Efficient Encoding of Graphics Primitives with Simplex-based Structures

Frank Yang, Yibo Wen

November 10, 2023

1 Abstract

Grid-based structures are commonly used to encode explicit features for graphics primitives such as images, signed distance functions (SDF), and neural radiance fields (NeRF) due to their simple implementation. However, in n-dimensional space, calculating the value of a sampled point requires interpolating the values of its 2^n neighboring vertices. The exponential scaling with dimension leads to significant computational overheads. To address this issue, we propose a simplex-based approach for encoding graphics primitives. The numbers of vertices in a simplex-based structure increase linearly with dimension, making it a more efficient and generalizable alternative to grid-based representations. Using the non-axis-aligned simplicial structure-property, we derive and prove a coordinate transformation, simplicial subdivision, and barycentric interpolation scheme for efficient sampling, which resembles transformation procedures in the simplex noise algorithm. Finally, we use hash tables to store multiresolution features of all interest points in the simplicial grid, which are passed into a tiny fully-connected neural network to parameterize graphics primitives. We implemented a detailed simplex-based structure encoding algorithm in C++ and CUDA using the methods outlined in our approach. In the 2D image fitting task, the proposed method is capable of fitting a giga-pixel image with 9.4% less time compared to the baseline method proposed by instant-ngp, while maintaining the same quality and compression rate. In the volumetric rendering setup, we observe a maximum 41.2% speedup when the samples are dense enough. We further showcase its potential applications including high-dimensional data compression and 3D reconstruction.

2 Introduction

Graphics primitives are building blocks used to create complex visual scenes in computer graphics. These primitives can be thought of as functions that map positional or directional information from $\mathbb{R}^m$ to attributes in $\mathbb{R}^n$ (Figure 1). For example, in image fitting, the graphics primitives function maps positional coordinates x and y

to the color profile RGB. In high dimensions, graphics primitives can also be used to represent occupancy networks [9, 4] and signed distance functions [12, 17].

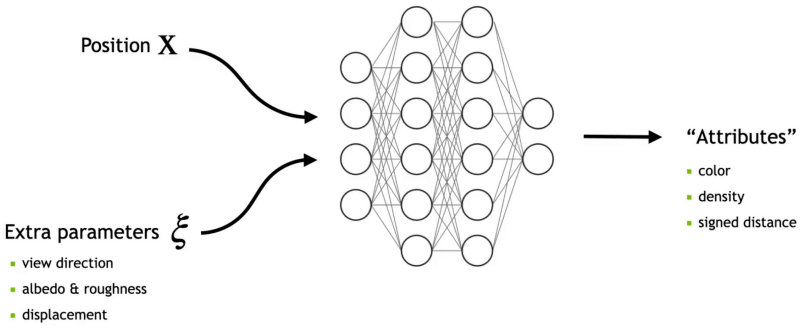


Figure 1: Illustration of graphics primitives

In recent years, Neural Radiance Fields (NeRF)[10] has emerged as a powerful technique for modeling graphics primitives using neural networks. NeRF uses multi-layer perceptrons (MLP) and techniques from differentiable rendering to learn a volumetric representation of complex scenes with only 2D image supervision. By taking both position and direction as the network input, NeRF can capture realistic view-dependent effects without any separation of shape and light in Figure 2. NeRF has been used for a variety of applications in the field of computer graphics.
The quality and performance characteristics of the mathematical representation are crucial for visual fidelity: we desire representations that remain fast and compact while capturing high-frequency, local detail. While neural networks, such as multi-layer perceptrons (MLPs), have shown great potential in modeling graphics primitives, they can struggle to capture high-frequency details due to their inherent smoothness. For example, when trained on a naive MLP, the performance of a NeRF rendering model can be poor, as indicated by a low peak signal-to-noise ratio (PSNR) score and long training time till convergence as shown in Figure 3.

$$(x, y, z, \phi, \theta) \rightarrow (R, G, B, \sigma)$$

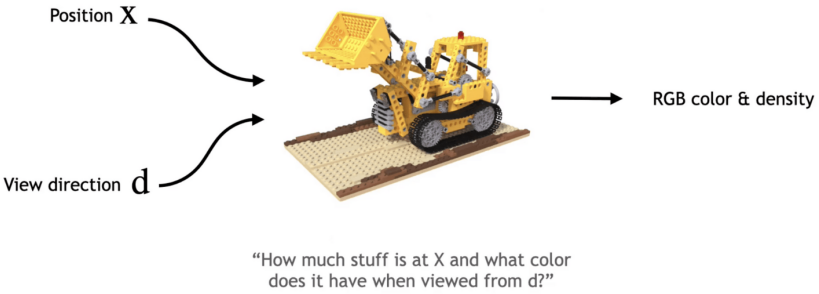


Figure 2: Example of a scene modeled using Neural Radiance Fields [10].

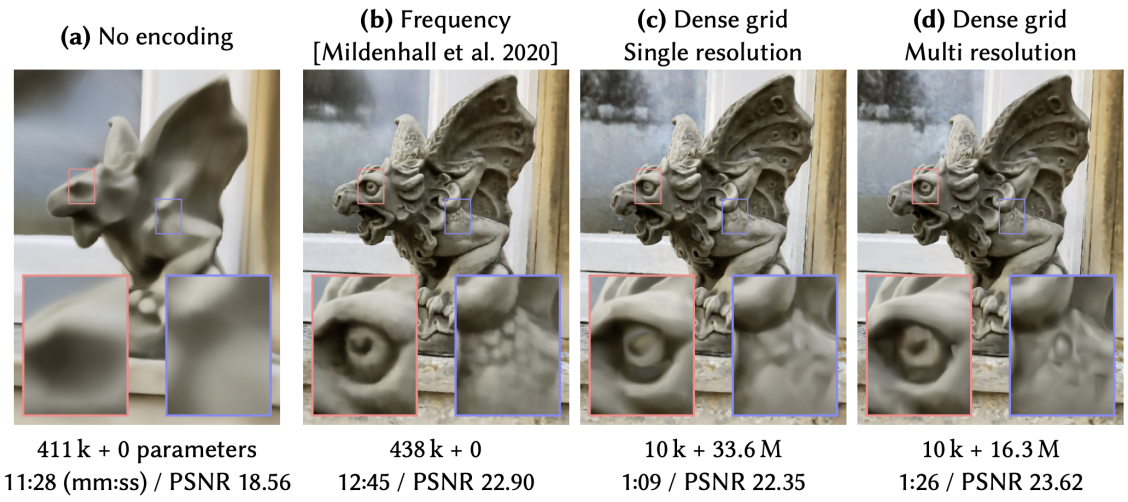


Figure 3: NeRF rending and respective performance on different encodings [11].

To overcome this limitation, neural networks are often paired with various encoding techniques, which map the input into higher dimensionality to allow a finer representation of high-frequency details. Existing examples of input encoding into high-dimensional space prove that attention components of recurrent architectures and, subsequently, transformers, are useful in identifying the location at which the neural network is currently processing. There are various popular variants of grid-based structures used for encoding graphics primitives, such as dense grid [16, 7], sparse grid [6, 15], octree [18], tensor decomposition [3], planar factorization [1, 2, 5], and multi-resolution grid [11]. Amongst all encoding methods, frequency encoding and parametric encoding have been widely adopted to encode the spatial-directionally varying light fields and volume density in the NeRF algorithm [11].
In the original NeRF paper, the authors proposed a frequency encoding technique to capture high-frequency details [10]. Inspired by the positional encoding in the transformer, frequency encoding encodes scalar positions as a multi-resolution sequence of sine and cosine functions.

$$enc(x) = (\sin(2^0 x), \sin(2^1 x), ..., \sin(2^{L-1} x), \cos(2^0 x), \cos(2^1 x), ..., \cos(2^{L-1} x)). \quad (1)$$

Because of such independence, frequency encoding has poor capability in generalizing patterns in high-dimensional space. An experimentation of the reconstruction takes a significant amount of epochs to converge [11]. Additionally, frequency encoding does not carry any trainable parameter and therefore requires a comparatively large fully connected MLP to represent the same amount of pixels. In Figure 3, the training took more than 12 minutes to converge.
Parametric encoding attempts to address the long training time by representing the neural network with an explicit dense-grid representation. Parametric encoding appends feature from the neighboring points in a grid into an auxiliary data structure, such as a tree. Theoretically, this arrangement trades a larger memory footprint for a smaller computational cost: the backward propagation through the MLP for

each trainable input encoding parameter is hardly significant in the overall training time. By reducing the size of MLP, such parametric models can typically be trained to converge much faster without sacrificing quality. However, the dense grid stores an excessive number of parameters, which consumes much more memory than the original neural network weights than necessary. Such an arrangement allocates as much memory on features to areas of empty spaces as it does to those areas near the surface. For 3D encoding scenarios, the number of parameters grows $\mathcal{O}(N^3)$, whereas the number of the surface area grows $\mathcal{O}(N^2)$. The number grows even faster in higher dimensions. Besides, the time complexity is not linearly scaled for a repetitive lookup in the dense grid in high-dimensional image reconstruction settings such as NeRF. The encoding achieves fast training and excellent representation at the cost of 33.6M parameters as shown in Figure 3.

