DEGREE PROJECT IN TECHNOLOGY,
FIRST CYCLE, 15 CREDITS
STOCKHOLM, SWEDEN 2018

Crowd Simulation Using Flow Tiles

LEO ENGE

FELIX LIU

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

DEGREE PROJECT IN TEKNIK,
FIRST CYCLE, 15 CREDITS
STOCKHOLM, SWEDEN 2018

Simulering av folkmassor med Flow Tiles

LEO ENGE

FELIX LIU

KTH ROYAL INSTITUTE OF TECHNOLOGY
SKOLAN FÖR ELEKTROTEKNIK OCH DATAVETENSKAP

Abstract

Crowd simulations are being used in an increasing number of different applications, like evacuation scenarios, video games and movie special effects This creates a demand for crowd simulators that are simple to use and accessible to users of varying backgrounds. We will study the flow tile method proposed by Chenney [1], which provides an intuitive way of interactively designing divergence free velocity fields for various applications. A reimplementation of Chenney's method will be given and the implementation will be evaluated in terms of user-friendliness and how well the use of static spatially defined velocity fields suits crowd simulation. Furthermore the possibility of using the velocity fields for other related applications such as mobile robotics will be touched on as well.

Sammanfattning

Simuleringar av folkmassor används i ett ökande antal olika tillämpningar, som evakueringsscenarion, datorspel och specialeffekter för film. Detta skapar en efterfrågan efter simulatorer som är enkla att använda och tillgängliga för användare från olika ämnesområden och bakgrunder. Vi kommer att studera flow tile-metoden som Chenney [1] föreslår. Metoden är ett intuitivt och interaktivt sätt att skapa divergensfria hastighetsfält för olika tillämpningar. En omimplementation av Chenneys metod kommer att ges och implementationen kommer att evalueras i termer av användarvänlighet och hur väl användningen av hastighetsfält som är statiska och definierade i rummet passar för simulering av folkmassor. Vidare kommer möjligheten att använda hastighetsfälten för andra liknande tillämpningar, som robotik, att diskuteras också.

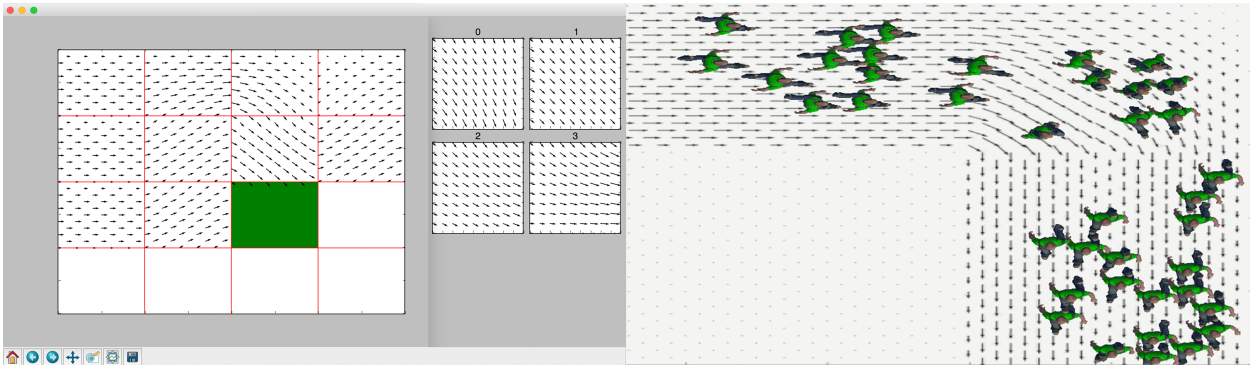


Figure 1: Image of the user interface for setting flow tiles to the left. Screenshot from the crowd simulation with agents moving along the velocity field to the right.

1 Introduction

Simulating large and dense crowds as realistically as possible is a big challenge with many different applications, including computer games, traffic design and evacuation planning. With increasing demand on being able to simulate crowds in a variety of situations and applications, a simple and intuitive way of being able to design flows is desirable. Because of the widely varying applications where crowd simulation is used, not all users might have a natural science or mathematical background. As such, they might not be familiar with the mathematics or theory behind fluid dynamics and other sciences used in crowd simulations. We aim to use the flow tile method proposed by Chenney in [1] to create a simple and intuitive way of designing velocity fields which targets a wide audience. A reimplementation of the flow tile method will be given, and we will evaluate the method in terms of user-friendliness and also the utility and realism of the resulting simulations.

1.1 Background and Related Work

The flow tile method uses a velocity field to drive the motion of the crowd forward, with the idea being that the user will interactively be able to design different velocity fields for the crowds to follow in an interactive way. The use of velocity fields to describe motion is commonly used in other applications such as fluid mechanics and particle simulations. In these applications, different physical properties of the motion can be described mathematically within the vector field defining the velocity. An example of this within fluid dynamics is describing incompressible fluids, fluids that cannot be compressed to higher densities by outside pressure, by the velocity field being divergence free. Furthermore divergence free vector fields can always be expressed using a vector potential, which in fluid dynamics is referred to as the stream function. The flow tile method uses both of these ideas, where incompressibility represents the way people tend to stay a certain distance away from each other and the stream function is used to store the discrete velocity field.

