

# GPU Crowd Simulation

Jeremy Shopf, Christopher Oat, Joshua Barczak  
Advanced Micro Devices, GPG



**SIGGRAPH**ASIA2008  
NEW HORIZONS

# Outline

- Motivation
- Global navigation
- Local navigation
- Spatial queries
- Conclusion

# Dynamic and Engaging World Through AI

- Global path finding along with local navigation and avoidance
  - Crowd dynamics similar to fluids
  - Agents “flow” towards the closest goal, along the path of least resistance
  - Froblins follow correct paths and do not “get stuck”
  - Local avoidance resolves fine scale collisions



The goal of the Froblins demo was to provide a path planning system for a large number of characters that was plausible and would be free of strange behaviors and glitches. Our solution was to use a hybrid system consisting of a global solver that would provide the shortest path to the goal on a coarse scale and a local system that would allow each agent to follow this path while navigating around other nearby agents and small-scale obstacles. We wanted everything to be dynamic, so in the demo we allow the user to place and move goals and obstacles such as poisonous gas.

# Video



# Global Navigation

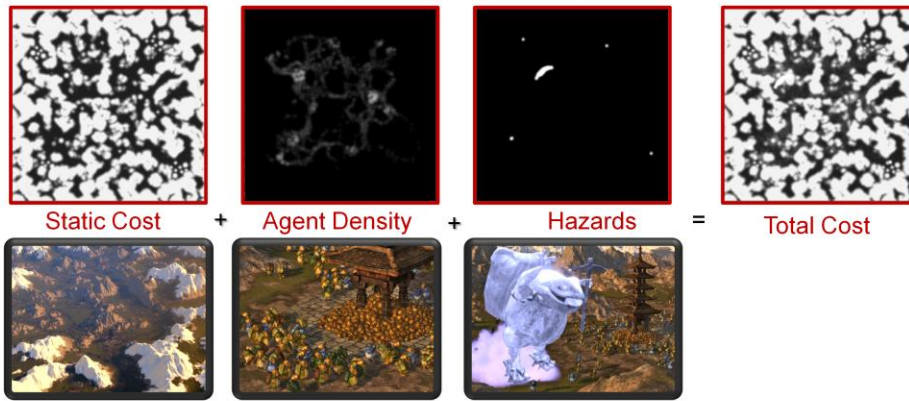
- Continuum approach [Treuille et al. 2006]
  - Smooth flow-like crowd movement
  - Congestion avoidance
  - Emergent behavior



For our global planner we chose a continuum approach similar to the Continuum Crowds paper at SIGGRAPH 2006. A continuum path planner has some inherent benefits such as smooth, flow-like crowd movement, easy incorporation of congestion avoidance, and exhibits some emergent phenomena such as lane formation. These type of behaviors add much realism so this type of approach was very attractive to us. The implementation in the 2006 paper allowed for large numbers of agents but we wanted something bigger and faster. Before I get to our implementation, let's discuss the details of a continuum approach.

# Continuum Model

- Model environment as positive cost function



The continuum method models the environment as a positive cost function that describes the cost or time of moving from one location to another nearby location. A region with a high cost should be prohibitive to an agent. In our application, we have three components to our cost function. The first, the static cost, contains the cost associated with the static terrain. Sharp slopes and mountain tops have a high cost while flat valleys have a very low cost. This component of the cost function also includes large structures such as tents and pagodas. The second component contains the cost of agent congestion. Each agent “splats” a radial function into the cost function additively. So if a large number of agents are in a certain area, the cost function is high in the location and other agents will be more apt to avoid that area. The last component is the Hazard component. This contains dynamic obstacles and hazards that the agents should avoid. This includes noxious gas clouds and the “Ghost Froblin” controlled by the user. All of these components are weighted accordingly and summed into a total cost function.

