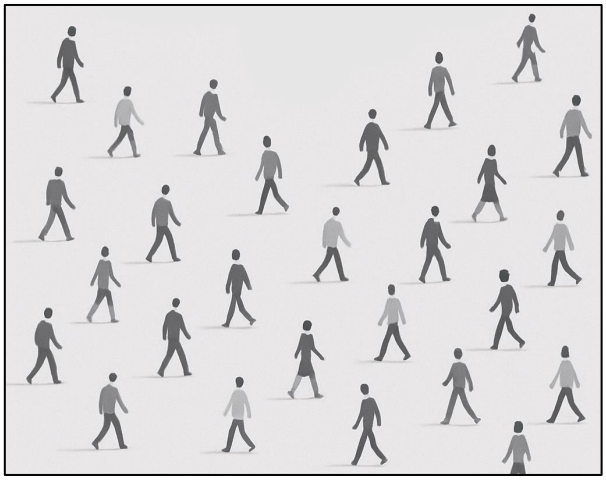
Crowd simulation

Oihan Abruña, Guillermo Núñez, Sebastian Paas and Iker Quintana



DigiPen
INSTITUTE OF TECHNOLOGY
EUROPE - BILBAO

1. Introduction

Crowd simulation is the study and modeling of the movement, behavior, and interactions of large groups of agents in a common environment. These simulations are widely used in multiple fields such as urban planning, animation, virtual reality, emergency evacuation modeling, and video game design.

Crowd simulations can be classified as **macroscopic** or **microscopic**. In macroscopic simulations, the crowd is treated as a continuous flow, focusing on shared properties instead of individual behaviors. Microscopic simulations treat each individual as a discrete, autonomous agent with its own goals, decision-making processes, and reactions to nearby agents or obstacles.

In microscopic simulations, **global navigation** and **local navigation** can be distinguished. Global path planning involves calculating an optimal route for each agent to reach its destination while taking environmental static obstacles into account. Local navigation handles real-time movement decisions in dynamic environments. This includes avoiding collisions with other agents, responding to crowd density or following other agents. Local navigation algorithms must balance realism, accuracy, and computational efficiency.

This paper focuses primarily on microscopic simulation techniques, particularly those related to local navigation. Exploring how agents avoid collisions and adapt to changing densities reveals how complex crowd behaviors can emerge from simple individual rules.

In particular, the paper aims to explore and compare multiple collision avoidance algorithms, analyzing their underlying principles, strengths, limitations, and suitability for different simulation scenarios.

2. Goal-directed steering

In crowd simulations, **goal-directed steering** refers to the process by which an agent navigates toward a specific target location. This is achieve using **seek behavior**, where the agent calculates a desired velocity pointing directly toward the goal and adjusts its current velocity based on it. The difference between the desired and current velocity produces a steering force that guides the agent smoothly toward the destination.

Although this mechanism is conceptually simple, it provides the foundation for more advanced navigation techniques. It establishes the basic locomotion pattern upon which additional behaviors such as collision avoidance are built.

Listing 1 Function to apply steering towards a desired velocity

```
Vec2 ApplySteering(Vec2 currentVel, Vec2 desiredVel,
float maxAcc, float dt)
{
    Vec2 acc = desiredVel - currentVel;
    float accLimit = maxAcc * dt;

    if (acc.Size() > accLimit)
        acc = accel.Normal() * accLimit;

    return currentVel + acc;
}
```

3. Collision avoidance

Collision avoidance is a crucial component of local navigation in crowd simulations. It refers to techniques that allow individual agents to navigate shared spaces without colliding with each other or other obstacles. It involves perceiving nearby obstacles and adjusting the movement to avoid potential collisions. This process must be executed in real-time and often under computational limitations, especially in large-scale simulations.

Various algorithmic approaches have been developed to address the collision avoidance problem, ranging from simple rule-based models to more mathematically sophisticated techniques. Each method differs in its assumptions about agent perception, prediction, and decision-making, resulting in **trade-offs between realism, computational cost, and scalability**.

In the following sections, we present an analysis and comparison of several collision avoidance algorithms.

4. Force based collision avoidance

Force based collision avoidance is a modeling approach where each agent is treated like a **physical particle** that experiences artificial forces. These forces guide their movement:

1. An **attractive force** pulls agents toward their goal or destination.
2. **Repulsive forces** push them away from static obstacles and other agents to avoid collisions.

This concept is inspired by Newtonian mechanics but adapted to simulate intelligent motion in agents like pedestrians or robots.

Universal Power Law

Figure 1: Universal Power Law in action.

The *Universal Power Law* [5] [4] model introduces a new idea: people try to **avoid spending unnecessary energy** when walking in crowds. They react more strongly to imminent collisions, and barely at all to those that might happen later, or not at all. It relies on the psychological idea that we react only to threats that are likely to occur in the near future.

The model introduces a concept called **time-to-collision** (τ), which measures how long it will take before two agents collide with each other, assuming they keep walking in the same direction and speed.

- If τ is small, the predicted collision is soon, people react with strong avoidance.
- If τ is large (more than 3 seconds), the collision may never happen, so it is usually ignored.

The interaction energy between pedestrians follows the following rule: as a potential collision gets closer in time, the avoidance force increases dramatically, but beyond a few seconds (about 2–3s of reaction time), agents stop reacting completely.

Examples

- Lane formation among people walking in opposite directions.
- Clogging near doorways (choke point).
- Synchronized movement in crowds without clear goals (like flocking).

Time to Collision
Given two agents with relative position $dp = p_j - p_i$ and relative velocity $dv = v_j - v_i$, and assuming both agents maintain their current velocity, the time to collision τ is computed as:

$$\tau = \frac{(-dp \cdot dv)}{|dv|^2} \tag{1}$$

- If $\tau < 0$, they are moving away from each other.
- If τ is very large or infinite, a collision is not expected.

Interaction Energy
This function defines how two agents interact. Energy increases sharply as τ gets smaller:

$$E(\tau) = \frac{k}{\tau^2} \cdot e^{-\tau/\tau_0} \tag{2}$$

where:
- k: A strength constant.
- τ_0: The time decay factor (around 2–3 seconds).

Avoidance Force
The avoidance force is computed as the gradient of the energy function:

$$F_{ij} = -\nabla_{p_{ij}} E(\tau) \tag{3}$$

At each simulation step, τ is computed by linearly projecting the paths of the agents i and j using their current velocities. A collision is considered to occur at a time $\tau > 0$ if the circular areas representing the pedestrians, with radii $R_i$and R_j , overlap.

$$F_{ij} = -\left[\frac{ke^{-\tau/\tau_0}}{|v_{ij}|^2\,\tau^2}\left(\frac{2}{\tau} + \frac{1}{r_0}\right)\right]\left[v_{ij} - \frac{|v_{ij}|^2\, r_{ij} - (p_{ij} \cdot v_{ij})\, v_{ij}}{\sqrt{(p_{ij} \cdot v_{ij})^2 - |v_{ij}|^2\left(|p_{ij}|^2 - (R_i + R_j)^2\right)}}\right] \tag{4}$$

This ensures that agents are pushed in the right direction to avoid imminent collisions efficiently.

5. Velocity based collision avoidance

Velocity obstacle collision avoidance algorithms and its descendants like *reciprocal obstacle avoidance (RVO), optimal reciprocal collision avoidance (ORCA)* and other hybrid algorithms were originally developed to avoid collisions between robots in the real world.

These algorithms focus on a microscopic approach in which each robot analyses the environment and makes its own decision based on his analysis. But the lack of coordination between the agents means that agents can make mistakes in their predictions and thus fail to cooperate properly. This was a **common issue** in the original velocity obstacle avoidance algorithm (*VO)* because each agent expected the others to be mindless objects traveling at constant velocity.

Reciprocal obstacle avoidance (RVO) would improve on this. *RVO* assumes that agents use the same decision process and will do half the work needed to avoid a collision. Each frame the algorithm analyses again the environment and refines its direction to avoid a collision. This iterative approach solved the problems with *VO*. However, *RVO* struggled with determining the closest and best approach vector. This can lead to agents oscillating and not finding the best side to dodge an obstacle.

The *ORCA* algorithm takes this into account and makes it a central insight for collision avoidance. *ORCA* has established itself in the gaming industry because it is relatively cheap to compute and can be easily combined with *A** to result in believable and performant navigation.

Optimal Reciprocal Collision Avoidance (ORCA)

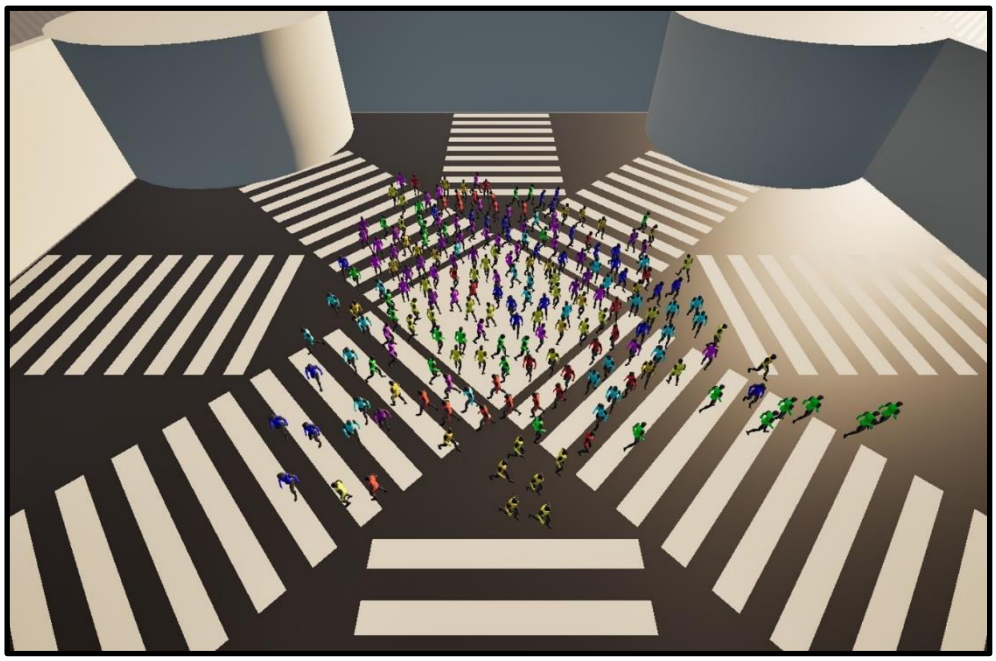


Figure 2: ORCA in action.

The ORCA [5] algorithm enables agents to move toward their goals while avoiding both static and dynamic obstacles through a three-step process:

1. First, it calculates the set of velocities that would result in a collision, known as the velocity obstacle (VO).
2. Second, it selects the optimal velocity that lies outside all velocity obstacles and is as close as possible to the agent's preferred velocity.
3. Finally, it updates the agent's velocity to this new, safe one.

For simplicity, agents and static obstacles are **modeled as 2D circles**. Each agent has knowledge of the positions and radii of all other agents, with the position representing the center of the agent's circle. This shared information allows agents to independently compute velocity constraints and avoid collisions in a decentralized and efficient manner.

Velocity Obstacle

A Velocity Obstacle (VO) is a construct in velocity space that defines **all relative velocities between two agents that would result in a collision** within a given time frame (the time horizon). If an agent's relative velocity lies within this region, it implies that the agents are on a path that will bring them too close to each other, typically, within a distance equal to the sum of their radii.

The VO can be visualized in 2D space as a cone (or triangle) emanating from the origin with a rounded peak. It represents all the velocities that will result in the moving circles (agents) intersecting within τ seconds, assuming constant velocities.

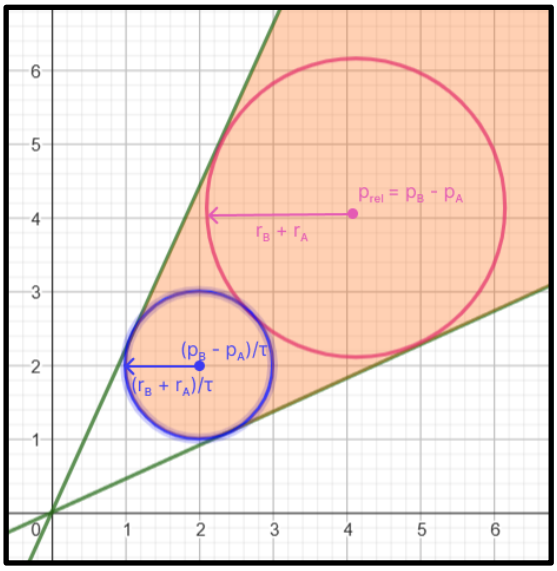


Figure 3: The velocity obstacle between agent A and B.

