Learning Based Infinite Terrain Generation with Level of Detailing

Aryamaan Jain\textsuperscript{1,2}, Avinash Sharma\textsuperscript{1,3}, K S Rajan\textsuperscript{1}
\textsuperscript{1} IIIT Hyderabad, India
\textsuperscript{2} Inria, Université Côte d’Azur, France
\textsuperscript{3} IIT Jodhpur, India
aryamaan.jain@inria.fr, avinashsharma@iitj.ac.in, rajan@iiit.ac.in

1. Introduction

Virtual world creation is essential for modern multi-media applications such as gaming, animation or AR/VR platforms. 3D Terrain modelling and rendering are at the core of generating large-scale realistic and immersive virtual worlds (as shown in Figure 1c) as it provides the foundational structure for the environment. Terrains define the physical surface features of the Earth’s crust, including mountains, valleys or plains and are popularly represented as a raster grid called Digital Elevation Models (DEM), where each cell in the grid represents a point on the surface and its elevation. Infinite terrain generation and rendering is a particular use-case finding application in games (such as Minecraft) or flight simulations.

Natural processes like erosion and weathering cause terrains to undergo various transformations, resulting in the development of intricate landscapes such as mountains, canyons, plateaus, and plains. Thus, a generated terrain needs to imitate these complex geometrical structures existing at multiple scales. This makes 3D terrain generation and authoring a challenging task. Existing techniques for infinite terrain generation involve procedural generation, which relies on mathematical algorithms to generate landscapes \cite{14, 44} but fail to model these processes. The recent surge of deep learning techniques has shown promise in generating more realistic terrain, as they can learn from real-world data to produce new, varied and realistic land-
scapes [13, 38, 63]. However, their use for generating infinite terrain is limited, where methods like [22] leverage procedural generation along with learning which reduces their realism. We bridge this gap by proposing a fully learning-based framework for generating infinite terrain.

In the context of rendering terrains, methods [5, 9, 40] use varying levels of detail to create real-time visualizations, particularly for real-time applications. Interestingly, terrain generation and rendering were typically attempted separately. Often, terrains were generated offline, followed by online rendering or even crude online procedural terrain generation algorithms (like Perlin noise) were used, which did not approximate real-world data very well [33, 35]. Our framework can generate realistic terrain with level-of-detailing learnt from real-world data and render it simultaneously in real-time.

In this paper, we present a framework to generate and render infinite terrain with level of detail (LOD) that is learnt from real-world data using deep learning methods. Our generative module can create terrain conditioned on their neighbourhood using image completion techniques like outpainting. Our enhancement module can refine the terrain progressively based on the LOD criterion. Furthermore, we propose a novel training strategy for terrain enhancement to incorporate quad-tree-based LOD. This also enables local and global context which minimizes errors along the terrain tile edges. These modules can be seamlessly integrated with quad-tree based rendering algorithms for real-time rendering, providing a realistic and unique user experience.

In summary, we propose a generative module for infinite realistic terrain generation, an enhancement module for progressive LOD and a seamlessly integrated quad-tree based rendering algorithm that is compatible with both the generative and enhancement modules for real-time rendering. We perform an elaborate quantitative and qualitative evaluation of the proposed framework. Our code is available at https://github.com/aryamaanjain/mr_terrain.

2. Related Work

Terrain Generation involves creating a surface mesh that accurately depicts the shape and elevation of the terrain and can broadly be categorized into procedural, simulation and example/learning based methods. Procedural generation involves algorithmic generation of landscapes that exploit self-similarity or fractal structure [34] of terrain. This includes noise functions like Perlin Noise [31, 42, 44, 45], Simplex Noise [14] or subdivision schemes [8, 30, 35] like the diamond-square algorithm [10, 11]. These algorithms are popular for infinite terrain generation but do not account for the physical processes that govern the formation of terrain thereby producing unrealistic landscapes. On the other hand, simulation-based methods [2, 37] use tools to simulate the natural process of erosion over a given terrain. They are typically used to simulate particular landforms such as fluvial V-shaped valleys [3, 49] or glacial U-shaped valleys [4]. They involve modelling the flow of water and sediment across the terrain and are grounded on physical laws. They are often slow and do not lend themselves to infinite terrain generation. In contrast, learning or example based methods [17, 64] use examples or sketches of terrains to generate new terrains. Deep-learning based solutions like GANs [12, 21, 36] have been leveraged for realistic and fast terrain generation [13, 59, 63]. Furthermore, VAEs [24] have also been employed for latent space manipulation of terrains [38]. However, application of learning-based techniques for infinite terrain generation remains sparse with a recent work [22] employing a combination of diffusion models [7, 18, 39, 51] and Perlin Noise for generating infinite terrain. Although [22] does improve upon other methods, diffusion models become a bottleneck for speed while their usage of Perlin Noise for consistency along edges reduces their realism.