## Continuum Approach

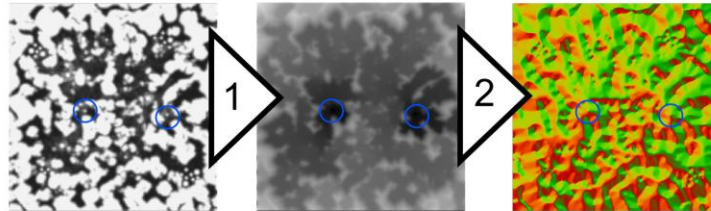
- Potential ( $\Phi$ ) = cost integrated along shortest path to goal.  
“Travel-time”
- $\Phi=0$  at goal, satisfies eikonal equation everywhere else:

$$\|\nabla \varphi(x)\| = F$$



Ok, well how is this cost function used to generate shortest paths to the goal? First a potential function is generated for the environment based on the cost function. The potential at a given location  $x$  is equal to the cost function integrated along the shortest path from  $x$  to the goal. So this function is the total cost or travel-time for an agent to get to the closest goal. How is this potential function generated? First, the potential is set to zero at the location containing the goal itself. This makes sense. If you are already at the goal, it doesn't cost anything to get there. Every where else, the potential is calculated to satisfy the eikonal equation, picture above. It is calculated such that the magnitude of the gradient of the potential function is equal to the cost function evaluated in the direction of the gradient ( $F$ ).

## Global Navigation Direction



By following the negative gradient of the potential, agents are guaranteed to follow optimal path [Kimmel and Sethian 2001]



We're talking more about the implementation of an eikonal equation solver in a second. First, how does one get the shortest path from the potential function? If you think of the potential function as the "travel-time" to the closest goal, the gradient of that function points in the direction in which the travel-time is increasing the most. So if you follow the negative of this gradient, you will always be moving as close to the goal as possible, which is logically the shortest path to the goal.



## Fast Marching Method [Tsitsiklis 1995]

- Similar to Dijkstra's
- One cell's potential calculated per iteration
- Not fast enough for our application
- Not readily parallelizable



The Continuum Crowds work used a technique known as the Fast Marching Method.

-This algorithm is very similar to Dijkstra's method.. The difference is in Step 2 where the potential is calculated. The problem of finding a solution to the Eikonal equation is a continuous one and Dijkstra's solves in the discrete domain and will always result in stairstep patterns. The Fast Marching Method computes the potential using a Upwind Finite Difference Approximation: solving a quadratic to compute the current potential based on the lowest potential neighbors along each axis.

## Fast Iterative Method [Jeong and Whittaker 2007]

- No ordered data structure
- Iteratively update *active* cells in parallel
- Active cells determined by convergence measure
- Active cells updated and culled in tiles



While the FMM was used in the Continuum Crowds work, the frame rates achieved were not suitable for interactive applications such as games. We sought to accelerate the eikonal solving step. The problem is the FMM is that it is not very parallelizable. This is due to the ordered data structure needed to track the cell with minimum potential and only that cell's potential can be determined each frame. The Fast Iterative Method is a recent work by Won-Ki Jeong and Ross Whitaker at Utah for solving the eikonal equation in parallel. This method maintains no ordered data structure. It works by updating the potential of many cells in parallel using the Upwind Finite Difference approximation also used in the FMM. Cells are removed from the active list in coherent blocks based on a convergence measure. Blocks of grid cells may be 'reactivated' if nearby blocks update cells that border them.

## Simplified Fast Iterative Method

- For smaller datasets, maintaining active tile list and testing convergence has negative performance impact
- Conservative iteration # determined empirically
- Solve 4 solutions simultaneously
  - 98% ALU utilization
- 5-39x speedup over CPU FMM\*

\*Configuration: AMD reference platform with AMD Athlon™ 64 X2 Dual-Core Processor 4600+, 2.40GHz, 2GB RAM, GPU: ATI Radeon™ HD 4870 Graphics. Motherboard: ASUSTek M2R32-MVP, Memory: DDR2-800 400 MHz, Operating System: Windows Vista® SP1.



