Scalable Grid-Based Crowd Simulation in Complex Multi-Layer Environments on the GPU

Thesis Report

Jesse Conijn

Delft University of Technology

TU Delft

Scalable Grid-Based Crowd Simulation in Complex Multi-Layer Environments on the GPU

Thesis Report

by

Jesse Conijn

4493516

Supervisors:	Prof. E. Eisemann
	Dr. L. Ferranti
Year:	2024 - 2025
Faculty:	Mechanical Engineering, Delft
	Computer Science, Delft

TU Delft

Contents

Scalable Grid-Based Crowd Simulation in Complex Multi-Layer Environments on the GPU

Jesse Conijn
TU Delft
Delft, The Netherlands

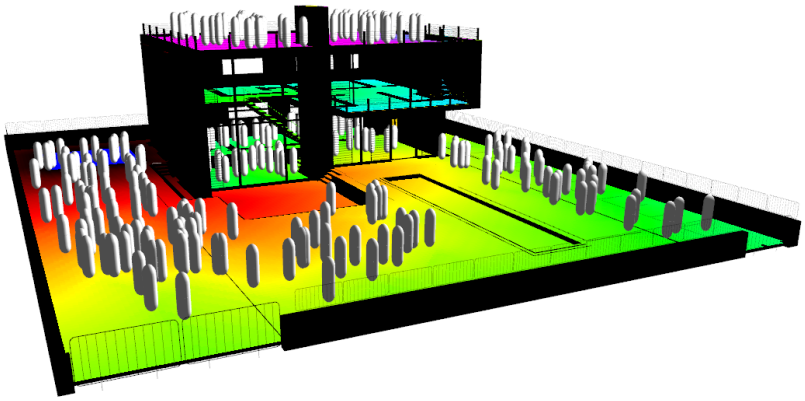


Figure 1: Real-time potential field projection in a densely populated, multi-level 3D scene.

Abstract

Simulating how crowds move through complex environments is essential for applications in urban planning, robotics, gaming, and safety analysis. However, many real-world spaces—such as multi-story buildings, staircases, and layered architectural designs—are too complex for traditional 2D or CPU-based crowd simulation methods, which often oversimplify geometry or become computationally infeasible. This thesis introduces a fully GPU-accelerated crowd simulation framework that efficiently handles complex, multi-layered 3D environments. Building on the Continuum Crowds algorithm, our method extracts walkable surfaces at different height levels, identifies vertical obstructions, and enables real-time navigation for thousands of agents. The system avoids common artifacts such as unrealistic wall clipping and enables realistic movement across layers. In performance benchmarks, our GPU implementation achieves a speedup between 5× and 1000× over the original CPU-based method, depending on scenario complexity, demonstrating strong scalability and consistent real-time performance. This makes it particularly valuable in domains such as robotics, where anticipating pedestrian flow is crucial for safe and intelligent robot navigation in dynamic environments.

Keywords

Computer Graphics, Continuum Crowds, Path Planning, Crowd Simulation, Multi-layer, Complex Environment, GPU, Robotics

1 Introduction

Simulating the movement of large crowds plays an essential role in applications such as video games [26], visual effects [9], robotics [22] and urban planning [6]. Realistic crowd simulation allows for better decision-making in complex environments and can enhance both the visual accuracy and the practical utility of virtual and real-world systems. In robotics, for example, it supports motion planning algorithms that must account for dynamic human environments, such as airports or hospitals.

Traditional crowd simulation methods often rely on agent-based models, where each individual is simulated separately using behavioral rules or local navigation techniques. While these approaches can produce natural movement patterns, they quickly become computationally expensive as the number of agents increases, especially in large or dynamic environments.

A significant breakthrough came with the Continuum Crowds algorithm, introduced by Treuille et al. in 2006 [25]. Unlike previous per-agent navigation models, Continuum Crowds treats the crowd as a continuous fluid-like density field. It uses a series of variable fields-like density, speed, and traversal cost-to compute a potential field, which then guides each agent along a gradient toward their goal. This approach shifts the computational focus from individual agents to grid-based fields, making it highly scalable and more suitable for real-time applications involving thousands of agents.

Despite its strengths, the original Continuum Crowds algorithm has limitations. First, it was designed to run on the CPU, which restricts its scalability in high-density scenarios. Second, the algorithm assumes a flat, 2D environment, making it unsuitable for complex, multi-layered spaces such as staircases, bridges, and multi-story buildings. As modern applications increasingly demand real-time simulations in rich 3D environments, these limitations become more pronounced.

Following Continuum Crowds came several other publications trying to improve this method, either focusing on speeding up the algorithm by translating it to the GPU, or on expanding it

to more complex multi-layer environments. Unfortunately, no research has yet succeeded in combining the two, since all methods that were optimized for the GPU were still only usable in simple 2D environments, and other methods that worked in multi-layered environments were incompatible with the GPU.

This thesis builds upon the Continuum Crowds framework and focuses on restructuring the algorithm to build a bridge between operating efficiently on the GPU and extending it to handle multi-layered environments. By leveraging the parallel processing power of modern GPUs and designing new techniques for layer detection and representation, the goal is to enable scalable, real-time crowd simulation in environments that reflect the complexity of real-world spaces. To our knowledge, this is the first approach to combine GPU-accelerated crowd simulation with support for complex, multi-layered 3D environments—bridging a gap previously left open by existing methods.

The remainder of this thesis is organized as follows. Section 2 reviews existing work on crowd simulation, scene representation, and GPU-based methods relevant to our approach. Section 3 provides an overview of the original Continuum Crowds algorithm and its underlying principles. Section 4 introduces our method for extracting a structured, multi-layered scene representation using GPU-based depth peeling, including the handling of vertical surfaces, inter-layer connectivity, and goal annotation. Section 5 describes how the original Continuum Crowds algorithm is adapted and integrated into the fully GPU-parallel framework. This includes the computation of variable fields, the parallel solution of the Eikonal equation, and the update of agents based on the resulting navigation fields. Section 6 presents the results of our method across various scenes, followed by a discussion in Section 7 that reflects on limitations, challenges, and opportunities for future work.

2 Related Works

Early crowd simulation systems such as Reynolds' *Boids* [23] demonstrated flocking behavior using simple rule-based interactions. Since then, more sophisticated agent-based approaches have emerged, including the Social Force Model for pedestrian dynamics [12], which models individuals as particles influenced by attractive and repulsive forces. Other notable methods include force-based models [11], velocity-based collision avoidance [7], and vision-based systems [21]. While effective for simulating local interactions and producing realistic motion, these agent-based methods often face scalability challenges in dense or geometrically complex settings.

To address such limitations, Treuille et al. [25] introduced the *Continuum Crowds* algorithm, modeling crowd motion as a potential field over a grid. This approach allows for global coordination and smooth navigation by solving the Eikonal equation over a discretized domain. Several extensions have adapted this framework to support anisotropic cost metrics [10], group behaviors [17], and GPU-accelerated computation [20].

Solving the Eikonal Equation on the GPU. Traditional methods for solving the Eikonal equation, such as the Fast Marching Method (FMM) [24], provide accurate solutions but rely on sequential updates, limiting their parallel efficiency. To address this, parallel-friendly variants like the Fast Sweeping Method (FSM) [28] and its parallel extensions [2] [3] enable grid-wide updates through directional passes. The Fast Iterative Method (FIM) [14], in contrast, is more compatible with GPU architectures because it avoids global sorting or directional sweeps. Instead, it uses a relaxed, asynchronous update strategy based on active sets, where only the texels that may change are iteratively updated. This allows for efficient parallelization without requiring tight synchronization or complex memory management. Improved versions [13] [8] further reduce computational overhead in large-scale simulations.

Navigating Complex Environments. Adapting crowd simulation to multi-layered 3D environments introduces additional challenges. Semantic models [15] [16] and layered navigation meshes [27] offer ways to model vertical connectivity, but they are often not optimized for GPU-based implementations.

To extract layered walkable surfaces efficiently, image-based and rasterization-based techniques are more GPU-friendly. Methods such as raycasting [19] and voxelization [4] allow for grid construction without relying on triangle-level queries. However, these typically capture only the nearest surface per pixel.

Depth Peeling [5] enables multi-layer surface extraction by successively removing the nearest geometry layer across multiple rendering passes. Bucket Depth Peeling (BDP) [18] and its adaptive variant improve performance by capturing multiple layers in a single pass using a fixed-depth bucket structure, reducing the number of rendering passes required and improving GPU utilization.

Our Contribution. This work builds upon the Continuum Crowds framework and extends it to general, multi-layered 3D environments. We introduce a depth-peeling-based GPU pipeline for extracting walkable surface layers, combined with texture-based inter-layer connectivity encoding, requiring as little user annotation as possible. We solve the Eikonal equation using an optimized GPU implementation of the Fast Iterative Method, and support multi-goal navigation and discomfort-aware boundary conditions. To our knowledge, this is among the first fully GPU-based implementations of the Continuum Crowds algorithm specifically adapted to handle multi-layered environments with minimal user input.

Notation. This report works with three main coordinate spaces:

- **World-space coordinates** (x,y,z): This is the primary 3D coordinate system used for positioning geometry. We follow OpenGL's convention, where the x-axis points right, the y-axis points upward (vertical), and the z-axis points into the screen (depth).
- **Texture-space coordinates** (u, v): These 2D coordinates lie in the range $[0, 1]^2$ and are used to sample from textures. Although sometimes denoted as (x, y) in the shader code for simplicity, we use (u, v) notation to clearly distinguish them from world coordinates. The origin is typically the bottom-left of the texture in OpenGL.
- **Screen-space coordinates** (x_s, y_s, z_s): These are defined in `gl_FragCoord`, where (x_s, y_s) are in pixel units and represent the fragment's position on the screen. The origin is at the bottom-left of the window, with $x_s \in [0, \text{width}]$, $y_s \in [0, \text{height}]$, and $z_s \in [0, 1]$ corresponding to the depth buffer range.

3 Continuum Crowds

The *Continuum Crowds* algorithm, introduced by Treuille et al. [25], presents a macroscopic approach to simulating crowds by modeling agents as a continuous fluid. Rather than treating individual agents as discrete entities, the algorithm represents the combined behavior of many agents using continuous scalar and vector fields defined over a spatial grid. This allows for efficient simulation of large-scale crowd behaviors while maintaining smooth, realistic movement patterns.

At the core of the Continuum Crowds model are two key components: a **potential field** and a **velocity field**. The potential field guides agents toward their goals, and is computed by solving the *Eikonal equation*, which encodes the travel cost from every cell to the goal location. The cost of traversing each cell is influenced by terrain factors (such as slope), environmental discomforts, and dynamic agent density. The velocity field is derived as the spatial gradient of the potential field, pointing in the direction of steepest descent and indicating how agents should move to minimize their travel cost.

To update the simulation, the algorithm evolves three fields over time:

- A **density field** that tracks how many agents occupy each location.
- A **speed field** that determines how fast agents can move, influenced by crowd density and environmental features.
- A **potential field** that guides agents toward their goal.
- A **velocity field** that prescribes agent movement based on potential.

This field-based model allows agents to implicitly avoid collisions and dynamically adapt to congestion. It also enables large-scale path planning and group navigation through complex environments without requiring explicit pathfinding for each agent.

In the following sections, we build upon this foundational model by extending it to support simulation in multi-layered 3D environments, leveraging modern GPU hardware to compute each field in parallel for high performance.

4 Scene Representation in Multi-Layered Environments

To enable real-time crowd simulation on the GPU, the environment must first be transformed into a structured, grid-based representation that supports parallel computation. This section describes the preprocessing steps required to extract walkable surfaces, identify vertical obstructions, encode inter-layer connectivity, and annotate navigation goals—all in a format optimized for GPU execution.

