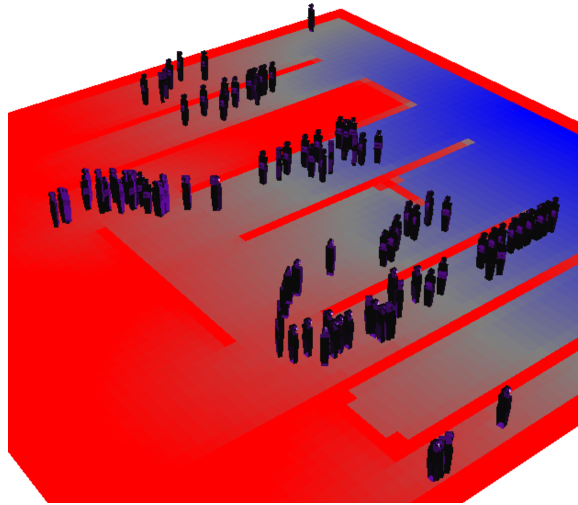


Figure 1: A crowd, with the motion-governing potential field for one group visible.

GPU Continuum Crowds

CIS 665 FINAL PROJECT FINAL REPORT

{mfickett, zarko}@seas.upenn.edu
University of Pennsylvania

Presented in [TCP06] is an algorithm for the simulation of large homogeneous crowds using a potential field model that simultaneously solves for global navigation and obstacle avoidance. When running on the CPU, the authors found that they could achieve smooth, interactive motion for large populations by simulating at five frames per second while rendering at twenty-four.

For this algorithm to see widespread use, however, we think that the brunt of computation should be moved to the GPU. This frees the CPU from spending cycles on what amounts to (in the context of games, at least) an environmental effect. To achieve this offloading, we will need to investigate methods to build the potential field in parallel.

1 Introduction

Convincing crowd simulation is important in producing believable realtime, interactive worlds. This implies three constraints: that the simulation is *convincing* (that is, it must only trick the observer into thinking that it is correct), that it is *realtime* (so it must not ask too much from the machine), and that it is *interactive* (so it should be able to deal with changes to the world made by the user and by other systems).

1.1 Objective

We plan to develop an implementation of the continuum crowds algorithm that computes as much as possible on the GPU and that can be used with minimal effort by application

developers, thus freeing the CPU for other tasks and, perhaps, enhancing simulation quality. We also plan to extend this algorithm in novel ways.

1.2 Technology

We will develop our project on relatively recent hardware (a Pentium M machine with a GeForce Go 6800 and Von Talon in HMS or machines in 100B) using either CUDA or Cg and Visual Studio 2005. We will use [JW07], as well as other available work, as a reference on solving the potential field equation. We would like to use Cg to allow the project to run on as many machines as possible; however, if we can't get an efficient implementation working with only Cg, we may need to move up to CUDA.

1.3 Design Goals

We intend to produce a functional reproduction of the continuum crowds algorithm while exploring some new extensions. In particular, we'd like to look at:

- Adapting the algorithm to higher dimensional space (either by creating new 3D textures or by coming up with ways to map non-planar 3D spaces to 2D space).
- Addressing environmental uncertainty (the original paper omits this)
- Dynamic level-of-detail for potential fields
- Adding richer goals or driving goals by some higher-level AI routine

We hope that the GPU implementation will provide enough of a speed-up that these extra features will be tractable in realtime.

1.3.1 Target Audience

Our project is targeted toward game and environmental simulation developers who want a way to drop in a believable crowd simulation.

1.3.2 User Goals and Objectives

As outlined in the introduction, the simulation should be believable or easily tweaked to be believable in the context that the developer is using it.

1.3.3 Tool Features and Functionality

The simulation would comprise a simulator core that handles the frame-by-frame updates (and any potential timeslicing) with hooks that developers can use to provide environmental data (such as collision maps or external agent positions).

2 Project Development Approach

2.1 Algorithm Details

The basic algorithm we'll be implementing simulates crowd behavior using potential fields. The smallest level of direct algorithmic control is a *group of people* rather than a single person. Each group gets its own potential field; every frame, the field is recalculated and individuals in a group are moved based on the potential field's gradient.

