GPU-Accelerated Procedural Terrain and Voxel Worlds: From Heightmaps to Work-Graph Pipelines

Hritvik JV
Department of Computer and Information Science
University of Oregon

***Abstract*—Procedural terrain and voxel worlds underpin many modern games and simulations, but generating and rendering massive, richly detailed landscapes in real time is only practical with GPU acceleration and parallel algorithms. This survey reviews key developments in GPU-accelerated procedural terrain, starting from early heightmap techniques such as geometry clipmaps and GPU noise evaluation, through fully volumetric voxel terrain with Marching Cubes and Transvoxel, to recent work on out-of-core gigavoxel paging and GPU work-graph based pipelines. We organize this paper around three main layers of the pipeline: parallel generation of scalar fields (noise-based and procedural content), GPU-accelerated surface extraction and level-of-detail (LOD) stitching, and streaming/virtualization techniques that enable worlds far larger than GPU memory. Throughout, we highlight common parallel patterns (data-parallel kernels, hierarchical level-of-detail structures, and task graphs), discuss trade-offs between quality, editability, and performance, and identify open challenges for future high-performance terrain engines.**

***Index Terms*—GPU computing, procedural terrain, voxel rendering, Marching Cubes, level of detail, work graphs, parallel algorithms**

I. Introduction

Open-world games and interactive simulations increasingly rely on procedural techniques to generate terrain and geometry on the fly. Rather than storing a fixed heightmap or mesh, developers define a compact procedural model and evaluate it at runtime to synthesize virtually unbounded landscapes. This approach reduces authoring effort, enables destruction and deformation, and supports infinite or very large virtual worlds. However, it also moves substantial computation into the inner rendering loop: every frame, the engine must generate or update terrain geometry at interactive rates.

Traditional CPU-based terrain generation struggles to meet these demands for large, high-resolution environments. Evaluating noise functions, combining multiple layers of terrain detail, and generating meshes for hundreds or thousands of tiles quickly saturates a single CPU core. Meanwhile, modern GPUs offer massive data parallelism and high memory bandwidth, making them a natural fit for the highly regular, grid-based computations involved in terrain and voxel rendering. Over the last two decades, this has motivated a progression from simple GPU-assisted heightmap rendering to fully GPU-driven pipelines for volumetric terrain, including level-of-detail (LOD), streaming, and advanced task scheduling.

This paper surveys that progression, with a focus on *GPU-accelerated procedural terrain and voxel worlds*. We start from early heightmap-based methods such as geometry clipmaps, which moved most of the terrain rendering work from CPU to GPU [1]. We then discuss GPU-based noise evaluation and procedural heightfield generation [2]–[4], followed by volumetric terrain approaches that operate on three-dimensional scalar fields and extract surfaces via Marching Cubes and related algorithms [6]–[8]. Building on this, we review techniques for LOD and crack-free stitching such as the Transvoxel algorithm [9], and systems that support dynamic, editable voxel terrain [10].

Modern engines also need to handle worlds that far exceed GPU memory capacity. This has led to out-of-core and paged voxel terrain systems that treat GPU memory as a cache over a much larger virtual volume [11]. Finally, we highlight recent work on GPU work graphs [12], which generalize data-parallel kernels into task graphs that can be generated and scheduled entirely on the GPU, opening new possibilities for fully GPU-driven procedural pipelines.

The contributions of this paper are threefold:

- We provide a structured overview of major techniques for GPU-accelerated procedural terrain, from early heightmaps to volumetric voxel systems.
- We analyze common parallel patterns and data structures (grids, clipmaps, octrees, chunk graphs) that recur across these techniques.
- We discuss open challenges and future directions for high-performance terrain engines, particularly in the context of emerging GPU work-graph APIs.

We begin by analyzing the computational patterns of heightmap rendering, which serve as a simpler 2.5D precursor to full volumetric solutions. From there, we examine the increased memory and bandwidth demands of 3D scalar fields, the necessity of sparse acceleration structures, and finally, the shift toward GPU-driven task scheduling

II. Motivation

Open-world procedural generation faces a constant tension between geometric detail and real-time performance. Traditional approaches, as explored in Section IV (Heightmaps), offer efficiency but fundamentally restrict structural complexity due to their 2.5D nature. Conversely, fully volumetric systems,

while enabling rich features like overhangs and dynamic destruction, introduce prohibitive memory and compute costs, often exceeding the capacity of a single GPU.