We begin by extracting a fixed number of horizontal scene layers using Depth Peeling [5], a rendering technique that captures multiple depth layers from a single viewpoint. We describe how this is adapted for our purposes in Section 4.1. Since near-vertical geometry is not well captured from a top-down view, we then introduce an additional step to identify vertical fragments using the same technique, but from side and front perspectives (Sec. 4.2). With this improvement to the scene representation, we proceed to compute connectivity between layers to determine where agents can traverse vertically (Sec. 4.3). Next, we apply boundary padding to mark areas near unwalkable surfaces as invalid, ensuring agents do not intersect with walls or fall between layers (Sec. 4.4). Finally, we describe a method for annotating agent goal regions within the scene through user interaction, enabling group-specific navigation targets (Sec. 4.5).

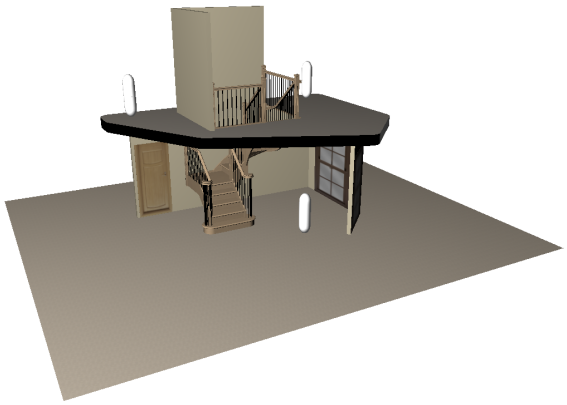


Figure 2: Scene 1, a multi-level environment. The stairs in the middle connect the two levels. The white pill-shaped models represent agents. [1]

4.1 Layer Extraction

To enable grid-based simulation over complex, multi-layered environments on the GPU, the environment must first be decomposed into a stack of structured 2D layers. This representation allows the complex 3D geometry of the scene to be reformulated into a regular grid structure, where each cell (or texel) can be independently processed in parallel. By working with layered 2D textures, we avoid the irregular memory access patterns and geometric queries associated with mesh-based navigation, making the pipeline more compatible with real-time GPU execution.

While alternative scene representations exist—such as navigation meshes, semantic maps, or full 3D voxel grids—these often require more extensive preprocessing or introduce scalability issues on the GPU. In particular, pointer-based or hierarchical structures are not directly compatible with GPU pipelines, which favor flat, texture-based data representations. Textures, along with uniform buffers and structured shader storage, remain among the few formats natively supported in fragment and compute shaders.

Our layered texture-based approach provides a practical balance: it captures key 3D navigational features such as elevation, slopes, and multi-level connectivity, while remaining efficient to process in parallel on the GPU.

Each layer corresponds to a horizontal slice of navigable geometry, captured from an orthographic top-down camera. These slices are then encoded into texture arrays, with each array storing a specific physical quantity such as world-space position, surface normal, depth value, semantic label, or walkability flag. To provide a consistent visual context throughout this thesis, Figure 2 shows the multi-level scene used for all subsequent illustrations and evaluations of the algorithm.

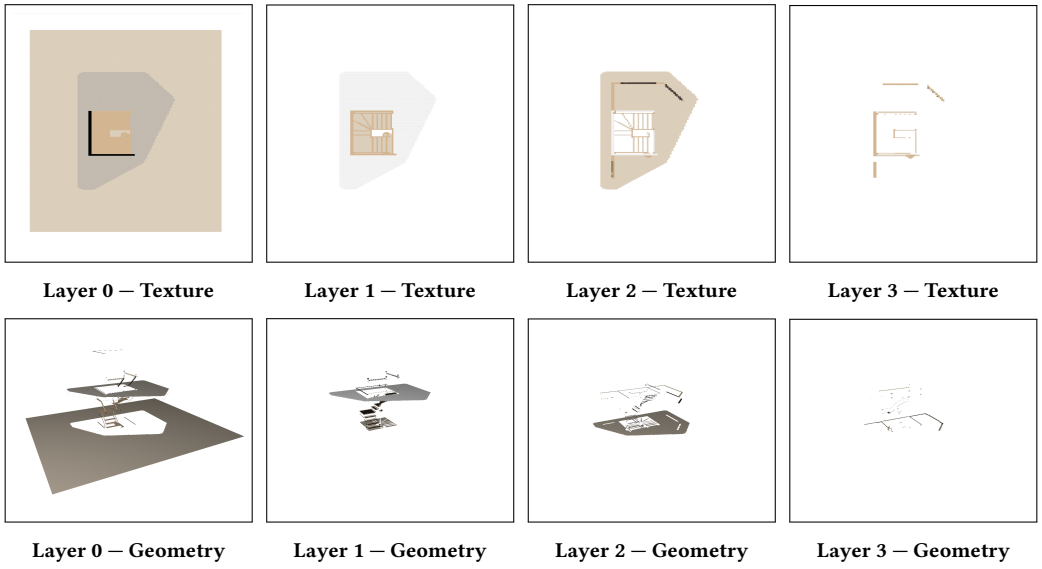


Figure 3: Depth peeling results for layers 0–3. The top row shows the captured surface colors in the `Color` texture array. The bottom row shows the corresponding geometry fragments assigned to each layer, visualizing the actual 3D surfaces that remain after peeling away higher layers.

In order to represent the scene as a structured stack of 2D layers suitable for GPU computation, a set of layered texture arrays is created, each with k layers. During rendering, **Multiple Render Targets (MRTs)** are used to write to different outputs from the fragment shader, with each output directed to a corresponding layer of a texture array.

Since different physical quantities (e.g., position, normal, flags) vary in dimensionality and data type, each texture array is configured with an appropriate internal format to match the size and precision of the stored data. For instance, position and normal vectors are stored using three-component (RGB) formats, while scalar fields such as fragment flags require only a single component.

This design allows the simulation to efficiently access and process different types of scene data in parallel on the GPU. For example, normals are used to filter out steep surfaces, goal IDs determine agent destinations, and discomfort values affect path planning. Choosing the correct format for each quantity ensures memory efficiency and avoids performance bottlenecks during shader operations. Table 1 summarizes the structure of the texture arrays used during the layer extraction stage.

Output	Format	Description
Position	RGB	World-space position vector
Normal	RGB	World-space surface normal
Grid	R	Encoded fragment flags
Goal	R	Goal IDs
Connection	RGBA	Inter-layer connections
Discomfort	R	Per-cell discomfort cost
Depth	Depth component	Z-buffer depth value

Table 1: Texture Arrays of Scene Layer Encodings

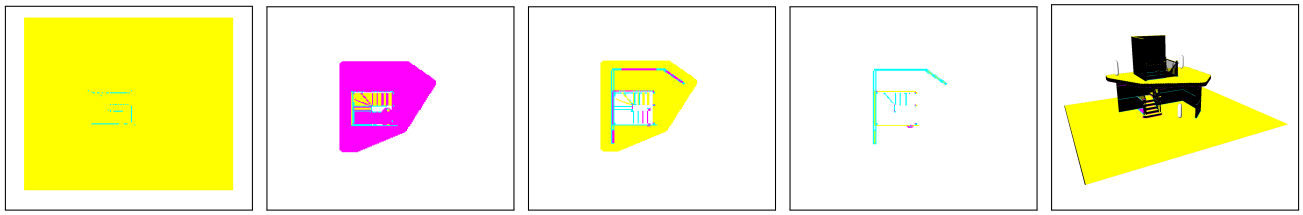
The spatial resolution (width × height) and the number of layers k for each texture array are chosen empirically to balance simulation accuracy and real-time performance. Higher resolutions provide finer accuracy for agent navigation and obstacle detection, but increase memory consumption and computational cost. Conversely, lower resolutions improve rendering performance at the cost of spatial detail. The number of layers k is primarily determined by the vertical extent of the environment and the chosen spatial resolution. It is selected to ensure complete coverage of all navigable surfaces while minimizing unused entries in the texture array. In practice, the chosen settings were selected through iterative testing to ensure sufficient fidelity for complex environments while maintaining interactive frame rates.

To extract individual layers of geometry from the scene, we employ the GPU technique Depth Peeling [5]. Traditionally used for order-independent transparency, Depth Peeling is repurposed here to progressively capture deeper fragments at each pixel from the camera's orthographic view.

To ensure numerical stability when sampling multiple depth layers, we apply a small depth offset (known as z-fighting prevention) to avoid visual artifacts caused by overlapping surfaces being rendered at nearly the same depth.

We denote:

- $D_i(x, y)$ the screen-space depth at pixel (x, y) in layer i
- $P_i(x, y)$ the world-space position of the fragment at that pixel in layer i
- $N_i(x, y)$ the world-space surface normal
- $G_i(x, y)$ the walkability flag assigned to that texel
- $H_i(x, y) = P_i^y(x, y)$ the vertical world height (i.e., the y-component of P_i)
- δ a small depth bias to prevent z-fighting (e.g., 10^{-3})



(a) Layer 0 (b) Layer 1 (c) Layer 2 (d) Layer 3 (e) Projection of layers

Figure 4: Visualization of the `Grid` textures across multiple layers. Yellow indicates walkable regions, Purple highlights texels that lie too close beneath higher layers, and Cyan marks surfaces that are too steep (i.e., too vertical) to walk on or found under these vertical surfaces. The final subfigure shows these values projected back onto the 3D scene for spatial context, with black areas indicating invalid regions or vertical fragments.

The first layer ($i = 0$) captures the closest visible fragments at each screen-space coordinate (x, y):

$$D_0(x, y) = \min_{p \in \text{scene}} \text{depth}_p(x, y) \tag{1}$$

where $\text{depth}_p(x, y)$ is the screen-space depth value of fragment p at pixel (x, y). This produces the first depth-peeling layer via a standard depth buffer pass.

Subsequent layers are computed by discarding fragments that are closer than the previously stored depth:

$$\text{if} \quad D_i(x, y) \leq D_{i-1}(x, y) + \delta \quad \rightarrow \quad \texttt{discard} \tag{2}$$

This ensures that each layer i retains only the next-deepest geometry, peeling away upper surfaces layer by layer. The process is repeated until all visible layers are written into the corresponding slices of the texture arrays.

To ensure the extracted surfaces are relevant for agent navigation, fragments are filtered based on two geometric criteria:

(1) **Surface Orientation**: Fragments with surface normals oriented too steeply relative to the up-axis (i.e., vertical walls or slopes) are flagged as non-walkable, since such surfaces cannot be realistically traversed:

$$\text{if} \quad |\mathbf{n} \cdot \mathbf{y}| < \theta_{\min} \quad \rightarrow \quad G_i(x, y) = 1 \tag{3}$$

where $\mathbf{n}$ is the world-space normal, $\mathbf{y}$ is the global up-vector $(0, 1, 0)$, and $\theta_{\min}$ is a tunable threshold that defines the minimum acceptable surface flatness.