Different approaches to the problem of being able to create velocity fields in an interactive way already exist. For instance, a method proposed by Patil et. al. in [5] allows a user to draw trajectories in the environment which is then used to create so called *navigation fields*. These navigations fields will then guide agents to specified goal positions in as smooth a way as possible. Another possible approach is to create trajectories and simulations based on real videos of actual crowds [6] [7]. This has the advantage of actually mimicking real crowd behavior but might be more limited in

terms of the possible simulations that can be created.

The flow tile method uses a continuum based model, in the form of the previously introduced velocity field, to drive the motion of the crowds. The continuum model for crowd behavior was originally proposed by Hughes in [10]. Other similar approaches include the work described in [9] and the aggregate method proposed by Narain et. al. [3]. These models have been further extended as well to incorporate the ability to model turbulence effects more accurately [16] [17]. This is of great importance in evacuation scenarios and other very high density simulations, since turbulent behavior has been observed in real crowds at very high densities and had previously not been able to be simulated with the continuum approaches. A common problem in modeling crowd behavior using continuum based, fluid like models is the lack of individuality of the agents that arise when modeling them as a homogeneous fluid. However, continuum based models can often handle larger and denser crowds compared to more discrete, individual methods of controlling the agent's motion.

Other types of simulations include those based on social forces models [8] [11], where the agents are affected by other agents and the environment in the form of forces pushing them in different directions. These forces could for example arise due to agents coming to close to an obstacle or another agent, making them move apart, or as an attractive force due to coming close to a friend or street performer or similar. These kinds of simulations obviously put much greater weight on inter-agent interactions that can appear in crowds, with less emphasis on being able to model large and dense crowds with satisfactory performance.

2 Implementation

The method of flow tiles, proposed by Chenney [1], is based around giving a user freedom to design velocity fields in an interactive way, while an underlying algorithm ensures that important physical properties of the flow, like incompressibility, are satisfied at all times. To achieve this a tiling algorithm is used, which will ensure that these constraints are always satisfied and that any incomplete tiling built by the user can always be completed in some way. This effectively takes away that responsibility from the user, so that the user can focus on designing the velocity field the way they want.

2.1 Tile Model

To ease the design of the velocity field it is broken up into square tiles, which we from here on will refer to as flow tiles. The tiles store information about the velocity field on their edges and corners. In the corners each tile will store a velocity, and on each edge the total flux across that edge. The flux is defined to be positive going to the right and up in the grid and the flux is restricted to being an integer multiple of some base flux Φ_b, such that the fluxes across each edge becomes

$$\begin{aligned}\Phi_{left} &= m_{left}\Phi_b \\ \Phi_{right} &= m_{right}\Phi_b \\ \Phi_{top} &= m_{top}\Phi_b \\ \Phi_{bottom} &= m_{bottom}\Phi_b,\end{aligned}$$

where $m_{left}, m_{right}, m_{top}, m_{bottom} \in \mathbf{Z}$. This gives a simple integer representation of the fluxes across each edge and makes the incompressibility constraint very simple,

$$m_{left} + m_{top} - m_{right} - m_{bottom} = 0. \qquad (1)$$

The negative signs in equation (1) come from the way we defined the flow to be positive going right or up. This way a positive flow on, for instance, the top of the cell is a flow going into the cell, which corresponds to a positive sign in (1). The base flux can be chosen by the user and adapted to different applications.

Furthermore, we restrict $m_{left}, m_{right}, m_{top}, m_{bottom}$ by choosing $m_{x,min}, m_{x,max}, m_{y,min}, m_{y,max} \in \mathbf{Z}$

such that

$$\begin{aligned} m_{x,min} \leq & m_{left} \leq m_{x,max} \\ m_{x,min} \leq & m_{right} \leq m_{x,max} \\ m_{y,min} \leq & m_{top} \leq m_{y,max} \\ m_{y,min} \leq & m_{bottom} \leq m_{y,max}. \end{aligned}$$

We emphasize here that $m_{left}, m_{right}, m_{top}, m_{bottom}$ are restricted to integer values meaning that these bounds on the fluxes define a *finite* set of possible flow tiles. When designing a given tiling we use only tiles within this finite set, which is advantageous since this restricts the number of tiles that can be placed in each slot to be finite, and also restricts the fluxes across the edges to not be unreasonably large. In practice it was found that even having relatively narrow bounds on the edge fluxes, say between -4 and 4, still provides enough flexibility to design most kinds of simple flows.

Finally, the velocities stored in the corner of each tile are also chosen from a finite set of possible corner velocities. This set can also be specified by the user and it was found that even having just a few choices will still permit the creation of a large variety of velocity fields.

2.2 Velocity Field

To be able to simulate the crowds we must have a continuous velocity field in each flow tile. Obviously we cannot store a completely continuous velocity field, but instead a discrete field of some kind can be stored, and from this the velocity can be calculated in the desired point. We begin by dividing each flow tile into a $n \times n$ grid as in figure 2. The edges of each flow tile are, as figure 2 suggests, at the flow tile-relative coordinates $x = -0.5, n + 0.5$, $y = -0.5, n + 0.5$ and these edges are in contact with the adjacent flow tile.