The state-of-the-art, multi-resolution hash encoding [11], combines both ideas to reduce waste and optimize complexity. The arrangement neither relies on a progressive update during training nor on prior knowledge of the geometry of the scene. Analogous to the multi-resolution grid in parametric encoding, multi-resolution hash encoding use multiple separate hash tables indexed at different resolutions, whose interpolated outputs are concatenated before being passed through the MLP. The reconstruction quality is comparable to the dense grid encoding, despite having $20\times$ fewer parameters.

To further improve the encoding training efficiency, we noticed that interpolation is inherently grid-based. Evaluating an input in n dimension requires 2^n neighboring vertices, leading to unnecessary computation. Inspired by simplex noise, we take advantage of simplex structures, which by definition is the polygon with the least number of vertices tiling an n-dimensional space. The simplex-based structure is more advantageous than the grid-based structure in encoding in two ways:

Fewer variables: In dense-grid encoding, the number of variables required increases exponentially with the dimensionality of the graphics primitives, making it impractical for high-dimensional problems. In contrast, a simplex-based structure uses only the n+1 vertices in n-dimension regardless of the dimensionality. This would lead to a significant improvement in computation speed.

Fewer artifacts: In simplex-based structure, the vertices of the simplex are typically well-separated and represent different combinations of variables. This reduces the correlation between variables, which in turn reduces the likelihood of artifacts arising due to the interactions between variables. Simplex-based encoding can more easily handle nonlinearities due to its ability to adapt to the shape of the solution space. In contrast, the dense-grid shape is fixed, leading to artifacts on discontinuities or sharp edges.

In later sections, we will further review the properties of grid and simplex structures on noises (Section 3) and present our proposed method with simplex-based encoding (Section 4) with the implementation of a simplex-based structure backed by multi-resolution hash encoding, a state-of-the-art method on graphics primitives

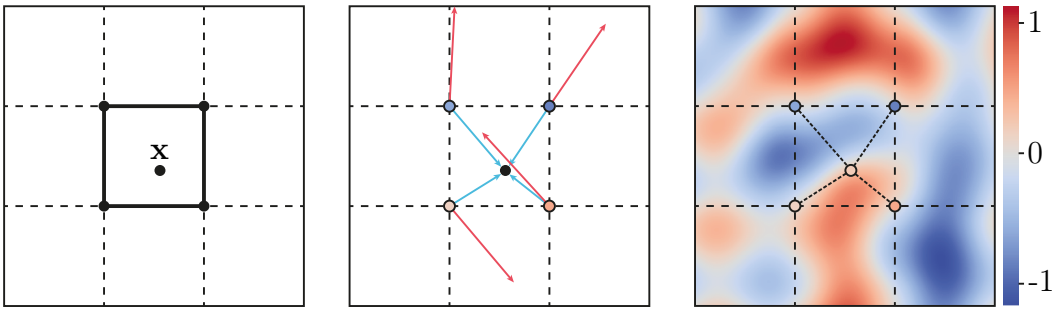
(Section 5). We then verify our performance and feasibility with multi-dimensional experiments on CPU and GPU (Section 6). We finally conclude with future works and discussions thereof (Section 7) with mathematical derivations of the proposed algorithm feasibility (Section 8).

3 Background

We observe that the feature retrieval procedure of the aforementioned methods shares similarities with the Perlin noise algorithm. Both utilize a grid-based structure and apply n-linear interpolation to obtain final output values. The main difference between them lies in the type of features used, with trainable features in one and randomly generated ones in the other. To address this limitation, we draw inspiration from simplex noise, a predecessor of Perlin noise that employs a simplex-based structure for graphics primitives. To provide a better understanding of our proposed method, we will briefly cover the background of both Perlin noise and simplex noise algorithms.

3.1 Perlin noise

Perlin noise [14], also known as classical noise, is a procedural generation algorithm first introduced by Ken Perlin in the 1980s. Due to its natural appearance and simple implementation, it has been widely adopted in computer graphics for generating visual content including texture, terrain, smoke, etc.



(a) Grid subdivision (b) Noise evaluation (c) Linear interpolation

Figure 4: Illustration of Perlin noise in 2D.

In n dimensional space, Perlin noise can be viewed as a pseudo-random mapping $\mathbb{R}^n \to \mathbb{R}$ and can be calculated with the following three steps as shown in Figure 4: (a) Grid subdivision. Given an input coordinate $\mathbf{x} \in \mathbb{R}^n$, we determine a grid cell that contains $\mathbf{x}$. The cell is a n dimensional hypercube with 2^n vertices in $\mathbb{Z}^n$ spanned by $\lfloor \mathbf{x} \rfloor$ and $\lceil \mathbf{x} \rceil$. (b) Noise evaluation. For each vertex $\mathbf{x}_i \in \mathbb{Z}^n$ of the hypercube, we generate a pseudo-random gradient vector of unit length $\mathbf{g}_i \in \mathbb{R}^n, ||\mathbf{g}_i|| = 1$ and a displacement vector $\mathbf{d}_i = \mathbf{x} - \mathbf{x}_i$. We then use the dot product $p_i = \mathbf{g}_i \cdot \mathbf{d}_i$ of the two vectors as the noise vector. (c) Linear interpolation. Finally, we use n-linear

interpolation to obtain the final noise scalar at $\mathbf{x}$. When higher-order smoothness is desired, instead of using n-quadratic or n-cubic interpolation, we can apply a more efficient smoothing function to the 2^n dot products on vertices. Particularly, Perlin noise uses the smoother-step function,

$$S_2(x) = 6x^5 - 15x^4 + 10x^3, 0 \leq x \leq 1, \tag{2}$$

which is C^2 smooth and has vanishing derivatives at the endpoints. Considering the procedures mentioned above, Perlin noise requires evaluation at 2^n vertices of the containing grid cell and calculation of 2^{n-1} weighted sums during interpolation. Therefore, the algorithm scales with $\mathcal{O}(2^n)$, which grows exponentially with dimension.

3.2 Simplex noise

Perlin noise, while very useful, suffers from exponential scaling across dimensions and directional artifacts in image reconstruction [13]. Those shortcomings inspire us to investigate a new reconstruction primitive: simplex noise. Rather than placing each input point into a cubic grid based on the integer parts of coordinate values, the input point is placed onto a simplicial grid, which is derived by dividing n-dimensional space into a regular grid of shapes with a minimum number of vertices (triangles in 2D, tetrahedrons in 3D, and so on). It's important to note that the number of simplex vertices in each dimension is $n + 1$, where n is the number of sizes.

Compared to Perlin noise, simplex noise can be generated with lower computational overhead, especially in higher dimensions. Simplex noise scales to higher dimensions with much less computational cost: the complexity is $\mathcal{O}(n^2)$ (depending on the sorting algorithm) for n dimensions. Simplex noise inherently produces fewer noticeable directional artifacts and is more isotropic (meaning it looks the same from all directions) than the Perlin noise. Both of these advantages over Perlin noise are crucial for a robust and computationally efficient image reconstruction algorithm.