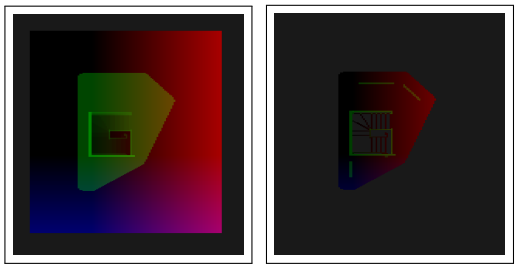
(2) **Vertical Separation**: A minimum vertical offset $h_{\min}$ is enforced to discard surfaces that are too close under prior layers. This prevents narrow gaps between stacked surfaces from being mistakenly treated as separate walkable layers:

$$\text{if} \quad H_{i-1}(x, y) - H_i(x, y) < h_{\min} \quad \rightarrow \quad G_i(x, y) = 2 \tag{4}$$

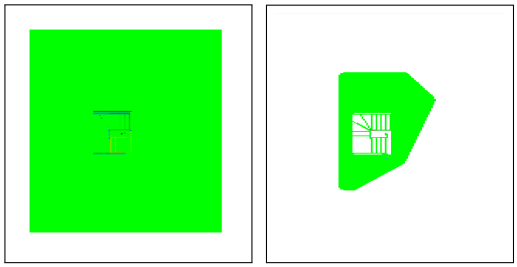
(3) All other fragments are considered walkable and assigned a walkability flag of 0: $G_i(x, y) = 0$

The walkability flag $G_i(x, y)$ is written to the corresponding coordinate in the `Grid` texture array.

It is important that all non-walkable texels are explicitly flagged (rather than discarded), as their vertical heights $H_{i-1}(x, y)$ are required in the next depth-peeling iteration. This enables the system to correctly evaluate the vertical separation for the next layer and avoid invalid walkable surfaces beneath obstructions.



Position — Layer 0 Position — Layer 2



Normal — Layer 0 Normal — Layer 2

Figure 5: Visualization of the `Position` and `Normal` textures for layers 0 and 2. The position textures encode world-space coordinates (red = x, green = y, blue = z), while the normal textures represent surface orientation in world-space. Together, they form the geometric basis for walkability analysis and navigation computation.

Figure 3 illustrates the resulting stack of (the first four) color texture layers (top) and visualizes the fragments captured with each layer in the 3D environment (bottom), while Figures 4 and 5 and show representative outputs of the grid, position and normal textures. Together, they highlight the layered scene representation produced by the extraction stage. These texture-based representations form the basis for all subsequent GPU-parallel computations, enabling fast and scalable access to spatial, semantic, and geometric information.

4.2 Identifying Vertical Fragments

One of the challenges with the top-down orthographic depth peeling approach is its inability to reliably detect vertical or near-vertical surfaces. From the perspective of the top-down camera, these surfaces have normals that point sideways rather than upwards, and are therefore invisible or misclassified, even though they may intersect or occlude walkable surfaces like floors. This limitation can lead to artifacts where agents are allowed to walk partially through walls, especially if the floor extends beneath these vertical obstructions.

To address this, we introduce a secondary analysis phase that identifies vertical fragments using two additional orthographic cameras—one facing the scene from the front and one from the side (see Fig. 6). In a GPU rendering pipeline, adding such camera views is straightforward and incurs minimal overhead, as the same depth peeling shaders can be reused from different viewpoints. We repeat the depth peeling process from both perspectives, storing position and normal data for each layer as before.

Next, we use a compute shader to analyze each texel in parallel, where each thread is assigned to one specific texel (Appendix 7.2). It iterates over the texels representing the fragments detected from the side and front views. For each texel, we check whether the normal deviates significantly from the vertical axis, i.e., if

$$|\mathbf{n} \cdot \vec{y}| < \tau.$$

We use a threshold of $\tau = 0.5$ to allow for some tolerance in surface orientation; this corresponds to surfaces steeper than approximately 60° from vertical, which helps avoid misclassifying slightly inclined walkable surfaces as vertical walls.

For each fragment flagged as vertical, we first determine where it maps to in the top-down view by projecting its world-space position into the top-down camera's texture space. This allows us to identify the corresponding texel in the original layer stack, where potential occlusion or interference with walkable surfaces may occur.

The projection follows a standard transformation pipeline in 3D graphics, consisting of two matrices: the `View` matrix (which aligns the camera with the scene), and the `Projection` matrix (which maps the 3D coordinates into a 2D viewing frustum). The fragment's world-space position $\mathbf{p}$ is first transformed into *clip space*:

$$\mathbf{p}_{\text{clip}} = \texttt{Projection} \times \texttt{View} \times \mathbf{p},$$

This clip-space position is then normalized by dividing by its fourth coordinate (w) to yield normalized device coordinates (NDC):

$$\mathbf{p}_{\text{ndc}} = \frac{\mathbf{p}_{\text{clip}}}{\mathbf{p}_{\text{clip}}.w}.$$

The resulting NDC values lie in the range $[-1, 1]$. These are converted to UV texture coordinates in the $[0, 1]$ range as follows:

$$(u, v) = 0.5 \cdot (\mathbf{p}_{\text{ndc}}.x, \mathbf{p}_{\text{ndc}}.y) + 0.5.$$

After these transformations, the resulting UV coordinates correspond to a specific texel in the top-down layer. From this location, we iteratively search downward through the stack to find a valid walkable surface that may be affected by the vertical geometry.

(a) Front

(b) Side

Figure 6: Visualization of the scene with orthographic view from front and side perspective.

Specifically, we look for the closest candidate that:

- lies below the vertical fragment (based on y position),
- is within a given vertical distance threshold (to avoid propagating effects too far), and
- has a sufficiently horizontal surface normal (i.e., $|\text{dot}(\mathbf{n}, \vec{y})| > 0.5$).

If a texel satisfies all criteria, it is marked as invalid in the `gridTextures` array, preventing agents from navigating into or through regions immediately adjacent to vertical walls. This additional invalidation step improves realism and addresses a key limitation of the original layer extraction method.

This solution effectively combines multiple orthographic perspectives to compensate for visibility limitations in any single view, resulting in a more complete and semantically correct multi-layered scene representation.

This step complements the top-down layer extraction, ensuring that vertical occlusions are accurately flagged in the GPU memory layout and eliminating navigation artifacts in real-time scenarios.

4.3 Layer Connectivity

With the scene rendered into multiple texture arrays encoding different physical quantities, the next step is to define inter-layer connectivity. This connectivity enables agents to traverse between layers when transitions are physically valid, such as via stairs, ramps, or other inclined surfaces.

While each layer is a flat representation of the scene in the form of a texture, agents must be able to move between layers to reflect real-world traversability. This requires detecting, for each grid cell (i.e., texel), whether neighboring cells in all four cardinal directions (North, East, South, West) exist on the same or another layer, and whether a connection between them is feasible.

A connection is considered valid from texel (x, y, c) to neighbor (x', y', i) in direction j, if:

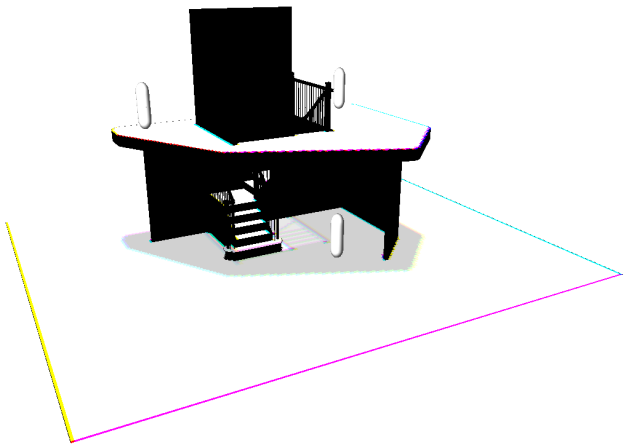


Figure 7: Visualization of the `Connection` textures, where the RGBA channels encode the layer index of each neighboring texel in the four cardinal directions (North, East, South, West). The values are normalized to the range $[0, 1]$, where 0 corresponds to the first layer, 1 to the last, and intermediate values are linearly mapped based on the total number of layers. Texels appear white or light grey when all neighboring directions connect to the same (non-zero) layer — with the brightness indicating which layer (e.g., white = highest layer). Colorful texels (e.g., purple, yellow, cyan) occur when neighboring directions connect to different layers, resulting in variation across the RGBA channels. Dark or black regions may indicate invalid or missing connections, or vertical fragments.

(1) The neighbor texel is within texture bounds,
(2) The neighbor is flagged as walkable in the corresponding `Grid` texture, and
(3) The height difference between the center and neighbor texel is within a threshold $h_{\max}$.

The parameter $h_{\max}$ defines the maximum vertical offset that an agent is allowed to traverse between layers. It determines whether the transition corresponds to a physically valid connection—such as a ramp, step, or slope—versus an unrealistic vertical jump or drop. In practice, this threshold is chosen empirically based on the expected step height or slope angle agents are allowed to climb. For example, if the grid resolution is 0.1 meters per cell, setting $h_{\max} = 0.2$ would allow transitions up to 20 cm, capturing typical stair heights.

Let:

- $\vec{o}_j$ be the offset for direction j (e.g., North = $(0, 1)$, East = $(1, 0)$, etc.),
- $H_c(x, y)$ be the height of the center texel at layer c,
- $H_i(x', y')$ be the height of the neighbor texel at layer i,
- $\Delta H = |H_c(x, y) - H_i(x', y')|$.

Then, the connection is regarded as valid if $\Delta H < h_{\max}$. If this condition is satisfied, the index of the connected layer i is saved in a vector at the index corresponding to direction j. If no valid connection is found, the component is set to -1.

The result is stored in the `Connection` textures. Each texel stores a four-component vector:

$$\texttt{Connection}(x, y, c) = (n, e, s, w)$$

Each component holds the index of the neighbor layer in the corresponding direction:

- **R** (red) = North
- **G** (green) = East
- **B** (blue) = South
- **A** (alpha) = West

For example, $\texttt{Connection}(x, y, 2) = (2, 3, 2, 1)$ means:

- North neighbor is on the same layer (2)
- East neighbor is on layer 3
- South neighbor is on the same layer (2)
- West neighbor is on layer 1

The result is shown in Figure 7, which is a directional connectivity map that makes inter-layer traversal by agents possible during simulation, with per-texel access optimized for GPU parallelism.

4.4 Boundary Padding

After computing the layer connectivity, an additional preprocessing step is required to ensure agents do not occupy positions that partially intersect with non-walkable geometry. While steep surfaces, low ceilings, or overlaps are filtered during the depth peeling stage, certain invalid configurations can still arise, particularly near walls or narrow architectural features where agents might clip into adjacent geometry. To avoid this, we introduce *boundary padding*—a mechanism to pad the area around non-walkable cells and designate surrounding cells as discomfortable.

Using a compute shader, the algorithm proceeds as follows:

- The four cardinal neighbors (north, east, south, west) are inspected using the `Connection` texture.
- If any of the neighboring cells are invalid (i.e., have a connection value of `-1`), the current texel is flagged as discomfortable.
- A discomfortable cell is marked by assigning it a discomfort value of ∞, making it functionally non-walkable during potential field computation. This is written to the `Discomfort` textures.

This ensures that even cells directly adjacent to obstacles are excluded from potential paths, preventing agents from moving too close to surfaces where artifacts like partial intersection with walls would occur. An example is shown in Figure 8.

In cases where agents are significantly larger than the cell size—specifically, when the agent radius exceeds twice the cell radius—a single layer of padding is insufficient. To address this, the padding step includes an extended pass: if a neighboring cell is invalid, the algorithm also inspects the neighbors of that neighbor. If these second-row neighbors are valid, they too are marked as uncomfortable. In this way, a thicker buffer zone is formed, ensuring that larger agents are equally constrained from crossing architectural boundaries.