Image Completion [20] techniques like inpainting [43, 57] and outpainting [58] have gained attention in recent years due to the advancements in deep learning techniques and form the basis of our infinite generation framework. Inpainting involves filling in missing parts of an image. It is a challenging task as the missing parts of the image can be irregular in shape and texture. On the other hand, outpainting involves generating new content beyond the boundaries of the input image. It can be used to generate high-resolution images from low-resolution inputs, as well as to create panoramic images from a single image.

Terrain Enhancement involves adding details to the terrain. Traditionally, this has been achieved through the use of simulation erosion models [37, 54] on top of a generated terrain, which can be time-consuming. Deep learning methods have become popular for terrain enhancement, with techniques such as [13] seeking to learn erosion features to accelerate the process. Terrain enhancement can also be framed as a terrain super-resolution problem, wherein a low-resolution terrain DEM is converted to a higher resolution. Methods such as [1, 25] employ ortho-photo and depth-map pairs to super-resolve low-resolution terrain tiles using attention feedback networks. Other methods, such as [22, 26] use only the depth map. In a broader perspective, there exists work on terrain enhancement incorporating style embeddings [63] as well as leveraging GANs [6, 56]. Although there is a wide array of techniques for enhancing terrains, they all process the terrain by breaking it into smaller tiles and then enhancing them separately. This causes an accumulation of errors along the edges [26], whereas our novel quad-tree based processing minimizes this issue.

Progressive super-resolution [27, 28, 53] is a type of im-
Figure 2. Overview of our framework: The terrain completion and enhancement modules are learnt from real-world datasets. The terrain completion module generates infinite terrain while the terrain enhancement module aids in level of detailing. The entire process is managed using a quad tree structure facilitating real-time rendering.

Figure 3. Illustration of the formulation and notations.

3. Method

An overview of our framework is given in Figure 2. The Terrain Completion Module (TCM) is trained to generate infinite terrain at a fixed resolution whereas the Terrain Enhancement Module (TEM) is trained to enhance the resolution of the terrain for LOD. We start with an initial fixed size DEM, which can either be taken from an existing dataset or generated via a terrain tile generation algorithm [13, 38]. We then employ the TCM to extend the initial DEM progressively based on the camera position enabling infinite generation. TCM generates terrain at the coarsest scale and needs to be enhanced for LOD based on criteria such as viewing position, direction and roughness of the terrain. The TEM is used to LOD the terrain. Finally, we use a quad-tree terrain rendering algorithm which is seamlessly integrated with the terrain completion and enhancement modules to render the terrain in real-time.

The notations used in the proceeding sections are illustrated in Figure 3. We denote the infinite terrain grid as $\mathcal{T}$. We get $s \times s$ tiles by indexing the grid as $\mathcal{T}_{i,j}$ where $i, j \in \mathbb{Z}$.

3.1. Terrain Completion Module

Initially, the grid $\mathcal{T}$ contains a single tile taken from an existing dataset or generated via a terrain tile generation algorithm. As the user moves along the grid, we populate $\mathcal{T}$ with tiles generated using the TCM within a radius $R$ from the user’s position. To generate an unavailable tile $c$, we access its available 8-neighbouring tiles $n$ and generate $c$ conditioned on $n$ using TCM. After generating $c$, we update $\mathcal{T}$ with the new tile. We loop similarly till the end.

An overview of the terrain completion module is given in Figure 4 and its inference is given in Algorithm 1.