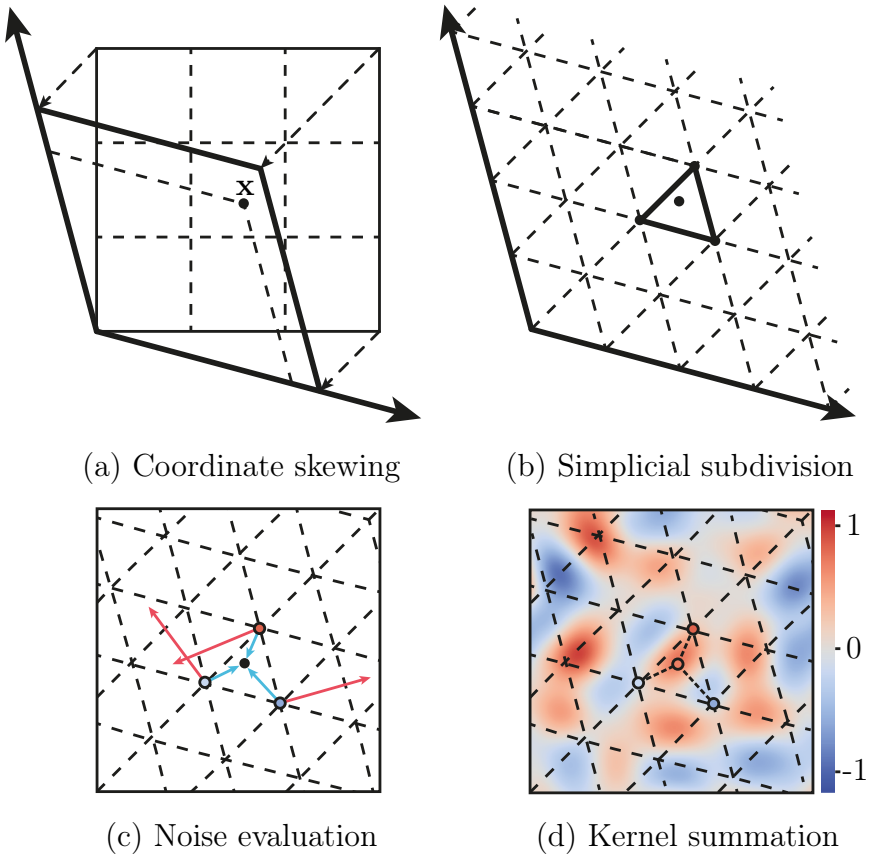
To generate a simplex noise in n dimensional space, the following 4 steps must be made as shown in Figure 5.

(a). Coordinate skewing: The coordinate axis in the n dimension is skewed such that the coordinate vector aligns with the simplex shape. In a 2D example, the x-y Cartesian coordinate is translated to a new u-v plane. The coordinate translation formula is given below,

$$\mathbf{x}' = \mathbf{x} + \mathbf{1}_n^T \cdot F_n \sum_i x_i, \quad F_n = \frac{\sqrt{n+1} - 1}{n}, \tag{3}$$

This has the effect of rearranging a hyper-cubic coordinate that has been squashed along its main diagonal such that the distance between the points (0, 0, ..., 0) and (1, 1, ..., 1) becomes equal to the distance between the points (0, 0, ..., 0) and (1, 0, ..., 0).

(b). Simplicial subdivision: Once the input coordinate is determined in the translated coordinate system, the surrounding lattice point of an input in the sim-



(a) Coordinate skewing (b) Simplicial subdivision

(c) Noise evaluation (d) Kernel summation

Figure 5: Illustration of Simplex noise in 2D.

plex grid is calculated via the following steps. First, take a floor and ceiling of the coordinates in the input. For input with coordinate $(x, y, z, ...)$ in the simplex coordinate, it lies in a simplex with at least coordinate spanned by $(\lfloor \mathbf{x} \rfloor, \lfloor \mathbf{y} \rfloor, \lfloor \mathbf{z} \rfloor ...)$ and $(\lceil \mathbf{x} \rceil, \lceil \mathbf{y} \rceil, \lceil \mathbf{z} \rceil ...)$. Then, the coordinate $(x_i, y_i, z_i, ...)$ are sorted in decreasing order. Start with $(\lfloor \mathbf{x} \rfloor, \lfloor \mathbf{y} \rfloor, \lfloor \mathbf{z} \rfloor ...)$, successively add 1 to the largest point in the coordinate until all $n + 1$ simplex points are found.

(c). Noise evaluation: At each vertex of the grid, a random gradient vector is assigned. To generate a noise value at a given point in space, the algorithm first determines which simplex shape contains the point, and then interpolates the gradient vectors at the vertices of that simplex to obtain a weighted sum. The resulting value is then scaled and smoothed to produce the final noise value.

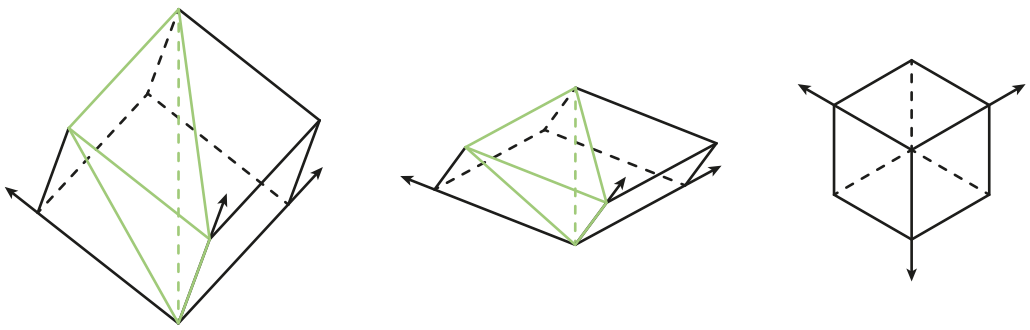
(d). Kernel summation: The input in the simplex coordinate is skewed back into the Cartesian coordinate system using the formula below,

$$\mathbf{x} = \mathbf{x}' - \mathbf{1}_n^T \cdot G_n \sum_i x_i', \quad G_n = \frac{1 - 1/\sqrt{n+1}}{n}. \tag{4}$$

Note that the translated unscrewed coordinate is precisely the coordinate of the original input in the orthogonal axes.

4 Proposed Method

To optimize the performance of sampling and interpolation, it may be beneficial to replace n-cubes with n-simplices, as simplices are defined as polygons with the least number of vertices in each respective dimension. For example, a simplex is a triangle in two-dimensional space and a tetrahedron in three-dimensional space. However, regular simplices that have equal edges cannot tile space beyond two dimensions. Furthermore, indexing the vertices of equilateral triangles in two dimensions is no easy task without careful manipulation. Fortunately, the simplex noise algorithm has provided a solid foundation for such operations. By making some adaptations, we can apply this methodology to a wider range of tasks, including parameterizing graphics primitives. In doing so, we can make full use of simplex-based structures to reduce computational costs and optimize memory usage during sampling and interpolation. For showing the correctness of our proposed method in arbitrary dimensions, we derive and prove the following theorems and lemmas (please refer to **Appendix A** for detailed proofs):



(a) 3-orthoscheme (b) Coordinate skewing (c) Simplicial subdivision

Figure 6: Illustration of the proposed method with coordinate skewing and simplicial subdivision in three dimensions.

Theorem 1. $S = \{\mathbf{x} \in \mathbb{R}^n : 0 \leq x_1 \leq \cdots \leq x_n \leq 1\}$ is a n-simplex.
Remark 2. Let π denote a permutation of $\{1, \cdots, n\}$, then $S_\pi = \{\mathbf{x} \in \mathbb{R}^n : 0 \leq \mathbf{x}_{\pi(1)} \leq \cdots \leq \mathbf{x}_{\pi(n)} \leq 1\}$ is a n-simplex. The n-simplex has $n+1$ vertices $\mathbf{v}_1, \cdots, \mathbf{v}_{n+1}$, where all entries at $\{\pi_1, \cdots, \pi_{i-1}\}$ of $\mathbf{v}_i$ are 1 and the rest are 0. Additionally, $\mathbf{v}_1$ is $\mathbf{0}^T$ and $\mathbf{v}_{n+1}$ is $\mathbf{1}^T$.
Theorem 3. All S_π are congruent.
Lemma 4. After coordinate transformation, all possible S'_π are still congruent.
Theorem 5. A n-hypercube can be triangulated into $n!$ disjoint congruent simplices.

4.1 Coordinate skewing

Similar to the simplex noise algorithm, we first apply coordinate transformation according to Equation 3. This skewing operation can also be rewritten as

$$\mathbf{x}' = \begin{bmatrix} 1+F_n & \cdots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \cdots & 1+F_n \end{bmatrix} \mathbf{x}, \quad F_n = \frac{\sqrt{n+1}-1}{n}, \tag{5}$$

where the sampled point $\mathbf{x} \in \mathbb{R}^n$ is multiplied by a constant matrix. As such, coordinate skewing is an affine transformation [1] that preserves the parallelism of planes and transforms the unit n-hypercube into an n-parallelpiped.
This transformation is crucial as it reduces distortion in simplex cells, resulting in the more balanced spatial division. Without this step, each cell would be a n-orthoscheme, which is a generalization of a right triangle in higher dimensions. The transformation leads to equilateral triangles in two-dimensional space and tetrahedrons with congruent isosceles faces in three-dimensional spaces. More specifically, this tetrahedron has 4 edges of the same length and another 2 edges of the same length which can be derived from the proof for Lemma 4. We can also calculate the ratio of the two different types of edges which is $\frac{\sqrt{3}}{2}$. Thus by using a coordinate transformation, we aim to avoid axis-aligned artifacts and increase quality in representing graphics primitives.