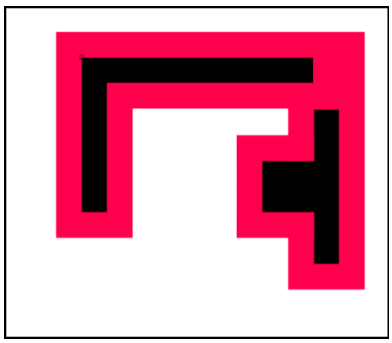


Figure 8: Example visualization of padding around invalid cells in the navigation grid. Black cells indicate non-walkable (invalid) regions such as walls. Red cells represent the surrounding discomfortable area created by the boundary padding step, which prevents agents from navigating too close to obstacles.

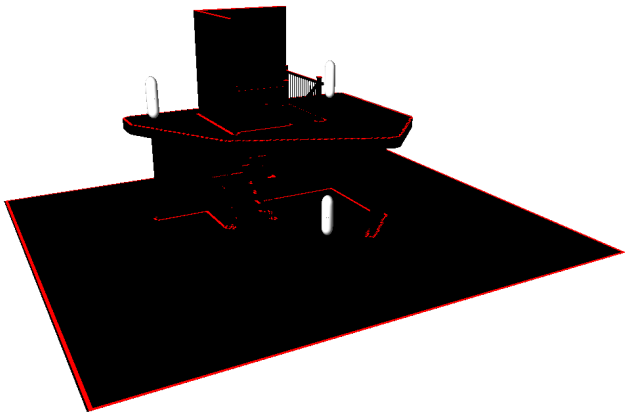


Figure 9: Visualization of the `Discomfort` textures, where the red cells indicate the padding around invalid cells.

This process is sensitive to the size of agents present in the simulation. If different agent groups have varying radii, the padding logic may be adjusted accordingly. For the current implementation, we assume a globally defined agent radius, and apply uniform padding based on that value.

The boundary padding field is written to the `Discomfort` texture array and used throughout the potential and speed computation shaders (Fig. 9). Cells that are assigned as discomfortable are treated as non-walkable and contribute no valid motion or path planning behavior. This technique, while conceptually simple, substantially improves the realism and robustness of the simulation, particularly in dense or tightly structured environments.

4.5 Agent Goal Annotation

To enable agents to navigate meaningfully through the environment, goal regions must be assigned in a way that is both flexible and GPU-compatible. To define these destination regions for agents, we introduce a method that allows the user to place *goal planes*—flat, horizontal meshes—slightly above the ground in locations where agents should eventually arrive (e.g., exits, points of interest). Each goal plane must be assigned a unique `goalID` (with values ≥ 0), identifying it as a distinct goal region. Since these goal meshes are not part of the scene itself, they do not participate in any of the texture arrays involved in physical layer extraction (positions, normals, connectivity, etc.).

During the standard layer extraction step (see Section 4.1), fragments belonging to goal plane meshes are rendered into all standard outputs (e.g., position, grid, depth) to maintain consistent layer structure across texture arrays. However, goal-specific information is handled separately: if a mesh has a valid `goalID`, this identifier is excluded from the standard MRTs and instead written into a dedicated `Goal` texture in a separate compute pass. This separation ensures that goal planes do not interfere with physical surface data while still being properly annotated for agent navigation.

Once the scene layers have been extracted, a separate compute shader processes only the goal planes. For each texel corresponding to a rendered fragment:

(1) The world-space position of the goal plane fragment is projected into the top-down UV space using the same orthographic camera setup used during layer extraction.
(2) The resulting UV coordinate is scaled to texel space, and for each layer (from top to bottom), the shader searches for the first valid walkable texel at the same texture coordinates with a height that is **lower** than that of the goal plane fragment.
(3) Once such a texel is found, the corresponding texel in the goal texture is assigned the `goalID` of the goal plane that was processed.

This method uses the position of the hovering mesh to define the goal *area* in the layer directly beneath it. As a result, the mesh must be placed slightly above the desired goal location so that it lies above (but not too close to) the underlying walkable surface. If the goal plane is placed too close, the target region may be incorrectly marked as non-walkable due to discomfort logic (i.e., it may be interpreted as lying too close to a higher layer). To prevent this, the goal plane must be placed higher than the system's height-difference threshold $h_{\min}$, ensuring correct goal annotation. While this solution is effective within our pipeline, it does require manual tuning and could be replaced by a more robust semantic tagging approach in future work.

This step decouples goal annotation from physical scene representation and ensures accurate mapping between annotated goal regions and the layered environment grid.

Together, these steps produce a complete, GPU-compatible representation of the environment. This structured data—stored in layered texture arrays—serves as the foundation for the simulation pipeline described in Section 5, where agent behavior is computed in parallel using the Continuum Crowds framework. For static environments, the layer extraction and encoding steps can be performed offline, reducing runtime overhead during the simulation and rendering loop.

Summary and GPU Integration

The steps outlined in this section collectively transform complex 3D environments into a structured, layered grid representation full of semantic data essential for navigation and simulation. By encoding scene data into texture arrays and using GPU-friendly operations like depth peeling and compute shaders, we produce a compact, parallel-accessible format that supports real-time performance. Each texel in this representation encapsulates sufficient geometric and semantic context—such as position, walkability, vertical connectivity, discomfort, and goals—for agents to reason about navigation.

This GPU-compatible structure serves as the foundation for the Continuum Crowds simulation framework described in the next section. The design ensures that all major computations—from field generation to agent updates—can be performed efficiently and entirely in parallel on the GPU.

5 GPU-Based Continuum Crowds Simulation

Building on the structured multi-layered scene representation from the previous section, we now proceed to simulate agent movement using the Continuum Crowds framework. This section presents a GPU-based implementation of the algorithm, designed to operate entirely in parallel over the extracted layered grid representation.

Unlike prior work, which has focused either on GPU acceleration in simple 2D environments or on CPU-based simulations in complex 3D spaces, our approach enables real-time, fully GPU-driven simulation in multi-layered 3D environments. This requires several novel adaptations to the Continuum Crowds model: we reformulate the variable field computations to support stacked texture arrays; solve the Eikonal equation using a parallel-friendly method compatible with our grid structure; and update agents efficiently across different height layers, while incorporating discomfort zones and inter-layer connectivity.

We begin by computing a set of dynamic variable fields over the grid, including a density and speed field (Sec. 5.1). These fields form the basis for solving the Eikonal equation (Sec. 5.2), which computes a scalar potential field that guides agents toward their goals via optimal paths. Finally, agents are updated in parallel based on the local gradient of the potential and local crowd conditions (Sec. 5.3). Together, these stages form a fully GPU-accelerated, scalable crowd simulation pipeline capable of handling complex multi-layered environments.

5.1 Variable Field Computation

In the previous section, we described how the environment is transformed into multiple stacks of layered textures using MRTs, where each layer corresponds to a 2D navigable grid and each texture array encodes a specific physical property (e.g., position, normal, walkability). Building on this representation, we now compute the core variable fields required by the Continuum Crowds algorithm on the GPU. These variable fields form the foundation for solving the Eikonal equation and ultimately guide agents through the environment toward their goals.

Just like in the previous section, each field is stored as a layered texture array. Again, each texture array must be configured according to the structure of the data it stores. Scalar fields such as density, average velocity magnitude, discomfort, and potential require only a single channel and are stored in the red (R) component. Directional fields such as speed and unit cost, which store values for each of the four cardinal directions (North, East, South, West), require four components and are stored in RGBA textures. The velocity field, which encodes movement direction, requires at least two channels (i.e. for the x- and z-direction), with an optional third component for the y-direction if vertical displacement is explicitly modeled (see Sec. 5.3).

Table 2 summarizes the texture arrays used for variable field storage.

Output	Format	Description
Density	R	Scalar density field
AVG Velocity	R	Average velocity magnitude
Speed	RGBA	Dir. speed values (N,E,S,W)
Potential (Read)	R	Read buffer for potential
Potential (Write)	R	Write buffer for potential
Velocity	RG/RGB	Agent movement direction vector

Table 2: Texture Arrays for Continuum Crowds Variable Fields

Once these texture arrays are initialized, compute shaders are used to populate them in parallel. Each texel corresponds to a cell in the simulation grid, mirroring the structure used in the layer connection stage (Sec. 4.3). The GPU-friendly architecture ensures that each variable field can be efficiently calculated across the entire environment at once.

The computation follows a specific dependency order:

(1) **Density Field** — Derived from agent presence and distribution, forming the basis for local interactions. Higher density slows down movement, influencing how agents flow through the environment.
(2) **Speed Field** — Computed from the local density and terrain slope. This field defines how fast an agent can move in each direction from a given location and is used as the cost metric in the Eikonal equation.
(3) **Potential Field** — Computed using the Fast Iterative Method (see Section 5.2). The potential field represents the cost to move from each location to a goal, and it is calculated by solving the Eikonal equation using the speed field as input.
(4) **Velocity Field** — Derived from the gradient of the potential field, this vector field determines the movement direction of agents at each texel.

In the following subsections, we describe the GPU-based computation of each of these variable fields in detail, highlighting the specific logic used in the compute shaders and how each field contributes to the overall simulation pipeline.

5.1.1 Density Field. In both real-world and simulated crowd behavior, pedestrians tend to avoid congested areas, as higher local densities lead to slower movement. Consequently, in the Continuum Crowds framework, speed is modeled as a density-dependent variable, and the accurate computation of a *density field* is a critical first step.

The original method [25] involves "splatting" agents onto a grid using simple techniques such as bilinear or Gaussian filters. However, in more complex or fine-grained simulations—particularly those with smaller cell sizes—such naive splatting may introduce visual or numerical artifacts. To address this, Jiang et al. [16] proposed a more robust two-step method for computing density.

First, for each agent i, an influence factor σ is computed, defining the number of grid cells around the agent that are affected by its presence. This is given by:

$$\sigma = \left\lceil \frac{(r_i - l/2)}{l} + 1 \right\rceil \tag{5}$$

depending on radius r_i of agent i and the grid's cell size l. The ceiling function ensures coverage of all neighboring cells within the agent's interaction range.

Secondly, for each cell, they compute a scale factor χ.

$$\chi = \frac{\text{dist}(\mathbf{P}_{\text{agent}}, \mathbf{P}_{\text{cell}})}{l} \tag{6}$$

Here, $\text{dist}(\cdot)$ denotes the Euclidean distance between the agent's position $\mathbf{P}_{\text{agent}}$ (projected to cell height) and the center position of the cell $\mathbf{P}_{\text{cell}}$. In [16], they make a distinction between the distance in x- and y-direction and take the larger of the two and divide it by the cell size. This results in axis-aligned symmetry, where our method gives a more desirable radially symmetric density around the agents.

Finally, the density contributed to a given cell is defined as:

$$\rho = \begin{cases} \dfrac{(1-a)\sin(2\pi X + \pi/2) + (1+a)}{2}, & 0 \le X \le 0.5 \\ a\left[1 - \left(\dfrac{X - 0.5}{\sigma - 0.5}\right)^2\right], & 0.5 < X \le \sigma \end{cases} \tag{7}$$

where a is the adjustment factor that controls the sharpness of the falloff.

In our GPU-based method, this logic is implemented in a compute shader, just like in the previous sections, where each thread is assigned to a single grid cell in the multi-layered representation. Figure 10 visualizes the density field projected onto the scene.

In a GPU implementation, computing density contributions from all agents for every grid cell can become a bottleneck, especially in large-scale simulations. To optimize performance, the algorithm restricts computation to a fixed influence radius $r_{\max}$, considering only agents within that range for each cell.

This localized computation allows each GPU thread to evaluate the density contributions for its assigned cell using only nearby agents, reducing memory access overhead and improving parallel efficiency. Agent data is typically passed via Shader Storage Buffer Objects (SSBOs) (Appendix 7.2), which allow for dynamic updates to agent properties during simulation.