Algorithm 1: Inference of TCM

| Input: $x, z$: Viewer position |
| $R$: Relevance radius |
| $\mathcal{T}$: Terrain grid |

| Data: $\mathcal{G}$: Generator |

| forall $(u, v) \mid ||(u, v)-(x, z)||_2 \leq R$ do |
| if not available($\mathcal{T}_{u,v}$) then |
| $n \leftarrow$ getNeigh($u,v,\mathcal{T}$) // get available neighbour tiles |
| $c \leftarrow \mathcal{G}(n)$ // predict center tile |

To train our model, we take a DEM tile $t$ and divide it into a $3 \times 3$ grid of tiles ($t_{11}, t_{12}, \ldots, t_{33}$). We then randomly mask the 8 boundary tiles to simulate a scenario of making them unavailable as we may encounter while running the system where $\mathcal{T}$ would be filled based on the trajectory of the user, producing arbitrary neighbourhoods. Subsequently, our objective is to predict the center of the grid $t_{22}$ given its neighbours. More specifically, our input consists of a stack of the DEM along with 3 masks which are
We optimize for L1 and GAN losses [21]. Denoting the discriminator available and 0 elsewhere.

1) predict mask: 1 for the \( t_{22} \) and 0 elsewhere, 2) available mask: 1 where \( t_{22} \) neighbours are available and 0 elsewhere and 3) unavailable mask: 1 where \( t_{22} \) neighbours are unavailable and 0 elsewhere.

We implement the generator \( G \) as a UNet [47] while the discriminator \( D \) is a sequential CNN followed by an MLP. We optimize for L1 and GAN losses [21]. Denoting the generated tile as \( \hat{t} \), we define the L1 loss as,

\[
\mathcal{L}_{L1}(G) = \| t_{22} - \hat{t}_{22} \|_1 \tag{1}
\]

Let the generated tile along with a neighbouring context be denoted by \( \hat{t}_{22} \) and ground truth tile along with context be denoted by \( t_{22} \). We define the GAN loss as,

\[
\mathcal{L}_{GAN}(G, D) = \min_G \max_D \log(D(t_{22}^c)) + \log(1-D(\hat{t}_{22}^c)) \tag{2}
\]

Taking an extended neighbouring context provides local and global coherence [20]. The final loss for the TCM is given by,

\[
\mathcal{L}_{TCM}(G, D) = \mathcal{L}_{GAN}(G, D) + \lambda_{TCM}\mathcal{L}_{L1}(G) \tag{3}
\]

\( \lambda_{TCM} \) is a hyper-parameter to adjust the contribution of the two terms.

**3.2. Terrain Enhancement Module**

Given a \( s \times s \) tile \( T_{i,j} \) at a coarse resolution, we progressively enhance it for LOD. More specifically, we divide our \( s \times s \) tile into four \( s/2 \times s/2 \) quadrant tiles and then enhance each quadrant tile to a \( s \times s \) tile separately. We repeat this process recursively, with each step enhancing the resolution by a factor of 2. This is further elaborated in **algorithm 2** and illustrated in **Figure 5**. The shouldEnhance function in **algorithm 2** depends on the roughness of the tile, intersection with view frustum, distance from the camera and availability of enhancement models beyond a certain depth in the recursion. We cache the output of TEM in a quad-tree data structure for faster inference. This technique allows us to enhance separate parts of the terrain with varying resolutions aiding in LOD.

**Algorithm 2: Recursive inference of TEM**

| Input: \( t \): Tile to render \( x, z \): Viewer position \( d \): Recursion depth |
|-----------------------------------------------|-----------------|-----------------|
| **Data:** \( E_0, E_1, E_2, E_3, E_4 \): Enhancement models |
| if shouldEnhance(\( t, x, z, d \)) then |
| \( \text{// LOD: divide & enhance quadrants of tile recursively} \) |
| TEM(\( E_d(t_{nw}) \), \( x, z, d+1 \)) \( \text{// north-east quadrant} \) |
| TEM(\( E_d(t_{ne}) \), \( x, z, d+1 \)) \( \text{// north-west quadrant} \) |
| TEM(\( E_d(t_{sw}) \), \( x, z, d+1 \)) \( \text{// south-west quadrant} \) |
| TEM(\( E_d(t_{se}) \), \( x, z, d+1 \)) \( \text{// south-east quadrant} \) |
| else |
| loadBuffer(\( t \)) renderTile(\( t \)) \( \text{// send tile to GPU} \) |