Computing the Velocity Obstacle
To compute the velocity obstacle between two agents, A and B, the following data is required:

- p_A, p_B: The positions of agent A and B.
- v_A, v_B: Their current velocities.
- r_A, r_B: Their radii.
- τ: The time horizon for planning.

From this, the lower end of the velocity obstacle (blue circle) and the trajectory of it (pink circle) that define the area of the velocity obstacle can be computed.

- Circle defining lower end: $center = \frac{(p_B - p_A)}{\tau} \quad radius = \frac{(r_B + r_A)}{\tau}$
- Circle defining trajectory: $center = p_B - p_A \quad radius = r_B + r_A$

Collision Checking with the Velocity Obstacle
Once the VO is constructed, we must determine whether agent A's current velocity will lead to a collision. We do so by calculating the relative velocity of B to A and test whether the point, the resulting vector is pointing too, is inside the velocity obstacle.

Checking a collision between a point and the two circles, that define the velocity obstacle, is straight forward. To determine whether the point is inside the velocity obstacle but outside the triangle, we can project the distance from the outer circle to the point and project that vector onto the closest edge.

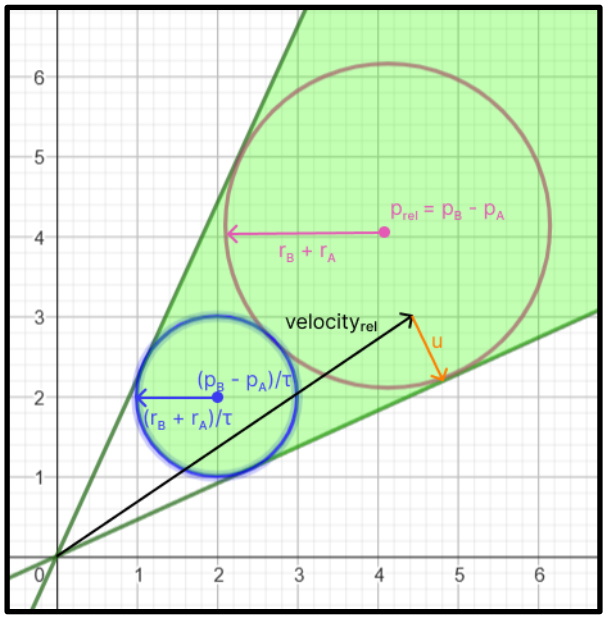


Figure 4: Collision resolution with the velocity obstacle.

Finding a safe velocity

Once we have determined that the agent is on a collision course, we need to find the velocity that will prevent it. The collision between a circle and point is simple to resolve:

$$direction = point - circleCenter$$
$$u = |direction| * (radius - |direction|) \quad (5)$$

u is the **smallest change necessary to prevent a collision**. However, if we change the velocity by u, we will only prevent the collision with one other agent and the new velocity might not be the best possible velocity to get to the target destination.

To take these restrictions into account we can define a half-plane of all the safe velocities for each agent we are on a collision course with. The direction of such a half-plane is perpendicular to u and close to the optimal velocity to the destination.

Both agents are expected to make half the change in direction to avoid a collision. For that reason, we only move half of u away from the optimal velocity. The half plane is defined by the direction and a point where n is the direction and p is a point on the line:

$$n = | u^{\perp} |$$
$$p = v_{optimal\ velocity} + u * \frac{1}{2} \quad (6)$$

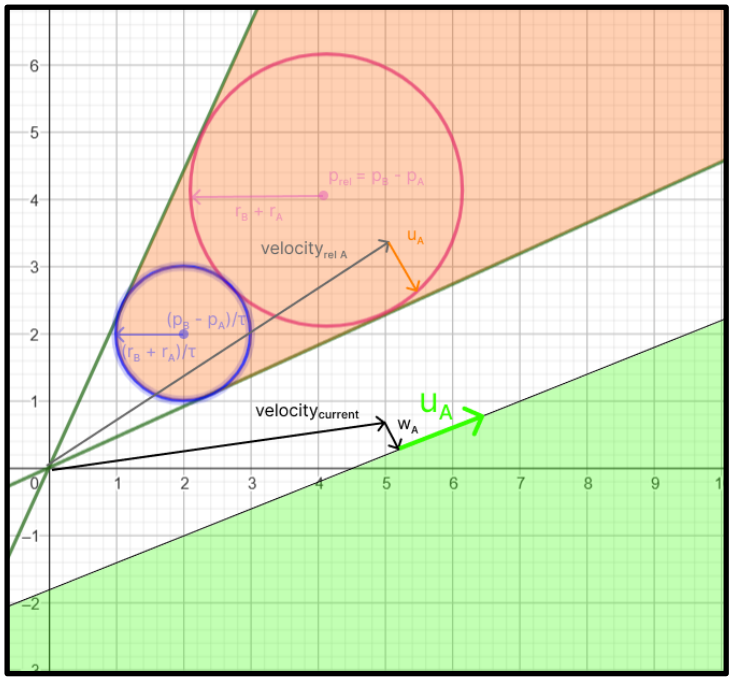


Figure 5: The space of safe velocities defined by a half plane.

The space of safe velocities is then further refined by combining the half planes from the velocity obstacles of all other agents.

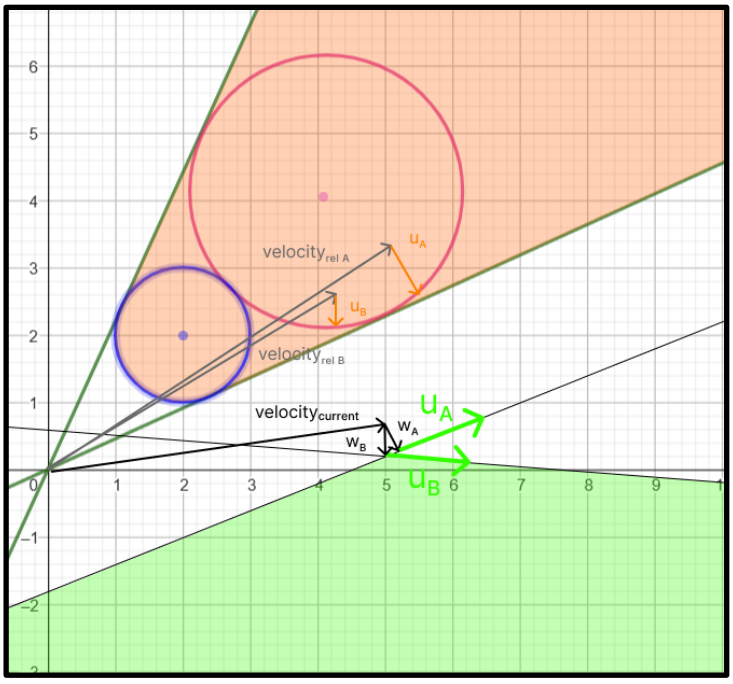


Figure 6: The space of safe velocities defined by two half planes.

Now we can find the best velocity to avoid the collision by finding the intersection between the lines of the half planes. We might find that depending on the velocity of other agents no solution is possible, or the half planes are pointing in the opposite direction. In these situations, we want to find the closest point**.**

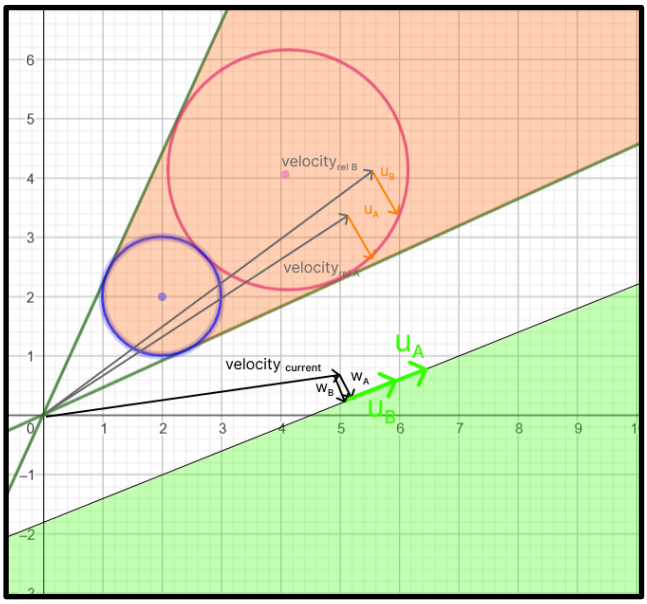


Figure 7: Two orca lines can be parallel or the same line.

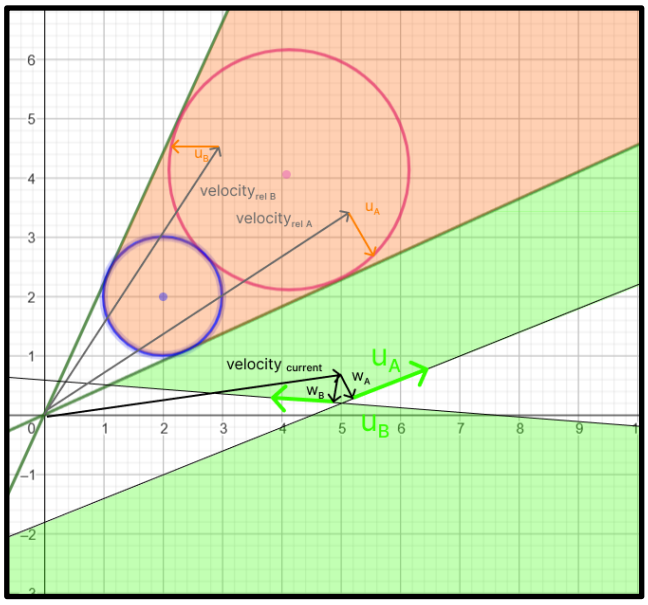


Figure 8: Two orca lines that oppose each other.

If we have one half plane with a point at $Plane_A$ with $Direction_A$ and another at $Plane_B$ with $Direction_B$ then we can find the intersection with the following:

$$denominator = Direction_A \times Direction_B$$

$$numerator = Direction_B \times (Plane_A - Plane_B)$$

$$safe\ velocity = Plane_A + \frac{numerator}{denominator} * Direction_A \quad (7)$$

Static obstacles
The edges of static obstacles like walls or pillar are defined by lines. For ORCA we **split them into points with a defined radius**.

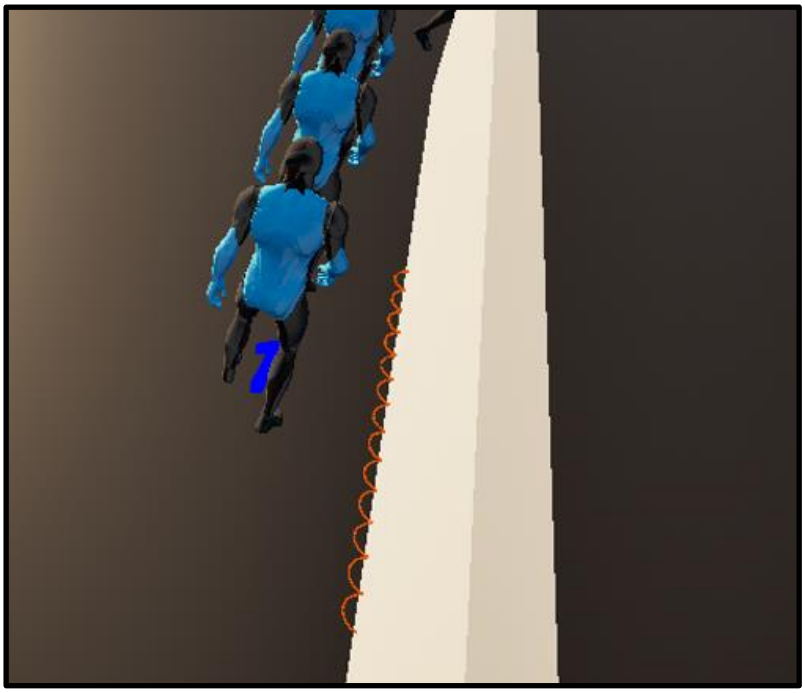


Figure 9: Wall colliders are split into smaller discrete circles.

We can handle these circles as we handle moving agents. The only difference is that an agent cannot expect the static obstacle to change its velocity. Therefor the only change to the previous formular is in the calculation of the point on the half plane. Instead of moving half of u away from the optimal velocity we move the complete distance of u away from it:

$$p = v_{optimal\ velocity} + u \quad (8)$$

ORCA, however, cannot handle path finding and agents get stuck on walls and pillars if the lines of the collider are perpendicular to the optimal velocity to the destination. They do not collide with them but also fail to find a way around them. This issue only arises if multiple points of a static obstacle are blocking the way. To fix this issue we used circular colliders for these edge cases.