The potential field is built from several components both internal and external to the simulation. It can include information meant for collision detection (so that crowds don't try to walk into buildings), varying topography (so crowds try not to climb steeper slopes), 'discomfort' (used to keep members of crowds from running into other members or external agents), flow speed, goal locations and density.

The greatest obstacle we will need to surmount in porting this algorithm to the GPU is finding an efficient way to construct the dynamic potential field from the unit-cost aggregate field. Fortunately, other work has been done on solvers for the necessary type of equation; for example, in [JW07] Jeong and Whitaker present a GPU-based eikonal equation solver that runs seven times faster than the fast marching algorithm referenced in the continuum crowds paper.

2.2 Target Platforms

We'll be developing on the platforms the project is intended to run on: PCs with modern graphics hardware. We'll try to keep to cross-platform libraries as much as possible. Again, we would like to use Cg if possible, but if that turns out to be too limiting we will implement features in CUDA.

2.3 Project Versions

2.3.1 Project Milestone Report (Alpha Version)

By April 9, 2007, we should have a working implementation of the paper's crowd simulation algorithm on the CPU and the GPU with no extensions. We should also have some simple visualization program working to show a crowd moving around.

2.3.2 Project Final Deliverables

By the end of the project, we hope to have a simulation kit comprising:

- An implementation of the paper's algorithm on the GPU (including comparisons to the CPU-based algorithm)
- An implementation of one of the heavier extensions (such as extending the algorithm to 3D or using higher-level AI), accompanied by a report on the results of this attempt and on any trouble we encounter in undertaking it.
- An implementation of LOD or uncertainty modeling
- A visualization program that shows that extension in action

- A means of controlling an agent or agents to demonstrate interactivity of the simulation

2.4 Work Plan

2.4.1 Project Milestone Report (Alpha Version)

At the first milestone report (April 9), we will deliver the alpha version of the simulator as outlined in 2.3.1. We will consider at this point whether we will add or remove features and decide on which extensions we intend to implement. During the last phase of alpha development we will begin to work on extensions.

2.4.2 Project Beta Version

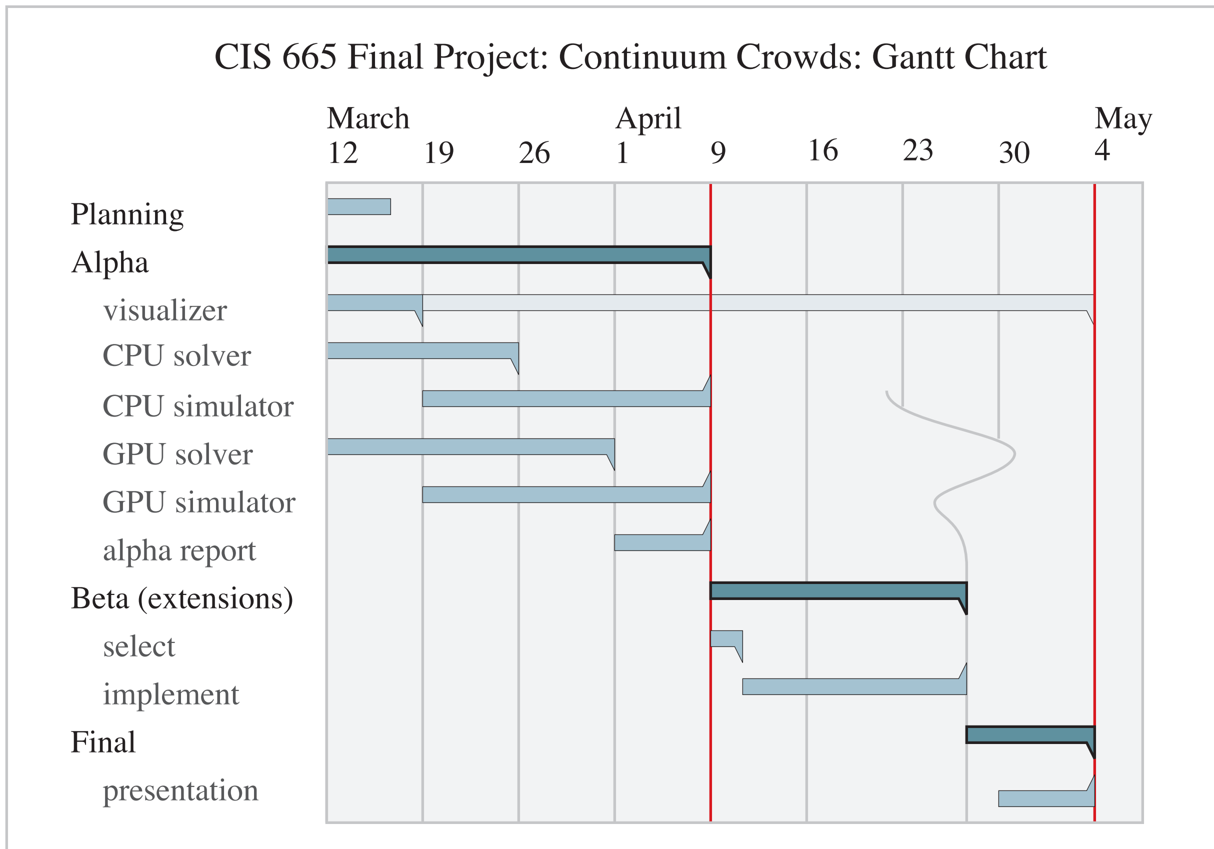
The beta version will be our feature-complete release. From then and until the final release we will spend time testing. While working up to this version we should be concentrating exclusively on extensions.