4.2 Simplicial subdivision

After transforming the input coordinate into simplex-based coordinates, we will need to locate the coordinates of neighboring vertices in the n-dimension hypercube. According to Theorem 5, in n-dimensional space, the hypercube can be split into $n!$ number of disjoint and congruent simplices. While congruency is not preserved under general affine transformation, we prove in Lemma 4 that under our coordinate skewing, the $n!$ simplices are still congruent. Assume the given point is sampled within the hypercube, then to locate the cell containing it, we need to determine which simplex cell among the $n!$ simplices it resides. For this step, we use a similar subdivision scheme from the simplex noise algorithm. Through direct sorting, we can find the corresponding vertices of the cell as shown in the proof for Remark 2. For a given point $\mathbf{x}$ inside the unit hypercube, we let $x_1, \cdots, x_n$ denote the sorted entries of $\mathbf{x}$ in descending order and record their respective original index. Coding-wise, we perform sorting for the input while preserving their index [2]. Then, we can obtain the $n+1$ vertices starting from $\mathbf{0}^T$ as the base vertex $\mathbf{v}_1$ [3]. By adding 1 to the base vertex at the index of the next largest entry of $\mathbf{x}$, we obtain the next vertex and use that as the new base. The process is repeated until we finally reach vertex $\mathbf{v}_{n+1}$, which should always be $\mathbf{1}^T$.

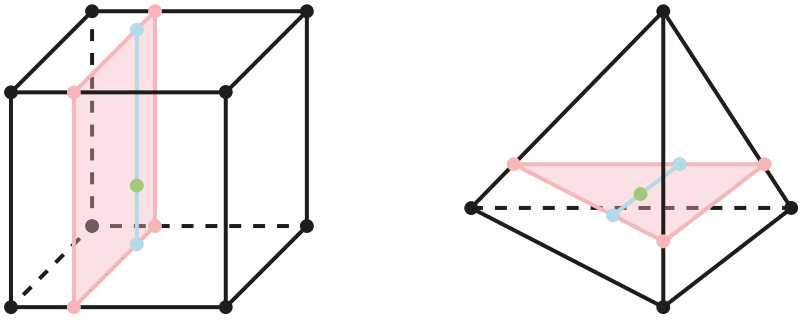
[1] A map f: X → Z is an affine map if there exists a linear map m_f : V → W such that m_f(x − y) = f (x) − f (y) for all x, y in X. Such transformation preserves the lines and parallelism

[2] In the implementation, bubble sort is used as the input dimension n is small enough, under which bubble sorting outperforms other methods

[3] After the 5.1 Scale Adjustment, we would always end with an input with all coordinates less than 1. 0^T is guaranteed to be the lower corner of the simplex structure

4.3 Barycentric interpolation

In order to learn the mapping for graphics primitives, instead of generating random gradient vectors as in noise algorithms, we retrieve learnable vectors at each vertex and interpolate them to obtain the final feature for the queried point. To maintain a similar interpolation scheme as the original trilinear interpolation, we didn't select kernel summation from the simplex noise algorithm. Instead, we derive our barycentric interpolation as an efficient alternative which is illustrated in the Figure 7.



(a) Trilinear interpolation (b) Barycentric interpolation

Figure 7: Illustration of linear interpolation for cube and tetrahedron.

In geometry, a barycentric coordinate system is a coordinate system in which the location of a point is specified by reference to a simplex, which makes it the perfect choice for our simplex-based structure. The barycentric coordinate can be found by expressing the point inside a simplex using a convex combination of the neighboring $n+1$ vertices and the coefficients of such combination are the local barycentric coordinate. Since the coordinate skewing is affine, the barycentric coordinate is preserved and the weights are given in the proof of Theorem 1 as follows,

$$w_1 = 1 - x_1, w_2 = x_1 - x_2, \cdots, w_n = x_{n-1} - x_n, w_{n+1} = x_n, \tag{6}$$

where the entries of $\mathbf{x}$ are already sorted in descending order from the previous step. Compared to the computationally expensive n-linear interpolation, the use of a simpler formula for weights in our proposed algorithm also enables a more efficient implementation.

4.4 Example Algorithm

Here we provide an example pseudo-code of our proposed method. The function takes in a point $\mathbf{x}$ inside the unit hypercube, retrieves values at its neighboring simplex vertices, and returns their interpolated values. With slight modifications, this example can be adapted using CUDA for more advanced scenarios including processing inputs in parallel, handling feature vectors, etc. For a detailed demonstration of

Python and sample outputs for each phase of our proposed method, please refer to **Appendix B**.

5 Implementation

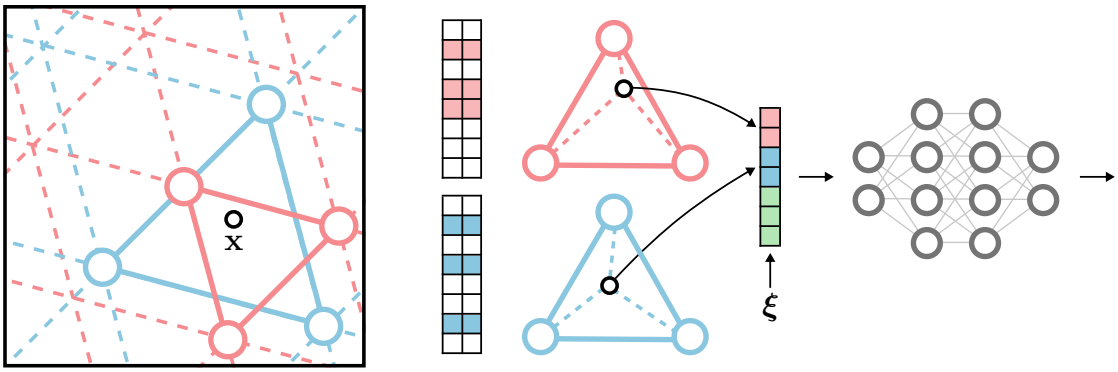


Figure 8: Illustration of the combined method in 2D adapted from Instant-NGP.

To validate our proposed structure for representing graphics primitives, we adopt the structural backbone from instant-NGP [11] and replace its explicit grid-based structure with a simplex-based structure. The demonstration of our implementation in 2D is given in Figure 8:
(a). Hashing of voxel vertices Find the surrounding neighbors at L different resolution levels on a 2D plane and assign indices to their corners by hashing their integer coordinates. **(b). Lookup** Look up the corresponding n-dimensional feature vectors from the hash tables H_L for all resulting corner indices. **(c). Barycentric Interpolation** Perform barycentric interpolation on neighboring coordinates according to the relative position of x within the respective l-th voxel. **(d). Concatenation** Concatenate the result of each level as well as any auxiliary inputs, producing the encoded MLP input. **(e). Neural Network** Backpropagate loss function through the MLP (e) the concatenation (d), the linear interpolation (c), and then accumulated in the looked-up feature vectors.

5.1 Scale Adjustment

In the multiresolution setup, sampling still takes place inside the unit n-cube. The difference is that the n-cube is divided accordingly to each level, or equivalently the input coordinate is scaled up accordingly. Then by simply taking its floor and ceiling, we can identify its local parallelepiped and continue with our proposed simplex algorithm within the unit n-cube.
Additionally, due to coordinate skewing, the vertex $\mathbf{1}_n^T$ is now $\sqrt{\mathbf{n+1}}_n^T$ in the new coordinate system. The resulting parallelepiped is smaller than the original hypercube and cannot cover our sampling volume entirely. Therefore, we need to use an

adjustment scale of $S_n = \sqrt{n+1}$ to avoid accessing points outside the simplicial grids.