## Implementation Thoughts

- High API overhead
  - API calls and state changes
  - Compute API would provide quicker thread launch
- Would benefit from shared memory
  - Perform several iterations for a tile in one launch  
[Jeong and Whitaker 2007]
  - Local Data Share (CAL/DX11/OpenCL)
- Will map easily to DX11 Compute Shader

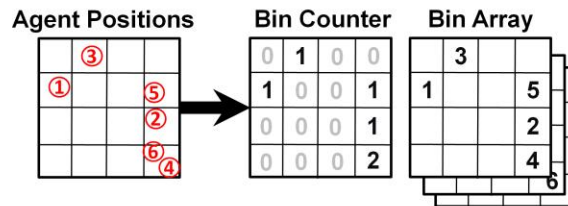
## Local Navigation and Avoidance

- Agents “sense” surroundings
- Update velocity based on positions and velocities of nearby agents/obstacles
- Inspired by Velocity Obstacle formulation [Fiorini and Shiller 1998]



We augment our coarse global navigation system with a fine-grained, higher frequency local navigation system. The basic idea here is that in the local system, agents “sense” their surroundings and update their velocities accordingly to prevent imminent collisions with other agents and obstacles. Before we can talk about how agents update their velocities, agents have to have a way to query what other agents and obstacles are nearby.

# Binning



- World space position mapped to 2D index
- Bin Counter: color buffer, tracks bin loads
- ID Array: depth array, stores binned IDs



We implement a GPU binning algorithm that sorts agents into spatial bins that can be queried by each agent. The output of this algorithm is a 2D color buffer containing the load/count of elements in each bin and a second texture array containing the IDs of the agents in that bin.

Our algorithm uses a 2D depth texture array and a single 2D color buffer to construct a data structure for storing agent information spatially. The depth texture array serves as our *Agent ID Array*. A given 2D texel address in this array serves as a bin. A single bin is a 1D array consisting of the same texel in each array slice. Each slice of the texture array contains a single agent ID (bin element). The agent IDs are stored in bins in ascending sorted order by agent ID. The number of agents that fall into a given bin may be less than the bin capacity (which is defined by the number of depth array slices). In order to efficiently query the agent IDs in a given bin we use store a bin count inside a color buffer.

## Binning Algorithm Overview

- Rasterize agents as point primitives to appropriate pixel
- Bind color buffer (bin counter) and depth texture array (ID Array)
- Render point with constant color and depth = ID
- Use depth testing to ensure one point is binned per pass (in sorted order)
- Additively blend points to color buffer to maintain bin counts

Please attend the “Parallel Computing for Graphics” course for more details  
Friday, 2:50pm  
Room 312



The binning data structure is populated by rasterizing a point primitive for each agent to the pixel that maps to the world-space position of the agent. The bin counter (a color buffer) is bound as the render target and the ID Array is bound as the depth buffer. The color of each primitive is a constant scalar and the depth of the primitive is set to  $\text{agentID} / \text{nTotalNumAgents}$ . The depth-test unit is used to reject all but the smallest ID and the color output is additively blended, effectively maintaining a count of how many agents ended up in each bin. There are many implementation details and performance optimizations that are left out here, but will be detail in the talk by Chris Oat in the “Parallel Computing for Graphics” course on Friday.

## Bin Queries

- Translate position to 2D index
- Fetch bin load from color buffer
- Fetch binned agent IDs from depth array
  - IDs are stored in array in sorted order

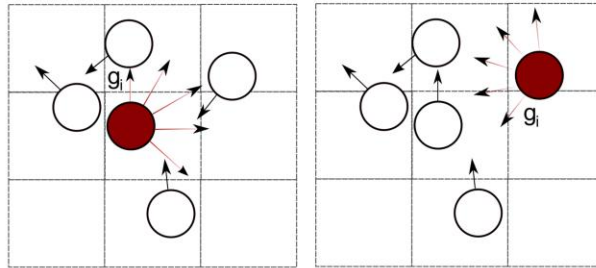


To find all agents near a particular world space position, the position is translated into a 2D bin address. Any translation function may be used, our world domain is square so a simple uniform grid was used to map world space positions to bins. Once the bin address is known, the bin load is read from the Bin Counter. This gives us the number of agents that are in the bin we are interested in. Finally, the each agent's ID in the bin is read from the Agent ID array.



## Agent Direction Determination

- Evaluate a fixed set of possible directions relative to global navigation direction



$g_i$  being the global direction determined by the global path planner.

We used five fixed directions in our implementation but one could easily use more for increased motion fidelity. This set of directions are rotated to align with the global direction,  $g_i$ . This way local avoidance is always relative to the direction the agent desires to move globally.