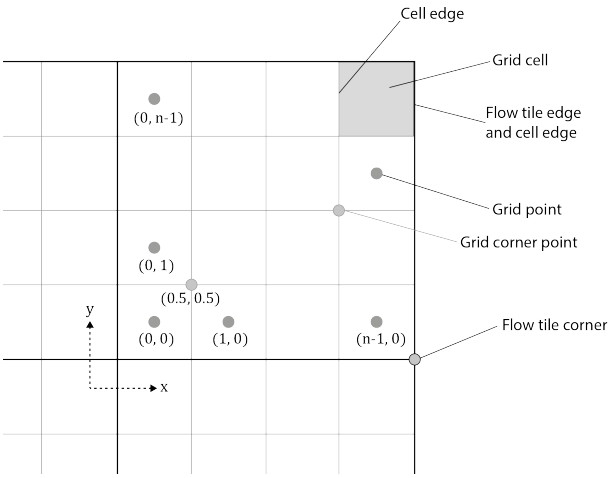


Figure 2: Flow tile and its inner grid

Stream Function

One idea on how to store the velocity field is to store the normal component of the velocity at each edge of the grid cells (see figure 2). From these values the velocity can be linearly interpolated at any given point. This would mean storing $2n(n+1)$ values per flow tile. Chenney [1] suggests a different approach, using the so called stream function. In a divergence-free field the velocity can be written as the curl of a vector field, its vector potential [14]. The curl is always perpendicular to the field it operates on and hence the vector potential is always normal to the plane in which the velocity field is in. The velocity field will always be in the xy-plane and therefore the vector potential must always be in the z-direction. We set the vector potential to be $S(x, y)\hat{\mathbf{z}}$, where $S(x, y)$ is the *stream function* and $\hat{\mathbf{z}}$ is the unit vector in the z-direction. According to the definition of the vector potential we then have

$$\mathbf{v}(x, y) = \nabla \times (S(x, y)\ \hat{\mathbf{z}}) = \left(\frac{\partial S}{\partial y}, -\frac{\partial S}{\partial x}, 0 \right). \quad (2)$$

Using the finite difference instead of the derivative we get

$$v_x(x, y) = S(x, y + 0.5) - S(x, y - 0.5) \quad (3)$$
$$v_y(x, y) = S(x - 0.5, y) - S(x + 0.5, y) \quad (4)$$

Chenney [1] suggests storing the stream function at each of the grid corner points and then interpolate the stream function for any other point. Using this approach has two advantages. The first is that only $(n+1)^2$ values needs to be stored in difference to the $2n(n+1)$ values for the above mentioned approach. The other advantage is that the stream function has a natural connection to the flux over the edges.

The flux over the edges are given as boundary conditions imposed by the user. It can be calculated by integrating the velocity component normal to the edge along the length of the edge. At the left edge the positive direction for the flux is into the flow tile. The flux at the left edge, f_{left} is therefore given by

$$f_{left} = \int_{edge} v_x(x,y)dy =$$

$$\int_{edge} \frac{\partial S}{\partial y}dy = \int_{S(-0.5,-0.5)}^{S(-0.5,n+0.5)} dS =$$

$$S(-0.5,-0.5) - S(-0.5,n+0.5) \quad (5)$$

We set $S(-0.5,-0.5) = 0$ and by the same method for each of the edges we get

$$S(-0.5,-0.5) = 0 \quad (6)$$
$$S(-0.5,n+0.5) = S(-0.5,-0.5) - f_{left} \quad (7)$$
$$S(n+0.5,-0.5) = S(-0.5,-0.5) + f_{bottom} \quad (8)$$
$$S(n+0.5,n+0.5) = S(-0.5,n+0.5) + f_{top} \quad (9)$$

From this we now have a stream function value in each corner of the flow tile. Bezier interpolation will be used to obtain the stream function values in the rest of the grid and to be able to use Bezier interpolation four stream function values are needed in each corner. The user has also set the velocity in each of the corners and to determine three more stream function values in each corner we assume that the velocity is constant over the whole grid cell in the corner. Rearranging (3) and (4) we get, for the bottom left corner,

$$S(-0.5,0.5) = v_x(-0.5,0) - S(-0.5,-0.5) = v_{x,bottom,left} - S(-0.5,-0.5) \quad (10)$$

$$S(0.5,-0.5) = S(-0.5,-0.5) - v_y(0,-0.5) = S(-0.5,-0.5) - v_{y,bottom,left} \quad (11)$$

$$S(0.5,0.5) = S(-0.5,0.5) - v_y(0,0.5) = S(-0.5,0.5) - v_{y,bottom,left}. \quad (12)$$

The velocity $\mathbf{v}_{bottom,left}$ is the velocity imposed by the user. In the same manner four stream function values can be determined in each corner giving a total of 16 stream function values. Given the 16 values bicubic Bezier interpolation can be used to determine the stream function values for each of the grid corner points.