Our primary motivation stems from the need to balance these factors in modern, interactive environments. To address the scalability challenges inherent in dynamic volumetric terrain, the procedural pipeline must be designed to prioritize parallel patterns, hierarchical Level-of-Detail (LOD), and efficient memory virtualization. This survey is therefore scoped to evaluate techniques—from early clipmaps to modern work-graph scheduling—that explicitly manage these High-Performance Computing (HPC) constraints on the GPU.

III. Background

A. Procedural Terrain and Voxel Worlds

Procedural terrain systems represent landscapes implicitly via functions rather than explicit meshes. A common abstraction is a scalar field

$$f(\mathbf{x}) = h(x, z) \quad \text{or} \quad f(\mathbf{x}) = d(x, y, z), \tag{1}$$

where h is a height function defined over the x–z plane, and d is a volumetric density or signed-distance field over 3D space. Rendering then becomes a matter of sampling f on a grid and extracting either a heightfield mesh or an isosurface.

Procedural models typically combine multiple building blocks [5]:

- Gradient noise (e.g., Perlin, simplex) and fractal Brownian motion for broad landforms.
- Domain warping and ridge functions for sharper features (ridges, valleys).
- Erosion or hydraulic simulation for more realistic terrain, at higher cost.
- Feature-specific primitives (plateaus, craters, caves) combined via constructive solid geometry or blending.

Voxel worlds extend this idea from 2.5D heightmaps to full 3D, representing the environment as a regular grid of volumetric cells (voxels). A voxel can store material ID, density, or other attributes. Volumetric terrain supports overhangs, caves, and destructibility in all directions, at the cost of increased memory and computation.

B. GPU Computing for Terrain

The execution model of modern GPUs is well suited for grid-based terrain computations. Thousands of threads execute the same program in parallel, each operating on independent data elements. The two primary interfaces are:

- *Graphics pipelines* (vertex, geometry, fragment shaders), used in early work to displace vertices or sample textures for heightmaps.
- *Compute APIs* (CUDA, OpenCL, DirectCompute, modern compute shaders), which expose general-purpose kernels over thread blocks and grids.

Procedural terrain workloads are largely data-parallel: each grid cell or voxel can be processed independently, with only local dependencies. This makes them ideal for single-instruction multiple-thread (SIMT) architectures. However, terrain systems also require:

- Efficient memory layouts for large grids or sparse structures.
- Hierarchical level-of-detail (LOD) structures (e.g., quadtrees, clipmaps, octrees).
- Streaming and paging to handle worlds larger than GPU memory.

In this context, we can view a terrain engine as a pipeline with three main stages:

1) **Scalar field evaluation**: compute height or density values on a grid, typically using noise and procedural functions.
2) **Surface extraction / meshing**: convert scalar samples into a renderable mesh (heightfield grid or isosurface).
3) **Streaming and LOD**: manage which portions of the world are resident at which resolutions, and handle transitions.

The following sections discuss each stage in more detail, organized chronologically.

IV. GPU-Accelerated Heightmap Terrain

Early GPU work on terrain focused on heightmaps, where each terrain point is defined by a single scalar height value. Heightmaps are simpler than full volumetric data and align well with traditional rasterization pipelines.

A. Geometry Clipmaps

A key milestone was the introduction of geometry clipmaps by Asirvatham and Hoppe [1], part of the GPU Gems 2 collection. Geometry clipmaps adapt the clipmap concept from texture mapping to terrain geometry. The terrain is represented as a set of nested, toroidal grids at successively lower resolutions centered around the viewer. Each level covers a larger area at lower resolution, allowing a vast terrain (tens of billions of samples) to be rendered while only a modest amount of geometry is stored on the GPU at any time.

The method exploits GPU vertex texture fetches to sample a heightmap stored in textures. All vertex positions are computed on the GPU from a small canonical mesh, with only the heightmap samples streamed or generated as needed. This shifts much of the workload from the CPU to the GPU: the CPU is mainly responsible for updating heightmap tiles and clipmap centers, while the GPU performs the heavy per-vertex computations.

In terms of parallelism, geometry clipmaps are highly data-parallel: each vertex fetches height data and computes its position independently. The main complexity lies in constructing the clipmap meshes so that transitions between levels are seamless and do not introduce cracks.