5.2 Hash Table Selection

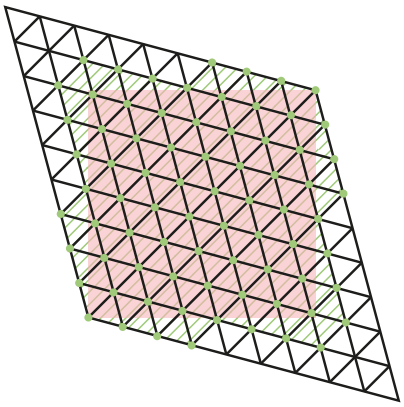
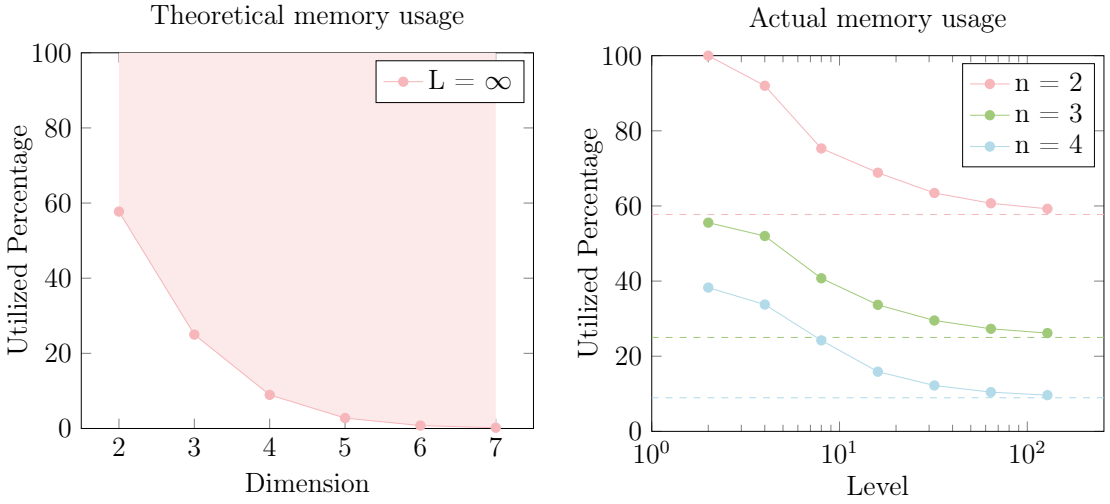


Figure 9: Illustration of an 10×10 simplicial grid in 2 dimension. If all 100 vertices are stored with an array, only 76 entries (colored in green) would be accessed and interpolated when sampling within the unit square (colored in red).

The choice of whether to use a dense grid or hash table can be task-specific for grid-based structures. However, using a dense grid to back a simplicial grid can be extremely inefficient. This is because only a portion of the vertices is accessed when we sample inside the unit n-cube, which leads to significant memory wastage if all vertex features are stored. Figure 9 provides a visual illustration of this in 2D. Assigning an order to the unused vertices to address this issue can cause unnecessary overhead, especially as it varies with grid size and dimension. To solve this problem, we have chosen to use a hash table. This approach avoids the need for explicit ordering, resulting in more efficient and scalable implementation of simplicial grids. To determine the size of the hash table when given dimension and level, we need to calculate the percentage of unused vertices. As the level increases, the volume covered by the used simplex vertices will converge to the volume of the hypercube. Assuming the grid is infinitely dense with at infinite level, the ratio of the two volumes would be 1. Then the percentage of vertices used in the simplex gird could be approximated by the volume ratio of the hypercube and the parallelpiped. The volume of the distorted n-cube, which is a parallelepiped as discussed in coordinate skewing, can be determined by $V = |\det(unskew(\boldsymbol{v_1}, \boldsymbol{v_2}, \ldots, \boldsymbol{v_n}))|$, where $\boldsymbol{v_1}, \boldsymbol{v_2}, \ldots, \boldsymbol{v_n}$ are the n-dimensional vectors that define the edges of the parallelepiped, and $|\cdot|$ denotes the absolute value of the determinant. In the skewed space, the vectors are on the axis and $(\boldsymbol{v_1}, \boldsymbol{v_2}, \ldots, \boldsymbol{v_n})$ is the identity matrix times S_n. By performing coordinate unskewing operation as Equation 4, we can obtain the vector coordinates in the original space. Hence, $V = |S_n^n \det(I_n - G_n C_n)| = S_n^n(1 - nG_n) = S_n^{n-1} = (n+1)^{\frac{n-1}{2}}$, where I_n is the identity matrix and C_n is the constant matrix of 1. Then, the ratio



(a) Theoretical percentage of the memory usage of simplex structures, which is $(n+1)^{-\frac{n-1}{2}}$. The volume ratio between the unit hypercube and the transformed parallelepiped is used for calculating the lower bound.

(b) Analytical percentage of the memory usage of simplex structures with different levels. The dashed lines correspond to the theoretical lower bounds in each dimension. Note that the results were calculated by sampling and may slightly underestimate.

Figure 10: Analysis of memory usage for simplicial structures with different dimensions and levels.

of the parallelepiped and the unit n-cube is $\frac{1}{V}$. As shown in Figure 14a, this ratio decays exponentially.
However, we would expect this ratio to be higher at lower levels due to discretization. We would like to derive an estimation that gives us a varying percentage at different levels. Due to limited memory, we only performed sampling in low dimensions. The result is reported in Figure 14b. As the level increases, the ratio quickly converges to the theoretical lower bound. Therefore, in practice, we could use a hash table with a size of the theoretical ratio to achieve similar quality compared to the collision-free implementation. We could also fix the hash table size and scale our level by $(n+1)^{\frac{n-1}{2n}}$, which is how we implemented it when compared with the baseline methods to guarantee equal memory size.

6 Experiments

To compare simplex-based and grid-based structures in multi-resolution hash encoding [11], we employ a variety of tasks with increasing dimensional inputs to provide comprehensive results. Our evaluation process involves comparing the training performance and benchmark kernel run-time of dense-grid and simplex multi-resolution backbone methods. Through these experiments, we would showcase the versatility of the simplex-based structure and highlight its superiority over traditional grid-based

encoding methods in various applications. We applied our method in the following tasks:

1. **Gigapixel image**: the network learns the mapping from 2D coordinates to RGB colors of a high-resolution image.

2. **Signed distance function (SDF)**: the network learns the mapping from 3D coordinates to the distance to a surface.

3. **Radiance and density fields**: the network learns the mapping from 3D coordinates with 2D viewing directions to the RGB radiance and density.

4. **High Dimension Noise**: the network learns the mapping from n-dimension coordinates to the predetermined noise value.

6.1 Gigapixel Image

Learning the 2D to RGB mapping of image coordinates to colors has become a popular benchmark for testing a model's ability to represent high-frequency detail. Recent breakthroughs in adaptive coordinate networks have shown impressive results when fitting very large images—up to a billion pixels—with high fidelity at even the smallest scales [8, 11]. We attempt to replicate the same experiment targeting to represent high-fidelity images with a hash table and implicit MLP in seconds to minutes. We begin by using the Tokyo gigapixel photograph as a reference image 11 and utilize simplex-based encoding to represent the image with hash maps and MLP parameters. Initially, the hash map features and the MLP weighting is randomly initialized so that the trained image appears noise-like. Through progressive weight back-propagation with ground truth and xy-to-RGB references appending, the network converges to the reference image with an indescribable difference. After 10,000 iterations, we are able to represent a 439M pixels image with only 7.9M trainable parameters, reaching a stunning 1.7 degrees of freedom in Figure 11.
Practically, the simplex-based and grid-based encoding yielded close to almost identical runtime and PSNR scores for this image over-fitting task. With 16 multi-resolution levels and 2^19 size of hash tables, after 10000 iterations, the simplex-based encoding obtains a PSNR of 29.94, whereas the grid-based encoding has a PSNR of 29.82. Additionally, 10000 iterations take 16.34 seconds for grid-based encoding and 14.94 seconds for simplex-based encoding. On 2D tasks like image overfitting, the grid-based structure requires information from 4 neighboring vertices, whereas the simplex-based structure has 3 neighboring vertices to interpolate with extra computational overhead on coordinate skewing and simplicial subdivision. While we do not expect any runtime improvement over instant-NGP, we would like to first use this experiment to verify the feasibility of simplex encoding. We use our analogous dense structure to check if we can produce similar results in both speed and quality. We then adopt the multi-resolution structure and compare the results with instant-NGP.