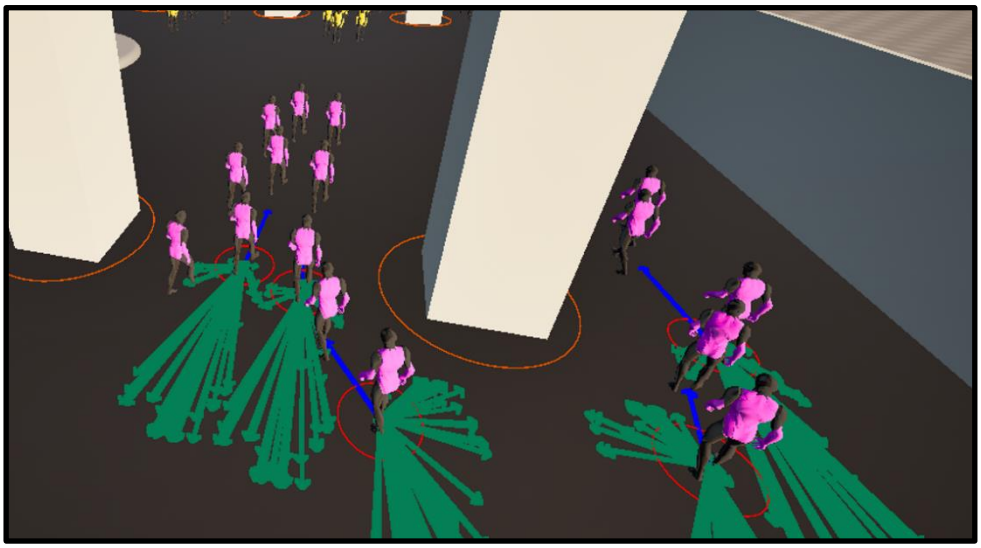


Figure 10: Adding circular colliders around pillars prevents agents from getting stuck on them.

Limitations

*A** or a similar pathfinding algorithm should be used in a game to find a path around static obstacles. ORCA should only consider static obstacles close to the agent to improve performance but still ensure local collision avoidance.

6. Grid based collision avoidance

Grid-based collision avoidance refers to algorithms that operate on a **discretized representation of the environment**, where the world is divided into a uniform grid of cells. Each cell encodes information about occupancy, navigability, or dynamic cost, allowing agents to make movement decisions by examining a limited neighborhood of adjacent cells. Unlike velocity-based approaches that rely on continuous motion and prediction, grid-based methods evaluate movement over discrete time and space steps, offering high spatial clarity and ease of implementation.

Grid-based approaches offer a more structured and deterministic framework, often at the cost of lower realism and less smooth motion. Resolution and update frequency are key challenges. Coarse grids can lead to choppy movement or deadlocks, while fine grids increase computational demands. Despite this, grid-based collision avoidance remains a robust choice in domains like cellular simulations, tactical AI, and tile-based games, where environmental structure and performance predictability are essential.

Cellular automata

Cellular automata [7] are computational models composed of a grid of cells, each of which can exist in a finite number of states. The state of each cell evolves over discrete time steps according to a local update rule, which considers the current state of the cell and the states of its neighboring cells. This simple rule-based evolution allows complex global behavior to emerge from purely local interactions. Each frame an agent has the option to move to one of neighboring cells, it will choose which cell to move to next depending on the Euclidian distance to the goal.

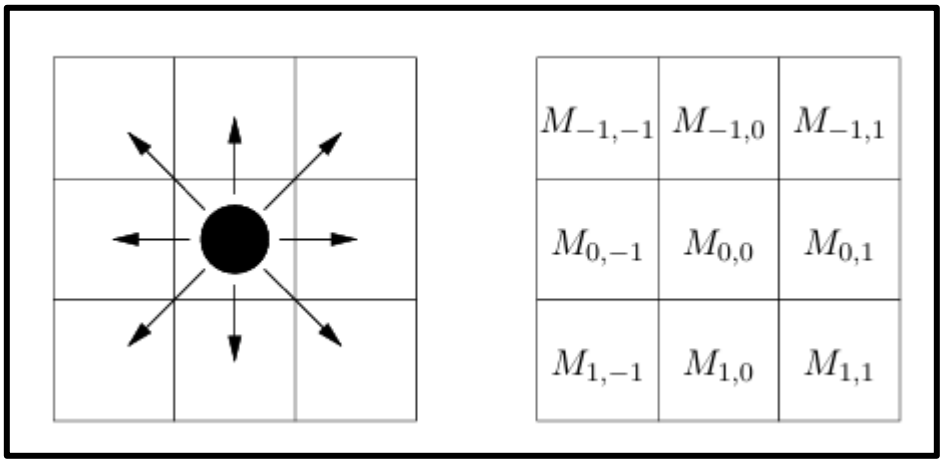


Figure 11: Agent neighbor cell selection.

Cellular automata rules

The rules for cellular automata define how each cell updates its state from one time step to the next based on its local environment. These rules are applied uniformly across the grid, meaning every cell follows the same logic. The rules used on the basic implementation of cellular automata for this project are:

- Each cell can either be empty or occupied by an agent or an obstacle.
- When an agent attempts to move to a neighbor cell, this cell becomes occupied before the next agents evaluate their neighbor cells.
- On each frame the 8 neighbor cells of each agent will evaluate the current state of their own neighbors to determine if a change of state is required.

Cell neighbors' rule

As mentioned in the previous section on the last rule, the current evaluated cell can change its stage temporarily depending on the state of the neighbor cells. This rule draws inspiration from models like **Conway's Game of Life**, where similar neighbor-based transitions are used to simulate population growth and decay. In the Game of Life, for example, a dead cell becomes alive if it has exactly 3 live neighbors. On this project it's the opposite, a cell becomes occupied/death if it has other X number of neighbors occupied. This rule allows agents to choose cells which are more probable to have more options for future movement.

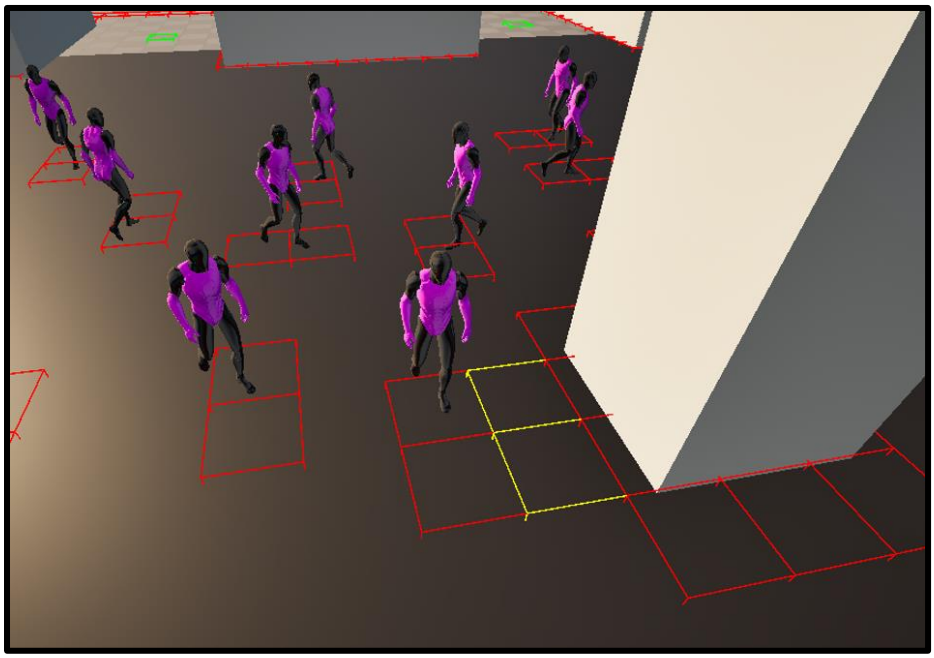


Figure 12: Yellow cells being temporarily occupied.

Cellular automata floor field
The floor field model is an extension of cellular automata used to simulate pedestrian movement and further enhance the way agents interact with the environment to reach their goal. The model introduces virtual fields overlaid on the grid to guide agents toward their goals. Each agent still moves based on the local rules described on the previous sections, but their decisions are influenced by values in these floor fields. There are two types of floor fields: **static** and **dynamic**.

Static floor field
The static floor field encodes the shortest distance from each cell to a target, such as an exit or a goal position in this case. The floor field is precomputed in this project as a list of distance each with an index which corresponds to the grid cell and the list does not change over time. Cells closer to the goal have lower values, and agents prefer moving to adjacent cells with lower static field values.

Dynamic Floor Field
The dynamic floor field is updated each frame and reflects the recent movement history of agents. When an agent moves, it increases the value of the dynamic field in the cell it leaves. Over time, this field decays or diffuses. Normally the dynamic floor field is used to attract other agents to the paths that other agents have followed creating a herd behavior, but on this project it was used the opposite way to entice agents to avoid other agents paths and thus avoid collisions between agents more regularly.

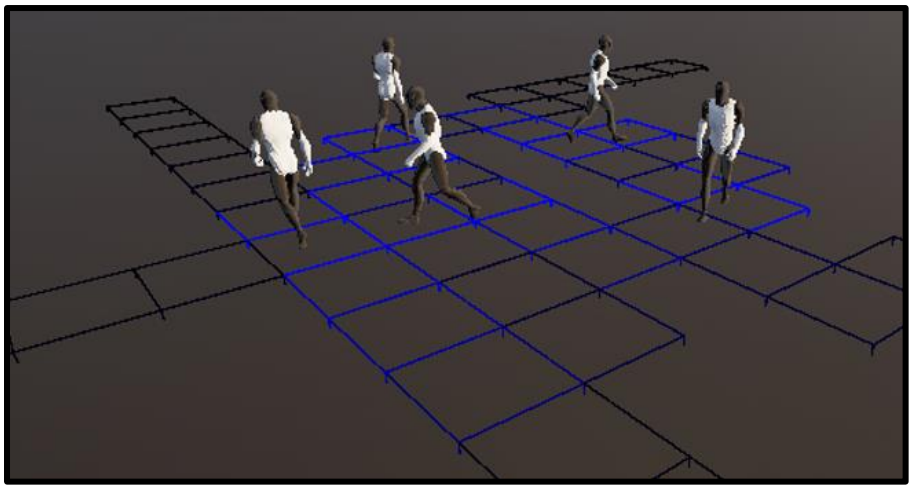


Figure 13: Agent dynamic field paths, the bluer the more recent.

To calculate which is the best neighbor cell each frame for the agents, using floor field, instead of the Euclidean distance the equation used is the following:

$$b = \mathrm{S} + \mathrm{w} * \mathrm{D}$$

where:

- b: The current score of the neighbor cell that is being evaluated.
- S: The value that the cell has in the Static floor field list.
- D: The value the cell has currently on the Dynamic floor field list.
- w: The weight applied to the dynamic floor field value in the equation, a lower weight means agents will prioritize moving towards the goal rather than avoiding other agents paths.

Other rules
During the implementation of cellular automata in this project other rules were applied to obtain a better result regarding collision avoidance. This rules involve certain penalties to the score previously mentioned above. The penalties are:

- If the neighbor cell is a diagonal cell.
- If the cell has more than Y amount of occupied neighbor cells.

The last penalty differs from the rules mentioned before in the sense that it does not mark the cell as occupied since it has less than X amount of neighbor cells being occupied, this penalty still allows the agent to move to this cell if there are no better options available, but by applying this penalty it discourages the agent to take this option if there is a better alternative with this penalty not being applied to that better cell option.

Static obstacles
Static obstacles, represented as line segments, are mapped to grid cells using the *Bresenham* algorithm.