We describe the training strategy of TEM in **algorithm 3** where we employ separate models \( E_i \) for separate resolutions. Each \( E_i \) is implemented as a CNN with skip connections [15]. Given a low-resolution input tile, we divide it into its 4 quadrant tiles, randomly select one quadrant tile and enhance it with \( E_0 \). We then take the enhanced tile and repeat the same process with \( E_1 \). We iterate over this process up to a limit employing \( T \) models \( E_0, \ldots, E_T \) (\( T=4 \) in our experiments) to enhance by a factor of \( 2^{T+1} \). We use a sum of L1 losses to optimize the parameters of \( E_i \) as described in **algorithm 3**.

**3.3. Terrain Rendering**

For each frame of rendering the terrain, we start by processing the user input and updating our camera and shader parameters. We only process terrain tiles in \( T \) which lie within a radius \( R \) from the camera position. As the camera changes position, we invoke the TCM to fill in the missing terrain tiles. For each tile within a distance \( R \), we cull the tiles that are not in the view frustum of the camera.
Algorithm 3: Single step of TEM training

Data: $t_{32}$: Low resolution input tile (32m)  
$t_{16}, t_{8}, t_{4}, t_{2}, t_{1}$: High resolution ground truth tiles  
$E_0, E_1, E_2, E_3, E_4$: Enhancement models  

idx ← (32,16,8,4,2,1)  // will aid in indexing  
l ← 0.0  // loss  
for $i$ ← 0 to 4 do  
    $q$ ← selectQuad($t_{idx[i]}$)  // select random quadrant of tile  
    $q' ← features(q)$  // add features to the quadrant  
    $t_{idx[i+1]} ← E_i(q')$  // forward pass  
    $l ← l + loss(t_{idx[i+1]}, t_{idx[i+1]})$  // accumulate loss  
Adam($\nabla l$)  // backprop & step

Algorithm 4: Single frame render step

Input: camera  
TCM, TEM  
$R$: Relevance radius  
$T$: Terrain grid  

ip ← userInput()  // position, speed, etc.  
updateCamera(camera, ip)  
updateShaders(camera)  

$x, z ← camera_x, camera_z$  
$TCM(x, z, R, T)$  // algorithm 1: fill unavailable tiles  
forall $(u, v) | \| (u, v) - (x, z) \|_2 \leq R$ do  
    $t ← T_{u,v}$  // extract tile to render  
    TEM($t, x, z, 0$)  // algorithm 2: LOD & render

render the rest with LOD using the TEM. This process is described in algorithm 4.

Quad-tree based rendering makes optimizations like view-frustum culling very efficient (illustrated in Figure 1b), as we can discard the tiles that are not visible to the user for rendering via an inexpensive frustum-tile intersection check. Furthermore, the discrete nature of LOD in quad-tree based rendering facilitates seamless integration with super-resolution algorithms (Figure 1a), which would be less straightforward to accomplish using alternative terrain rendering algorithms.

4. Experiments and Results

Setup: We implemented our system in Python and used OpenGL/GLSL for rendering. Our models were trained using PyTorch on an NVIDIA GeForce RTX 2080 Ti GPU. To evaluate the real-time performance, we conducted experiments using a system with an NVIDIA GeForce RTX 3050 Laptop GPU, 12th Gen Intel Core i5 CPU (2.50 GHz) and 8GB RAM.

Dataset: The experiments employed the USGS National Map 3DEP Downloadable Data Collection dataset. This publicly accessible dataset comprises 100002 DEM tiles at a 1-meter resolution. We downloaded approximately 700 GB of data, which was then subjected to a data cleaning process resulting in a dataset size reduction to approximately 600 GB. The dataset was cropped to the nearest power of two, yielding a final dataset size of 81922 at 1-meter resolution.