5.1.2 Speed Field. In the Continuum Crowds model, the local speed of a pedestrian is influenced by two competing factors: the underlying terrain (modeled via *topographical speed*) and the local crowd density (modeled via *flow speed*). The *speed field* $f(\mathbf{x}, \theta)$ captures this relation and is used to determine how fast an agent can move from position $\mathbf{x}$ in direction θ. The field is computed per cell and per direction, resulting in a four-component field (for North, East, South, and West) stored in the RGBA channels of the `Density` texture array.

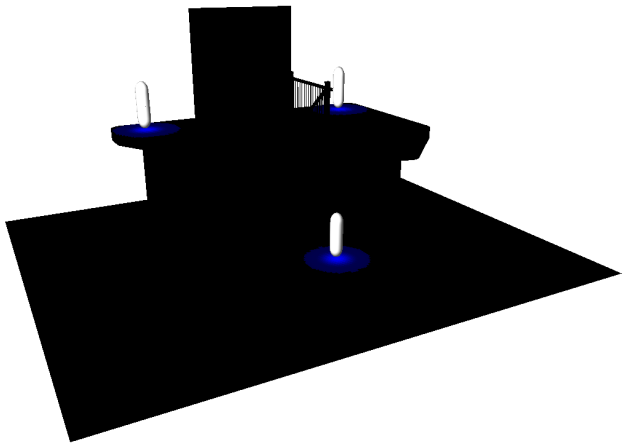


Figure 10: Projection of the `Density` textures onto the scene geometry. Blue regions indicate local agent density, fading with increasing distance from agents. For illustration purposes, we used $a = 0.3$, $r = 0.75$, and $l = 1.0$ — significantly larger than the actual cell size — to highlight spatial variation.

Following the original formulation by Treuille et al. [25], the speed field is defined piecewise based on the density ρ at a neighboring location $\mathbf{x} + r\mathbf{n}_\theta$, where r is a fixed radius offset and $\mathbf{n}_\theta$ is the unit vector in direction θ. This offset ensures that agents do not bias their own speed by self-contribution:

Topographical Speed. When the density at the neighboring point is low (i.e., $\rho \le \rho_{\min}$), the speed is controlled by the slope of the terrain:

$$f_T(\mathbf{x}, \theta) = f_{\max} + \frac{\nabla h(\mathbf{x}) \cdot \mathbf{n}_\theta - s_{\min}}{s_{\max} - s_{\min}} (f_{\min} - f_{\max}) \tag{8}$$

where $\nabla h(\mathbf{x}) \cdot \mathbf{n}_\theta$ is the directional slope (computed using the height difference between neighboring cells), and $f_{\min}$, $f_{\max}$ define the lower and upper bounds for speed.

Flow Speed. When the density is high ($\rho \ge \rho_{\max}$), the speed is determined by the average local flow:

$$f_{\bar{v}}(\mathbf{x}, \theta) = \bar{\mathbf{v}}(\mathbf{x} + r\mathbf{n}_\theta) \cdot \mathbf{n}_\theta \tag{9}$$

where $\bar{\mathbf{v}}$ is the average velocity field evaluated at an offset location and projected onto the direction of motion.

Interpolated Speed. For intermediate densities ($\rho_{\min} < \rho < \rho_{\max}$), the final speed is linearly interpolated between the topographical and flow speeds:

$$f(\mathbf{x}, \theta) = f_T(\mathbf{x}, \theta) + \left(\frac{\rho(\mathbf{x} + r\mathbf{n}_\theta) - \rho_{\min}}{\rho_{\max} - \rho_{\min}} \right) (f_{\bar{\mathbf{v}}}(\mathbf{x}, \theta) - f_T(\mathbf{x}, \theta)) \tag{10}$$

The speed field is computed in a compute shader similar to the density field. The process for computing the speed field in each direction is as follows:

(1) **Walkability Check:** The shader first checks whether the current cell is walkable by evaluating both the `Grid` and `Discomfort` texture arrays at the same texture coordinate. If not walkable or has a discomfort value above a certain threshold, the cell's speed vector is set to `(-1, -1, -1, -1)` and skipped.
(2) **Directional Iteration:** For each of the four directions (North, East, South, West), the connection texture is used to determine the correct layer index of the neighboring cell, which accounts for inter-layer traversal.
(3) **Neighbor Evaluation:** In turn, the neighbor cell's walkability and discomfort are checked. If not valid, the direction is skipped. Otherwise, the algorithm walks through subsequent neighbor cells up to a radius r, checking at each step:
    - Validity of connection index (not `-1`)
    - Walkability and discomfort level

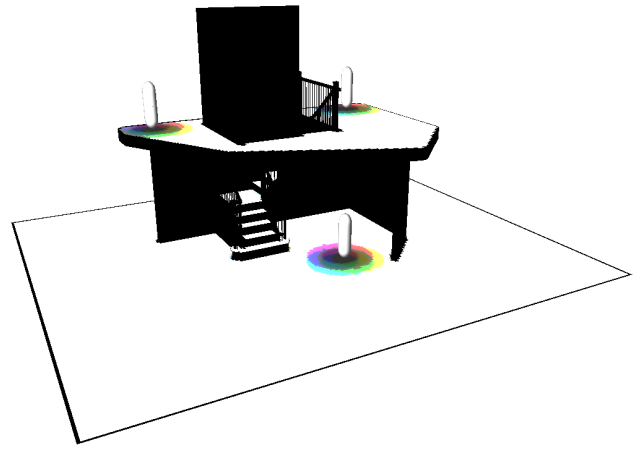
    The last valid cell within the radius is used to sample the density and average velocity required to evaluate flow speed or interpolated speed.
(4) **Speed Computation:**
    - If the sampled density $\rho \leq \rho_{\min}$, compute topographical speed based on slope.
    - If $\rho \geq \rho_{\max}$, compute flow speed by projecting the average velocity onto the direction.
    - Otherwise, linearly interpolate between both speeds.
(5) **Storage:** The computed speed values for all four directions are packed into a `vec4` and stored in the corresponding texel of the `Speed` textures.

This implementation ensures that agent speeds reflect both physical constraints (e.g., slope and walkability) and dynamic factors (e.g., crowding and flow), while supporting multi-layer environments and GPU acceleration. Figure 11 illustrates the computed speed field in two complementary ways. Figure 11a shows the smoothed speed magnitudes projected onto the scene geometry, revealing the influence area around agents. Figure 11b visualizes the dominant speed model (flow vs. topographical) in each direction, with intensity indicating the degree of flow-based influence.

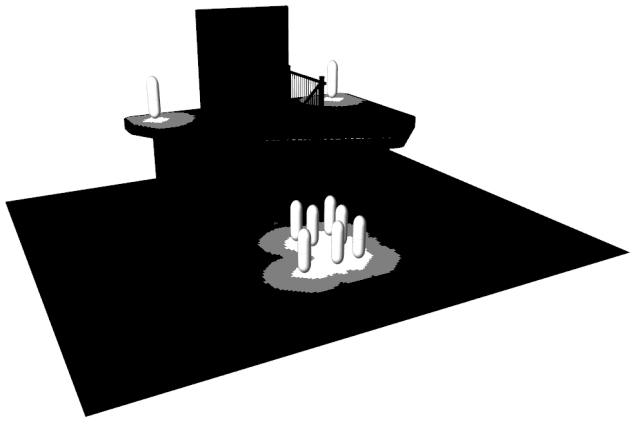
This speed field then forms the basis for solving the Eikonal equation, which we use to compute globally consistent travel times across the environment.

5.2 Solving the Eikonal Equation

The Eikonal equation lies at the core of the Continuum Crowds algorithm, defining the scalar potential field $\phi(\mathbf{x})$ that guides agents toward their goals. In our approach, we solve this equation using the *Fast Iterative Method* (FIM), a GPU-friendly alternative to the traditional Fast Marching Method (FMM). The FIM is well-suited for parallelization, allowing the entire field to be updated efficiently over several iterations.



(a) Projected speed magnitudes



(b) Flow vs. topographical influence

Figure 11: Visualizations of the speed field. (a) Projected speed magnitudes showing smoothed agent influence areas. (b) Classification of directional speed influences: darker regions use terrain-based (topographical) speed, brighter regions use density-based (flow) speed, and intermediate tones represent interpolated influence based on local density.

Prior to solving the Eikonal equation, goal regions are annotated using the method described in Section 4.5. This produces a `Goal` texture array where each texel contains the goal ID it belongs to, or `-1` if it is not part of a goal.

To support multiple agent groups—each with distinct goals—we allocate a separate set of potential textures arrays for each group. Specifically, for each group g, we maintain:

- `PotentialRead[g]`: the potential field from the previous iteration.
- `PotentialWrite[g]`: the output field for the current iteration.

This is needed because, as we update the potential field, we need to both read from and write to the potential texture array. To avoid a-synchronized reading and writing, we separate it into two texture arrays. Then, for each frame, before computing the potential field, the data from `PotentialWrite` is copied to `PotentialRead`, to keep it updated.

To update the potential field, we dispatch a compute shader where each thread corresponds to a single texel. This shader performs two logical phases: an *initialization phase* (on the first frame), and an *iterative update phase* for all subsequent frames.

Initialization Phase. During the first frame, the potential field must be initialized. For each texel, the shader performs the following logic:

(1) If the texel is not walkable (as determined by the `Grid` textures) or its discomfort value exceeds a specified threshold, the potential is set to ∞.
(2) If the texel's goal texture contains the current group ID g, it is marked as a goal texel and assigned $\phi = 0$.
(3) All other texels are initialized with $\phi = \infty$.

Iterative Update Phase. On subsequent frames, the shader performs the core FIM update:

(1) For each texel, skip if not walkable or marked discomfortable.
(2) If the current texel's previous potential value is 0.0 (i.e., a goal), retain this value.
(3) Otherwise, gather values from the four cardinal neighbors using the `Connection` textures.
(4) For each valid neighbor:
    - Fetch the previous potential value.
    - Compute the movement cost using:
    $$c_i = \frac{\alpha \cdot s_i + \beta + \gamma \cdot d_i}{s_i}$$
    where s_i is the speed from the `Speed` texture array and d_i is the discomfort value.
(5) We find the neighbors with the least costly adjacent cell along both the x- and y-axes:
$$m_x = \arg\min_{i \in \{W,E\}} \{\phi_i + C_{M \to i}\} \quad m_y = \arg\min_{i \in \{N,S\}} \{\phi_i + C_{M \to i}\}$$
(6) Use these upwind directions to solve for the larger solution to phi_M for grid cell M in the quadratic equation
$$\frac{(\phi_M - \phi_{M_x})^2}{C_{M \to m_x}} + \frac{(\phi_M - \phi_{M_y})^2}{C_{M \to m_y}} = 1 \tag{11}$$
(7) Write the resulting ϕ_M value to the `PotentialWrite` texture.

This iterative approach enables efficient propagation of values from the goal areas, as all valid texels are updated in parallel. There is no need for centralized coordination or prioritization of updates—the wavefront of ϕ values naturally expands outward over successive iterations (Fig. 12).

This implementation enables real-time updates of the potential field directly on the GPU, allowing agents to respond continuously to changes in terrain, density, and discomfort without centralized coordination. The single-iteration-per-frame approach ensures stability while preserving responsiveness in complex, dynamic environments

Although the compute shader can theoretically be run multiple times per frame to accelerate convergence, we observed that a single iteration per frame produces stable results and avoids visual artifacts such as agent jitter. Running multiple iterations per frame introduced instability, likely due to a combination of factors: the asynchronous nature of the Fast Iterative Method (FIM), the absence of convergence control between passes, and potential race conditions from parallel writes in GPU memory. These effects can lead to non-smooth updates in the potential and velocity fields, which manifest as flickering or oscillatory agent behavior. By limiting the solver to a single pass per frame, we ensure gradual, stable propagation of the potential field while maintaining real-time performance.