Listing 2 Function to pass static obstacles from line segments to occupied cells.

```
void PlotLineBresenham(FIntPoint p0, FIntPoint p1, int
resolution)
{
     int x0 = p0.X;
     int y0 = p0.Y;
     int x1 = p1.X;
     int y1 = p1.Y;

     int dx = FMath::Abs(x1 - x0);
     int dy = FMath::Abs(y1 - y0);
```

```
     int sx = x0 < x1 ? 1 : -1;
     int sy = y0 < y1 ? 1 : -1;

     int err = dx - dy;

     while (true)
     {
          if (x0 >= 0 && x0 < resolution && y0 >= 0 && y0 <
resolution)
          {
               int index = CellIndex(x0, y0, resolution);
               Grid[index] = CellState::occupied;
          }

          if (x0 == x1 && y0 == y1) break;

          int e2 = 2 * err;
          if (e2 > -dy) { err -= dy; x0 += sx; }
          if (e2 < dx) { err += dx; y0 += sy; }
     }
}
```

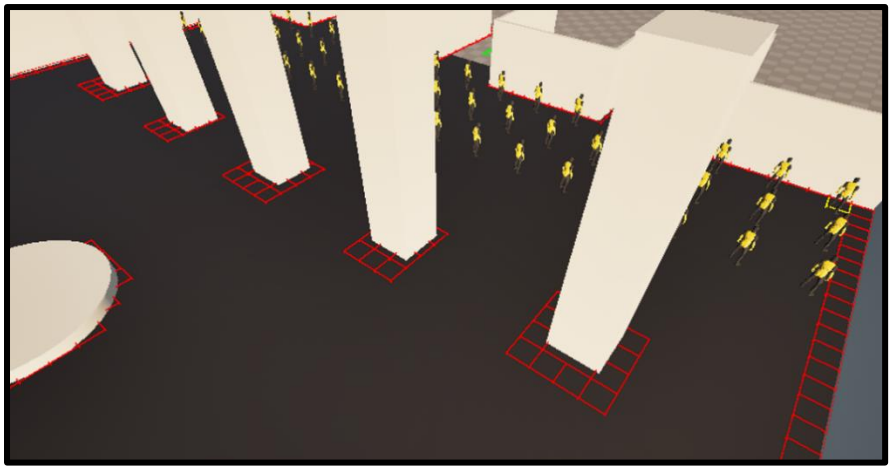


Figure 14: Static obstacles as occupied grid cells.

7. Detailed collision prediction

Most traditional collision-avoidance algorithms work under the **assumption that agents will continue along linear, straight-line paths** in the future. These algorithms often assume that the current velocity of agents is representative of their future motion and thus base their motion prediction on the assumption of constant velocity. This approach becomes inaccurate in more dynamic settings where agent behavior can change frequently.

More advanced algorithms address the limitations of this linear trajectory assumption by incorporate more sophisticated motion models that account for changes in speed, direction, and intent over time, enabling more accurate prediction of agent movement and more effective avoidance of potential collisions with dynamic obstacles.

Probability based collision avoidance (WarpDriver)

The approach suggested by *WarpDriver* [2] constructs a collision probability field around each agent, which represents the likelihood of colliding with other agents across space and time. Given these fields, local collision avoidance strategies can be executed using gradient descent techniques. This means an agent can move away from areas of high collision probability, selecting safer paths.

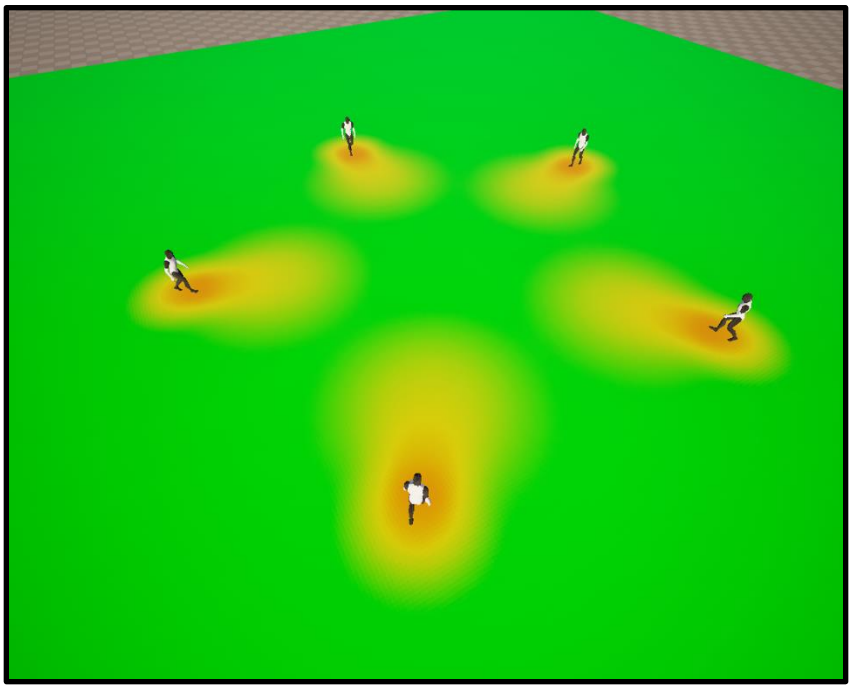


Figure 15: Agent probability fields.

To construct the probability field that represents the potential for collision several properties are considered.

Position

The strength of the probability field is highest near the agent's position and decreases with distance. This distribution is modeled using a *Gaussian* function centered at the agent's location.

$$P(d) = e^{\left(-\frac{d^2}{2\sigma^2}\right)} \quad (9)$$

$$d = \| r - \mu \| \quad (10)$$

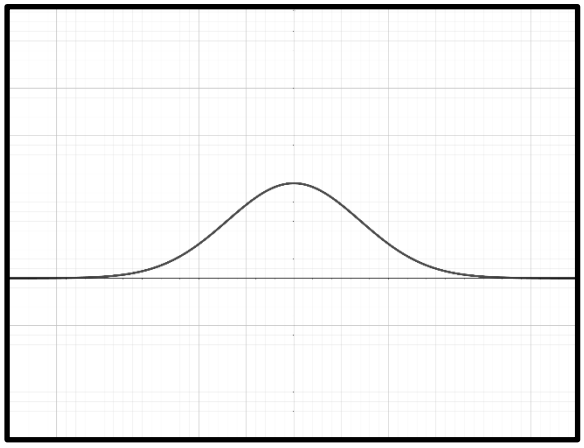


Figure 16: Probability field function.

where:

- $r = (x, y)$: The point in space where the probability is being evaluated.
- μ : The agent's position.
- d : The distance between the point and the agent.
- σ : The spread or decay rate of the probability field.

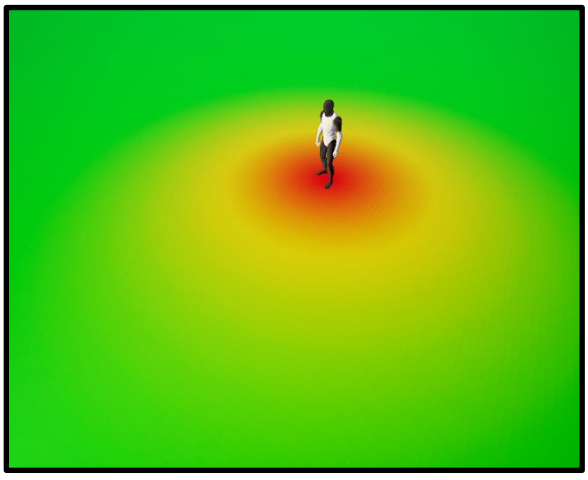


Figure 17: Probability field with σ = 200.

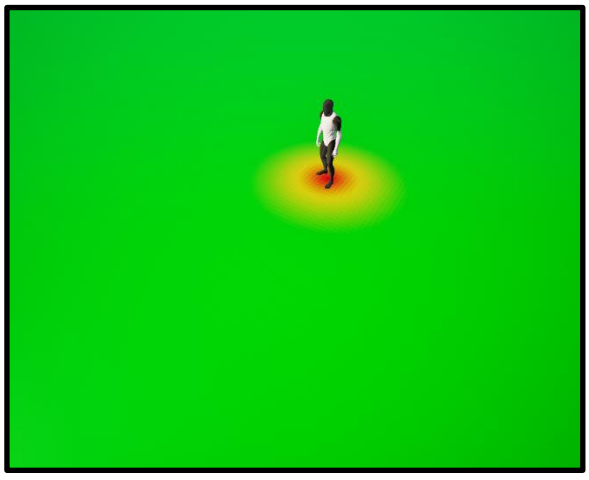


Figure 18: Probability field with σ = 60.

Size

Each agent is modeled as a cylindrical entity with a defined radius, representing its physical occupancy in the environment. The radius defines the minimum area around the agent that is considered occupied and at high risk of collision. Any point within this radius is assumed to have a high probability of being inside the agent's physical boundary.

$$d = \| r - \mu \| - R \tag{11}$$

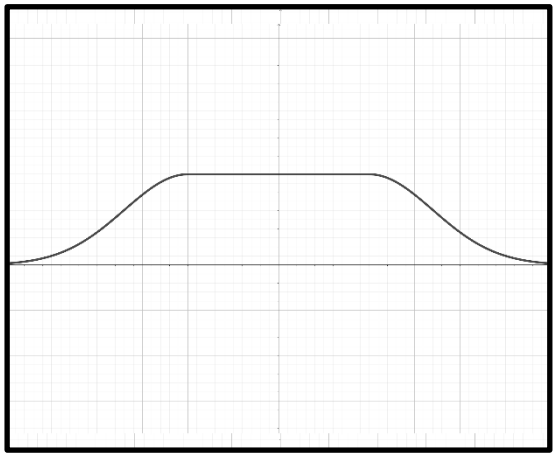


Figure 19: Probability field function with radius.

where:

- R : The radius of the agent's cylindrical body.

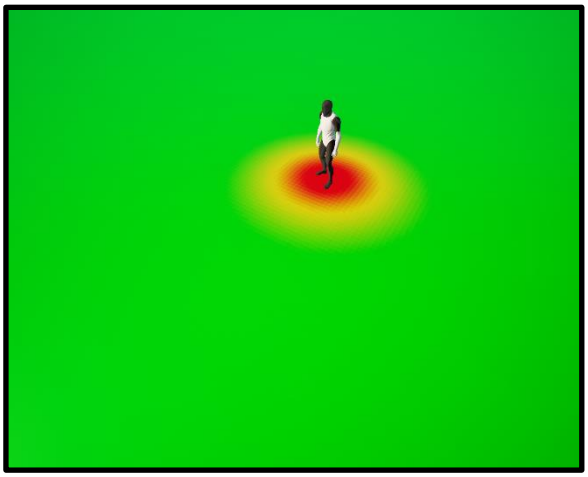


Figure 20: Probability field with R = 50.

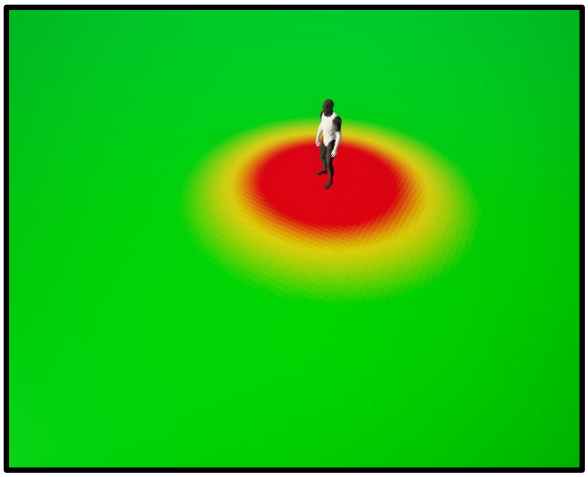


Figure 21: Probability field with R = 150.

Velocity
To predict the future positions of an agent and compute the corresponding probability fields over time, the agent's velocity is used, assuming the velocity remains constant over the time horizon.

$$\mathrm{r(t)} = r_0 + v \cdot t \quad (12)$$
$$\mathrm{d(t)} = \| r(t) - \mu(t) \| \quad (13)$$

where:

- $r(t)$: The predicted position of the agent at time t .
- r_0: The current position of the agent.
- v : The constant velocity vector of the agent.

Time horizon
The probability field accounts for future predictions by considering a time horizon T, which represents how far into the future the agent's position and collision probabilities are projected. To estimate probability distribution over time, the time horizon is divided into discrete N intervals, allowing to compute a series of future probabilities at sampled time points.

$$\Delta t = \frac{T}{N} \quad (14)$$

where:

- T : The total duration in the future for which probabilities are considered.
- N : The discrete time steps that are used within the time horizon.
- Δt : The time step representing the interval between consecutive samples.

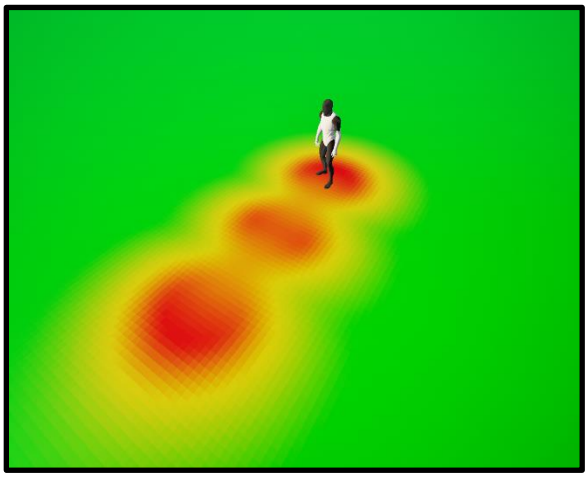


Figure 22: Probability field with T = 3 and N = 3.