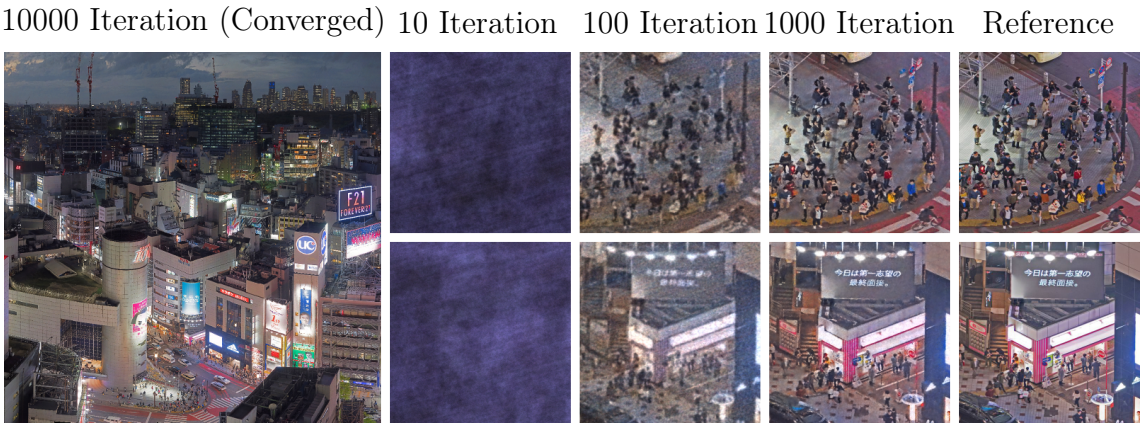


Figure 11: Optimization results from fitting an RGB image with 439M pixels (21450 × 21450). We use the same configurations with 7.9M trainable parameters (7.87M + 7k). Tokyo gigapixel photograph ©Trevor Dobson (CC BY-NC-ND 2.0)

6.2 Volumetric Rendering

A more useful application is volumetric rendering, which computes the pixel color of a ray by integrating over transmittance and density. Given a ray vector $\mathbf{r}$ parameterized by distance t and viewing direction $\mathbf{d}$, volumetric rendering computes its final pixel color $C(\mathbf{r})$ by

$$C(\mathbf{r}) = \int_{t_n}^{t_f} T(t)\sigma(\mathbf{r}(t))\mathbf{c}(\mathbf{r}(t), \mathbf{d})dt, \quad T(t) = \exp(-\int_{t_n}^{t} \sigma(\mathbf{r}(s))ds), \tag{7}$$

where σ is the density and T is the transmittance. Unlike raytracing and rasterizing, volumetric rendering is inherently differentiable which enabled us to learn 3D shape with only 2D supervision.

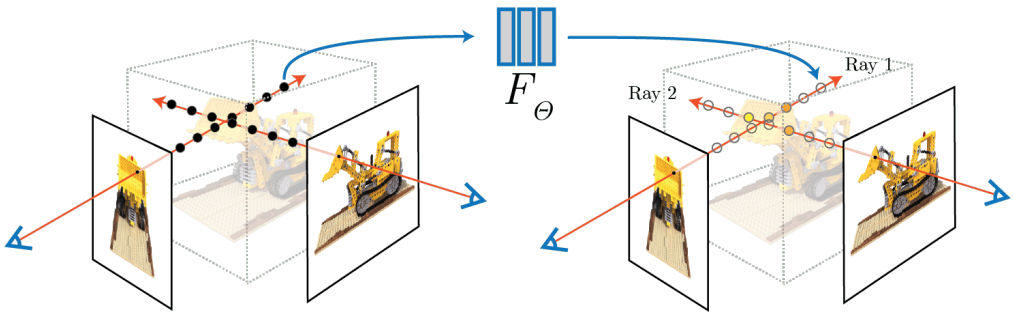
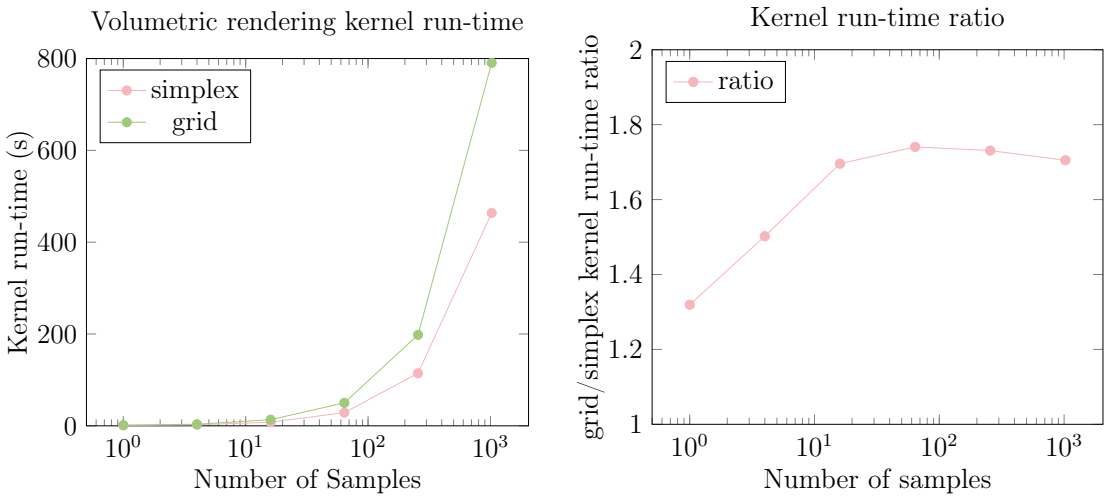


Figure 12: Illustration of rendering with neural radiance fields using volumetric rendering

This is first used for 3D reconstruction in NeRF [6], where the 3D scene is represented by a neural network. The network takes in 3D position and 2D direction and outputs the volume density of the particle at the position as well as RGB radiance at that position viewed from the given angle. To render an image, the network is

queried multiple times at discrete points along the ray and their density and color are obtained. Using a volumetric rendering equation, the samples are composited into the final ray color. Finally, the L2 loss is computed based on ground truth pixel color and through gradient-descent, the network learns this 3D scene.



(a) The comparison between the volumetric rendering performance shows that simplex-based structure is consistently faster

(b) The grid-to-simplex run-time ratio plateaus at 1.8 - approaches the theoretical upper bound at 3D

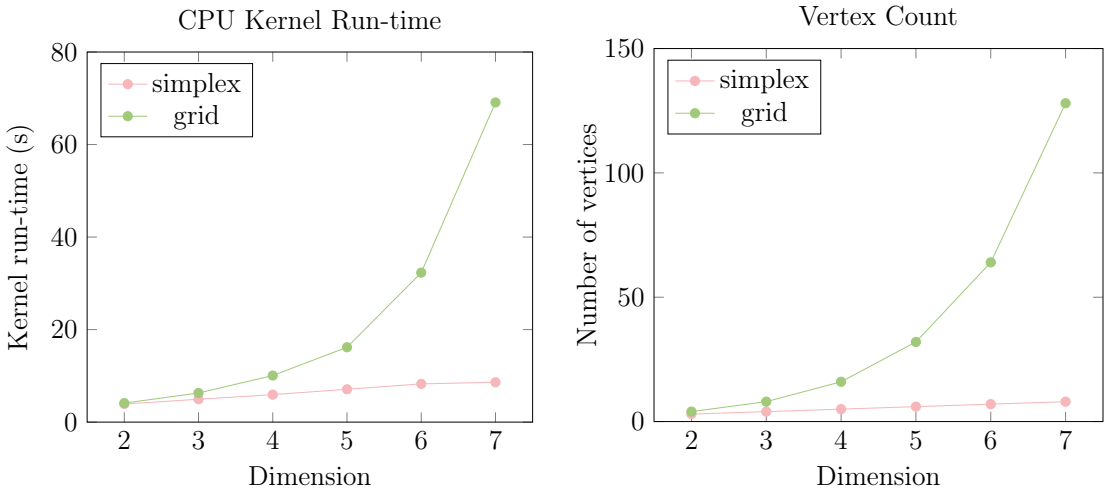
Figure 13: Analysis of memory usage for simplicial structures with different dimensions and levels.

Note that in the volumetric rendering equation, transmittance T in Equation 7 depends on previously sampled densities and cannot be evaluated in parallel. This makes our implementation extremely efficient because as the number of samples increases, the runtime of the entire algorithm would approach the theoretical bounds, which is a 2-time speed up in 3D (with 4 vertices in simplex vs 8 vertices in the grid). Through repetitive sampling, our computational overhead for sorting coordinates for each vertex in the Simplicial Subdivision phase (Refer to **Section 4.2**) becomes negligible. This effect is proven in Figure 13. This can allow faster training and rendering for NeRF without any loss in quality.