4.1. Implementation Details

Terrain Completion Module: The generator of TCM is a UNet with a bottleneck of 1024 channels. The encoder layers consist of [4, 64, 128, 256, 512, 1024] channels, while the decoder layers contain [1024, 512, 256, 128, 64, 1] channels. To enhance training stability, batch normaliza-
tion was incorporated into the standard UNet architecture. The discriminator of TCM comprises a sequential CNN followed by an MLP. The discriminator consists of a head, a body, a tail and a classifier. The head converts the 4-channel input to 64 channels. The body maintains 64 channels across a series of 3 Conv-BatchNorm-LeakyReLU-Conv-Maxpool layers. Finally, the tail reduces the number of channels to 1 and passes flattened features to a two-layer MLP. We employed the Adam optimizer [23] with a learning rate (LR) of 0.0002 for the generator and 0.0001 for the discriminator, setting the parameters \(\beta_1\) and \(\beta_2\) to 0.5 and 0.999, respectively. An LR scheduler was employed to reduce the LR by 25% every four epochs. We trained the model using a batch size of 8 for 256 epochs. We set the \(\lambda_{TCM}\) value to 10.0 in Equation 3.

Terrain Enhancement Module: TEM is composed of a set of 5 CNN models for enhancing 2m, 4m, 8m, 16m and 32m DEMs. Each model contains 47,497 trainable parameters to ensure real-time performance during inference. A single model in the TEM is comprised of three main components: the head, body, and tail, similar to the TCM. The head component is a convolutional layer that takes a 4-channel input, consisting of a DEM to enhance and 3 additional features and outputs 32 channels. The three additional channels in the input include the gradient along the x-axis, the gradient along the z-axis and the L1 norm of the gradient. The body component is composed of 4 Conv-ReLU layers, which maintain 32 channels. The tail component includes a Pixel-Shuffle layer [50] to increase the resolution by a factor of two, followed by a convolutional layer. A skip connection connects the head and tail components and the output is a residual that is added to the bicubic-upsampled input. The model is trained for 128 epochs with a batch size of 32, utilizing the Adam optimizer with an LR of 0.001 and a scheduler that reduces the LR by 20% every two epochs.

4.2. Terrain Completion

Comparative Assessment: We adopt the Fréchet Inception Distance (FID) metric [16] to quantitatively evaluate the TCM and present the corresponding results in Table 1. Our analysis involves a comparison with methods generating infinite terrains, encompassing Procedural and Kernel Blending approaches. Procedural algorithms such as Perlin, Simplex, and Ridge Noise functions lack adherence to geological laws or learning from real-world DEMs, resulting in comparatively higher FID. Furthermore, we illustrate Perlin and Ridge Noise terrain on the top row of Figure 6 where we observe their lack of distinctive features such as realistic valleys and ridges. Kernel Blending [22] represents a hybridization of learning-based and procedural techniques. While performing better than Procedural methods, their effectiveness is constrained due to their reliance on procedural techniques to ensure continuity along tile edges. As depicted in Figure 6, Kernel Blending approaches demonstrate a degree of realism towards the center but suffer from procedural technique influence towards the edges. TCM with 31 million parameters achieves significantly lower FID scores, being fully learning-based. An important insight is that, unlike other techniques where tile edges are approximately at mean elevation, such a constraint does not hold for the TCM as seen in Figure 6. We do not compare against simulation-based methods since they do not lend themselves to infinite generation.

Ablation Study: We conduct an ablation study to justify the choice of our parameters, as shown in Table 1. Our findings reveal a decreasing trend in FID as the ratio of the number of parameters of the Generator \(G\) and Discriminator \(D\). The number of channels in \(G\)'s bottleneck (U-Net) is represented as \(G_{width}\).