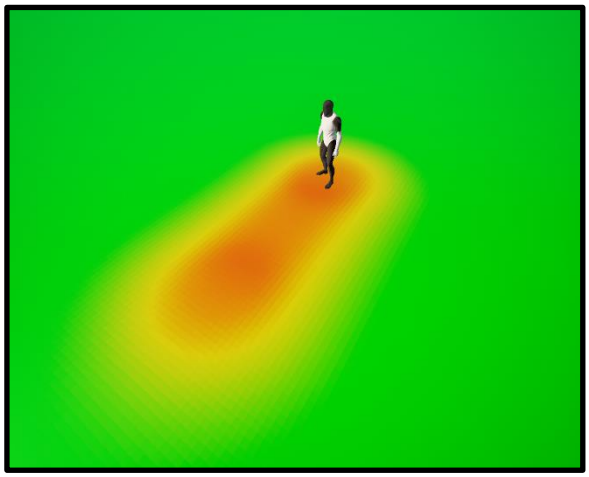


Figure 23: Probability field with T = 2 and N = 8.

Time uncertainty
As predictions extend further into the future, uncertainty about the agent's exact position increases. To model this temporal uncertainty, two key modifications are introduced in the probability field at each time step:

1. Reduced Confidence Near the Agent
The maximum probability at distances less than the agent's radius R is decreased over time to reflect reduced certainty about the agent occupying that specific region in the future.

$$\alpha(t) = \frac{1}{1+\lambda t} \quad (15)$$
$$P' = \alpha(t) \cdot P \quad (16)$$

where:
- $a(t)$: The decay in time.
- λ : The decay rate.
- P': The new probability.

2. Increased Spread
The standard deviation σ of the *Gaussian* field increases over time, reflecting growing positional uncertainty. This makes the probability field more diffuse as time progresses.

$$\sigma(t) = \sigma + \gamma \cdot t \quad (17)$$

where:
- $\sigma(t)$: The standard deviation of the gaussian field over time.
- γ : The uncertainty rate.

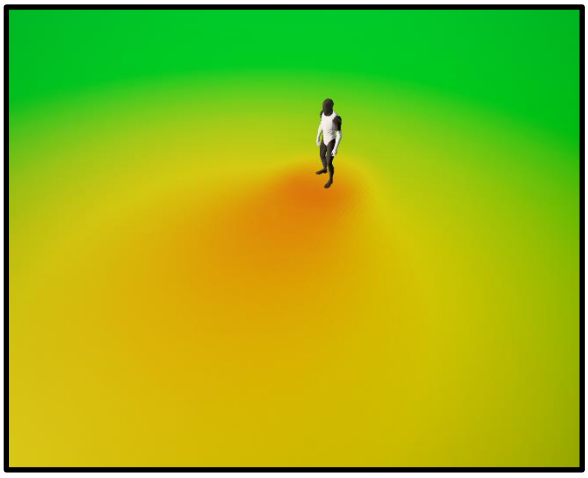


Figure 24: Probability field with λ = 0.1 and γ = 100.

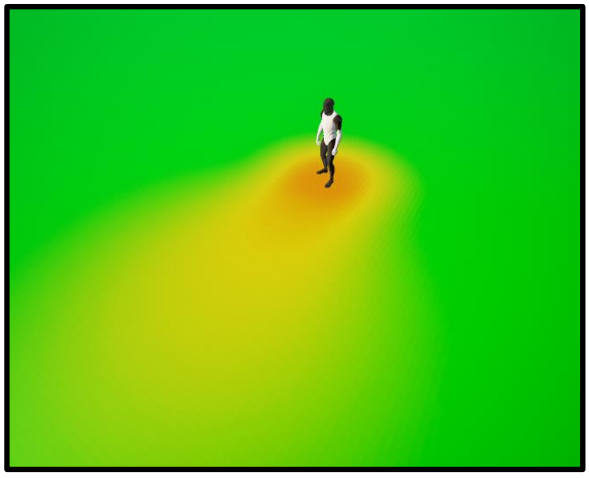


Figure 25: Probability field with λ = 0.5 and γ = 20.

Others
It is worth noting that additional factors could be incorporated into the model to capture more complex behaviors or uncertainties. However, for the purposes of this research, we focus on this core set of properties, as they provide a practical balance between computational efficiency and representational accuracy.

Collision probability gradient
To compute the gradient of the collision probability in an efficient way, partial derivatives are taken with respect to the spatial coordinates x and y. Since the collision probability is defined using an exponential function, these derivatives can be computed analytically in a straightforward manner, leading to the following expressions:

$$\nabla P = \left(\frac{\partial P}{\partial x}, \frac{\partial P}{\partial y}\right) \tag{18}$$

By applying the chain rule, the gradient can be expanded as:

$$\nabla P = \frac{dP}{d\beta} \cdot \frac{d\beta}{dd} \cdot \nabla d$$

$$\nabla d = \left(\frac{\partial d}{\partial x}, \frac{\partial d}{\partial y}\right)$$

where:

- $\beta = -\frac{d^2}{2\sigma^2}$
- $d = \sqrt{x^2 + y^2} - R$

Computing the derivatives:

$$\frac{\partial d}{\partial x} = \frac{x}{\sqrt{x^2 + y^2}}$$

$$\frac{d\beta}{dd} = -\frac{d}{\sigma^2}$$

$$\frac{dP}{d\beta} = P$$

$$\frac{\partial P}{\partial x} = \frac{dP}{d\beta} \cdot \frac{d\beta}{dd} \cdot \frac{\partial d}{\partial x} = -P \cdot \frac{d}{\sigma^2\sqrt{x^2+y^2}} \cdot x \tag{19}$$

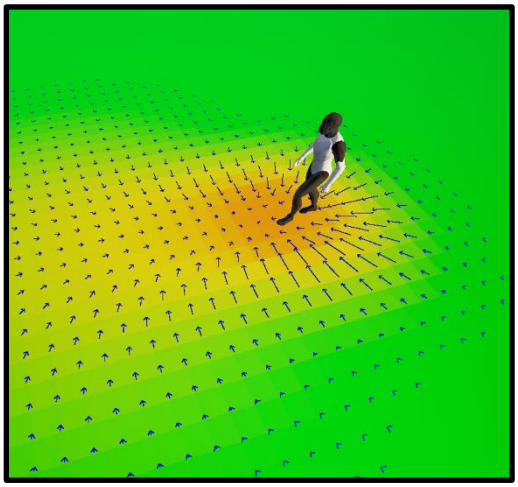


Figure 26: Collision probability gradient.

Combination of collision probability fields

To evaluate collision risk for each agent, individual pairwise collision probabilities are combined into a single overall probability P_T and gradient ∇P_T. The combination is performed cumulatively using the following formulas:

$$P_{bc} = P_b + P_c - P_b \cdot P_c \quad (20)$$

$$\nabla P_{bc} = \nabla P_b + \nabla P_c - P_b \cdot \nabla P_c - P_c \cdot \nabla P_b \quad (21)$$

where:

- P_b and P_c are the probabilities of collision with agents or obstacles b and c , respectively.
- ∇P_b and ∇P_c are their corresponding gradients.
- P_{bc} and ∇P_{bc} are the combined probability of collision and gradient.

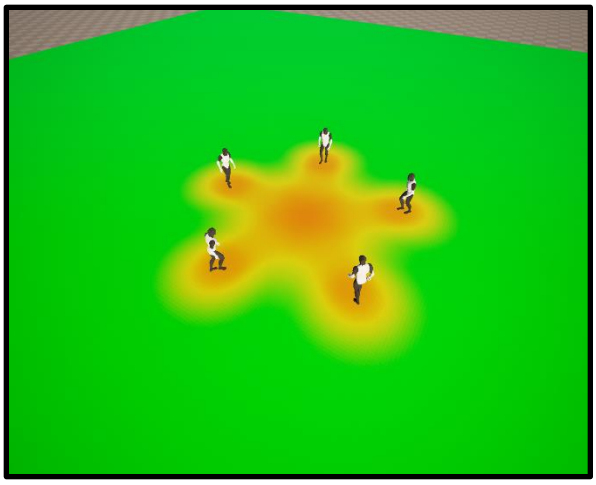


Figure 27: Combined probability fields.

Overall computations
For each agent, collision probabilities are sampled along its predicted trajectory over a defined time horizon T. Based on these samples, the following key quantities are computed:

Normalization factor:

$$n = \int_0^T P_T(t) \cdot dt \tag{22}$$

Overall collision probability and probability gradient:

$$P = \frac{1}{n}\int_0^T P_T(t)^2 \cdot dt \tag{23}$$

$$\nabla P = \frac{1}{n}\int_0^T P_T(t) \cdot \nabla P_T(t) \cdot dt \tag{24}$$

Point of application:

$$s = \frac{1}{n}\int_0^T P_T(t) \cdot d(t) \cdot dt \tag{25}$$

Collision avoidance
Once the collision probability and its gradient have been computed, the agent's new velocity is determined by minimizing the overall risk of collision. This involves selecting a velocity that results in the lowest possible collision probability by performing one step of gradient descent to modify its trajectory.

$$q = s - \alpha \cdot P \cdot \nabla P \tag{26}$$
$$v' = q - r \tag{27}$$

where:

- α : Scaling factor that controls how strongly the agent adjusts its trajectory in response to the collision gradient.

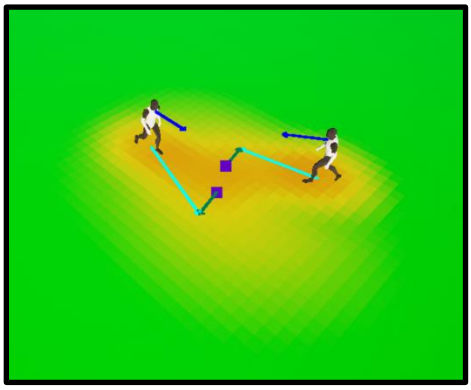


Figure 28: Agent collision avoidance considering the current velocity (blue arrow) to compute the new velocity (cyan arrow), using the negative gradient in the point of application (purple point).

Static obstacles with non-cylindrical shapes
While dynamic agents are modeled as cylindrical entities for simplicity, static obstacles often have irregular or non-cylindrical shapes that require a more flexible representation.

To accurately represent static obstacles with arbitrary shapes, their **boundaries can be modeled as a set of line segments** that approximate or encapsulate their geometry. Instead of using a circular distance metric, the probability field around each obstacle is computed based on the distance to the nearest line segment.

$$d(t) = dist(r(t),\, L) \tag{28}$$

where:

- $d(r,\, L)$: Distance from the sampled position to the line segment.
- L : The closest line segment of the shape.

This formulation ensures that the highest probability density is concentrated along the edges of the obstacle, and it decays smoothly with distance. To avoid overestimating the collision probability at the intersection of line segments, each sampled point **considers only the maximum** probability among all segments, rather than combining them.

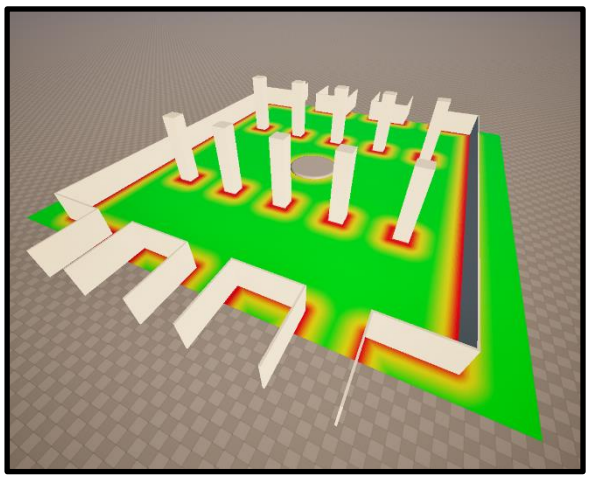


Figure 29: Probability field of static obstacles represented as line segments.

Function Overview

The following function illustrates the core logic used to determine a new safe velocity for an agent. It operates by integrating over the collision probability fields generated by both dynamic and static obstacles. The resulting velocity steers the agent away from high-risk regions using a gradient descent approach.