Using directions “to the right” of the global navigation direction allows us to ignore the resolution of which agent will turn each way.

## Agent Direction Determination

- Calculate fitness of each discrete direction
- Fitness based on minimum time to collision and angle relative to global navigation direction
- Choose direction with greatest fitness

$$fitness(v_i) = w_i t(v_i) + (g_i \cdot v_i).5 + .5$$



Fitness function is :  $F(v) = w_i * t(v) + (g_i \cdot v) * .5 + .5$

Where  $w_i$  is a per-agent weight that affects preference to move in the global direction or avoid nearby agents. This can be used to tweak aggressiveness of individual agents.

$t()$  is the function that determines time to collision

$g_i$  is the global navigation direction

$v$  is the direction to be evaluated

So the first term factors in obstacles and the second term factors in the angle relative to the global direction

## Agent Direction Determination

- Calculate fitness of each discrete direction
- Fitness based on minimum time to collision and angle relative to global navigation direction
- Choose direction with greatest fitness

$$fitness(v_i) = \boxed{w_i t(v_i)} + (g_i \cdot v_i).5 + .5$$



## Agent Direction Determination

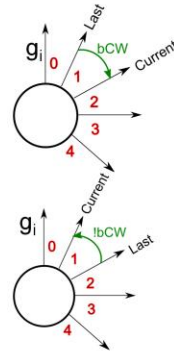
- Calculate fitness of each discrete direction
- Fitness based on minimum time to collision and angle relative to global navigation direction
- Choose direction with greatest fitness

$$fitness(v_i) = w_{it}(v_i) + (g_i \cdot v_i).5 + .5$$



# Oscillation Damping

- Only allow local direction changes to be in the same relative direction within N frames
- Achieved by weighting the fitness function by relative change in direction
- Track last direction choice and it's relative change from the previous direction per agent



$$fitness(v_i) = (w_i t(v_i) + (g_i \cdot v_i) \cdot 0.5 + 0.5) O_i$$

# Oscillation Damping

## Weighting the fitness function

nLastDir = <from texture>;

bLastTurnCCW = <from texture>;

bUpdate = ( nFramesMod == nFrame );

If( !bUpdate && bThisTurnCCW != bLastTurnCCW )

    fFitness = 0;



# Oscillation Damping

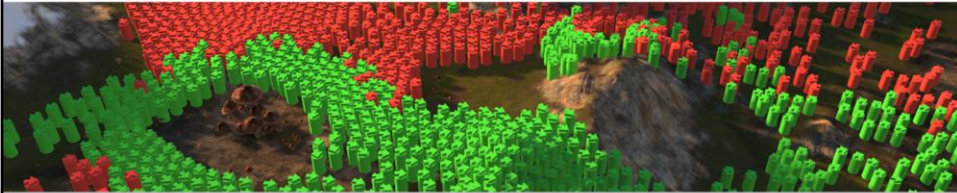
## Updating agent state

$\text{bLastTurnCCW} = ( \text{nDir} < \text{nLastChoice} ) ? 0 : 1;$

$\text{nLastTurn} = \text{nDir};$

## Results

- Real-time frame rates (>20fps) on ATI Radeon™ HD 4870\*
  - >65,000 agents simulated or
  - 3,000 highly detailed Froblins rendered and simulated or
  - 16,384 simplified agents rendered and simulated



\*Configuration: AMD reference platform with AMD Athlon™ 64 X2 Dual-Core Processor 4600+, 2.40GHz, 2GB RAM, GPU: ATI Radeon™ HD 4870 Graphics, Motherboard: ASUSTek M2R32-MVP, Memory: DDR2-800 400 MHz, Operating System: Windows Vista® SP1.





# Conclusions

- Massive crowd simulation entirely utilizing the GPU
- Novel binning algorithm
  - Please attend the “Parallel Computing for Graphics” course for more details. Friday, 2:50pm, Room 312.



# Thank You

- Game Computing Applications Group – Natalya Tatarchuk, Chris Oat, Joshua Barczak, Abe Wiley
- Dan Abrahams-Gessel
- Won-Ki Jeong

