Implementing Position-Based Real-Time Simulation of Large Crowds

Tomer Weiss*
University of California, Los Angeles
*tweiss@cs.ucla.edu

***Abstract*—Various methods have been proposed for simulating crowds of agents in recent years. Regrettably, not all are computational scalable as the number of simulated agents grows. Such quality is particularly important for virtual production, gaming, and immersive reality platforms. In this work, we provide an open-source implementation for the recently proposed Position-based dynamics approach to crowd simulation. Position-based crowd simulation was proven to be real-time, and scalable for crowds of up to 100k agents, while retaining dynamic agent and group behaviors. We provide both non-parallel, and GPU-based implementations. Our implementation is demonstrated on several scenarios, including examples from the original work. We witness interactive computation run-times, as well as visually realistic collective behavior.**

***Index Terms*—crowd simulation, collisions avoidance, constraints**

I. Introduction

Simulation of virtual agents, and crowds of agents, is important in multiple visual media domains, from Virtual and Augmented reality, games, to educational platforms. Simulating crowds is driven by a combination of algorithms, each focusing on different aspects [6]. Such aspects include visual animation and rendering of crowd agents; modeling of the crowd to be simulated; and computing movement of each agent within the crowd.

We address the task of implementing a framework for computing virtual agent movements; i.e., in each time step of the simulation, each agent must determine in which direction to move such that realistic individual and collective behaviors result. Agents share the same environment. Therefore, they can interact, and potentially collide with each other. A goal of a algorithm that computes agent movement is to avoid such collisions. Unfortunately, computing collision-free agent motion is difficult, due to the complexity of such dynamic interactions, especially in dense settings.

Researchers have proposed various collision avoidance methods [1], [2], [5], [8]. Recently, Karamouzas et al. [3] proposed a collision avoidance method based on experimental observations of human crowds. The method is an explicit force-based scheme, where collision avoidance is a function of inter-agent time-to-collision. Weiss and colleagues [8] proposed a similar approach, using Position-Based Dynamics constraints to simulate agent dynamics, for up to 100k agents, in real-time. Additionally, their approach is capable in simulating both sparse, and dense scenarios.

Here, we present an implementation of Position-based crowd simulation [7], [8]. Our solution is open source[1], and is implemented with C++, for both non-parallel, and GPU-enabled machines. As far as we are aware, this is the first open source implementation of a Position-based dynamics framework for crowd simulation [4].

II. Design and Implementation

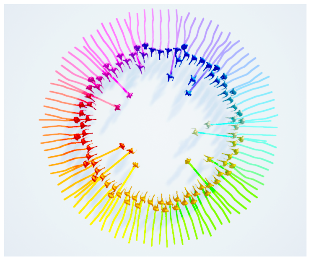
Fig. 1: Agents locomote toward their antipodal goal on the circle.

A. Structure

Our goal was to allow users to easily run simulation scenarios. To that end, a scenario can either be hard-code, or loaded from a file. The primary components of each simulation scenario are:

1) Spatial Environment: Crowd simulations scenarios are conducted in various environments. To simplify implementation, we assume the environment is a plane. Agents are free to locomote within the environment, except for obstacles. We define such obstacles with line segments, that are impassable for agents. The environment is discretized via a spatial *hash-grid* [8]. Grid boundaries are predetermined, according to the environment. The grid is further divided to cells, that are later used to determine agent collisions (Section II-B).

```
class Simulation
{
  Obstacle **obstacles;
  Grid *grid;
  Agent **agents;
  ...
}
```

2) Locomoting Agents: Each agent has several simulation properties, including *current* and *goal* position, *velocity*, *preferred velocity*, *mass*, and *group membership*.

In each time step, agent advance from the current, to the goal position. All agents start locomoting from standing still. If there are no potential collisions, agents locomote with their preferred velocity directly to their goal. Otherwise, agents preform local collision avoidance according to the position-based constraint framework. Agent's mass is used in

[1]https://github.com/tomerwei/pbd-crowd-sim

```
class Agent
{
  float2 x;    //Current Position
  float2 goal; //Goal Position
  float2 v;    //Current Velocity
  float vPref; //Preferred Velocity
  float mass;
  int group;
  ...
}
```

constraint resolve. Typically, constraints are between pairs of agents. In case a constraint needs to be satisfied, both agents contribute to the correction according to their corresponding mass (Section II-A2).

Once an agent reaches its goal position, it halts. Additionally, our computational framework requires other parameters, as reported in table 2 in [8].