Discrete Velocity Field Grid

From the stored stream function values the velocity can easily be calculated for some of the points in the flow tile using (3) and (4). Far from all desired velocities though can be calculated from the stored stream function values. Chenney [1] suggests that the velocity can be determined in any point by interpolating the stream function for the points needed. Ideally this would be done using Bezier interpolation, but that proves to be too time consuming to be done for each agent in runtime. Another idea is to do linear interpolation on the stream function since that is less time consuming. Doing linear interpolation on the stream function would result in two steps of calculations, first interpolating the stream function in four points and then calculating the velocity using these values. Instead we suggest storing the velocity directly in a discrete grid and linearly interpolate the velocity in any other point from this grid. This will only require interpolating the velocity in one point each time. It will consume more memory as

we store $2 \times n^2$ values instead of $(n+1)^2$ values as for the Stream function. But memory is not as big a problem as runtime calculation speed.

Even though the stream function will not be used to calculate velocities at runtime it will still be used to generate the discrete velocity field. This is because the stream function, as explained above, easily can be chosen to satisfy the edge flux conditions and since it can used for the Bezier interpolation.

2.2.1 Bezier Interpolation

One dimension

In one dimension, *Bezier interpolation* is a method for interpolating a smooth curve from a set of so called control points [15]. If three control points are used the interpolated curve will be a quadratic curve and if four control points are used the interpolated curve will be cubic. In figure 3 we have the four points P_0, P_1, P_2, P_3 and the aim of the cubic Bezier interpolation is to find a cubic curve that starts in P_0 tangent to the linear curve from P_0 to P_1 and ending in the point P_3 tangent to the linear curve from P_2 to P_3. The distance between the points P_0 and P_1 should represent the rate at which the curve moves towards point P_1 before turning towards point P_2, and the same for the distance between the points P_2 and P_3.
To satisfy all of these conditions the one-dimensional cubic Bezier curve is defined as

$$\mathbf{B}(t) = (1-t)^3\mathbf{P}_0 + 3(1-t)^2 t\mathbf{P}_1 + 3(1-t)t^2\mathbf{P}_2 + t^3\mathbf{P}_3 \quad (13)$$

where $P_i = (x_i, y_i)$ are the control points and t is a parameter with $0 \leq t \leq 1$. To find the y-value representing a given x-value the Bezier interpolation can be stepped through with small Δt until B_x equals the given x-value, then B_y will be the desired y-value.

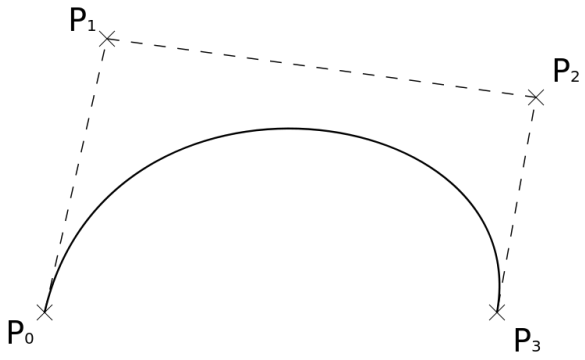


Figure 3: Example of a Bezier curve interpolated from four controlpoints.

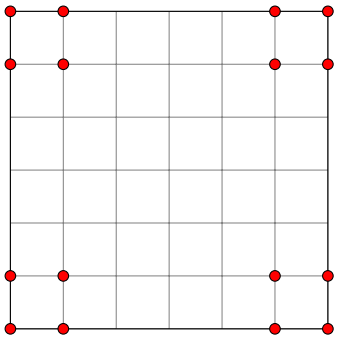


Figure 4: A flow tile grid with red points representing the control points where the stream function value is set.

Two dimensions

The Bezier interpolation can be extended to two dimensions. For this, 16 control points will be necessary instead of the four needed in the one-dimensional case. In this project we have the special case where all the 16 control points are neatly aligned in a grid, see figure 4. The two-dimensional Bezier interpolation can then be done as a repetitive implementation of the one-dimensional Bezier interpolation. First, with the grid in figure 4 in mind, one-dimensional cubic interpolation is done along the two top rows and the two bottom rows to get the situation in figure 5. Then, one-dimensional cubic Bezier interpolation is

done for each column until the entire grid is filled.

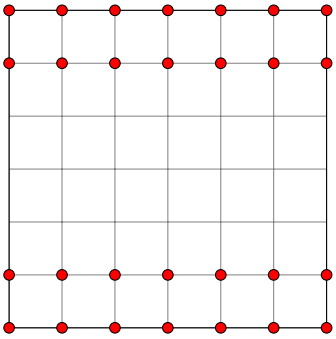


Figure 5: A flow tile grid after the first set of Bezier interpolations. Red points represent the points where the stream function value is set.

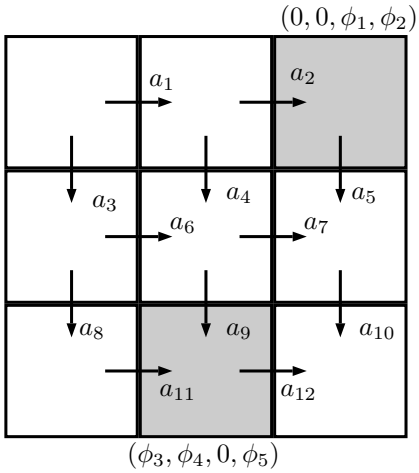


Figure 6: An incomplete tiling. The gray tiles are occupied and the occupying tile's fluxes are listed, in clockwise order starting from the top, next to them.