2.4.3 Project Final Deliverables

Our final deliverables (May 4) will include:

- Source and documentation for the simulator
- Source and documentation for the visualization demo
- Slides for the project presentation

2.4.4 Gantt Chart



3 Results

3.1 GPU Solver

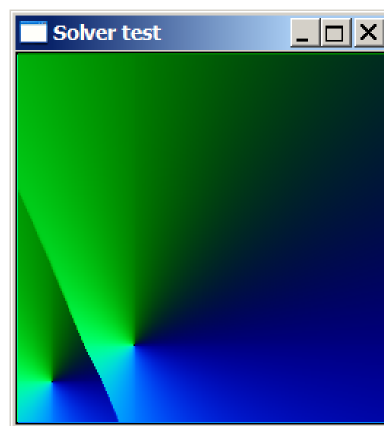


Figure 2: The gradient of a two-goal potential field as shown by our test code

The GPU solver was reasonably successful, achieving an average of a 3.4x decrease in runtime when compared with the CPU fast marching solver (in further comparison to [JW07], which claims about a 7x decrease). Larger input datasets had a greater difference

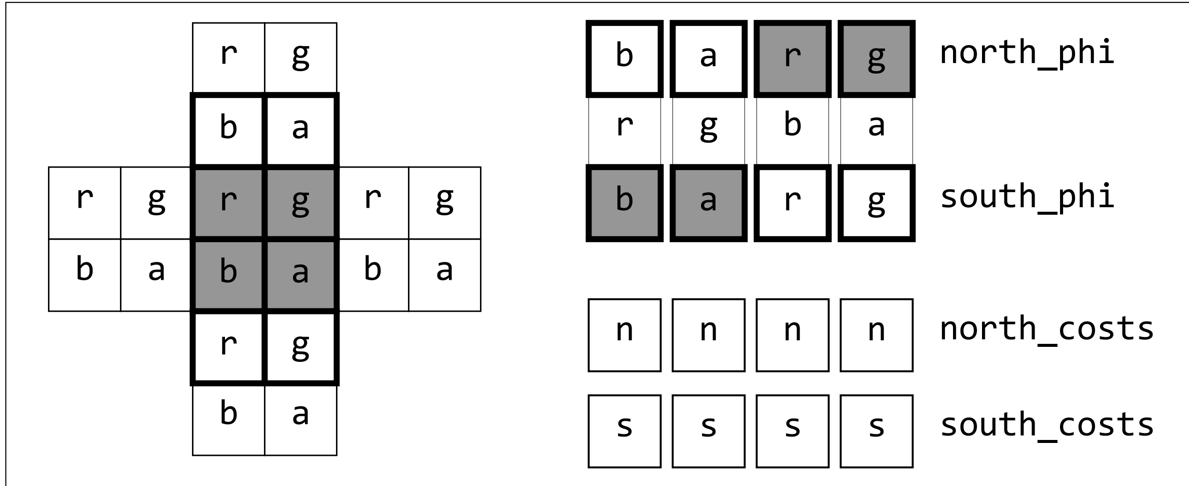


Figure 3: Calculating with multiple potential values in one fragment

in speed (favoring the GPU), perhaps explaining the apparent difference in benchmark performance (the data in [JW07] were roughly 256 times the size of our *square* and *cloudy* benchmarks). The GPU algorithm currently introduces some artifacts into its output data that can be lessened in severity by changing parameters (yielding a small but not insignificant decrease in performance). Though we planned to develop a fully 3D version of the solver, we were not able to complete it for lack of time: for the first half of the project we worked on getting the algorithm to run quickly enough to be useful, and in the second half we worked on lessening the severity of the artifacts in the output.

3.1.1 The Algorithm

The algorithm maintains an active list of groups of fragments ('tiles') that it enables and disables based on whether that tile has converged. To initialize, the algorithm must execute a reduction on an initially provided potential field to see which of these tiles should be active from the start (these will be the ones containing goals). In its inner loop, the algorithm has three stages. In the first stage, every active fragment is updated by examining its neighbors and (if necessary) lowering its potential according to the equation from [TCP06]:

$$(\frac{\phi_M - \phi_{m_x}}{C_{M \rightarrow m_x}})^2 + (\frac{\phi_M - \phi_{m_y}}{C_{M \rightarrow m_y}})^2 = 1$$

In the second stage, the 1-neighbors of each active tile are enabled and all active fragments are again updated. Finally, a reduction on the current potential field identifies which tiles contain fragments that have not converged. These tiles become the new active frontier. The algorithm continues to spin until there are no active tiles.

As suggested by [JW07], we try to keep as much on the GPU as possible. All tile operations are drawn out of vertex buffers, including reductions. We use hardware occlusion tests to determine when there are no active tiles remaining (by attempting to draw the field with tile-filtering on and then checking to see if no fragment was drawn). When tile overlapping is enabled we turn on depth checks so as to keep from having to recalculate known fragments. We also exploit the GPU's data-parallel instructions within fragment shaders to update four cells in the potential field at once. [JW07] did this in three