6.3 Kernel Analysis

To investigate the performance of our core implementation, we compare the kernel run time with baseline implementation on both CPU and GPU. For the CPU, we used an Intel Core i7-8700K CPU with 6 cores and 12 threads, running at 2.6 GHz with 16 GB of RAM (2019 Macbook Pro 16 inch). We implemented both the baseline method (grid implementation) and our proposed method (simplex hash implementation) on this CPU and measured the kernel run-time of both methods for various dimensional inputs. We used c++ for our implementation and measured the kernel run-time using the chrono module.

For each dimension, we use 2^{27} cells and randomly sample 2^{10} data points inside the n-dimensional structure. In order to produce a result in the seconds level, we perform the computation for each method 1000 times. Note that the side length of the grid is $\sqrt[n]{2^{27}}$. For a 3-dimensional input, for example, the side length is $\sqrt[3]{2^{27}} = 2^9$. For each dimension, we run the experiments 5 times to calculate the average of the kernel run-time for each method. The experiment result is summarized in Figure 14.



(a) Simplex run-time scales linearly with dimension, while grid run-time scales exponentially

(b) Simplex vertices scales linearly with dimension, while grid vertices scales exponentially

Figure 14: Analysis of memory usage for simplicial structures with different dimensions and different levels. Both graph exhibits the same pattern with dimension

According to the graph, the kernel run-time of simplices scales much better with dimension. The simplex-based encoding scales linearly because its number of vertices also scales linearly with dimension. On the other hand, the kernel run-time for grid-based structure scales exponentially - matches our observation on the exponential growth of the number of vertices with respect to dimension. This gives us a huge competitive advantage against grid-based encoding in high-dimensional tasks such as NeRF and SDF.

7 Discussion

7.1 Limitation

We did not observe the same speedup of our implementation over baseline methods on GPU as on CPU. One possible reason for this might be thread divergence caused by our sorting algorithm in the simplicial division. Although simplicial-based algorithms offer good scalability with dimension, the irregularity of n-simplices increases

with dimensionality. For instance, in two-dimensional space, we use equilateral triangles, while in three-dimensional space, we use isosceles tetrahedrons. However, in higher dimensions, the simplices used are far from regular, leading to a possible decrease in its ability to represent graphics primitives while persevering quality. Moreover, the coverage of the simplicial grid over the unit hypercube decreases exponentially with dimensionality, which is undesirable and requires more sophisticated handling compared to a grid-based structure.

7.2 Future work

Further investigations are needed to evaluate the performance boost of the simplex-based algorithm. In the paper, we didn't benchmark the backward kernel for our implementation and reason about its possible speedup effect on the overall training. While we also tried to keep our code changes to the original grid-based implementation to the minimum, more optimizations scheme on GPU can be devised and implemented for simplex algorithms.
Our simplex-based structure is orthogonal to almost all other popular encodings such as frequency encoding, and dense and sparse structures. For different tasks, suitable combinations can be chosen to further improve speed or memory usage.
For improving quality, we can add an additional offset vectors to each vertex and allow them to be learnable. We can also add random rotation at each level in the multiresolution implementation.

7.3 Conclusion

In this paper, we present a new approach for parameterizing graphics primitives using a simplex-based structure that offers significant advantages over traditional algorithms. By representing primitives as simplices, we are able to reduce the memory access and interpolation complexity, resulting in more efficient implementation. Through repetitive experiments and benchmarking, we show that our approach scales exceptionally well with dimensionality, making it particularly well-suited for tasks such as volumetric rendering. We believe that the simplicity, efficiency, and versatility of our approach make it an exciting avenue for future research in graphics primitives and beyond.

References

[1] Ang Cao and Justin Johnson. Hexplane: a fast representation for dynamic scenes. *arXiv preprint arXiv:2301.09632*, 2023.

[2] Eric R. Chan, Connor Z. Lin, Matthew A. Chan, Koki Nagano, Boxiao Pan, Shalini De Mello, Orazio Gallo, Leonidas Guibas, Jonathan Tremblay, Sameh Khamis, Tero Karras, and Gordon Wetzstein. Efficient geometry-aware 3D generative adversarial networks. In *arXiv*, 2021.

[3] Anpei Chen, Zexiang Xu, Andreas Geiger, Jingyi Yu, and Hao Su. Tensorf: Tensorial radiance fields. In *European Conference on Computer Vision (ECCV)*, 2022.

[4] Zhiqin Chen and Hao Zhang. Learning implicit fields for generative shape modeling. *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019.

[5] Sara Fridovich-Keil, Giacomo Meanti, Frederik Warburg, Benjamin Recht, and Angjoo Kanazawa. K-planes: Explicit radiance fields in space, time, and appearance. *arXiv preprint arXiv:2301.10241*, 2023.

[6] Peter Hedman, Pratul P. Srinivasan, Ben Mildenhall, Jonathan T. Barron, and Paul Debevec. Baking neural radiance fields for real-time view synthesis. *ICCV*, 2021.

[7] Stephen Lombardi, Tomas Simon, Jason Saragih, Gabriel Schwartz, Andreas Lehrmann, and Yaser Sheikh. Neural volumes: Learning dynamic renderable volumes from images. *ACM Trans. Graph.*, 38(4):65:1–65:14, July 2019.

[8] Julien N. P. Martel, David B. Lindell, Connor Z. Lin, Eric R. Chan, Marco Monteiro, and Gordon Wetzstein. Acorn: Adaptive coordinate networks for neural scene representation. 2021.

[9] Lars Mescheder, Michael Oechsle, Michael Niemeyer, Sebastian Nowozin, and Andreas Geiger. Occupancy networks: Learning 3d reconstruction in function space. In *Proceedings IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2019.

[10] Ben Mildenhall, Pratul P. Srinivasan, Matthew Tancik, Jonathan T. Barron, Ravi Ramamoorthi, and Ren Ng. Nerf: Representing scenes as neural radiance fields for view synthesis. In *ECCV*, 2020.

[11] Thomas Müller, Alex Evans, Christoph Schied, and Alexander Keller. Instant neural graphics primitives with a multiresolution hash encoding. *ACM Trans. Graph.*, 41(4):102:1–102:15, July 2022.

[12] Jeong Joon Park, Peter Florence, Julian Straub, Richard Newcombe, and Steven Lovegrove. Deepsdf: Learning continuous signed distance functions for shape representation. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[13] Ken Perlin. Chapter 2 noise hardware.

[14] Ken Perlin. An image synthesizer. *SIGGRAPH Comput. Graph.*, 19(3):287–296, jul 1985.

[15] Sara Fridovich-Keil and Alex Yu, Matthew Tancik, Qinhong Chen, Benjamin Recht, and Angjoo Kanazawa. Plenoxels: Radiance fields without neural networks. In *CVPR*, 2022.

[16] Cheng Sun, Min Sun, and Hwann-Tzong Chen. Direct voxel grid optimization: Super-fast convergence for radiance fields reconstruction. In *CVPR*, 2022.

[17] Delio Vicini, Sébastien Speierer, and Wenzel Jakob. Differentiable signed distance function rendering. *Transactions on Graphics (Proceedings of SIGGRAPH)*, 41(4):125:1–125:18, July 2022.

[18] Alex Yu, Ruilong Li, Matthew Tancik, Hao Li, Ren Ng, and Angjoo Kanazawa. PlenOctrees for real-time rendering of neural radiance fields. In *ICCV*, 2021.