2.3 Tiling Algorithm

The tiling algorithm, proposed by Chenney in [1], is a way to ensure that a complete tiling can be built step by step while ensuring that all the physical constraints, like incompressibility, will be met in the end. This is done by the algorithm searching for valid tiles to place in a given position in an incomplete tiling. A user is then free to choose from this set of valid tiles, and no matter what choice the user makes, the resulting unfinished tiling will always be completable using only the tiles in the finite tile set we restrict ourselves to after choosing the minimum and maximum fluxes as described in 2.1. Basically, we use a function which takes an incomplete tiling and an open tile slot as input and returns a set of valid tiles from the tile set for the slot. To achieve this we will use what are called linear programs.

2.3.1 Linear Program

Briefly, a *linear program* is an optimization problem of the form

$$\begin{aligned} \textbf{Maximize:} \quad & f(\mathbf{x}) = \mathbf{c}^T\mathbf{x} \\ \textbf{Constraints:} \quad & A\mathbf{x} \leq \mathbf{b} \\ & \mathbf{x} \geq \mathbf{0}. \end{aligned} \tag{14}$$

Here, $\mathbf{c}, \mathbf{x}, \mathbf{b} \in \mathbf{R}^n$, A is an $n \times n$-matrix with real entries and $f(x)$ is a scalar function called the objective or goal function. The example in 14 is given in *canonical form* and it can be shown that a much more general set of problems can be converted to this form. For instance, it is possible to have equalities in the constraints on the variables, variables are allowed to be bounded by negative values and can be bounded both from above and below and the goal function could also be sought to be minimized. Linear programs where the variables are restricted to integer values, as is the case in our tiling algorithm, are called *integer programs* and are in general significantly more difficult to solve. Integer programming is known to be NP-complete.

Constraints

The algorithm uses integer programs to which we add different constraints corresponding to different restrictions in the tilings, e.g. incompressibility. The variables in our integer programs are the edge fluxes in the tile grid. An example of what the constraints in the integer program correspond to can be seen in figure 6, which shows an incomplete tiling, and the fluxes across each edge which are the variables in our

integer program. For this tiling we would get the following constraints due to our requirement that the flow be divergence free.

$$\begin{aligned} a_1 + a_3 &= 0 \\ a_1 - a_2 - a_4 &= 0 \\ a_3 - a_6 - a_8 &= 0 \\ a_4 + a_6 - a_7 - a_9 &= 0 \\ a_5 + a_7 - a_{10} &= 0 \\ a_8 - a_{11} &= 0 \\ a_{12} + a_{10} &= 0 \end{aligned}$$

Note that the gray placed tiles in figure 6 are already assumed to be divergence free, so we do not add the incompressibility constraint for them. Furthermore we have constraints on each of the variables due to either existing tiles in the grid, or our choice of $m_{x,min}, m_{x,max}, m_{y,min}, m_{y,max}$. These will be as follows for our example in figure 6.

$$\begin{aligned} m_{x,min} \leq & a_1, a_6, a_7 \leq m_{x,max} \\ m_{y,min} \leq & a_3, a_4, a_8, a_{10} \leq m_{y,max} \\ & a_2 = \phi_2 \\ & a_5 = \phi_1 \\ & a_9 = \phi_3 \\ & a_{11} = \phi_5 \\ & a_{12} = \phi_4 \end{aligned}$$

The inequality constraints correspond to the specified minimum and maximum fluxes, and the equality constraints are due to existing tiles in the grid, since two adjacent tiles must have the same flux on the edge that they share.

Finding Valid Tiles

For each edge in the tile grid we use two integer programs which are identical apart from the fact that one maximizes the valid flux for that edge and one minimizes it. This gives a range of valid fluxes for each edge of the tile to be placed, from this all tiles with fluxes in these ranges are generated. These however are not guaranteed to be valid tiles for the slot, so further filtering is needed. The filtering is done by adding constraints corresponding to each candidate tile to the linear program. For each edge that the candidate tile would occupy we restrict the flux of that edge to be exactly that of the candidate tile. After this the resulting integer program is tested for feasibility, basically we test if it has a solution. If the resulting model is feasible the tile is returned as a possibility for the given slot, and we continue testing the rest of the candidate tiles the same way.

Network Flow and Performance

The kinds of integer programs we deal with are of the same types as those in network flow problems, where one has a graph with edges pushing flow onto the vertices in the graph. In our case we can think of the tiles as vertices and edges between the tiles as edges in the graph. Network flow problems also require the net flow into each vertex to be zero, which is equivalent to the incompressibility constraint in our case. Our problem of maximizing and minimizing the flux across a given edge can be viewed as a special case of the maximum flow problem. The integral flow theorem for this problem guarantees, since we have integer constraints on our flows, that the maximum and minimum flows obtained are integers [4]. This is of great benefit to our algorithm since we do not have to use expensive integer program solvers, but can instead use a significantly more efficient general simplex linear program solver. To note is that there exist more specialized algorithms for maximum flow problems, e.g. Ford-Fulkerson's algorithm, but in our case using these were not found to be necessary since the simplex solver gave satisfactory results.