In practice, each frame advances the potential field incrementally, while the agent update step (see Section 5.3) ensures that crowd motion is responsive to evolving environmental conditions. As the density and discomfort fields change, the speed field and thus the potential field adapt accordingly.

After each update of the potential field ϕ, a velocity field is derived to guide agent motion. This is done with the following steps:

(1) Sample the scalar potential ϕ at the current texel and its four cardinal neighbors (left, right, top, bottom), using the `Connection` texture to ensure valid sampling between connected layers.
(2) Compute the gradient $\nabla\phi$ using finite differences:
$$\nabla\phi = \left(\frac{\phi(x+1,y) - \phi(x-1,y)}{2}, \frac{\phi(x,y+1) - \phi(x,y-1)}{2} \right)$$
(3) Normalize and negate the gradient to obtain the velocity direction:
$$\mathbf{v}(x,y) = -\frac{\nabla\phi}{\|\nabla\phi\|}$$
(4) Scale by the local speed $s(x,y)$ from the `Speed` texture array:
$$\mathbf{v}(x,y) = s(x,y) \cdot \mathbf{v}(x,y)$$
(5) Store the result in the `Velocity` texture array.

This process yields a per-texel velocity vector pointing in the steepest descent direction of the potential field, scaled by the agent's allowed movement speed at that location. The resulting velocity field serves as the basis for agent motion in the next simulation step (Fig. 13).

5.3 Agent Movement and Update

After computing the navigation fields, each agent must update its position based on the velocity field derived from the potential gradient. This is performed entirely on the GPU using a compute shader, where each thread is responsible for updating a single agent. The agents' transformation matrices and velocity vectors are stored

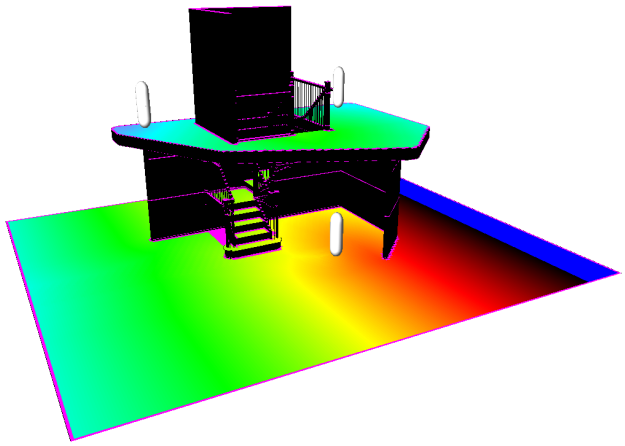


Figure 12: Projection of the `Potential` field onto the scene geometry. The blue region marks the agents' goal, from which the potential propagates outward. The color gradient visualizes this propagation in the order: black → red → yellow → green → cyan → magenta. Invalid or unreachable cells are highlighted in purple.

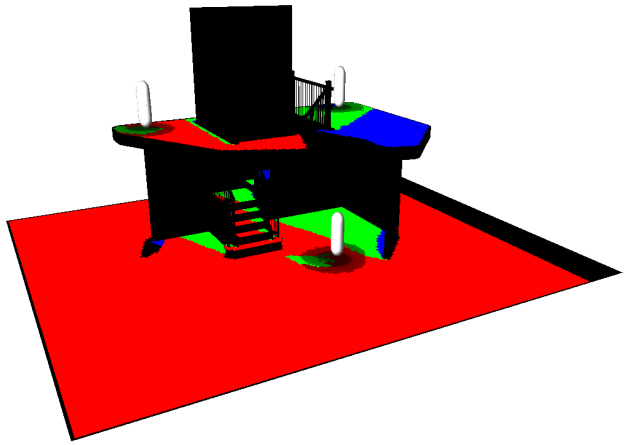


Figure 13: Projection of `Velocity` field. Red is positive x-direction, blue is positive z-direction, green is negative x-direction.

in Shader Storage Buffer Objects (SSBOs) to avoid data transfer overhead between GPU and CPU.

Due to the multi-layered structure of the environment, determining the correct texel that corresponds to an agent's position is non-trivial. The method begins by projecting the agent's world-space position into the orthographic camera space (used during layer extraction) to compute the 2D texture coordinates:

(1) The agent's position $\mathbf{p}$ is transformed by the top-down view-projection matrix to get the projected coordinate $\mathbf{p}_{\text{proj}}$.
(2) Perspective division is applied: $\mathbf{p}_{\text{proj}} \leftarrow \mathbf{p}_{\text{proj}} / \mathbf{p}_{\text{proj}}.w$.
(3) The normalized texture UV coordinates are obtained using: $\text{texUV} = 0.5 \cdot \mathbf{p}_{\text{proj}}.xy + 0.5$.

To resolve the appropriate layer index, all layers are iterated. For each layer, the corresponding grid cell is checked for:

- **Walkability:** Verified using the `Grid` texture array.
- **Height proximity:** The absolute height difference between the agent's current height and the sampled height from the `Position` texture array.

The layer with the smallest valid height difference is selected as the one the agent currently occupies. The resulting 3D texture coordinate (including layer index) is then used to sample navigation fields such as the velocity and potential.

Once the agent's texture coordinate is resolved, the corresponding velocity is fetched from the `Velocity` texture array. This velocity encodes directional influence derived from the potential field (see Section 5.2). The agent's position is then updated using an explicit Euler integration step:

$$\mathbf{p}_{\text{new}} = \mathbf{p}_{\text{old}} + \mathbf{v} \cdot \Delta t \tag{12}$$

Here, Δt is the frame time passed sent from the CPU, and $\mathbf{v}$ is the sampled velocity at the agent's current texel.

For updating the vertical component (height), a straightforward solution is employed. Instead of estimating vertical movement velocity (which can be noisy and cause agents to float above or sink below the surface), the new Y-coordinate is directly sampled from the `Position` texture array using the updated 2D coordinates and re-running the layer-matching projection function. The final Y-position is then corrected by adding a model-specific height offset (e.g., the distance from the ground to the model's center).

The updated agent position and velocity are written back to the corresponding SSBOs:

- **Position:** Stored in the translation component of each agent's transformation matrix.
- **Velocity:** Stored in a separate velocity SSBO, to be reused for calculating average velocity in subsequent simulation steps (see Section 5.1.2).

By keeping all agent data and navigation logic entirely on the GPU, the system supports real-time, large-scale crowd simulations with minimal latency and no CPU-GPU synchronization overhead.

This fully GPU-driven pipeline—from field generation to agent movement—ensures responsiveness to dynamic environments while maintaining high performance across complex, multi-layered scenes.

6 Results

We evaluated the performance and effectiveness of our proposed method across several test scenarios. This section presents key results from those evaluations, highlighting the combined effects of the hybrid approach.

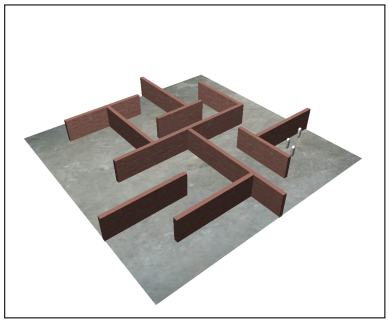
All experiments were conducted on a desktop machine equipped with an Intel Core i7-7700HQ CPU (2.80GHz), 8 GB of RAM, and an NVIDIA Quadro M1200 GPU with 4096 MB of video memory. The system was implemented in C++ using OpenGL, with a custom

engine responsible for things like rendering, shader orchestration, GPU resource management, simulation control, and a lightweight entity–component system. This engine enabled efficient integration of multi-pass rendering and compute shader stages, forming a flexible foundation for the GPU-accelerated crowd simulation framework. Standard support libraries were used where appropriate, such as for windowing, user interface, and texture loading.

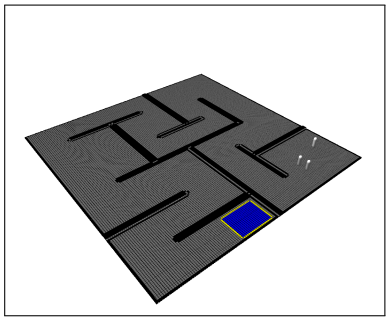
While the test machine is not a state-of-the-art desktop, relying on a mobile CPU and a mid-range GPU, the relative performance difference between our method and the original approach remains highly significant. The results reported here are therefore conservative estimates: on modern, high-end hardware, both methods would achieve higher absolute frame rates, but the performance gap in favor of our method is expected to be even more pronounced.

To ensure a fair comparison, all tests were performed in static environments. This consistency was crucial for fair benchmarking, allowing us to compare both methods under nearly identical environmental conditions. Most evaluations were carried out using Scene 3, the maze (Figure 14), with a 3D version used for our method, and an identical 2D version used for the original method, making it ideal for direct comparison.

The results focus on several performance metrics, including frame rate (FPS), compute time, memory consumption, and the completeness of captured data.



(a) 3D version



(b) 2D version

Figure 14: Scene 2 (maze) as a 3D (left) and 2D (right) representation, to compare results for our method and the original method.

6.1 Scene Representation Quality and Generalization

One of the core strengths of our method lies in its ability to robustly represent a wide variety of 3D environments using a unified texture-based grid system. Thanks to several key additions—such as multi-angle layer extraction (Section 4.2) and automatic padding around non-walkable regions (Section 4.4)—the system is capable of accurately capturing complex spatial layouts, including overhangs, stairs, and multi-level geometry.

In practice, this means that for a wide range of scenes—including the staircase (Scene 1), the maze (Scene 2), and the villa (Scene 3, Figure 15)—the full set of texture arrays (e.g., `Position`, `Grid`, `Potential`, etc.) can be computed automatically without any manual intervention. As long as the camera positions and grid resolution are chosen appropriately, the system is able to extract all necessary spatial and semantic data from the 3D scene. This allows seamless initialization of the simulation pipeline and makes the method highly generalizable across different types of environments.

This level of automation and robustness is essential for scaling the method to more diverse and dynamic 3D scenes in the future, and it significantly reduces the setup overhead compared to traditional approaches that require manual tagging, segmentation, or precomputed navigation data.

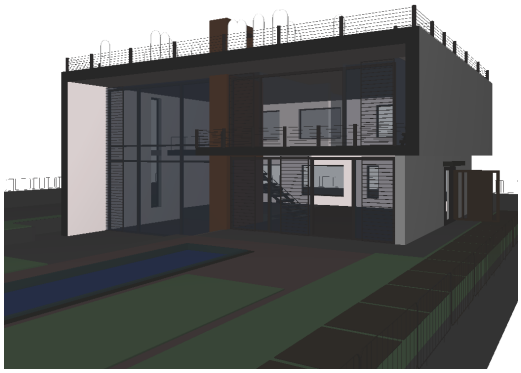


Figure 15: Scene 3: a multi-story villa.

6.2 Scalability with Agent Count

To evaluate how well our method scales with increasing numbers of agents, we measured the frames per second (FPS) across a range of agent counts in Scene 2 with a 200×200 cell grid size. The agent counts tested were: 5, 10, 50, 100, 200, 500, 1000, and 2000. Figure 16 compares the FPS achieved by our GPU-based method with the original CPU-based implementation, plotted on a logarithmic scale due to the significant performance gap.