<table>
<thead>
<tr>
<th>Method</th>
<th>FID ↓</th>
<th>(G_{width})</th>
<th>(#G / #D)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perlin Noise [45]</td>
<td>489.79</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Simplex Noise [14]</td>
<td>529.171</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Ridge Noise</td>
<td>530.373</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Kernel Blend [22]</td>
<td>274.749</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>TCM2M</td>
<td>386.024</td>
<td>256</td>
<td>0.388</td>
</tr>
<tr>
<td>TCM2M</td>
<td>215.854</td>
<td>256</td>
<td>1.536</td>
</tr>
<tr>
<td>TCM8M</td>
<td>148.892</td>
<td>512</td>
<td>31.334</td>
</tr>
<tr>
<td>TCM31M</td>
<td>114.376</td>
<td>1024</td>
<td>126.294</td>
</tr>
</tbody>
</table>

Table 1. Quantitative evaluation of infinite terrain generation techniques using the FID ↓ [16] metric. The FID increases with the ratio of the number of parameters of the Generator \(G\) and Discriminator \(D\). The number of channels in \(G\)'s bottleneck (U-Net) is represented as \(G_{width}\).
features and body size. We cap the maximum size of the generator to 31 million parameters as networks beyond this size inhibit real-time inference.

**User Study:** For subjective evaluation, we conducted a user study involving 16 participants with varying levels of experience in terrain analysis. To evaluate the effectiveness of our approach, we employed a first-preference experiment methodology. Each participant was presented with 5 pairs of terrain tiles, where each pair consisted of a terrain generated using Perlin noise and TCM, randomly ordered. The participants expressed a clear preference for the terrain tile generated with TCM in 93.75% of the cases, highlighting the superior performance of our method. We have included a more detailed discussion of the user study in the supplementary material.

### 4.3. Terrain Enhancement

**Comparative Assessment:** We present a comparison of terrain enhancement based on RMSE and PSNR metrics in Table 2. AFND [25] and TRCAN [22] are tile-based super-resolution methods, *i.e.*, they enhance the entire tile for a particular factor at once, whereas our quad-tree based method progressively enhances the quads of a terrain up to a factor. AFND and TRCAN were originally designed for enhancement factors of 8, but we have trained them for factors up to 32. Our experimental results demonstrate that the TEM outperforms AFND and TRCAN for enhancement factors greater than 2 highlighting the effectiveness of the TEM for high enhancement factors.

**Ablation Study:** We report the results on ablation tests for TEM in Table 2. Our final model is denoted as TEM$_{47K}$. In TEM$_{dem}$, we pass the DEM as input without any features, which leads to lower scores. In TEM$_{prop}$, we back-propagate with only the last model’s loss without any direct supervision for other models. Specifically, we use the loss term $\|t_1 - t_1\|_1$, where $t_1$ and $t_1$ are the predicted tile and ground truth tiles at 1m, instead of the summation of L1 losses for all resolutions as given in algorithm 3. Despite the gradient flow through all the models, we observe poor performance for them. Interestingly, we not only observe a performance drop for factors lower than $\times 32$ but also find that the performance is not the best for $\times 32$ enhancement factor. These observations further support the potential of a progressive architecture for terrain enhancement.

**Tile Edge Errors:** Tile-based methods [22, 25] are known to exhibit high errors along tile boundaries, especially for high enhancement factors such as $\times 32$. This leads to a degradation in metrics like RMSE/PSNR. As illustrated in Figure 8, our proposed quad-tree based processing manages to subdue this effect. We enhance a 32m DEM to 1m using tiles-based and quad-tree based methods, compute their residual error with respect to the ground truth and plot them. We can observe particularly higher errors along the edges for the tile-based method.

**Memory Constraints:** One key difference between the methods prevalent in the literature of natural image super-resolution and the problem we are tackling is scale. While super-resolution on natural images has mainly been attempted for factors of up to $\times 8$ [29, 62], for terrains as in our case, the factor can be as high as $\times 32$. This makes
<table>
<thead>
<tr>
<th>Method</th>
<th>×2 (16m)</th>
<th>×4 (8m)</th>
<th>×8 (4m)</th>
<th>×16 (2m)</th>
<th>×32 (1m)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RMSE↓</td>
<td>PSNR↑</td>
<td>RMSE↓</td>
<td>PSNR↑</td>
<td>RMSE↓</td>
</tr>
<tr>
<td>Bicubic</td>
<td>0.587</td>
<td>39.225</td>
<td>0.545</td>
<td>39.801</td>
<td>0.553</td>
</tr>
<tr>
<td>AFND [25]</td>
<td>0.451</td>
<td>40.783</td>
<td>0.459</td>
<td>40.662</td>
<td>0.489</td>
</tr>
<tr>
<td>TRCAN [22]</td>
<td>0.368</td>
<td>41.937</td>
<td>0.408</td>
<td>41.124</td>
<td>0.436</td>
</tr>
<tr>
<td>TEMdem</td>
<td>0.463</td>
<td>40.37</td>
<td>0.423</td>
<td>41.045</td>
<td>0.413</td>
</tr>
<tr>
<td>TEM_{prop}</td>
<td>1.489</td>
<td>27.774</td>
<td>0.862</td>
<td>33.006</td>
<td>0.743</td>
</tr>
<tr>
<td>TEM_{2K}</td>
<td>0.417</td>
<td>40.817</td>
<td>0.395</td>
<td>41.495</td>
<td>0.384</td>
</tr>
<tr>
<td>TEM_{22K}</td>
<td>0.414</td>
<td>40.884</td>
<td>0.394</td>
<td>41.548</td>
<td>0.381</td>
</tr>
</tbody>
</table>