dimensions (by packing multiple Z-slices per pixel), but we were able to do it in only two. As in Figure 3 (where the shaded pixel is the one currently being calculated), we pack potential values into each color channel. By using Cg's vector conditional operator we can efficiently find the values associated with m_x and m_y as required by the crowds equation: for example, with values in vectors as in the figure, we could find four minimum ϕ-values at once using the expression

```
(north_phi + north_costs < south_phi + south_costs) ? north_phi : south_phi
```

3.1.2 The Interface

Early on, we developed a small framework to wrap calls to Cg and OpenGL to make it easier to develop our code. We knew that we would be dealing with lots of different shaders and render targets and keeping track of everything using raw OpenGL calls seemed like it would cause too many problems. It was also one of our stated goals to write code that could be easily integrated into other projects. The resulting framework, which we've called CgShim, ended up working fairly well for these purposes. It abstracts textures, FBOs, programs, and other necessary GPGPU objects into C++ objects. High-level commands can be issued to the objects; for instance, we have a single method on our FBO object that (when provided with the necessary program, tile grid, and program argument parameters) will execute a parallel reduction. Should we wish to port our code to a different or more abstract graphics API, we would only need to reimplement the virtual methods defined in the various CgShim objects. The CgShim framework is (as is much of our code) documented inline using doxygen.

The final interface to the GPU solver was designed to work in a pipeline with other GPGPU components. It accepts CgShim FBO and texture objects as input and writes to a specified CgShim FBO as output. Internally, the solver does a little bit of caching to maintain an initial list of active tiles (these are where the goals are located in the potential field and therefore are where the field should start growing from). This cache is only updated when the client of the solver notifies it that the goals have changed positions. There is also a provision to allocate the solver a certain amount of time it's allowed to work through a sort of manual timeslicing. This is meant to be used in situations where the cost of executing the solver must be amortized over a number of frames.

3.1.3 Performance

big A 1024^2 pixel field with square areas of differing cost.

cloudy A 256^2 pixel field with random cost.

square The 'traditional' field; 256^2 with two small goals and equal cost everywhere.

maze A 512^2 pixel field, most of which is occupied by a maze. It has equal cost floors and infinite cost walls.

tiny A 64^2 pixel field with equal cost everywhere; there is a single source near the middle.