B. Procedural Heightfields on the GPU

While geometry clipmaps assume an existing heightmap, subsequent work focused on generating heightmaps procedurally on the GPU. Geiss [2] presented a system for generating

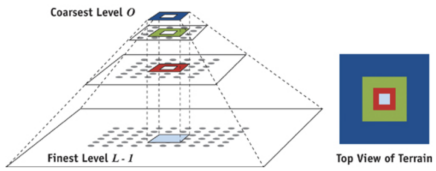

Figure 2-1 How Geometry Clipmaps Work

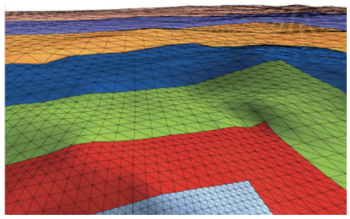
Figure 2-2 Terrain Rendering Using a Coarse Geometry Clipmap

Fig. 1: The Geometry Clipmap structure. Terrain is partitioned into nested, multi-resolution rings centered on the viewer. Only the innermost ring (L0) is fully detailed, while coarser rings (L1, L2, etc.) cover larger areas at lower resolution, demonstrating early use of the GPU to manage LOD and streaming [1].

complex procedural terrains using the GPU. Rather than storing a precomputed heightmap, the terrain is defined by a combination of noise functions, ridged multifractals, and other procedural primitives evaluated entirely on the GPU. This allows for infinite, non-repeating terrain and enables interactive adjustments to procedural parameters.

Geiss's chapter uses GPU fragment shaders and render-to-texture passes to evaluate noise at various scales, combine them, and produce heightfields and auxiliary maps (e.g., slope, masks). The emergence of this technique aligns with the ALU operations becoming significantly cheaper than texture fetches. The approach exploits the fact that each pixel (height sample) can be computed independently, making it well suited to fragment shaders and later compute shaders. The work also hints at volumetric extensions, treating the terrain as a 3D scalar field.

Li et al. [3] took a similar idea into the CUDA programming model. They described a parallel algorithm using Perlin noise superposition for terrain generation on CUDA. Their method assigns each heightmap sample to a separate CUDA thread, which evaluates multiple octaves of Perlin noise and combines them. The paper compares CPU and GPU implementations and shows substantial speedups for large terrain grids. It also discusses issues such as memory access patterns and the cost of Perlin noise evaluation.

Backes et al. [4] examined real-time massive terrain generation using procedural content. Their system uses the GPU both to synthesize heightmap tiles (via procedural noise and erosion-inspired filters) and to render them, combined with instancing for vegetation and other objects. The paper emphasizes parameterization and interactivity while maintaining real-time performance.

C. Hardware Tessellation and Displacement

While geometry clipmaps managed LOD via CPU-side grid updates, the introduction of hardware tessellation in DirectX 11 and OpenGL 4 shifted this responsibility entirely to the GPU graphics pipeline. In this model, the CPU submits coarse patch geometry (e.g., a low-resolution quad tree). The GPU's Hull Shader (or Tessellation Control Shader) calculates tessellation factors per edge based on camera distance, and the fixed-function tessellator subdivides the geometry.

Finally, the Domain Shader (or Tessellation Evaluation Shader) samples the heightmap texture to displace the newly generated vertices [14]. This approach allows for continuous, crack-free LOD without the complex CPU-side buffer management required by clipmaps. However, it introduces new HPC challenges regarding thread divergence, as adjacent patches may require vastly different levels of subdivision, leading to load imbalances in the shader pipeline.

D. Parallel Patterns in Heightmap Terrain

Heightmap-based terrain on the GPU tends to share several parallel patterns:

- **Embarrassingly parallel sampling**: each height value is computed independently, mapping directly to threads or fragments.
- **Multi-pass composition**: different noise layers or filters are often applied in separate passes, writing intermediate results to textures.
- **Tiled streaming**: large heightmaps are split into tiles, and only tiles near the camera are updated or streamed.

These techniques demonstrate that GPUs can handle both the generation and rendering of heightfields for large terrains. However, heightmaps cannot represent caves, overhangs, or fully 3D destructible environments, which motivated a shift toward volumetric representations.

V. Volumetric Voxel Terrain and Surface Extraction

Volumetric terrain represents the world as a three-dimensional scalar field, typically sampled on a regular grid of voxels. This enables features like caves and arches, but also increases memory use and computational cost. GPU acceleration is therefore even more critical.

A. Scalar Fields and Volumetric Terrain

In a volumetric terrain system, the terrain is defined by a function $d(x, y, z)$, where the surface is an isosurface (commonly $d = 0$). Positive and negative values correspond to different materials (e.g., solid vs. empty). The scalar field can be defined procedurally via noise and constructive solid geometry, loaded from simulation data, or derived from signed-distance primitives.