Table 2. Quantitative evaluation of terrain enhancement using the RMSE↓ (in meter) and PSNR↑ (in dB) metrics for upsampling factors from ×2 to ×32.

the direct application of methods from natural image super-resolution literature to terrains non-trivial. For example, as in our case, the input DEM is a 256×256 raster grid. Directly super-resolving it by a factor of 32 will give an output of dimensions 8192×8192, which in most cases would be intractable for training due to the memory constraints of the GPU VRAM. Our proposed method is similar to progressive super-resolution, but rather than processing the entire tile, we process quadrants of the tile. This ensures that the enhancement module receives input of the same dimension at each stage unlike [27, 28, 53], where the dimensions of the input increases at successive stages limiting their scalability for higher enhancement factors due to memory constraints. Our formulation has the added advantage of being compatible with the quad-tree-based terrain rendering algorithms which have been researched quite extensively.

4.4. Rendering Details

Our framework uses OpenGL/GLSL for rendering. Our system is written in Python, enabling seamless integration with PyTorch. In Figure 1a, the LOD is showcased through a mesh representation. To optimize performance, we employ quad-trees for view frustum culling, as depicted in Figure 1b from an overhead perspective. The normals are calculated efficiently using gradients rather than cross-products and objects like trees are added using geometric instancing. We showcase a rendered scene generated and processed within our framework using Terragen [46] in Figure 1c.

In the absence of batching, TCM computes 1 inference in 0.032s on the GPU and 0.344s on the CPU. Similarly, TEM computes 1 inference in 0.0019s on the GPU and 0.017s on the CPU. In addition, we profiled the system performance by traversing a random path and recording the frames per second (FPS) at different points as shown in Figure 9. We observed occasional drops in FPS below 100 at specific points along the path, as indicated by red circles, corresponding to TCM and TEM inferences.

Figure 9. FPS (bottom) observed along a randomly traversed path (top). The red circles indicate the FPS dropping below 100.

5. Conclusion

We introduced a framework for learning-based infinite terrain generation and level-of-detailing compatible seamlessly with quad-tree based terrain rendering. The generative module enables the creation of vast and diverse landscapes, while the enhancement module ensures that the terrain is rendered at the appropriate level of detail. The seamless rendering algorithm provides a visually appealing and immersive experience for users. Overall, our framework offers a powerful toolset for game developers, virtual reality enthusiasts, and other applications that require high-quality large-scale terrains. Furthermore, the quad-tree based enhancement module may be extended to other domains where the raster sizes are typically large, such as the medical domain due to our memory-efficient processing. Another possible direction of exploration is the integration of other rendering algorithms or enhancing the framework’s control capabilities.
References


computer vision and pattern recognition, pages 4681–4690, 2017. 7
[34] Benoit B Mandelbrot and John W Van Ness. Fractional brownian motions, fractional noises and applications. SIAM review, 10(4):422–437, 1968. 2
[40] Renato Pajarola. Overview of quadtree-based terrain triangulation and visualization. 2002. 2, 3
[46] Planetside. Terragen 4, 2023. 8
[60] Liwei Zhang, Jiachen Zhou, Juzhong Tuo, Biao Wang, and Yuchang Sun. A multilevel terrain rendering method based