We tested the performance of the solver on the CUDA machine in HMS (an Intel Xeon running at 3GHz with a GeForce 8800 GTX). The CPU solver is a no-frills implementation of fast marching using a Fibonacci heap to maintain the frontier. Our results, though not

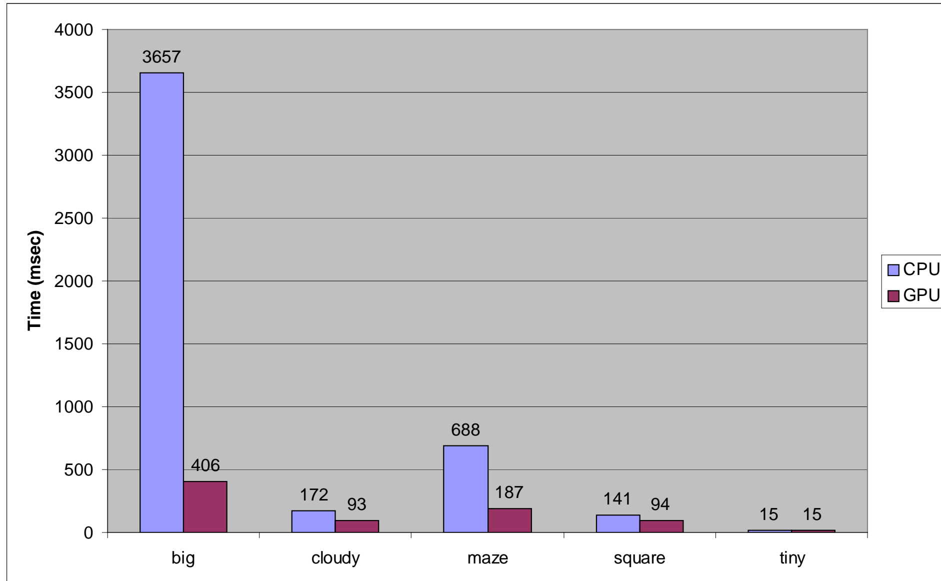


Figure 4: Algorithm performance on benchmarks (64x64 tiles, $\frac{1}{4}$-overlap)

as optimism-inspiring as those in [JW07], still suggest that using the GPU to approximate the 2D potential field (especially when larger fields are needed) is a viable option.

The GPU solver beats the CPU solver more and more decidedly as the size of the input increases. Added field complexity doesn't seem to affect the GPU solver's runtime, at least with small datasets (*cloudy* and *square* were random and uniform, respectively, but both ran in about the same amount of time).

3.1.4 Drawbacks, Future Work

A major drawback of the way we've implemented the solver is that it introduces artifacts into the output field. For our purposes these were not significant enough to impede the solver's usefulness for crowd simulation, but they may be unacceptable for other uses (or larger-scale uses). Allowing active tiles to overlap, decreasing the epsilon used to determine when values have converged, and changing tile size all can help in decreasing the prevalence of the artifacts. These changes come at a cost, but it does not appear to be very weighty: increasing overlap for *big* to one half of a tile caused it to run in 703 msec, about a 1.7 time increase. This is still slower than the CPU solution, which ran in 3657 msec.

While developing the algorithm, we kept the possibility of a full 3D extension in mind. We were easily able to extend the internal propagation fragment program (though one conditional operator required upwards of eight branches and we had to provide another texture that contained costs in the Z dimension). The tile-based frontier management system will also extend (and indeed was extended in [JW07]) with some work. We began to implement a 3D-to-2D map where Z-slices were lined up one after another in a rectangular texture. Tiles became volumes that could pierce through a number of these slices. An additional reduction step is necessary to 'flatten' out the Z-slices, after which our regular 2D reduction would work. We did begin work on the 3D solver and were able to get to this reduction step but did not have sufficient time to complete the algorithm. We were also able to get a 3D version of the fast marching algorithm running on the CPU (but doing this was very simple).

If we had more time to spend on the solver, we would have liked to finish the 3D solver and to work out how better to deal with the artifacts that appear in the 2D solver's output. It would have also been interesting to try a delayed-update scheme, where instead of updating the whole potential field we would update only interesting tiles (like those with people on them). Exploiting this temporal coherence might drive down computation time without having too terrible an effect on simulation believability.