Listing 2 Function to compute safe velocity based on collision probability

```
FVector2D ComputeSafeVelocity(const ACrowdAgent* agent,
const std::vector<ACrowdAgent*>& agents,
const std::vector<LineSegment>& staticObstacles,
const FWarpDriverParameters& params)
{
     // Initial position of this agent
     FVector2D cPos = FVector2D(agent->GetActorLocation());

     // Accumulators
     float N = 0.0f;
     float sumP2 = 0.0f;
     FVector2D sumPG(0, 0);
     FVector2D sumPR(0, 0);

     for (int i = 0; i < params.numSamples; i++)
     {
          float t = i * params.dt;

          // Get the sample position
          FVector2D pos = ComputeProjectedTrajectory(cPos,
                     agent->mVelocity, t);

          // Static obstacles
          float staticProb = 0.0f;
          FVector2D staticGrad(0.0f, 0.0f);
          StaticCollisionProbabilityAndGradient(pos, params,
          staticObstacles, staticProb, staticGrad);

          // Dynamic obstacles
          float collisionProbability = staticProb;
          FVector2D collisionGradient = staticGrad;
          DynamicCollisionProbabilityAndGradient(pos, t,
          params, agents, collisionProbability,
          collisionGradient);
```

```
            // Accumulate
            N += collisionProbability * params.dt;
            sumP2 += collisionProbability *
                     collisionProbability *
                     params.dt;

            sumPG += collisionGradient *
                     collisionProbability *
                     params.dt;

            sumPR += pos * collisionProbability *
                     params.dt;
        }

        // If no collision risk, keep velocity
        if (N <= 0.001f)
            return agent->mVelocity;

        // Normalize
        float invN = 1.0f / N;
        float p = sumP2 * invN;
        FVector2D grad = sumPG * invN;
        FVector2D s = sumPR * invN;

        // One gradient-descent step
        FVector2D offset = params.alpha * p * grad;
        FVector2D safePos = s - offset;

        // Compute the safe velocity vector from the agent's
        initial position to the safe position
        FVector2D safeVel = safePos - cPos;
        return safeVel;
}
```

Optimizations
The core of the probability field in this method relies on a *Gaussian* decay function of the form:

$$P(d) = e^{\left(-\frac{d^2}{2\sigma^2}\right)}$$

While this formulation is mathematically elegant and expressive, the exponential function exp(x) is computationally expensive and can significantly impact real-time performance when evaluated repeatedly over multiple agents.

To address this, we employ the fast exponential approximation proposed by *Schraudolph* [3] , which provides a compact and efficient alternative based on manipulating the *IEEE-754* floating-point representation. The core idea is to exploit the logarithmic encoding of floating-point numbers and reinterpret multiplication in the exponent as bitwise operations.

The complete transformation used to approximate e^x is:

$$i = a \cdot x + (b - c) \tag{29}$$

where:

- $a = \frac{2^{23}}{\ln 2} \approx 12102203$
- $b = 127 \cdot 2^{23} \approx 1065353216$ (bias for single-precision floats)
- c : Tunable parameter that adjusts the approximation's characteristics (Root-mean-square (RMS) relative error is minimized for $c \approx 60810$)

It is crucial to prevent overflow due to the rapid growth of the exponential curve. In particular, the input must be clamped to a safe range before applying the approximation. For single-precision floats (*IEEE-754*), the maximum representable value is approximately $3.4 \cdot 10^{38}$. The exponential function reaches this magnitude at approximately e^{88}. Therefore, to maintain numerical safety, the input is clamped to the range $[-88.0,\ 88.0]$.

Although the approximation sacrifices some precision (especially for large magnitudes) it is sufficiently accurate in the region of interest. In practice, this results in orders of magnitude performance improvements with negligible impact on the behavior of the probability field, particularly given that very small probabilities are treated as zero in the system.

Listing 3 Function to compute fast exponential approximation.

```
float EXPF(x)
{
    uint32_t i = (uint32_t)(EXPF_A * x + (EXPF_B - EXPF_C));
    return (float)i;
}
```

8. Algorithm evaluation

To evaluate the performance and effectiveness of the various collision avoidance algorithms, a series of simulation scenarios have been designed. In each scenario, multiple agents are initialized with predefined goals. Once an agent reaches its goal, it is removed from the simulation. The simulation ends when all agents have reached their destination and exited the environment.

Evaluation Metrics

The following metrics are used to quantitatively evaluate each algorithm's performance:

- **Collision Count**: The total number of collisions between agents. This metric reflects the algorithm's ability to maintain spatial separation and avoid physical contact.
- **Simulation Completion Time**: The total time (in seconds) required for all agents to reach their respective goals. This indicates the overall efficiency of the crowd movement.
- **Computational Cost**: Measured by the average and minimum frame rate during the simulation, this metric captures the algorithm's runtime performance.

These metrics offer a balance between safety (minimizing collisions) and efficiency (minimizing time), which are critical in practical applications of crowd simulation.

All measurements were carried out under comparable conditions, as far as reasonably possible, given that the algorithms differ in nature and each has distinct tunable parameters. Despite these algorithm-specific differences, the experiments were carried out using the same hardware and identical goal-directed steering forces to ensure a fair comparison.

Testing environments

The algorithms have been evaluated across a **variety of testing environments** designed to challenge different aspects of local navigation and collision avoidance. Each environment presents unique constraints and dynamic conditions that test the robustness, adaptability, and efficiency of the algorithms under diverse scenarios.

1. Circle

This scenario consists of five agents initially positioned evenly along the perimeter of a circle. Each agent is assigned a target directly opposite its starting point on the circle. As a result, all agents converge toward the center, where their paths intersect, creating potential for high-density interactions.

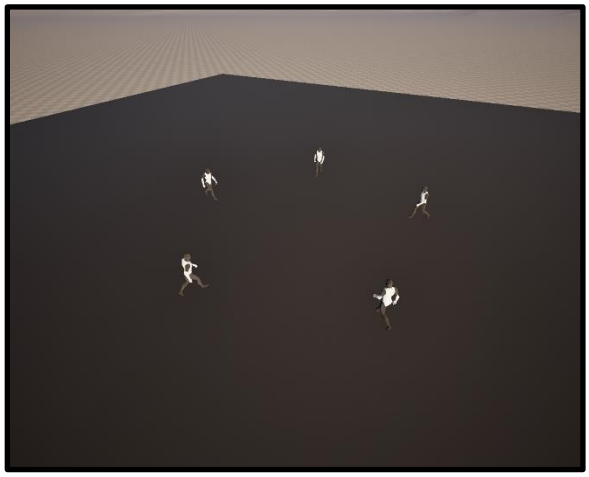


Figure 30: Circle testing environment.

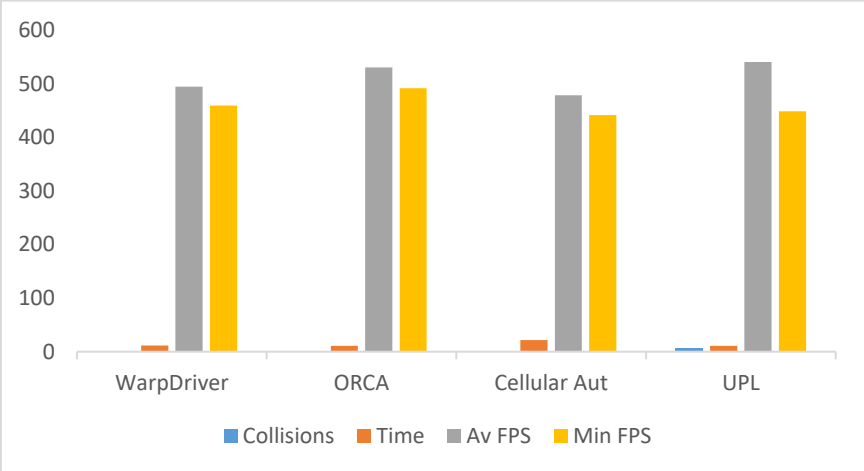
Comparison

WarpDriver and *ORCA* demonstrated the best overall superiority. Both algorithms managed to avoid collisions entirely, navigating the dense central convergence zone with smooth and symmetric trajectories.

Cellular Automata also avoided collisions but took nearly twice as long to complete the simulation due to its less direct movement patterns.

In contrast, *UPL* performed the worst in terms of safety, with a high number of collisions. While the completion time was comparable to the faster algorithms, the agent behavior was noticeably more erratic and less symmetrical.

Graph 1: Circle testing environment metrics.



2. Fork Path

This scenario features a fork-shaped environment with two lateral branches merging into a single main path. Agents are placed in each of the side branches and are assigned a target at the end of the main branch. As they move toward their goal, agents from both branches converge at the junction, creating a merging situation that tests the agents' ability to navigate shared space and resolve path conflicts.

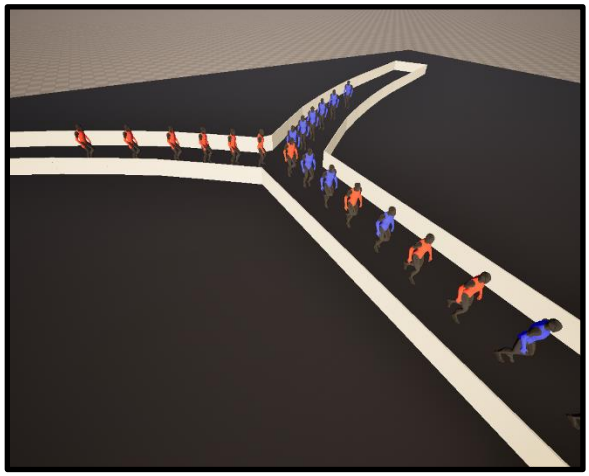


Figure 31: Fork testing environment.

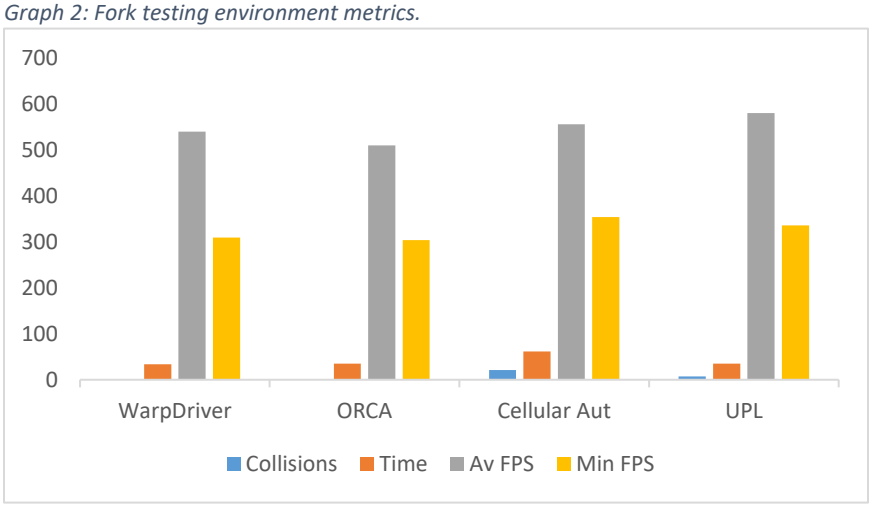
Comparison

WarpDriver and *ORCA* performed similarly in terms of collision avoidance, both achieving zero collisions. However, *WarpDriver* produced smoother and more orderly agent behavior since agents tended to queue neatly at the junction, reducing congestion and creating a more structured flow.

UPL completed the simulation with only a few collisions and was the fastest in terms of computational performance. However, agents' movement was less smooth, with agents often merging more aggressively, resulting in minor overlaps and reduced coordination.

Cellular Automata showed clear limitations in this scenario. The discretization of space made it difficult for agents to consider curves and merge fluidly, resulting in delayed movements and awkward path adjustments. As a result, the simulation took nearly twice as long to complete and had a noticeable number of collisions

Graph 2: Fork testing environment metrics.



3. Queue

This scenario consists of a main linear path with multiple perpendicular side rooms, each containing several agents. Additional agents are placed directly along the main path. All agents share a common target located at the end of the main path. As agents from the side rooms attempt to merge into the main flow, they must wait for suitable gaps, creating a queue-like dynamic that challenges priority handling, congestion resolution, and fairness in path access.

