. 13, pp. 600–613, 2004.

APPENDIX A
APPENDIX: ADDITIONAL OUTPUTS

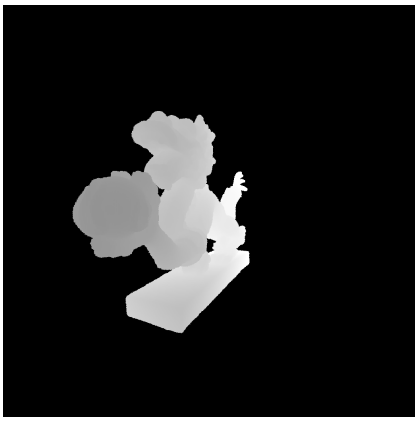
(a) Cloud 0

(b) Cloud 1

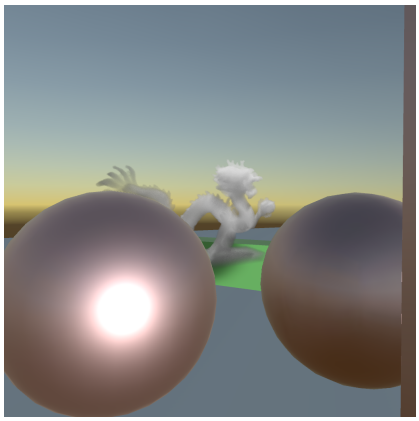
(c) Dragon

Fig. 12: Additional Qualitative Results

APPENDIX B
APPENDIX: DEPTH MASKING



(a) Dragon Depth Mask

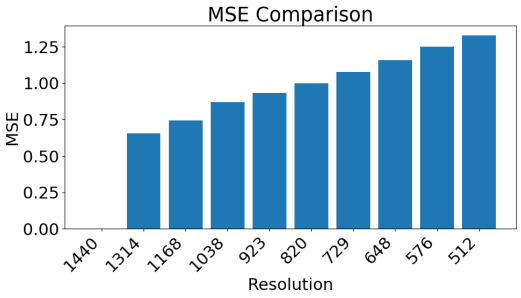


(b) Dragon Behind Unity Geometry

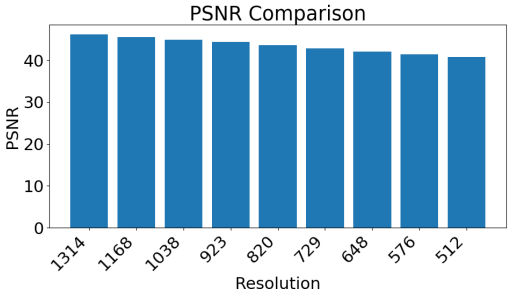
Fig. 13: Depth Mask and Working depth masking

APPENDIX: IMPLEMENTATION DETAILS

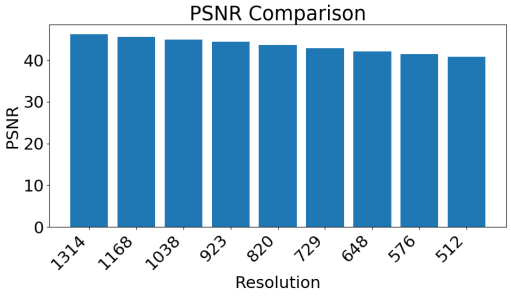
A. Resolution



(a) Mean Square Error of different resolutions

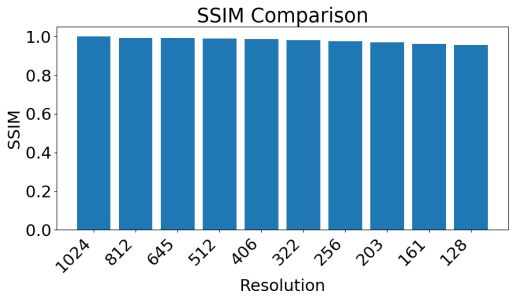


(b) Peak Signal-to-Noise Ratio of different resolutions

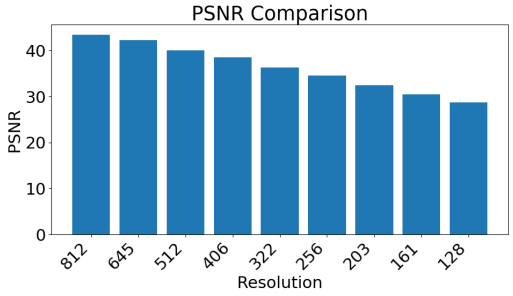


(c) Structural Similarity Index Measure of different resolutions

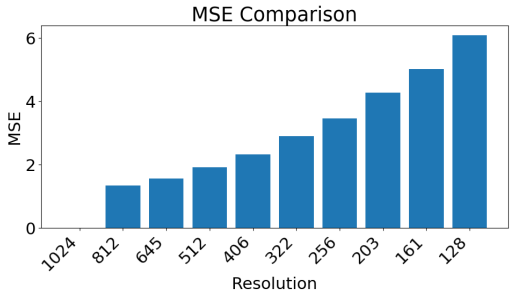
Fig. 14: Various metrics against resolution



(a) SSIM of different volume resolutions



(b) PSNR of different volume resolutions

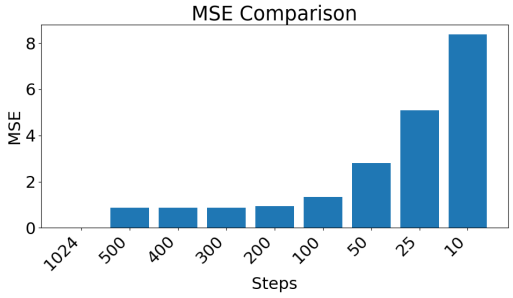


(c) MSE of different volume resolutions

Fig. 15: Various metrics against volume resolution

B. Volume resolution

C. Ray-Marching



(a) MSE Ray