3.2 GPU Simulator

The GPU simulator is certainly a successful alternative to the CPU simulator, and includes an interesting extension in dynamic map discovery and goal assignment. It can be seen to run at speeds better than that of the CPU simulator, and this without any significant effort towards streamlining (having focused on functionality and extensions).

3.2.1 GPU Simulator Performance Numbers

The following performance numbers were taken on Windows XP with 2x2.8Ghz processors and an NVidia GeForce 6800 (in Moore 100B).

At a 64^2 grid size and 80 crowd members, the GPU simulator vastly outperforms the CPU simulator, at 10fps v. 3.5fps, respectively. In the CPU setup, the simulator is the limiting factor (with the CPU solver about a third of the frame time); generating the velocity and speed fields are the most expensive steps, followed by generating the cost field. At 512 members, the CPU simulator drops to about 3fps, but the per-cell operations are still more expensive than the per-member (splatter) operations by a factor of about two. Switching to the GPU simulator, with the CPU solver, runs the same (64^2, 512) at just over 8fps; splattering and solving are by far the most expensive steps. With the all-GPU setup, this simulation runs at just over 7fps, with splattering about as expensive as solving (or just more).

On the GPU, the most expensive part of the splattering step is the nominal portion: converting people to splats, especially the predictive discomfort splatting, since this involves multiple repetitions of splattering with varying amounts of velocity projection. Running on the CUDA machine in HMS (with a GeForce 8800), the 10fps increases to around 35fps, and field resolutions as high as 256x256 and 8000 members run at about 15fps; splattering is relatively better as compared to solving the potential field, but is still the most expensive simulation step.

3.2.2 Algorithm Highlights

For the most part, the GPU simulator (like the CPU version) closely follows the equations governing crowd behavior presented in [TCP06]; however, there are some particularly interesting points, and some limitations, primarily in the splattering step. Crowd member positions were stored packed tightly in a 2D texture (approximately in a normal array); however, for crowd member interaction (via density, average velocity, and predictive discomfort) these stored coordinates had to be converted to in-location values.

Splattering was accomplished by drawing vertices from a VBO, indexed by one coordinate, and applying a vertex shader that looked up the appropriate position from texture and moved the vertex. (Additional information was stored in the vertex' color.) Then, a fragment shader blurred the output of that initial splattering according to