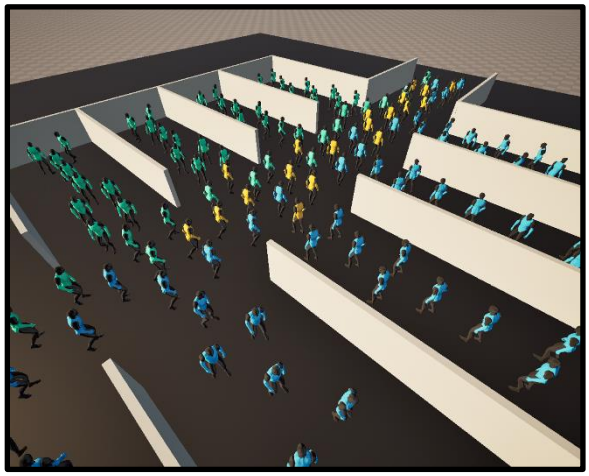


Figure 32: Queue testing environment.

Comparison

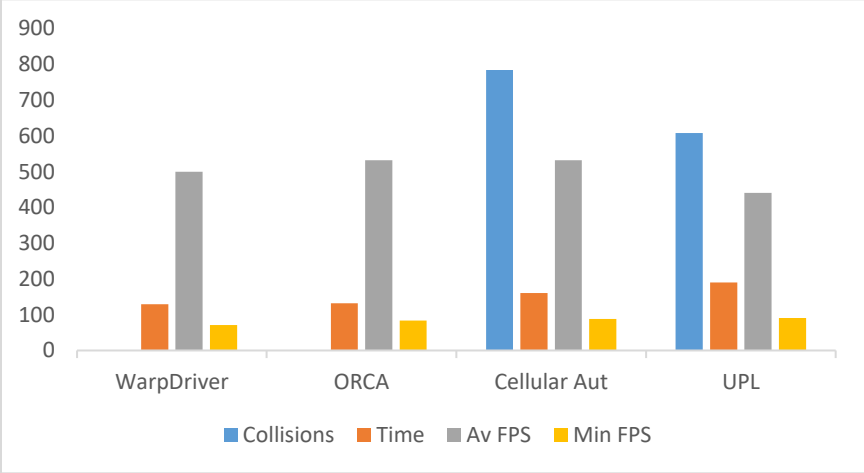
WarpDriver performed well under high-density conditions, handling congestion with organized queuing behavior. However, toward the end of the simulation, when only a few agents remained, it struggled with convergence.

ORCA was noticeably more aggressive when merging into the main path. While the total simulation time was only slightly longer than *WarpDriver*'s, the final segment exhibited less spatial coordination, with agents often competing for space rather than letting

others go first.

Cellular Automata and *UPL* both showed significant difficulties in this environment. *Cellular Automata* suffered from frequent collisions and long delays, largely due to its rigid movement patterns and poor adaptability to dynamic congestion. *UPL*, also recorded a high number of collisions.

Graph 3: Queue testing environment metrics.



4. Hall

This scenario involves two large groups of agents positioned on opposite sides of a wide hall. Each group is assigned exit targets on the opposite side, requiring them to cross paths. The environment includes several static obstacles such as columns, mimicking real architectural constraints. This setup generates dense bidirectional flow and tests the agents' ability to handle congestion, navigation around obstacles, and collective movement under realistic spatial constraints.

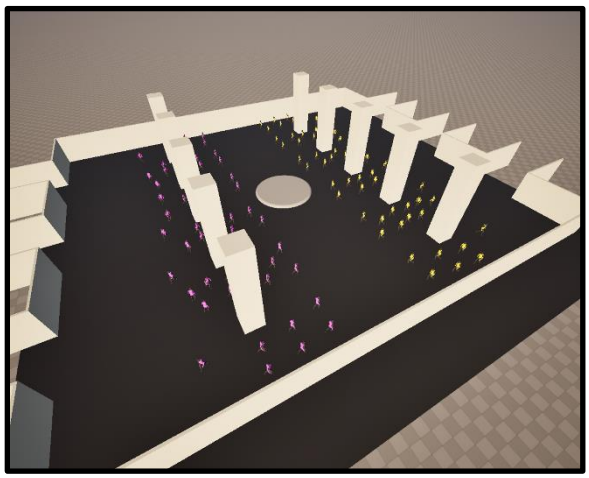


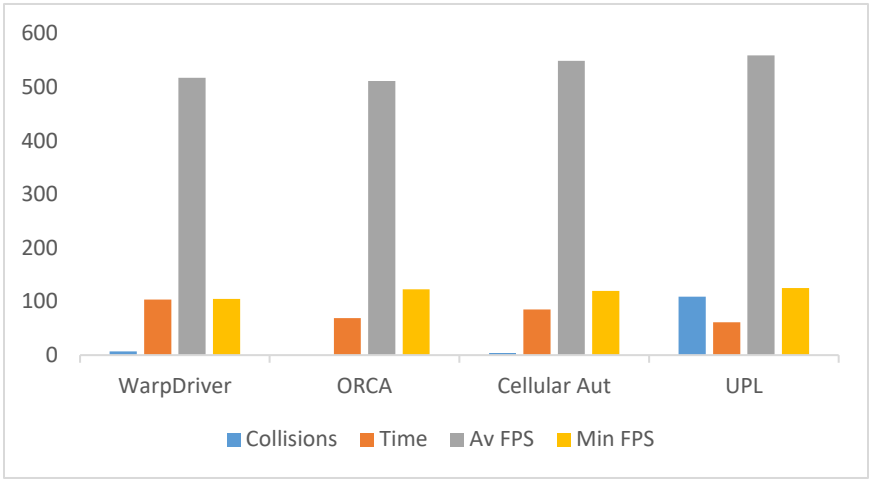
Figure 33: Hall testing environment.

Comparison

WarpDriver showed a strong tendency to prioritize collision avoidance, which increased the overall simulation time. However, created well-structured lanes of movement. *ORCA* outperformed the rest in terms of efficiency and safety, but, toward the end of the simulation, the accumulation of agents near the targets led to some congestion.

Cellular Automata achieved relatively few collisions but had less coordinated movement, since the group tended to spread out unevenly across the hall, leading to a scattered flow. With *UPL* agents moved quickly and forcefully, often ignoring personal space and pushing through other agents coming from the opposite direction.

Graph 4: Hall testing environment metrics.



5. Crossing

This scenario simulates a busy four-way intersection with all roads intersecting at the center. Large groups of agents start from the four diagonal and cardinal directions, each aiming to cross to the opposite side. The crossing creates a situation where multiple flows intersect simultaneously, testing the agents' ability to navigate in dense, multidirectional crowds.

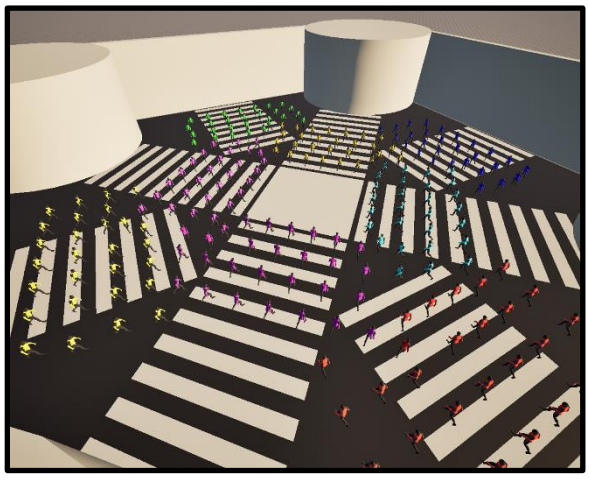


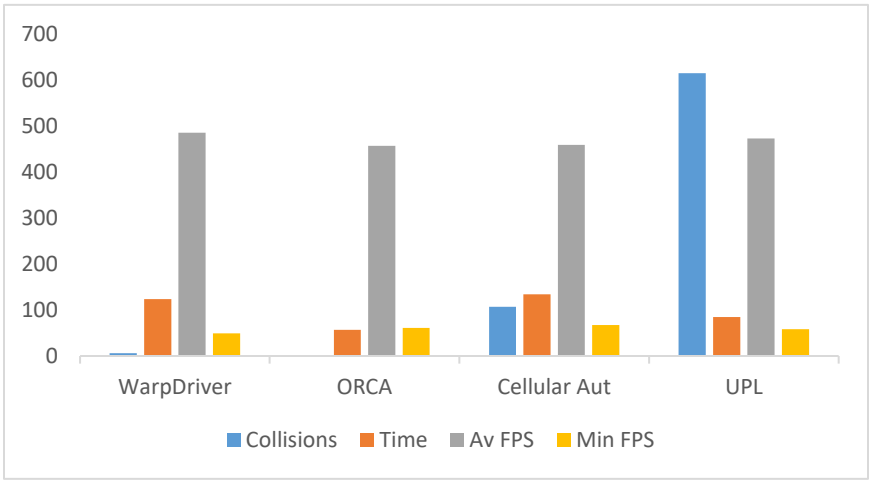
Figure 34: Shibuya crossing testing environment.

Comparison

WarpDriver agents tended to avoid collisions by taking long, curved paths, which made the overall simulation slower. In such an open space, they often wandered more than necessary before reaching their goals. *ORCA* handled the crossing much more efficiently. Agents moved directly and organized themselves fairly well. However, early in the simulation, there was noticeable congestion at the center.

Cellular Automata struggled with coordination. Agents spread out too much and did not follow clear paths, making the movement look disorganized. This led to a high number of collisions and the longest simulation time overall. *UPL* showed very aggressive behavior in this test. Agents often made sharp, sudden turns to avoid others, resulting in abrupt movement and many collisions.

Graph 5: Shibuya crossing testing environment metrics.



Overall Conclusions

WarpDriver performs reliably in narrow, high-density environments where agents need to move with precision and form organized flows. It handles congestion effectively and maintains good separation between agents. However, in open spaces, its overly cautious behavior leads to agents taking long detours to avoid collisions, which slows down progress and sometimes causes delays near the end of a simulation. As the number of agents increases, it also becomes more computationally demanding, which can be a limitation for larger scenarios.

ORCA is slightly more efficient computationally and generally works better in open environments. It produces smooth and direct paths thanks to its linear trajectory assumptions, which are often effective. However, this same simplicity becomes a drawback in very dense situations, where agents can get stuck in place, balancing each other out without resolving the congestion. Still, it offers a good balance of speed and quality in many typical scenarios.

Cellular Automata is lightweight and fast to compute but shows clear limitations in environments where obstacles or movement do not align with its underlying grid. In dense areas, agents tend to hesitate and behave less naturally, especially at intersections or around corners. Agents also tend to spread out too much, making the overall crowd behavior feel less realistic and more fragmented.

UPL is the fastest algorithm in terms of computation, but it comes at the cost of safety and realism. Agents often push through each other, leading to more collisions. While the generated paths can appear smooth at times, the algorithm applies forces too aggressively, which results in sudden, unnatural changes in direction. This makes it feel more chaotic, particularly in complex or high-density situations.

9. Following behavior (Queuing)

Another aspect of human locomotion that impacts crowd simulation is how **individuals follow one another** in environments such as corridors or queues. Specifically, a person adjusts their speed in a particular way when walking behind someone else.

Most standard local behavior algorithms do not explicitly model this kind of **speed adaptation**. Instead, they allow each agent to move toward its goal while avoiding collisions, with any resulting speed change emerging implicitly from this process.

However, the probability-based collision avoidance algorithm *WarpDriver* addresses this case with surprising success. Since agents adjust their trajectories, and thus their speed, based on the estimated probability of collision, they naturally slow down in high-risk areas and may even come to a near-stop while waiting for safe passage. As a result, this specific pattern of behavior **emerges indirectly** as a consequence of the collision avoidance mechanism itself.

This implicit handling of speed control facilitates the natural emergence of structured formations such as queues. As agents reduce speed in response to high local collision probabilities, spatial ordering arises where agents align behind one another while maintaining safe distances. This results in queue-like patterns without explicitly programming such formations.

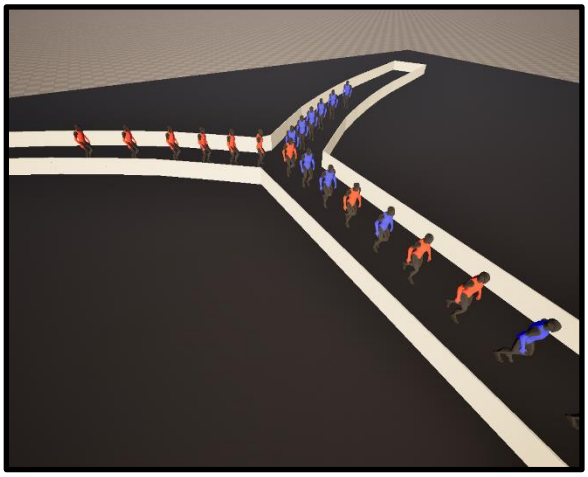


Figure 35: Agents forming a queue in a fork path.