Several works explore volumetric terrain generation on the GPU. Engström [7] investigated volumetric terrain generation with Marching Cubes on the GPU, focusing on extracting meshes from large scalar fields in real time. Lengyel's dissertation [8] provides a comprehensive framework for voxel-based

terrain, combining procedural generation, LOD, and dynamic edits.

B. Marching Cubes on the GPU

The Marching Cubes algorithm is a classic method for extracting an isosurface from a scalar field. It considers each cell of the grid, determines the pattern of inside/outside values at cell corners, and consults a precomputed lookup table to generate triangles. The algorithm is highly parallel, as each cell can be processed independently given access to its eight corner values.

Cirne and Pedrini [6] presented a methodology for accelerating Marching Cubes on GPUs. They described a pipeline where:

1) A classification kernel determines which cells intersect the isosurface and computes the number of generated triangles per cell.
2) A prefix-sum (scan) computes offsets into a global vertex buffer.
3) A generation kernel writes triangle vertices using the lookup tables.

To improve performance, they explored auxiliary spatial data structures (such as trees) to focus computation on regions where the surface exists.

Their results show significant speedups compared to CPU implementations, especially for large volumetric datasets. This pattern of classification–scan–generation is common in GPU isosurface extraction and is applicable to terrain as well as medical or scientific data.

While Cirne and Pedrini optimized the pipeline, Marching Cubes is famously difficult to load-balance becuase the number of triangles outputted per voxel varies(0-5 tris). This created an SIMT inefficiency

Andersson [13] studied hybrid CPU/GPU strategies for Marching Cubes, analyzing when it is beneficial to offload work to the GPU and how to partition the volume. Hybrid approaches can be useful when CPU resources are otherwise idle or when GPU memory is constrained.

C. Preserving Sharp Features: Dual Contouring

A major limitation of Marching Cubes is its inability to reproduce sharp features; because vertices are placed via linear interpolation along edges, the resulting mesh always appears smoothed or ”blobby.” To address this, Ju et al. proposed Dual Contouring (DC) [15]. Unlike Marching Cubes, which generates polygons based on edge intersections, Dual Contouring generates a vertex inside each voxel that exhibits a sign change.

DC utilizes Hermite data (exact surface normals at intersection points) to solve a Quadratic Error Function (QEF). This minimizes the error of the vertex position relative to the surface tangents, allowing the reconstruction of sharp edges and corners. (Note: While Ju et al. theoretically solve the sharp feature problem, implementing a numerically stable QEF solver within the constraints of a compute shader remains a significant barrier to adoption compared to the simple lookup tables of Marching Cubes.)

TABLE I: Comparison of Isosurface Extraction Algorithms

Feature	**Marching Cubes (MC)**	**Dual Contouring (DC)**
Parallel Pattern	Table Lookup, Highly Data-Parallel	QEF Solver (Linear System)
Vertex Placement	Linear Interpolation on Edges	Optimal (Minimizing Quadratic Error)
Feature Preservation	Poor (Smooth Mesh)	Excellent (Sharp Corners and Edges)
GPU Efficiency	High. Fast, Constant-Time	Low. Requires Atomic Ops, Divergent
Crack-Free LOD	Requires Custom Transition Cells	Inherently Simpler LOD Structure

From an HPC perspective, DC is significantly more expensive to parallelize than Marching Cubes. While Marching Cubes relies on constant-time table lookups, DC requires solving a small linear system (typically via Singular Value Decomposition or QR decomposition) for every active voxel. Implementing robust, numerically stable QEF solvers within GPU kernels remains a complex challenge, often requiring hybrid approaches where the GPU identifies active cells and the CPU or a specialized compute shader solves the QEFs for complex geometry.

D. Voxel Terrain and LOD: Lengyel's Framework

Lengyel's dissertation [8] laid much of the groundwork for real-time voxel terrain in games. He proposed a voxel-based terrain system with several key components:

- A modified Marching Cubes algorithm that eliminates certain ambiguities to simplify implementation and improve consistency.
- A multi-resolution representation of voxel data, with LOD levels aligned to powers of two.
- A crack-free LOD transition scheme, later generalized into the Transvoxel algorithm.
- Support for dynamic edits by updating voxel data and regenerating affected regions.