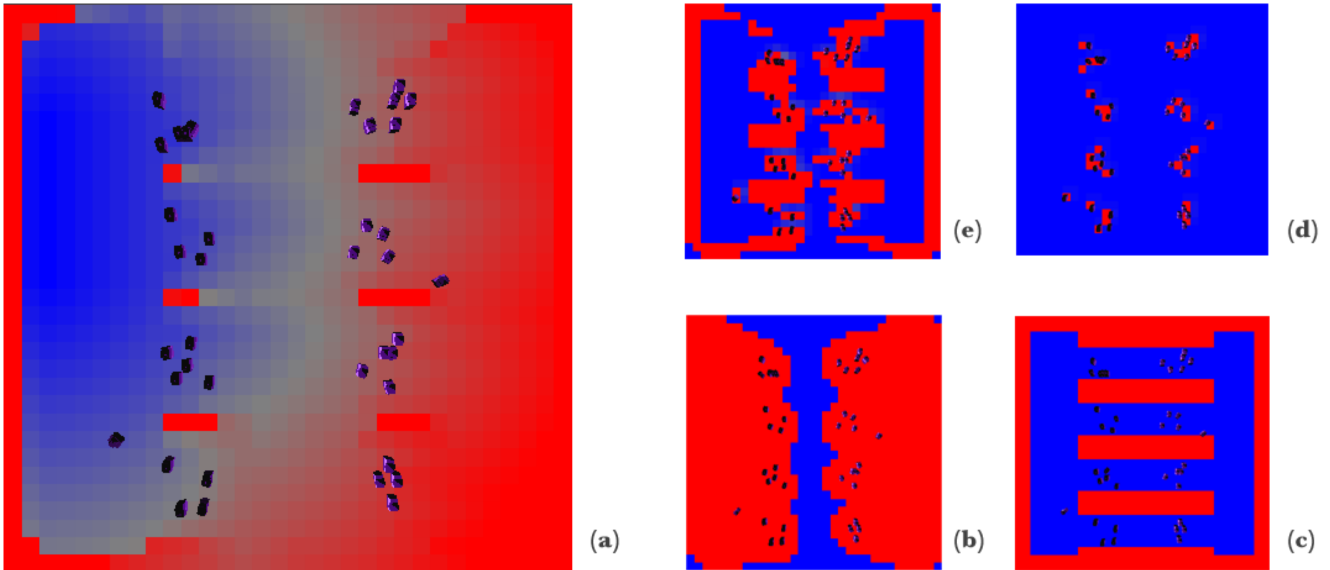


Figure 5: The potential field (a) is derived from the masked discomfort field (e) - and other fields that feed into the cost field. The mask (b) is applied to the original discomfort field, which is derived from a base set of 'walls' (c) and splattering (d).

equations from [TCP06]. This method for splattering has the drawback that multiple crowd members in the same cell won't contribute ideally to density and other fields, and any extra values stored in color will be blended. However, in practice there seems to be sufficient information transferred for reasonable movement.

3.2.3 Dynamic Discovery & Goals

In addition to the basic simulation (as derived from [TCP06]), we extended the simulation to support dynamic discovery of the map (discomfort and height fields) and associated dynamic assignment of goals. For both aspects, there is a mask, representing anything a crowd member has seen (this derives from splattering). For map discovery, the equivalent of a logical *and* is performed between the mask and the maps; what has not been seen is discounted in cost equations. Goal assignment takes place in two steps. First, the end goals are drawn against the mask. If visible, these end goals are used (as is the case without dynamic goal assignment). If not, an edge detection is run on the mask, and goals are assigned at the mask edge where there aren't walls (high discomfort).

With dynamic discovery and goal assignment enabled, the crowd members can be seen to explore the map and, when they find the goals, change course to convene at the end-goal location; the effect is very satisfactory. Visibility is currently tested crowd- instead of group-wide; changing this would be relatively simple, but would require significant overhead (and thus was not pursued).

Although not quite as expensive as splattering, dynamic goal assignment is a close second (to one splatter-step, though not to predictive discomfort splattering). Enabling dynamic map discovery significantly *increases* frame rate initially, owing probably to a simpler cost field and more widely distributed goals (meaning quicker parallel convergence) for the solver. However, once most of the map is discovered, the simulation runs at about 9.5fps (down from about 16.5fps), as compared to 10fps without dynamic discovery/assignment.

3.2.4 Limitations and Future Work

We weren't able to extend the simulator to 3D, although much of the code is generalized to any number of dimensions, and the equations are certainly all general. The major hurdle in converting dimensions was and would be texture formatting (which we explored to some extent) and, in a related vein, reworking some shaders which already make use of three or four channels per fragment.

Since most of the simulator work was spent on implementing features, little was spent on streamlining. Given that splattering is the most expensive step, investigating alternative methods is probably relevant, and experimenting with the number of discomfort-prediction steps is a key tradeoff to investigate. As far as GPU bandwidth, aside from reading back the positions (and, for informative display, the potential fields), little does and less needs to be moved on and off the GPU; this is particularly true given the GPU solver's pipelined interface. An interesting area to explore would be configuring the viewer to use instancing and/or geometry shaders, so the positions could be provided to it on-card (possibly along with potential, height, and other fields for display), thus lowering or eliminating transfer time.

The dynamic goal assignment's test for end-goal visibility also could be improved greatly; at worst, it should use a parallel reduction to test for goals which passed the mask; and at best could use an occlusion test, discard non-goal points, and simply count drawn fragments to determine goal visibility in a single pass.

Aside from speed, the flexibility of interpretation of the discomfort field lends itself to exploration. One interesting experiment would be to implement path-wearing. Adding interfaces for interactivity and easier input of initial configurations would also make the simulator more user-friendly.

References

[JW07] Won-Ki Jeong and Ross T. Whitaker. A fast eikonal equation solver for parallel systems. In *SIAM conference on Computational Science and Engineering*, 2007.

[TCP06] Adrien Treuille, Seth Cooper, and Zoran Popovic. Continuum crowds. *ACM Transactions on Graphics*, 25(3):1160–1168, 2006.