To solve the integer programs the open source lp_solve library[1] was used, which includes a standard simplex linear program solver. The ability to make small modifications to integer programs provided in the lp_solve library is especially useful when solving for particular tile slots. This since only the objective function changes when computing the valid flux ranges for each of the four edges, the constraints depend only on the configuration of the unfinished tiling. The lp_solve library also includes ways to

[1] `http://lpsolve.sourceforge.net/5.5/`

test the integer programs for feasibility, which is necessary when filtering the valid tiles to be placed in each slot.

2.3.2 Boundary Conditions

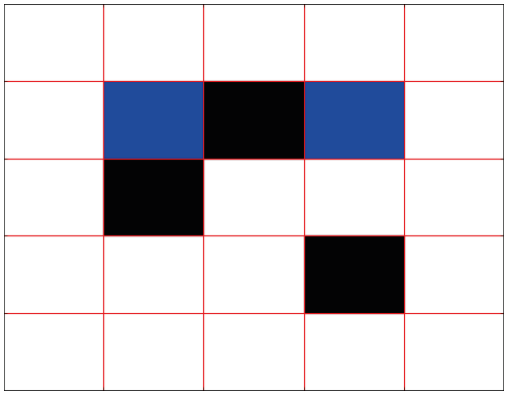


Figure 7: User view of specifying obstacles, black squares are obstacles already in the environment and blue squares are tiles where new obstacles can be placed.

One of the advantages of the flow tile method is the ability to specify boundary conditions before the construction of the velocity field. This is done by simply adding constraints to specific edges in the linear program. In the example given in figure 6, we have implicitly included the boundary condition that all edges on the boundary of the entire field should have zero flux. We are, however, not restricted to boundary conditions on the edge of the tiling, it is also possible to set boundary conditions on internal edges in the grid. These could for example correspond to physical obstacles that the agents are not able to pass through. Boundary conditions also turn out to be of great importance in the tiling algorithm, due to the fact that they restrict the number of valid tiles to place in each slot. It was found that in a linear program without boundary conditions the number of valid tiles per slot became very high, often in the hundreds, even when the range of valid fluxes was chosen restrictively. Most of these valid tiles were often completely irrelevant to the type of scenario that was being created, only resulting in much slower performance of the program.

Adding obstacles to the tiling can be done by choosing tile slots that the obstacle occupies and constraining the fluxes on all the edges of the occupied tiles to be zero in the linear program. This will effectively include the obstacles in the tiling, since nothing will be able to flow into those tiles. However, some care must be taken when adding obstacles, since they can often not be added arbitrarily due to the fact that the resulting linear program might not be feasible. To deal with this problem we can test all the slots in the tiling one by one by adding the obstacle constraint and testing the linear program for feasibility, the same way we did when filtering valid tiles in the tiling algorithm. This way we can create a visualization for the user which tiles obstacles can be placed in, see figure 7.

2.4 Overlapping Tilings and Collision Detection

Since the velocity fields are incompressible, when only a single velocity field is used to drive the crowd collision detection is not usually necessary, since the incompressibility of the flow will automatically ensure that agents stay reasonably apart from one another. However, a limitation of this is that the agents paths can never cross, due to the fact that streamlines cannot intersect. This can be addressed by layering tilings on top of each other, having different agents follow different velocity fields. This however, means that collision detection will be required in the overlapping regions.

The collision detection used in our implementation was a simple pairwise collision avoidance similar to the one used by Shabo in [2]. Each agent is assigned a collision radius, and when another agent comes within that distance, the agent's velocity is adjusted according to the following formula:

$$\vec{v} = v_{ref} \frac{\vec{r_1} - \vec{r_2}}{||\vec{r_1} - \vec{r_2}||}.$$

$\vec{r_1}$ and $\vec{r_2}$ are the positions of the pair of agents and v_{ref} is a parameter which needs to be set according to different implementations. Furthermore, to improve performance, collision detection is not performed pairwise between all agents in the entire simulation, but only within each flow tile. That is, each agent will check for collisions with other agents in the same flow tile. If an agent is within the collision radius of the edge of a flow tile, however, collision detection is also performed with the agents in that neighboring cell.

3 Evaluation

3.1 User Experience and Interface

One of the main advantages of the method is the possibility for an inexperienced user to create tilings and simple flow fields. Since the user can get a clear view of what the tiling under construction looks like, as well as being able to see the flows in the candidate tiles for each slot, no real knowledge about the underlying mechanism is required.

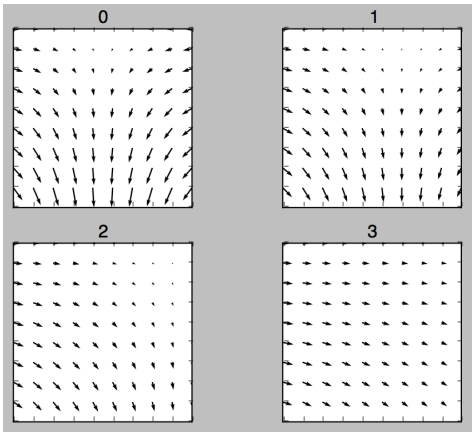
One issue when it comes to the ease of use of the method is setting boundary conditions. Boundary conditions cannot be set arbitrarily, since then the resulting linear program might not be feasible. This will happen if the boundary conditions set are so restrictive that there is no way to create a tiling which has incompressible flow and satisfies all the boundary conditions. On the other hand, if the boundary conditions are not restrictive enough a very large number of tiles can become valid for each slot making the choice for the right one difficult. We feel that special care should be taken in how the user can interact with the program to set the boundary conditions, since the boundary conditions are an important part of the tiling algorithm.