The system divides the world into blocks or chunks of voxels, each meshed independently using Marching Cubes. LOD is achieved by varying the voxel resolution per chunk depending on distance from the viewer. To avoid cracks between chunks with different resolutions, specialized transition meshes are generated at boundaries.

Although Lengyel's dissertation is not exclusively GPU-focused, many of its ideas map naturally onto GPU implementations. Mesh extraction per chunk can be performed in parallel, and LOD transitions can be handled using precomputed lookup tables similar to Marching Cubes.

E. The Transvoxel Algorithm

Lengyel later published the Transvoxel algorithm [9], which formalizes the crack-free stitching between meshes at different resolutions. The algorithm distinguishes between:

- *Regular cells*, handled by a modified Marching Cubes variant.
- *Transition cells*, which occur along boundaries between LOD levels and require special cases.

For transition cells, the algorithm defines an extended set of patterns and lookup tables that generate triangles aligned with both the coarse and fine grids. This ensures that meshes from adjacent LOD levels match perfectly without gaps or overlaps. The paper provides tables and code for both regular and transition cells.

On the GPU, Transvoxel can be implemented as an extension of the standard Marching Cubes pipeline. Each cell determines whether it is a regular or transition cell based on LOD information, and selects the appropriate lookup table. The classification and prefix-sum pattern remains the same, but the generation step must support both cell types.

F. Dynamic and Editable Voxel Terrain

Sanford's senior project on Dynamic Voxel Based Terrain Generation [10] explores editable voxel terrain using an octree structure. The system maintains a hierarchical voxel representation and supports real-time edits such as digging or building. When voxels change, affected nodes are marked dirty and their meshes are regenerated.

From a parallel computing perspective, dynamic edits introduce several challenges:

- Efficiently identifying which regions must be remeshed.
- Performing localized Marching Cubes or similar algorithms on the GPU for those regions.
- Synchronizing updates between CPU-side data structures (e.g., octrees) and GPU buffers.

Sanford's work is primarily CPU-centric, but subsequent systems and engines have moved more of this logic onto the GPU, using compute shaders to regenerate meshes for modified chunks.

VI. Sparse Acceleration Structures

A dense grid of 1024^3 voxels requires over a gigabyte of memory even with 1 byte per voxel. To render high-resolution worlds, terrain engines must exploit the sparsity of the data: most of a volumetric world is either empty air or solid underground, with the interesting surface occupying a thin manifold.

A. Sparse Voxel Octrees (SVO)

Standard octrees require pointer chasing, which is the enemy of GPU performance due to high latency and poor cache locality. Sparse Voxel Octrees (SVOs) reduce memory complexity from $O(N^3)$ to $O(N^2)$ by only allocating memory for nodes that contain the surface. Laine and Karras [16] demonstrated an efficient GPU implementation for ray-casting SVOs. Their approach replaces the traditional pointer-based tree (which causes poor memory coalescence on GPUs) with a linearized memory layout.

However, modifying an SVO in real-time is computationally expensive. When a user edits the terrain, the engine must traverse the tree, allocate or free nodes, and update pointers. On a GPU, this requires atomic operations and careful memory management to prevent fragmentation. Consequently, SVOs are often used for static geometry, while dynamic terrain relies on chunk-based hashmaps.

B. GPU Spatial Hashing

To avoid the tree traversal latency of octrees, modern engines often use spatial hashing [17]. The 3D world coordinates of a chunk (x, y, z) are hashed to an index in a linear buffer. This allows $O(1)$ access to voxel data without pointer chasing.

This technique is particularly effective for "infinite" procedural worlds. Since the hash map has a fixed size, it acts as a rolling cache. As the camera moves, old chunks collide with new chunks in the hash table, and the engine simply overwrites the old data. This maps perfectly to the GPU's memory architecture, allowing compute shaders to query terrain density via efficient texture fetches or buffer loads without traversing deep hierarchical structures.

VII. Streaming and Out-of-Core Voxel Worlds

Realistic open worlds often exceed the memory capacity of a single GPU. To manage this, engines must stream data in and out of GPU memory based on camera position and view. For voxel terrain, this leads to out-of-core or paged systems where only a subset of the world is resident.