Our method maintains high frame rates even with large numbers of agents. Starting from ~380 FPS with 5 agents, the frame rate only gradually declines to around 99 FPS with 2000 agents—still comfortably real-time. Even with 10.000 agents, our method would still run at 25 FPS. In contrast, the original method shows a steep

decline: from 3.8 FPS with 5 agents to less than 0.1 FPS with 2000 agents.

This exponential divergence in performance is primarily due to the way the original method computes the density field: it processes each cell sequentially on the CPU and, for every cell, calculates the distance to every agent. As the number of agents increases, this nested loop structure becomes increasingly costly. In contrast, our method leverages GPU compute shaders and texture arrays to parallelize the update process. Rather than having the CPU iterate over every cell and agent combination, our approach assigns a thread to each agent and computes its contribution to the surrounding field in a single pass. This massively reduces redundant computations and significantly lowers the per-agent overhead, enabling real-time performance even with thousands of agents.

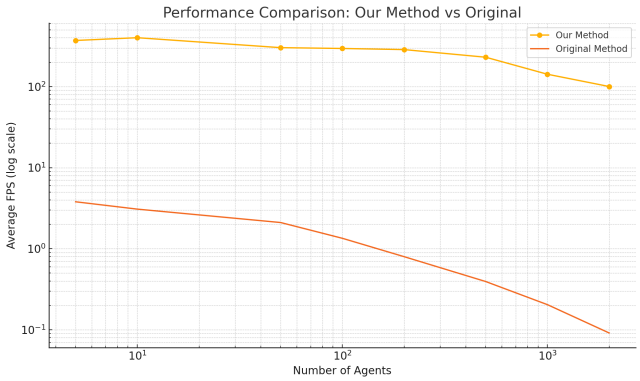


Figure 16: Comparison of FPS vs. number of agents (logarithmic scale) between the original CPU-based method and our GPU-based method.

6.3 Scalability with Grid Resolution

To evaluate the scalability of both methods with respect to scene resolution, we measured the average frame rate (FPS) across various grid sizes while keeping the number of agents fixed at 50. The tested grid sizes were 50×50, 100×100, 200×200, 500×500, and 1000×1000. Results are shown in Figure 17.

Our method consistently outperformed the original method across all resolutions. While the original CPU-based implementation already struggles at 200 × 200 with an average FPS of approximately 2.4, our method maintains well over 300 FPS at the same resolution. Even at 1000 × 1000, our GPU-based method runs at more than 200 FPS, making it viable for real-time simulation at high resolutions.

Similar to the results in the previous subsection, the original method becomes impractical at high resolutions due to its sequential, cell-by-cell updates and the need to compute distances to every agent for each cell. This leads to a dramatic slowdown as the grid size increases.

In contrast, our method leverages GPU parallelism to update the entire field concurrently, resulting in significantly higher performance. However, for grid sizes beyond 1000 × 1000, performance begins to degrade slightly. This is primarily because the number of

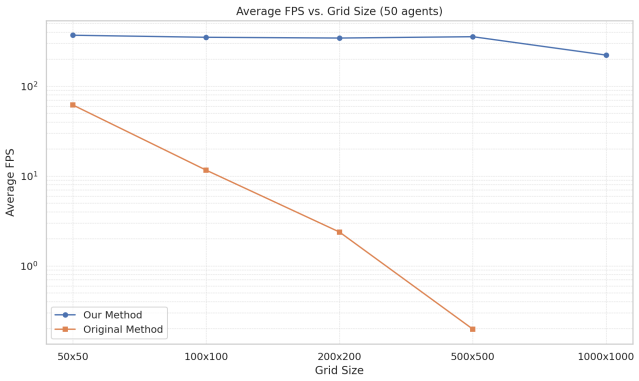


Figure 17: Comparison of FPS vs. grid size (logarithmic scale) between the original CPU-based method and our GPU-based method.

grid cells exceeds the number of threads that can execute in parallel, requiring individual threads to process multiple cells sequentially. Additionally, as texture sizes grow, memory bandwidth becomes a limiting factor—reading from and writing to large textures increases pressure on the GPU's memory subsystem, which can become a bottleneck.

Still, these results make our technique far more suitable for large-scale crowd simulations in complex 3D environments, where higher resolution is often required to accurately model narrow passages, stairs, or layered terrain.

6.4 Effect of Multi-Angle Views on Obstacle Detection

As described in Section 4.2, we introduced a method for identifying vertical fragments using additional front and side views during the depth peeling stage. To evaluate the effectiveness of this technique, we measured the number of discomfort cells detected with and without these additional perspectives.

Across all tested scenes, incorporating the front and side layers consistently resulted in more comprehensive discomfort coverage. Table 3 summarizes the difference for three representative scenes: the staircase (Scene 1, used in the rest of the report), the villa (Scene 2), and the maze (Scene 3). The improvement is especially notable in Scene 3, where the number of discomfort cells more than doubles.

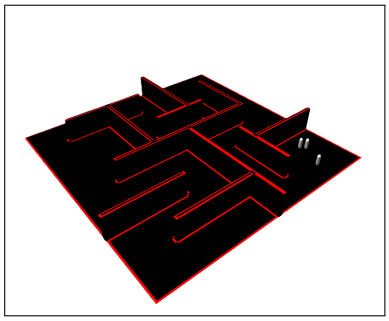
Table 3: Number of discomfort cells in different scenes with and without front/side layers

Configuration	Scene 1	Scene 2	Scene 3
Without front/side layers	1514	3029	1813
With front/side layers	1897	3431	3682

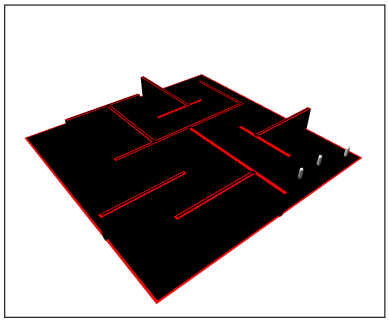
Figure 18 provides a visual comparison. On the left, the additional views allow the system to correctly tag the space beneath walls and near vertical structures as discomfort zones. Without these

layers (right), those areas remain untagged, resulting in incomplete obstacle representation.

This omission has practical implications. When computing the `Potential` field, walls that lack proper discomfort tagging do not sufficiently repel the propagation of the field. As shown in Figure 19, this allows the field to extend beneath walls—resulting in agents that may appear to walk through solid geometry. The inclusion of multi-angle layer extraction effectively eliminates this unrealistic behavior.



(a) With extra layers



(b) Without extra layers

Figure 18: Comparison of `Discomfort` textures with and without multi-angle (front and side) layers.

6.5 GPU Memory Consumption

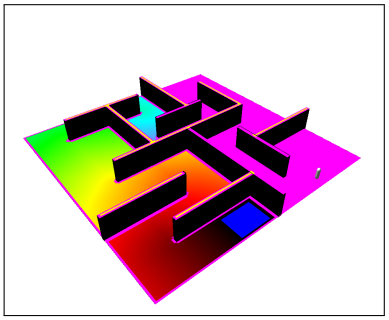
Our method relies heavily on GPU-resident 3D texture arrays, which are accessed and updated during various shader passes. Since GPU memory (VRAM) is typically more limited than system RAM, it's essential to quantify and manage the memory footprint of these textures.

Each floating-point value occupies 4 bytes. Therefore, a single texel in a texture is:

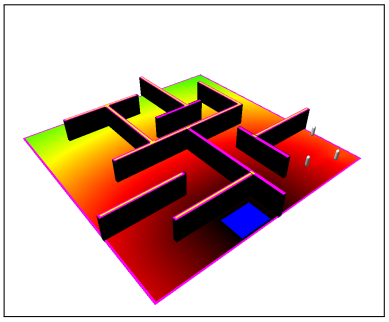
$$\text{Texel Size (bytes)} = 4 \times \text{Number of Channels}$$

To compute the total memory usage of a 3D texture array, we multiply the size of a single texel by the total number of texels across the array's width, height, and depth (layers), as shown in Equation 13.

$$\text{Memory} = 4 \times \text{channels} \times \text{width} \times \text{height} \times \text{layers} \tag{13}$$



(a) With extra layers



(b) Without extra layers

Figure 19: Comparison of `Potential` fields. Without discomfort tagging from side/front layers, walls do not block propagation effectively.

Tables 1 and 2 summarize the texture formats used in our implementation. For example, the `Position` texture uses an RGB format, meaning each texel occupies $4 \times 3 = 12$ bytes. Given a grid resolution of 200×200 and 10 layers, the total memory usage for this texture would be:

$$12 \times 200 \times 200 \times 10 = 4.8\,\text{MB}$$

Summing the contributions from all textures listed in Tables 1 and 2 for a $200 \times 200 \times 10$ grid results in approximately **38 MB** of total GPU memory usage. In the context of modern GPUs, which typically provide 8–24 GB of VRAM, this memory requirement is negligible.

However, as grid resolution increases, so does memory usage. For instance, a grid of 2000×2000 with the same number of layers results in approximately **3.8 GB** of texture memory. While this is still manageable on contemporary GPUs, it highlights the importance of efficiency and possible future optimizations, especially when scaling to very large environments or supporting multiple concurrent simulations.

When comparing this to the memory usage of the original method, we can estimate the footprint by measuring the size of a single Cell object, which is approximately 250 bytes. Multiplying this by the number of grid cells ($200 \times 200 \times 10$) results in roughly 100 MB of CPU-side memory.

Although this is significantly more than the 38 MB used in our GPU-based approach, the higher memory footprint is less problematic in practice, as modern CPUs typically have substantially more available RAM than GPUs. Therefore, the impact of this increased usage is limited — though it may still influence performance when scaling to very large environments or high agent counts.

6.6 Summary

The results across all test scenarios clearly demonstrate the advantages of our proposed method over the original CPU-based implementation. In particular, we observe major improvements in speed, scalability, and automation, all while maintaining low memory usage and high environmental fidelity.

Key findings include:

- Achieves real-time performance with 2000 agents on a 200× 200 cell grid running at over 99 FPS.
- Offers a significant and scalable speed-up over the original method, demonstrating improvements ranging from 5× to over 1000× depending on grid resolution and agent count.
- Reduces GPU memory usage to well under 100 MB for standard grids.
- Improves obstacle detection via multi-angle layer extraction.
- Generalizes effectively across a range of complex 3D environments with minimal user annotation.

Together, these results highlight the method's suitability for real-time, large-scale crowd simulation in diverse and geometrically complex environments.

7 Discussion and Conclusion

This thesis has presented a novel GPU-based extension to the Continuum Crowds algorithm, designed to support scalable, real-time crowd simulation in complex, multi-layered 3D environments. By integrating a range of GPU-accelerated techniques—including depth peeling, compute shaders, and layered texture arrays—the method achieves a significant leap in performance, flexibility, and generalization compared to the original CPU-bound formulation.

7.1 Discussion

The results have demonstrated that the system is highly scalable both in terms of agent count and grid resolution. It consistently maintains real-time performance even in scenarios where the original method becomes unusable. These gains are largely due to the use of parallel compute shaders and GPU-friendly data structures, which allow each stage of the simulation pipeline—field computation, Eikonal solving, and agent updates—to be fully parallelized.

One of the most notable strengths of the proposed system is its generalization capability. Thanks to the automatic scene representation pipeline—including multi-angle layer extraction and boundary padding—the system can ingest arbitrary 3D environments and extract all necessary simulation data without manual annotation. This makes it applicable not just to synthetic benchmarks but to real-world architectural spaces as well.

While the proposed method offers clear performance advantages and automation benefits, several challenges and limitations remain that should be acknowledged.