Overall the visualization is key for the user-friendliness of the program. At each step it is possible to create an intuitive visualization for the user of the current progress and the candidate tiles, as can be seen in figure 8a and 8b. If only simple scenarios are required the boundary conditions issue can be addressed by having a number of simple preset boundary conditions that the user can choose from depending on the scenario to be created. In conclusion, we do think that Chenney's flow tile approach can be implemented in a way such that even more inexperienced user can be able to create simple velocity fields for different applications.

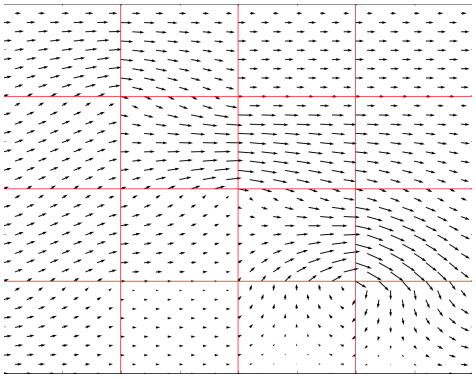
3.2 Emergent Behavior

The resulting movement of the agents did have many similarities to the road map approach used by [2] at moderate densities, especially when having only one tiling which the agents follow. When multiple tilings were layered on top of each other, the lack of incompressibility really shows in the regions where agents collide, since the only adjustment to the agents' velocities due to other agents was done when they collided. In real crowds people can plan ahead and anticipate a collision before it occurs and adjust accordingly. Having some way to introduce incompressibility in the colliding regions might help with this. One possibility could be to use the unilateral incompressibility constraint described in [3] in the overlapping regions to help the agents stay reasonably apart from each other. The aggregate way in which the method of Narain et. al. [3] is built up makes it especially suited to integrating parts of their work, like the unilateral incompressibility, into other simulations.

The phenomenon of simulated crowds moving in a very homogeneous way is definitely present in our implementation as well. This is especially clear when the agents are moving in a straight line, the small variations in velocity within each tile does not really become noticeable. However, when the agents are made to turn, the agents do seem to take more varied paths and have a more varied appearance. This should be due to the fact that within each tile, the speed at which the agents travel is quite similar throughout. When a turn is introduced it produces a noticeable difference due to the fact that some agents need to travel a further distance to make the turn, while still carrying a similar speed to the other



(a) User view of the candidate tiles for a slot.



(b) User view of a finished 4x4-tiling after construction.

Figure 8

agents. This leads to some agents taking longer to complete the turn and thus gives a more varied appearance to how the agents move.

We do believe that a crowd which in a visual way appears somewhat realistic can be achieved using Chenney's flow tile method, especially for moderate densities. While the movement of the crowd might not accurately represent how real humans would move around but when looked at the movement might appear realistic to a general viewer.

4 Discussion

4.1 Realism and Applications

One of the main ways the method of using flow tiles enables the integration of a very intuitive user interface is the fact that the motion of the agent is controlled on a spatial level, rather than by features that vary over time like density or relative distance to other nearby agents. The velocity field created using the flow tile method is constant in space and time and do not depend on the dynamics of how the crowd moves. This enables the velocity field, and in turn the way the agents move, to be determined by a user before the simulation is run.

Human Crowds

A trade-off due to this is that more dynamic ways of simulating crowds more accurately mimic the way crowds of people interact in reality, where each person adjusts the way they move around in real time based on what people around them are doing, obstacles etc. The constant velocity field used in the flow tile method does not directly take these things into account. All in all, the flow tile method should be more suitable for application where behavior that precisely mimics real crowds is not necessary. This could include applications within digital entertainment such as video games or CGI in movies. These kinds of applications could also benefit from the ease of use of the method greatly, since professionals within these fields might not be familiar with the intricacies of crowd dynamics and simulation. In those applications, a somewhat realistic appearance of the crowd might be enough, and small imperfections in the simulation might not be as noticeable.

Robotics

Another possible application where the spatial approach could be useful is within the field of robotics. This is an emerging field in this age when automation of simple tasks becomes more and more widespread.

The use of velocity fields to guide robots is something that has been discussed before, see for instance the work by Kelly et al. [12] or Dixon et al. in [13]. These both describe the use of a *desired velocity field* to guide the robots around the environment for which they propose control algorithms to steer the robots along the velocity fields. These approaches both propose some restrictions on the velocity fields in regards to smoothness and boundedness but these should without too much issue be able to be met with the flow tile approach. However, some care might be needed in the interpolation of the velocity field between the tiles' edges to ensure a smooth transition between the tiles. As an added benefit these kinds of applications do not require the velocity fields to be designed to mimic human movement. Integrating the flow tile method with the use of velocity fields to guide mobile robots is an interesting application, which could probably benefit from the intuitive interface within which the velocity fields are designed as well.