8 Appendix

Appendix A. Mathematical proofs to simplex-based structures

Theorem 1. *$S = \{\boldsymbol{x} \in \mathbb{R}^n : 0 \leq x_1 \leq \cdots \leq x_n \leq 1\}$ is a n-simplex.*

Proof. Let M be an upper triangular $(n+1) \times (n+1)$ matrix with only 0 and 1. Then for any $\mathbf{x} \in S$, the solution to

$$\begin{bmatrix} 1 & \cdots & 1 & 1 \\ 0 & 1 & \cdots & 1 \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & 1 \end{bmatrix} \boldsymbol{\lambda} = \begin{bmatrix} 1 \\ x_n \\ \vdots \\ x_1 \end{bmatrix}, \quad \boldsymbol{\lambda} \in \mathbb{R}^{n+1}, \tag{8}$$

should be $\boldsymbol{\lambda} = [1 - x_n, x_n - x_{n-1}, \cdots, x_2 - x_1, x_1]^T$. Consider points $\mathbf{v}_1, \cdots, \mathbf{v}_{n+1}$, where the first $i+1$ entries of $\mathbf{v}_i$ are 1 and the rest are 0. Then M can also be expressed as $\begin{bmatrix} 1 & \cdots & 1 \\ \mathbf{v}_1 & \cdots & \mathbf{v}_{n+1} \end{bmatrix}$. The solution shows that $\mathbf{x}$ can be written as a linear combination of the $n+1$ points. Addi tionally, given that $0 \leq x_1 \leq \cdots \leq x_n \leq 1$, every entry of $\boldsymbol{\lambda}$ is no less than 0. With the addition constraint $\sum_i \lambda_i = 1$, we conclude that $\mathbf{x}$ can be written as a convex combination of the $n+1$ points and hence is inside the convex hull C of points $\mathbf{v}_1, \cdots, \mathbf{v}_{n+1}$. Therefore, we have $C \subseteq S$. Let $\mathbf{x}' \in C$. Therefore, there exists such $\boldsymbol{\lambda} \in \mathbb{R}^{n+1}$ s.t.

$$\begin{bmatrix} | & \cdots & | \\ \mathbf{v}_1 & \cdots & \mathbf{v}_{n+1} \\ | & \cdots & | \end{bmatrix} \boldsymbol{\lambda} = \mathbf{x}', \quad \sum_{i=1}^{n+1} \lambda_i = 1, \quad \lambda_i \geq 0 \text{ for all } i. \tag{9}$$

Then, we have $x'_j = \sum_{j+1}^{n+1} \lambda_j$, which indicates that $0 \leq x'_1 \leq \cdots \leq x'_n \leq 1$. Therefore, we have $\mathbf{x}' \in S$ and hence $S \subseteq C$.
In conclusion, $S = C$. Since the $n+1$ vertices $\mathbf{v}_1, \cdots, \mathbf{v}_{n+1}$ are affinely independent points, its convex hull C is a n-simplex and so is S. □

Remark 2. *Let π denote a permutation of $\{1, \cdots, n\}$, then $S_\pi = \{\boldsymbol{x} \in \mathbb{R}^n : 0 \leq x_{\pi(1)} \leq \cdots \leq x_{\pi(n)} \leq 1\}$ is a n-simplex. The n-simplex has $n+1$ vertices $\boldsymbol{v}_1, \cdots, \boldsymbol{v}_{n+1}$, where all entries at indices $\{\pi_1, \cdots, \pi_{i-1}\}$ of $\boldsymbol{v}_i$ are 1 and the rest are 0. Additionally, $\boldsymbol{v}_1$ is $\boldsymbol{0}^T$ and $\boldsymbol{v}_{n+1}$ is $\boldsymbol{1}^T$.*

Theorem 3. *All possible S_π are congruent.*

Proof. For any S_π, consider its $n+1$ vertices $\mathbf{v}_1, \cdots, \mathbf{v}_{n+1}$ as defined in Remark 2. Any two vertices of S_π is an edge and it has $\frac{n(n+1)}{2}$ edges with different lengths. Let d_k denote the distance between two difference vertices $\mathbf{v}_i$ and $\mathbf{v}_j$, where $i > j$ and $k = i - j$. As shown in Remark 2, consecutive vertices only differ by 1 at one entry, and we have $d_1 = \sqrt{1} = 1$. Similarly, vertices that differ by k in their order have k more (or less) 1 and $d_k = \sqrt{k}$. Therefore, for any S_π, it contains $n + 1 - k$ edges with length $\sqrt{k}$ where k is from 1 to n. Since all simplices have the same edges with each other, they are congruent regardless of the order of π. □

Lemma 4. *All possible S'_π in the transformed coordinate system are still congruent in the original coordinate system.*

Proof. We first obtain the vertex coordinates of S'_π in the original coordinate system using coordinate unskewing in Equation 4. For its vertex $\mathbf{v}'_i$, every entry is subtracted by $(i-1)G_n$. Using similar notation as above, $d'_1 = \sqrt{(1-G_n)^2 + (n-1)G_n^2}$. Similarly, $d'_k = \sqrt{k(1-kG_n)^2 + (n-k)(kG_n)^2}$. Therefore, for any S'_π, it contains $n+1-k$ edges with length d_k where k is from 1 to n. Since all transformed simplices still have the same edges with each other, they are congruent regardless of the order of π. □

Theorem 5. *A n-cube can be triangulated into $n!$ disjoint congruent simplices.*

Proof. Based on Remark 2, the hypercube $[\mathbf{0}, \mathbf{1}]^n$ fully contains all S_π where π ranges over all possible $n!$ permutations of $\{1, 2, ..., n\}$.
Assume, to the contrary, that two different simplices S_π, S_{π^*} with their two corresponding permutations π, π^* intersect each other. Then $\exists \mathbf{x} \in \mathbb{R}^n$, s.t. $\mathbf{x}$ is strictly in the interior of both S_π and S_{π^*}. By sorting the entries of $\mathbf{x}$, if the order satisfy both constraints from S_π, S_{π^*}, there must exist two entries with the same value. Therefore as the inequality constraints are not strictly satisfied, $\mathbf{x}$ has to be on the surface of both S_π, S_{π^*}. Contradiction.
Since there are $n!$ such permutations, there are $n!$ simplices with the disjoint interior contained by the hypercube. Together with Lemma 4, the hypercube can be triangulated into $n!$ disjoint congruent simplices. □

Appendix B. Python Demonstrations

```python
#1. coordinate skewing
input = [0.4,0.5,0.3]
print('Before coordinate skewing: ', input)

F = (math.sqrt(len(input) + 1) - 1) / len(input)
total = sum(input)

for i in range(len(input)):
    input[i] = input[i] + total * F

print('After coordinate skewing: ', input)


#2. simplicial division
vertices = []
weights = []

transformed_input = [[input[i], i] for i in range(len(input))]
transformed_input.sort(reverse=True)

print('After simplicial division: ', transformed_input)

start_input = [math.floor(input[i]) for i in range(len(input))]
start_weight = 1


#3. Barycentric interpolation
vertices.append(start_input.copy())
for i in range(len(transformed_input)):
    start_input[transformed_input[i][1]] += 1
    vertices.append(start_input.copy())

    weights.append(start_weight - transformed_input[i][0])
    start_weight = transformed_input[i][0]

weights.append(transformed_input[-1][0])

print('Calculated vertices in simplex coordiante: ', vertices)
print('Calculated weights: ', weights)
```

Figure 15: Python demo on simplex encoding

```
Before coordinate skewing:  [0.4, 0.5, 0.3]
After coordinate skewing:  [0.8, 0.9, 0.7]
After simplicial division:  [0.9, 0.8, 0.7]
Calculated vertices in simplex coordiante:
 [[0, 0, 0], [0, 1, 0], [1, 1, 0], [1, 1, 1]]
Calculated weights:  [0.1, 0.1, 0.1, 0.7]
```

Figure 16: Python output

Appendix C. Trilinear and barycentric interpolation

Algorithm 1 Trilinear Inpoterlation

```
1: function TRILINEAR INTERPOLATION(x, points)
2:     n ← length(points)
3:     if n = 1 then
4:         return points[0].value
5:     end if
6:     i ← 0
7:     while i < n − 1 and x > points[i + 1] do
8:         i ← i + 1
9:     end while
10:    t ← (x − points[i])/(points[i + 1] − points[i])
11:    y0 = TrilinearInterpolation(x, points[0 : i + 2])
12:    y1 = TrilinearInterpolation(x, points[i : n])   ▷ Continue calculation in n-1 dimension
13:    return y0 ∗ (1 − t) + y1 ∗ t
14: end function
```

Algorithm 2 Barycentric Interpolation

for i **do** in n+1 dimensions
  $F_n = \frac{\sqrt{n+1}-1}{n}$
  $x_i = x_i + 1_i^T \cdot F \sum_i x_i$ ▷ Coordinate Skewing
  start-weight = 1
  initialize w = $w_1, w_2, w_3, ...w_n$
  f = bubble-sort($f_1, f_2, f_3, ...f_n$)
  $\lfloor x \rfloor$ = Floored coordinates $x_1, x_2, x_3, ...x_n$
  if i = 0 **then**
    $coordinate_i$ = the index of largest coordinate in $\lfloor x \rfloor$
    w_i = start-weight - $coordinate_i$
    start-weight = f_1
  else
    $coordinate_i$ = the index of i-th largest coordinate in $\lfloor x \rfloor$
    $x_{coordinate_i}$ += 1
    w_i = start-weight - $coordinate_i$ ▷ Simplicial Subdivision
  end if
end for
g = 2 features for point **x**
x-feature = $w \cdot g$ ▷ Return feature for point x