Goal Area Definition. The approach we used to define the goal region—through a hovering mesh plane captured in the depth peeling stage—proves functional, but it is not without caveats. Because the discomfort detection relies on geometric relationships between layers, placing the goal mesh too close to the underlying ground can trigger false positives in marking the discomfort zones. Specifically, the area beneath the goal mesh may be marked as "too close to the previous layer" and thus treated as non-walkable. This prevents the potential field from propagating outward, effectively disabling goal attraction in those regions.

A workaround was to position the goal mesh higher than the system's height-difference threshold. While effective, this solution requires scene-specific tuning and reduces generalizability. Future work could explore more robust and semantically-driven ways to define goal areas—e.g., tagging goal regions using metadata or incorporating post-processing steps to override local discomfort logic.

Implementation Complexity. It is also important to note that, despite its performance benefits, the proposed GPU-based system introduces significantly more complexity compared to traditional CPU-based crowd simulation methods. Creating and managing numerous GPU texture arrays, synchronizing shader dispatches, and maintaining consistent data formats across rendering and compute pipelines require advanced graphics programming knowledge. In contrast, conventional CPU implementations typically involve straightforward data structures (e.g., 2D arrays of cell objects) and are easier to understand, debug, and extend.

This added complexity may pose a barrier for adoption in smaller projects or educational contexts. However, we argue that this trade-off is justified in scenarios where real-time performance and large-scale simulations are critical. In such cases, the performance gains and memory savings demonstrated in this work offer a compelling incentive to invest in the GPU-based approach.

Scene Update Cost. Although the layered scene representation provides rich geometric information for agent navigation, its construction relies on Depth Peeling, which requires multiple rendering passes. In this implementation, scene extraction is performed offline to avoid runtime cost. However, in use cases involving dynamic environments—such as destructible geometry or moving obstacles—this step would need to be repeated during simulation, reducing the system's overall FPS advantage. Future work could explore more efficient layer extraction methods, such as Bucket Depth Peeling or Adaptive Depth Peeling [18], or investigate incremental update strategies to better support dynamic scenes without full reprocessing.

Memory usage. While memory usage remains relatively low for standard grid sizes, very high resolutions (e.g., beyond 2000×2000) can approach the limits of GPU memory, potentially requiring optimization strategies such as sparse textures or other methods of compression. Additionally, while the method handles vertical navigation via layered connections, it currently assumes static geometry and does not yet support dynamic obstacles or moving platforms.

In terms of implementation, the system depends heavily on hardware capabilities. Although performance is excellent even on mid-range GPUs, some operations—such as multi-pass depth peeling or

large-scale texture updates—may still benefit from more powerful hardware.

7.2 Conclusion

In summary, this thesis contributes the first fully GPU-based implementation of Continuum Crowds adapted for complex, multi-layered 3D environments. It:

- Offers up to three orders of magnitude speed-up over the original CPU method.
- Maintains real-time performance for thousands of agents.
- Scales effectively with grid size and agent count.
- Requires minimal user input due to automated scene processing.
- Accurately handles multi-level traversal and obstacle avoidance.

These advancements make the method highly suitable for applications ranging from game development and robotics simulations to large-scale virtual environments.

Future work may focus on dynamic environment support, further performance optimization (e.g., texture streaming), and integration with reinforcement learning or semantic scene understanding to enhance agent behavior. Still, as it stands, the system already represents a robust and practical step forward in real-time crowd simulation for complex environments.

References

[1] [n. d.]. David s2 v1 - Download Free 3D model by Arek @ Vonka Stairs (@A.Vonka) — sketchfab.com. https://sketchfab.com/3d-models/david-s2-v1-96f19c33b3f343a383d02fbcfa048368. [Accessed 25-08-2025].

[2] Stanley Bak, Joyce McLaughlin, and Daniel Renzi. 2010. Some improvements for the fast sweeping method. *SIAM Journal on Scientific Computing* 32, 5 (2010), 2853–2874.

[3] Miles Detrixhe, Frederic Gibou, and Chohong Min. 2013. A parallel fast sweeping method for the Eikonal equation. *J. Comput. Phys.* 237 (2013), 46–55.

[4] Elmar Eisemann and Xavier Décoret. 2006. Fast scene voxelization and applications. In *Proceedings of the 2006 symposium on Interactive 3D graphics and games*. 71–78.

[5] Cass Everitt. 2001. Interactive order-independent transparency. *White paper, nVIDIA* 2, 6 (2001), 7.

[6] Tian Feng, Lap-Fai Yu, Sai-Kit Yeung, KangKang Yin, and Kun Zhou. 2016. Crowd-driven mid-scale layout design. *ACM Trans. Graph.* 35, 4 (2016), 132–1.

[7] Paolo Fiorini and Zvi Shiller. 1998. Motion planning in dynamic environments using velocity obstacles. *The international journal of robotics research* 17, 7 (1998), 760–772.

[8] Tor Gillberg. 2011. A semi-ordered fast iterative method (SOFI) for monotone front propagation in simulations of geological folding. In *MODSIM2011, 19th International Congress on Modelling and Simulation*. 641–647.

[9] Stephen Gustafson, Hemagiri Arumugam, Paul Kanyuk, and Michael Lorenzen. 2016. Mure: fast agent based crowd simulation for vfx and animation. In *ACM SIGGRAPH 2016 Talks*. 1–2.

[10] Stephen J Guy, Min Kim, Ming C Lin, and Dinesh Manocha. 2012. Clearpath: Highly parallel collision avoidance for multi-agent simulation. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. 177–187.

[11] Dirk Helbing, Illés Farkas, and Tamas Vicsek. 2000. Simulating dynamical features of escape panic. *Nature* 407, 6803 (2000), 487–490.

[12] Dirk Helbing and Peter Molnar. 1995. Social force model for pedestrian dynamics. *Physical review E* 51, 5 (1995), 4282.

[13] Yuhao Huang. 2021. Improved fast iterative algorithm for eikonal equation for GPU computing. *arXiv preprint arXiv:2106.15869* (2021).

[14] Won-Ki Jeong and Ross T Whitaker. 2008. A fast iterative method for eikonal equations. *SIAM Journal on Scientific Computing* 30, 5 (2008), 2512–2534.

[15] Hao Jiang, Wenbin Xu, Tianlu Mao, Chunpeng Li, Shihong Xia, and Zhaoqi Wang. 2009. A semantic environment model for crowd simulation in multilayered complex environment. In *Proceedings of the 16th ACM Symposium on Virtual Reality Software and Technology*. 191–198.

[16] Hao Jiang, Wenbin Xu, Tianlu Mao, Chunpeng Li, Shihong Xia, and Zhaoqi Wang. 2010. Continuum crowd simulation in complex environments. *Computers & Graphics* 34, 5 (2010), 537–544.

[17] Kwon Lee, Jinxiang Chai, Pieter S Reitsma, Jessica K Hodgins, and Nancy S Pollard. 2007. Group behavior from video: A data-driven approach to crowd simulation. *ACM Transactions on Graphics (TOG)* 26, 3 (2007), 1–9.

[18] Fang Liu, Meng-Cheng Huang, Xue-Hui Liu, and En-Hua Wu. 2009. Efficient depth peeling via bucket sort. In *Proceedings of the Conference on High Performance Graphics 2009*. 51–57.

[19] Tomas Moller and Ben Trumbore. 1997. Fast, minimum storage ray-triangle intersection. *Journal of Graphics Tools* 2, 1 (1997), 21–28.

[20] Rahul Narain, Ariel Golas, Sean Curtis, and Ming C Lin. 2009. Aggregate dynamics for dense crowd simulation. In *ACM Transactions on Graphics (TOG)*, Vol. 28. ACM, 122.

[21] Jan Ondřej, Julien Pettré, Anne-Hélène Olivier, and Stéphane Donikian. 2010. A synthetic-vision based steering approach for crowd simulation. *ACM Transactions on Graphics (TOG)* 29, 4 (2010), 1–9.

[22] S. Poddar, C. Mavrogiannis, and S. S. Srinivasa. 2023. From crowd motion prediction to robot navigation in crowds. In *2023 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 6765–6772.

[23] Craig W Reynolds. 1987. Flocks, herds and schools: A distributed behavioral model. In *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*. 25–34.

[24] James A Sethian. 1999. Fast marching methods. *SIAM review* 41, 2 (1999), 199–235.

[25] Adrien Treuille, Seth Cooper, and Zoran Popović. 2006. Continuum crowds. *ACM transactions on graphics (TOG)* 25, 3 (2006), 1160–1168.

[26] Branislav Ulicny and Daniel Thalmann. 2002. Towards interactive real-time crowd behavior simulation. In *Computer Graphics Forum*, Vol. 21. Wiley Online Library, 767–775.

[27] Wouter Van Toll, Roy Triesscheijn, Marcelo Kallmann, Ramon Oliva, Nuria Pelechano, Julien Pettré, and Roland Geraerts. 2016. A comparative study of navigation meshes. In *Proceedings of the 9th International Conference on Motion in Games*. 91–100.

[28] Hongkai Zhao. 2005. A fast sweeping method for eikonal equations. *Mathematics of computation* 74, 250 (2005), 603–627.

Appendix A: Compute Shaders in GLSL

Modern graphics hardware enables not only traditional rendering tasks but also general-purpose parallel computation through *compute shaders*. In contrast to vertex and fragment shaders, which are tied to the graphics pipeline, compute shaders allow for more flexible, high-throughput parallel processing on the GPU.

A compute shader is a GLSL shader stage that runs independently of the traditional rendering pipeline. It allows arbitrary read and write access to textures, buffers, and images, and is ideally suited for tasks that benefit from *massive data parallelism*, such as simulations, image processing, or physics computations.

In this project, compute shaders are used extensively for:

- Preprocessing connectivity information, discomfort values and agent goal areas.
- Computing variable fields like density and speed fields.
- Calculating potential fields (Fast Iterative Method) and velocity fields.
- Update agent movements.

A compute shader is launched over a 1D, 2D, or 3D grid of workgroups. Each *workgroup* contains a fixed number of *local invocations*, defined with the `layout(local_size_x = ..., ...)` directive. Each shader thread can access its unique global invocation ID using:

```
ivec3 texCoord = ivec3(gl_GlobalInvocationID);   (14)
```

This maps one thread to one texel (or voxel) in a texture array, enabling massively parallel computations such as updating the potential field or checking neighboring cells.

Compute shaders use `imageLoad` and `imageStore` operations to directly read and write to textures:

```
imageStore(myTexture, texCoord, vec4(value));
vec4 value = imageLoad(myTexture, texCoord);
```

This is crucial for updating intermediate outputs such as `potentialWrite`/`poten` `gridTextures`, and `discomfortTextures`. In contrast, read-only access (e.g., for previous potential values or normals) typically uses `texture` or `texelFetch` functions.

Although compute shaders support memory synchronization (e.g., `barrier()`, `memoryBarrier()`), this implementation generally avoids such requirements. Each thread works independently on a texel, and no intra-group coordination is necessary in the majority of cases.

This approach allows for highly parallelized, real-time simulation of crowd behavior in complex, layered environments.

Appendix B: Shader Storage Buffer Objects (SSBOs)

Shader Storage Buffer Objects (SSBOs) are OpenGL buffer objects that allow shaders to read and write large blocks of data. Unlike Uniform Buffer Objects (UBOs), SSBOs support more flexible and dynamic storage, making them suitable for storing per-agent state like transformation matrices or velocity vectors. They enable efficient GPU-to-GPU data access across different shader stages, reducing the need for CPU intervention and improving simulation performance in parallel compute workflows.