A. Performance Constraints in Streaming

The effectiveness of any out-of-core system relies on its ability to hide latency (τ) by overlapping I/O and compute. The total time required to make a required chunk of size C visible on the GPU can be expressed as:

$$T_{\text{load}} = T_{\text{read}} + T_{\text{transfer}} + T_{\text{generate}} \tag{2}$$

Where T_{read} is the time spent fetching data from disk (asynchronously), T_{transfer} is the time to move C across the PCIe bus, and T_{generate} is the time for GPU meshing and finalization. The maximum tolerable latency, τ_{max}, is defined by the frame time ($\frac{1}{\text{FPS}}$) minus a safety budget (δ): $\tau_{\text{max}} \approx \frac{1}{\text{FPS}} - \delta$. The main constraint is memory bandwidth and latency between storage, CPU memory, and GPU memory.

B. Gigavoxel Paged Terrain

Pecoraro's thesis on GigaVoxel Paged Terrain Generation and Rendering [11] presents a system that treats the world as a large virtual volume, divided into bricks (small 3D blocks of voxels) that can be paged between disk, main memory, and GPU memory. The GPU stores only a cache of bricks around the viewer; when the camera moves, new bricks are requested and old ones may be evicted.

Key ideas include:

- A hierarchical page table mapping virtual voxel coordinates to physical bricks, similar to virtual texturing.
- Background threads and asynchronous I/O to load bricks without stalling rendering.

- GPU-side ray casting or isosurface extraction over the cached bricks.

While the thesis uses a specific rendering technique, the broader pattern is widely applicable: treat the GPU as a limited cache over a much larger procedural or stored dataset, and use indirection to access bricks. This method however, does mean that the bottleneck has been shifted from GPU compute units to PCIe bus

C. Chunking and Streaming Patterns

Voxel-based engines often divide the world into fixed-size chunks or regions (e.g., 32^3 or 64^3 voxels). For each chunk, the engine maintains:

- Voxel data (either stored explicitly or generated procedurally on demand).
- A mesh, generated via Marching Cubes or similar.
- Metadata including LOD level and timestamps for streaming.

At runtime, chunks near the camera are kept in memory and potentially at higher resolution; distant chunks may use coarser LOD or be unloaded entirely. Chunk management is typically CPU-driven, but GPU compute shaders or asynchronous compute queues can be used to generate voxel data and meshes for newly loaded chunks.

The parallelism here is multi-level:

- Across chunks: multiple chunks can be generated or meshed in parallel.
- Within chunks: each voxel cell can be processed independently during scalar field evaluation and surface extraction.

The main constraints are memory bandwidth and latency between storage, CPU memory, and GPU memory. Systems such as Gigavoxel terrain [11] explicitly model these constraints and design multi-stage pipelines to hide latency.

D. Integration with LOD and Transitions

Streaming interacts closely with LOD. When a higher-resolution chunk is loaded, the engine must ensure that its neighbors at coarser resolutions are still present until transition meshes can be generated. Conversely, unloading a chunk requires updating neighbors to prevent cracks.

Transvoxel [9] provides a robust algorithmic framework for LOD transitions, but practical systems must manage the lifecycle of chunks and meshes around these boundaries. Many engines use ring-based regions or clipmap-inspired structures in 3D, combining ideas from heightmap geometry clipmaps with volumetric chunking.

VIII. GPU Work-Graph Pipelines for Procedural Geometry

The techniques discussed so far primarily use data-parallel kernels operating over arrays or grids. Recent GPU architectures and APIs introduce more flexible execution models based on task graphs, enabling more dynamic and irregular workloads to stay on the GPU.

A. Work Graphs and Device-Side Task Generation

Kuth et al. [12] proposed a system for real-time procedural generation with GPU work graphs. Prior to work graphs, the CPU had to be interrupted to schedule new chunks. Work graphs allow the GPU to become a self-feeding state machine, eliminating the latency of the CPU-GPU synchronization loop. Work graphs are a feature introduced in recent graphics APIs (for example, in DirectX and vendor-specific extensions) that allow GPU kernels to enqueue new work items and form a graph of tasks. Unlike traditional compute shaders, which must be dispatched from the CPU, work graphs permit device-side scheduling and dynamic task generation.

In their paper, the authors demonstrate several applications of work graphs to procedural geometry, including vegetation, instanced meshes, and other content. The core idea is to represent the procedural pipeline as a graph of nodes, where each node encapsulates a specific computation (e.g., generating tiles, refining patches, spawning instances). Nodes can spawn other nodes, enabling adaptive refinement and irregular workloads without CPU intervention.