B. Computing a Time Step

The simulation is divided to discrete time steps (Listing 1). At each time step, we: (1) Save agents positions (2) Compute agent movements (3) Copy data from GPU to local CPU memory. Note that the last step is only needed with a GPU implementation, in case we want to save our results for later analysis. For saving agents positions, we create a new file where each line contains the agent identifier, and planar position.

```
planAgentVelocity();
updateNeighbours();
For (int i = 1; i < iterations; i++)
{
    calculateStiffness(i);
    projectConstraints();
}
updatePositions();
updateVelocity();
```

Listing 1: Simulation time step

1) Agent Movements: Agent velocity in each time step is derived by interpolating between agent's current velocity, and the agent's preferred velocity toward its locomotion goal: `v * (1 - Ksi) + Ksi * goalV`, where `Ksi` is the interpolating factor, which we experimentally set to 0.05. Given the agent's velocity, we can derive its position in the next time step, along with potential colliding agents. We can find potential projected neighbours by employing the hash-grid (Section II-A1).

2) Collision Avoidance: In case of predicted collisions between agents, we apply positional constraints. In [8], the authors describe a variety of positional constraints. For brevity, we present our GPU implementation for the short-range collision avoidance constraint (Listing 2).

Our current constraint implementation has no user-defined variable types, to simplify memory allocation details. Our repository includes further system documentation, such as details of other constraints, and an extended description of the architecture.

III. Discussion

We ran our implementation on a 2.5 GHz Intel Core i7 CPU. For GPU experiments, we used a NVIDIA GeForce GT

```
__device__ void pbdCollisionConstraint(
int       i,       //agent identifiers
int       j,
float     margin,  //minimum distance allowed
float     wi,      //typically wi=wj=0.5
float     wj,
float2 *  x,       //current agent positions
float2 *  deltaX,  //positional correction buffer
int    *  deltaXCtr//positional correction counter
)
{
    float f = distance(x[i], x[j]) - margin;
    if(f<0)
    {
        float2 contact=make_float2(0.f,0.f);
        contact.x=(pos[i].x-pos[j].x)/d;
        contact.y=(pos[i].y-pos[j].y)/d;
        atomicAdd(&deltaX[i].x,-wi*contact.x*f);
        atomicAdd(&deltaX[i].y,-wi*contact.y*f);
        atomicAdd(&deltaXCtr[i],1);
        atomicAdd(&deltaX[j].x,-wj*contact.x*f);
        atomicAdd(&deltaX[j].y,-wj*contact.y*f);
        atomicAdd(&deltaXCtr[j],1);
   }
}
```

Listing 2: Constraint implemented as a device function, which is called from a CUDA kernel.

750M. Our current GPU implementation is in CUDA. In the future, we intend to implement OpenCL, and Unity compute shader support.

Tuning a constraint-based simulation is difficult, since there are multiple constraint parameters that affect the simulation quality. Furthermore, agent movement needs to be smoothed by clamping with a maximum acceleration or velocity, which requires multiple experimental iterations. Therefore, we employed the parameters described in previous work to short-cut the implementation process.

Finally, we plan a web implementation of our algorithm, to allow more public experimentation. It will also be worthwhile to combine our framework with recent machine learning platforms.

References

[1] P. Charalambous and Y. Chrysanthou. The pag crowd: A graph based approach for efficient data-driven crowd simulation. In *Computer Graphics Forum*, volume 33, pages 95–108. Wiley Online Library, 2014.

[2] L. Hoyet, A.-H. Olivier, R. Kulpa, and J. Pettré. Perceptual effect of shoulder motions on crowd animations. *ACM Transactions on Graphics (TOG)*, 35(4):53, 2016.

[3] I. Karamouzas, B. Skinner, and S. J. Guy. Universal power law governing pedestrian interactions. *Physical review letters*, 113(23):238701, 2014.

[4] M. Müller, B. Heidelberger, M. Hennix, and J. Ratcliff. Position based dynamics. *Virtual Reality Interactions and Physical Simulations (VRIPHYS)*, 18(2):109–118, 2007.

[5] J. Ondřej, J. Pettré, A.-H. Olivier, and S. Donikian. A synthetic-vision based steering approach for crowd simulation. *ACM Transactions on Graphics (TOG)*, 29(4):123, 2010.

[6] N. Pelechano, J. M. Allbeck, and N. I. Badler. Virtual crowds: Methods, simulation, and control. *Synthesis lectures on computer graphics and animation*, 3(1):1–176, 2008.

[7] T. Weiss, C. Jiang, A. Litteneker, and D. Terzopoulos. Position-based multi-agent dynamics for real-time crowd simulation. In *Proceedings of the Tenth International Conference on Motion in Games*, page 10. ACM, 2017.

[8] T. Weiss, A. Litteneker, C. Jiang, and D. Terzopoulos. Position-based real-time simulation of large crowds. *Computers & Graphics*, 78:12–22, 2019.