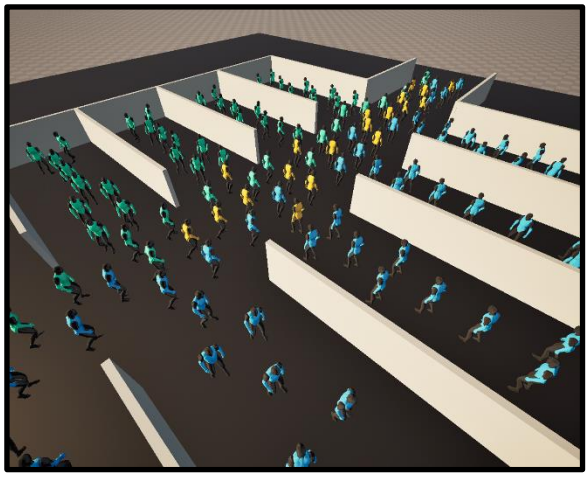


Figure 36: Agents waiting to join a queue.

10. Density Awareness (DenseSense)

Pedestrian movement is known to be **highly influenced by crowd density**, with individual speeds typically decreasing as the surrounding density increases. To reflect such dynamics, some algorithms dynamically adjust an agent's preferred velocity based on local density or flow conditions, which represent the direction toward an agent's immediate goal.

A key concept in this context is the *Fundamental Diagram*, which describes the inverse relationship between pedestrian speed and density, as the crowd becomes denser, individuals slow down. This relationship is environment-independent and corresponds to a relationship between density and speed.

DenseSense [4] builds upon this principle by incorporating a density-aware filter that enables more human-like responses in congested environments. This filter can be applied alongside any collision avoidance algorithm.

It refines the preferred velocities by considering dynamic local factors such as nearby agents. This leads to more natural trajectories and enhances the realism of simulated agent behaviors.

Natural walking speed

An agent's natural walking speed can be estimated using a model that employs a linear combination of physiological and psychological factors, specifically, stride length and personal space:

$$V(S) = \max\left(\|v_0\|, \left(\frac{S \cdot \alpha}{(1+\beta)}\right)\right) \tag{30}$$

where:

- v_0 : The current preferred velocity.
- S : The available space.
- α : The stride factor, which refers to the relationship between stride length and walking speed.
- β : The stride buffer, which reflects a preference for personal space.

Density computation

A *Gaussian* density function is employed to quantify the influence of each neighboring agent on the perceived density at a given point:

$$w(d) = \frac{1}{\sqrt{2\pi\sigma}}\, e^{\left(-\frac{d^2}{2\sigma^2}\right)} \tag{31}$$
$$d = \| r - \mu \|$$

where:

- $r = (x, y)$: The point in space where the density is being evaluated.
- μ : The agent's position.
- d : The distance between the point and the agent.
- σ : The spread or decay rate of the density.

The total density is computed as follows:

$$\rho = \sum_{i=1}^{N} w(d_i) \quad (32)$$

Humans are known to exhibit an **elliptical personal space**, with a stronger preference for maintaining space in the forward direction. To capture this behavior in the simulation, the personal space of agent is modeled using an anisotropic transformation of the relative positions of its neighbors. [7] [4] [4] [4] [4] [7]

$$d' = 2.5(d - D) \quad (33)$$
$$D = (d \cdot v_\theta) \cdot v_\theta \quad (34)$$

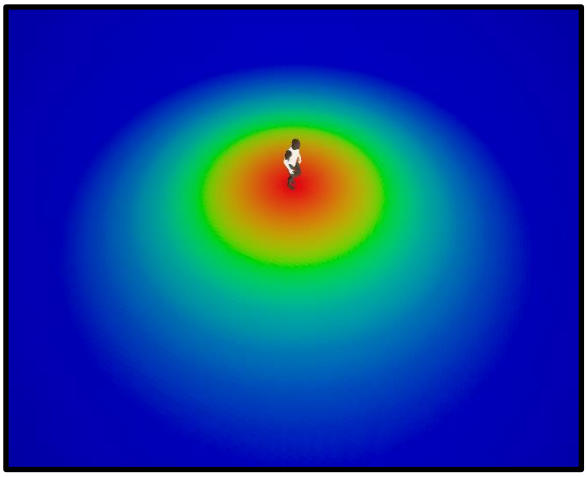


Figure 37: Crowd density heatmap.

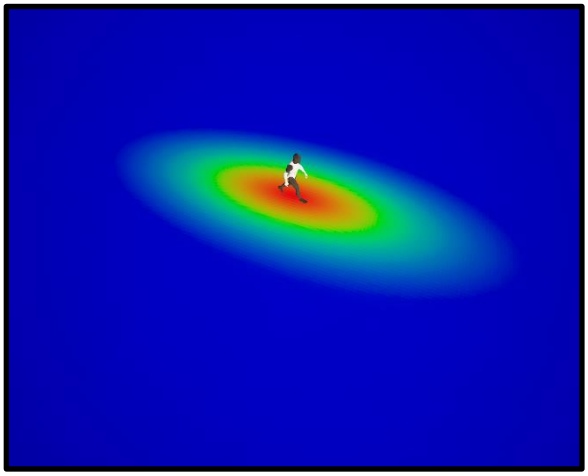


Figure 38: Crowd density heatmap considering elliptical personal space

Selection of preferred velocity

At each simulation step, the algorithm constructs an arc centered around the input preferred velocity.

Figure 39: Arc centered at input preferred velocity

The agent selects the velocity that minimizes the distance to its goal over a given time horizon, while adhering to the velocity constraints imposed by the *Fundamental Diagram*. The adherent velocity can be computed as:

$$v_{FD_\theta} = \frac{v_\theta}{\|v_\theta\|} \cdot \mathrm{V}\left(\frac{\rho_\theta}{w}\right) \tag{35}$$

where:

- v_θ : The velocity in the direction defined by angle θ.
- ρ_θ : The density in the direction defined by angle θ.
- w : The agent's width.

From the arc, it selects a velocity that minimizes a defined cost function:

$$\arg\min_{\theta \in \Theta}\left(\left\|g - \left(\mathrm{p} + v_{FD_\theta} \cdot \Delta t\right)\right\|\right) \tag{36}$$

where:

- Θ : The set of all candidate angles on the arc.
- g : The goal position.
- p : The agent's current position.
- Δt : The time step.

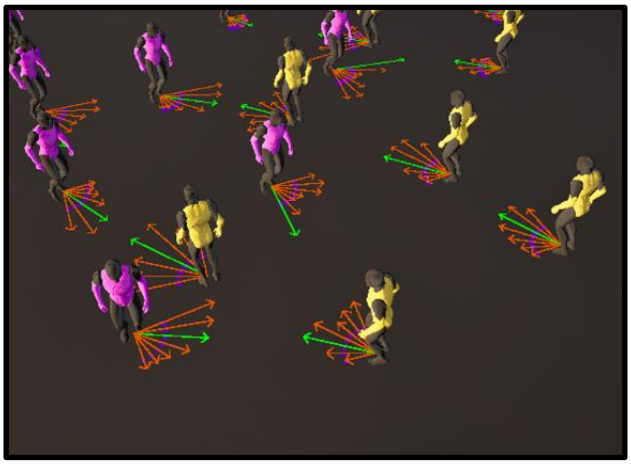


Figure 40: Agents choosing sampled velocity based on crowd density.

Density maps

Density maps are a valuable tool for visualizing the spatial distribution of agents within a simulation environment. They highlight regions of varying crowd density, allowing to easily identify congested or underutilized areas within the level.

A common technique for generating density maps is the use of **heatmaps**. A color intensity is assigned to each point in the environment based on the normalized density of nearby agents.

To ensure the resulting heatmap values are consistent and comparable, the computed densities are normalized by dividing them by the maximum density value across all sampled points. [7] [7] [7] [7] [7] [7] [7] [7] [7] [7] [7] [7]

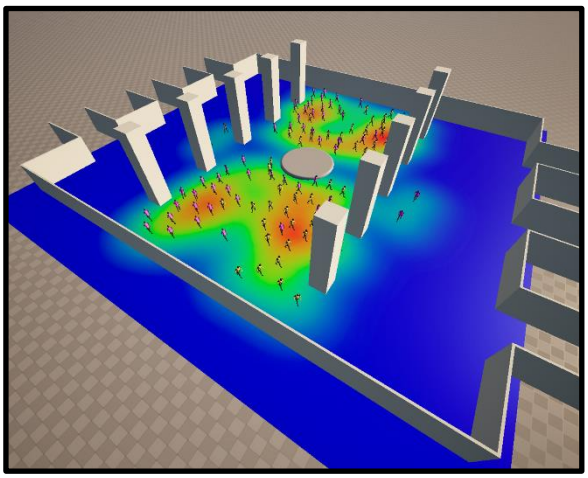


Figure 41: Crowd density heatmap example.

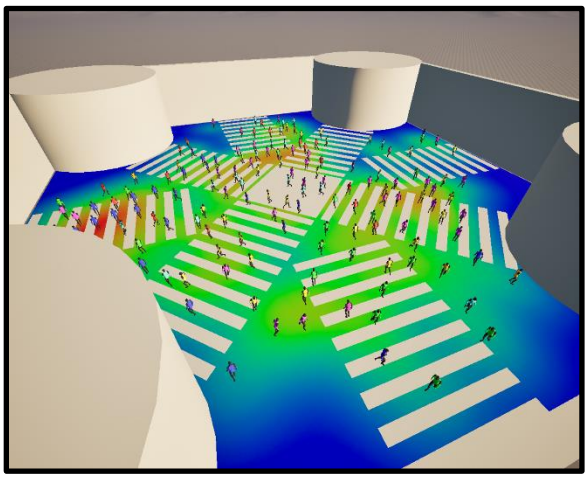


Figure 42: Crowd density heatmap example.

11. Conclusions

Achieving a realistic and efficient crowd simulation is a complex task. In this research, various methods have been studied aimed at reaching this goal, with a particular focus on collision avoidance techniques.

The simulation has been approached by considering only local navigation, modeling the behavior of each individual independently without global path planning or group coordination.

Different approaches have been tested, each with their own strengths and weaknesses, and it was found that their effectiveness varies depending on the scenario. For example, some algorithms performed better in high-density environments, while others were more suitable for open spaces.

An exhaustive comparison has been carried out, evaluating multiple metrics for each collision avoidance algorithm across a range of environments, each with distinct challenges and requirements. Interestingly, some methods led to the spontaneous emergence of complex group behaviors, such as the formation of organized queues.

Furthermore, the study considered how density awareness influences individuals' perception of their surroundings, which consequently affects their desired speed toward a given goal. These insights contribute to a deeper understanding of crowd dynamics and may guide the development of more adaptive and realistic simulation systems.

12. Future work

As for future work, incorporating global navigation strategies and group behavior models could significantly enhance the realism and applicability of the simulations.

While this study focused on local navigation and individual decision-making, real world crowd movement often depends on a combination of local reactions and global path planning. Integrating global navigation would allow agents to make more informed, long-term decisions rather than reacting just to their immediate surroundings.

Additionally, modeling group behaviors would provide a more accurate representation of social dynamics in real crowds.

These extensions could improve the system's ability to simulate large-scale, and complex environments

13. References

[1] Wouter van Toll, Julien Pettré. Algorithms for Microscopic Crowd Simulation: Advancements in the 2010s. Computer Graphics Forum, 2021, Eurographics 2021, 40 (2), 10.1111/cgf.142664. hal-03197198

[2] WOLINSKI D., LIN M. C., PETTRÉ J.: WarpDriver: Context aware probabilistic motion prediction for crowd simulation. ACM Trans. Graph. 35, 6 (2016), 164:1–164:11.

[3] N.N. Schraudolph, A fast, compact approximation of the exponential function, Neural Computation. 11 (1999) 853–862.

[4] BEST A., NARANG S., CURTIS S., MANOCHA D.: DenseSense: Interactive crowd simulation using density-dependent filters. In Proc. ACM SIGGRAPH/Eurographics Symp. Computer Animation (2014), Eurographics Association, pp. 97–102.

[5] VAN DEN BERG J. P., GUY S. J., LIN M. C., MANOCHA D.: Reciprocal n-body collision avoidance. In Proc. 14th Int. Symp. Robotics Research (2011), pp. 3–19. 6, 8, 9, 11

[6] KARAMOUZAS I., SKINNER B., GUY S. J.: Universal power law governing pedestrian interactions. Phys. Rev. Lett. 113 (2014), 238701:1–5. 6

[7] D. Helbing, I. J. Farkas, and T. Vicsek, "Simulating dynamical features of escape panic," cond-mat/0102397, 2001.