The core performance benefit of the GPU work-graph pipeline is the near-elimination of host-side dispatch latency. For N procedural chunks requiring M steps of refinement and meshing, the conventional total overhead, $O_{\text{conventional}}$, scales linearly with the number of necessary CPU dispatches (D):

$$O_{\text{conventional}} = \sum_{i=1}^{D} (\tau_{\text{API}} + \tau_{\text{sync},i}) \tag{3}$$

Where τ_{API} is the driver overhead and $\tau_{\text{sync},i}$ is the synchronization cost (often a full CPU-GPU round-trip). By contrast, work graphs internalize the dispatch logic, making the overhead nearly constant, dominated only by the initial submission and final synchronization. This design enables a substantial reduction in overhead for highly irregular and dynamic workloads.

B. Applicability to Terrain and Voxel Worlds

Although Kuth et al. [12] do not focus exclusively on terrain, their framework is highly relevant. A terrain engine can naturally be expressed as a hierarchy of tasks:

- High-level nodes decide which regions or chunks need to be (re)generated based on camera position and LOD policies.
- Mid-level nodes evaluate scalar fields over tiles or bricks, using noise and procedural functions.
- Low-level nodes perform surface extraction (e.g., Marching Cubes) and write meshes into GPU buffers.

With work graphs, all of these tasks can be orchestrated on the GPU. This reduces CPU–GPU synchronization overhead and allows the terrain pipeline to react more quickly to camera motion or edits. It also opens the door to more sophisticated scheduling and load balancing strategies, such as prioritizing chunks in the current view frustum or adaptively refining areas of high curvature.

C. Relation to HPC Task Graphs

The move from pure data-parallel kernels to work graphs mirrors trends in high-performance computing more broadly, where task-graph runtimes manage dependencies among heterogeneous tasks across CPUs and accelerators. Work graphs can be seen as a GPU-resident task-graph runtime specialized for graphics and compute workloads, offering:

- Reduced launch overhead for small, frequent tasks.
- Better support for irregular and adaptive computations.
- Closer coupling between procedural generation and rendering.

For procedural terrain and voxel worlds, this suggests a future where the entire pipeline—from high-level world layout down to per-voxel operations—can be encapsulated in a GPU-driven task graph with minimal CPU coordination.

IX. Discussion and Open Challenges

The evolution from CPU-based heightmaps to GPU-driven voxel worlds has enabled much richer and larger virtual environments, but several challenges remain.

A. Quality vs. Performance vs. Memory

Noise-based terrain is efficient and compact, but can appear artificial without additional processing (e.g., erosion). Simulating erosion or other physical processes at runtime is expensive, and precomputing them undermines the benefits of procedural generation. Balancing realism and performance remains an open problem, particularly when the scalar field must be evaluated many times per frame across multiple LOD levels.

Memory is another major constraint. Voxel representations require substantially more storage than heightmaps, especially for high-resolution volumes. Sparse data structures and paging systems [11] mitigate this, but introduce complexity in streaming and cache management.

B. Editability and Destruction

Destructible terrain is a key motivation for voxel worlds. However, supporting arbitrary edits imposes strict requirements on data structures and meshing:

- Edits can invalidate LOD assumptions and require updates across multiple resolutions.
- Chunk-based meshing must avoid visible seams when neighboring chunks are updated at different times.
- Parallel update algorithms must avoid race conditions when multiple edits occur concurrently.

Some systems address this by restricting edits to coarse resolutions or using hybrid representations (e.g., a base terrain plus sparse edits). Fully general, high-resolution, destructible voxel terrain at scale remains a challenging target.

C. Streaming and Latency Hiding

Streaming systems rely on overlapping I/O, CPU processing, and GPU computation. Achieving smooth camera movement without visible popping or stalls requires careful tuning of:

- Prefetch distances and LOD thresholds.
- Asynchronous transfers between storage, CPU memory, and GPU memory.
- Budgeting for chunk generation and meshing per frame.

Work-graph-based approaches [12] may help by allowing the GPU to manage some of this scheduling internally, but they do not eliminate the need for well-designed streaming policies.

D. Integration with Other Systems

Terrain and voxel engines rarely operate in isolation. They must integrate with:

- Physics engines (collision detection against voxel or mesh surfaces).
- AI systems (pathfinding on dynamic terrain).
- Rendering features (shadows, global illumination, atmospheric effects).

Each integration point introduces additional data dependencies and synchronization constraints. For example, physics may require a simplified collision representation that must be updated when terrain changes, while GI may need access to the voxel density field.

X. Conclusion

GPU-accelerated procedural terrain and voxel worlds have evolved significantly over the past two decades. Early work on geometry clipmaps and GPU-based heightfields demonstrated that terrain rendering could be moved largely onto the GPU [1], [2]. Subsequent research extended this to full volumetric terrain, combining procedural scalar fields with GPU-accelerated isosurface extraction and sophisticated LOD schemes such as Transvoxel [8], [9]. Out-of-core and paged voxel systems then addressed the challenge of worlds larger than GPU memory [11], while recent work on GPU work graphs suggests new ways to express entire procedural pipelines as device-side task graphs [12].

Across these developments, several common themes emerge: the use of data-parallel kernels for scalar field evaluation and meshing, hierarchical structures for LOD, and multi-stage pipelines for streaming and caching. At the same time, important challenges remain regarding realism, editability, streaming, and integration with other engine systems.

For high-performance computing and parallel programming, this domain offers a rich set of real-world workloads that blend regular grid computations with hierarchical and task-based patterns. While voxel engines have traditionally been memory-bound, the emergence of GPU Work Graphs suggests the future bottleneck will shift back to pure compute, as we can now procedurally generate infinite detail on-chip without ever touching main memory. Future work is likely to explore tighter integration between procedural generation, simulation, and rendering, leveraging both data-parallel and task-graph paradigms on emerging GPU architectures

REFERENCES

[1] A. Asirvatham and H. Hoppe, "Terrain rendering using GPU-based geometry clipmaps," in *GPU Gems 2*, M. Pharr and R. Fernando, Eds. Addison-Wesley, 2005.

[2] R. Geiss, "Generating complex procedural terrains using the GPU," in *GPU Gems 3*, H. Nguyen, Ed. Addison-Wesley, 2007.

[3] H. Li, X. Tuo, Y. Liu, and X. Jiang, "A parallel algorithm using Perlin noise superposition method for terrain generation based on CUDA architecture," in *Proc. Int. Conf. on Materials Engineering and Information Technology Applications (MEITA)*, 2015.

[4] G. C. Backes, T. A. Engel, and C. T. Pozzer, "Real-time massive terrain generation using procedural erosion on the GPU," in *Proc. 17th Brazilian Symp. on Computer Games and Digital Entertainment (SBGames), Computing Track*, 2018.

[5] T. J. Rose and A. G. Bakaoukas, "Algorithms and approaches for procedural terrain generation – a brief review of current techniques," in *Proc. 2016 8th Int. Conf. on Games and Virtual Worlds for Serious Applications (VS-GAMES)*, 2016.

[6] M. V. M. Cirne and H. Pedrini, "Marching cubes technique for volumetric visualization accelerated with graphics processing units," *Journal of the Brazilian Computer Society*, vol. 19, no. 3, 2013.

[7] L. P. Engström, "Volumetric terrain generation on the GPU," Master's thesis, KTH School of Computer Science and Communication, 2015.

[8] E. S. Lengyel, "Voxel-based terrain for real-time virtual simulations," Ph.D. dissertation, University of California, Davis, 2009.

[9] E. S. Lengyel, "Voxel-based terrain for real-time virtual simulations," Ph.D. dissertation, University of California, Davis, 2009.

[10] T. Sanford, "Dynamic voxel based terrain generation," Senior Project, California Polytechnic State University, San Luis Obispo, 2015.

[11] A. N. Pecoraro, "GigaVoxel paged terrain generation and rendering," Master's thesis, DigiPen Institute of Technology, 2015.

[12] B. Kuth, M. Oberberger, C. Faber, D. Baumeister, M. G. Chajdas, and Q. Meyer, "Real-time procedural generation with GPU work graphs," *Proc. ACM Comput. Graph. Interact. Tech.*, vol. 7, no. 3, 2024.

[13] P. Andersson, "Rendering with Marching Cubes – looking at hybrid solutions," Bachelor's thesis, Umeå University, 2012.

[14] I. Cantlay, "DirectX 11 terrain tessellation," NVIDIA Whitepaper, 2011. [Online]

[15] T. Ju, F. Losasso, S. Schaefer, and J. Warren, "Dual contouring of Hermite data," in *Proc. ACM SIGGRAPH*, 2002.

[16] S. Laine and T. Karras, "Efficient sparse voxel octrees," in *Proc. ACM SIGGRAPH Symp. on Interactive 3D Graphics and Games (I3D)*, 2010.

[17] S. Lefebvre and H. Hoppe, "Perfect spatial hashing," *ACM Transactions on Graphics (TOG)*, vol. 25, no. 3, 2006.