4.2 Improvements and Future Work

As discussed in sections 2.3.2 and 3.1, boundary conditions in the tilings are of great importance to the building of the tilings. Thus, to make the method as user friendly as possible an interactive way of setting boundary conditions is a big possible improvement to the method. At this stage, an interactive way of adding obstacles has been implemented, but having the user also be able to set the fluxes across specific edges in an interactive way would be highly beneficial. This could improve the possibility of designing flow fields around specific environments and situations, allowing velocity fields to be created with specific scenarios in mind.

Further improvements could be made when it comes to interpolating velocities in the discrete points they are stored to form a continuous velocity field. An issue can arise here in the edges of the tiles, since the only constraint on an edge that is shared between two tiles is that the flux across the edge is the same from both tiles. This does not, however, impose a strict restriction on the direction the flow approaches the edge from, since flux is a one dimensional quantity, and as such cannot contain detailed information about the two dimensional quantity that is the velocity around it. A sharp difference between the direction of velocities close to an edge of a flow tile is therefore possible, which in turn leads to the velocities interpolated from points on two sides of an edge to be unnatural and not fitting with the rest of the field. A more advanced interpolation scheme than the linear one used in this implementation could create a smoother transition in velocities between these kinds of edges which should lead to an increased perceived realism of the simulations.

5 Conclusions

The method of using flow tiles to simulate crowds will not be able to simulate crowds as realistically as some of the other methods mentioned in section 1.1. The more goal oriented approaches, where each agent is walking from a given starting position and has a goal position in mind, more closely mimic the way humans walk in real life. This makes it possible to create more complex simulations where agents are moving from different starting points to different goals, while some internal models make sure to handle things like collision detection and interactions between agents. Where the flow tile method shines, however, is when it comes to building simple flows interactively without much prior knowledge about crowd simulations. When it comes to realism, every single agent might not move in a way you would expect to see in a real crowd, but as a whole the crowd could move in a way that at least appears realistic. This kind of simulation can have applications as well, the most obvious application being computer games and other types of CGI and digital entertainment in general. In these applications the appearance of the crowd is key, rather than the actual behavior.

There are a couple of improvements that can be made to make the simulations more realistic. As mentioned in section 4.2 the interpolation algorithm can be developed further to make the velocities at

the edges of two adjacent flow tiles match better. Also a more heterogeneous crowd behavior can be achieved using multiples layers of tiling with agents following different velocity fields. This will require more refined collision avoidance than used in this project. These are the two major improvements to reality that should be implemented. Still, the method of flow tiles will not be the best method to achieve absolute realism, but with advances in fields like robotics and digital entertainment, absolute realism might not be required for all applications

Being able to create divergence free in general can have applications outside of crowd simulation as well. With continuing advances in the field of robotics, velocity fields being used to control mobile robots is a real possibility. Within robotics, realistic movement is often not required either, but it could still benefit greatly from the simple way in which Chenney's flow tile method permits the design of velocity fields. All in all, we do feel that the flow tile method has a lot of interesting possible applications, especially were simplicity in design is required.

References

[1] Chenney, S. 2004, Flow Tiles, *Proceedings of the 2004 ACM SIGGRAPH / Eurographics symposium on computer animation*, pp. 233-242.

[2] Shabo, J. 2017, *High Density Simulation of Crowds with Groups in Real-Time*, Master's thesis, KTH Royal Institute of Technology.

[3] Narain, R., Golas, A., Curtis, S. and Lin, M.C. 2009. Aggregate Dynamics for Dense Crowd Simulation. *ACM Trans. Graph. 28*, 5, Article 122 (December 2009), 8 pages.

[4] Jungnickel, D. (2008) *Graphs, Networks and Algorithms* 3rd ed., Berlin: Springer, p. 156.

[5] Patil, S., van den Berg, J., Curtis, S., Lin, M. C. and Manocha, D., 2011, Directing Crowd Simulations Using Navigation Fields, *IEEE Transactions on Visualization and Computer Graphics*, vol. 17, no. 2, pp. 244-254.

[6] Bisagno, N., Conci, N., and Zhang, B., Data-Driven crowd simulation, *2017 14th IEEE International Conference on Advanced Video and Signal Based Surveillance* (AVSS), Lecce, 2017, pp. 1-6.

[7] Lee, K.H., Choi, M.G., Hong, Q., Lee, J., Group behavior from video: a data-driven approach to crowd simulation, 2007, *Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pp. 109-118.

[8] Helbing, D. and Molnár, P. Social force model for pedestrian dynamics. *Phys. Rev. E* **51**, pp.4282-4286, May 1995.

[9] Treuille, A., Cooper and S., Popović, Z., Continuum Crowds, *ACM Trans. Graph. 25*, 3 (July 2006), pp. 1160-1168.

[10] Hughes, R.L., 2003, The Flow of Human Crowds, *Annu. Rev. Fluid Mech. 35*, pp.169-182.

[11] Gayle, R., Moss, W., Lin, M.C. and Manocha, D., 2009, Multi-Robot Coordination using Generalized Social Potential Fields, *2009 IEEE International Conference on Robotics and Automation*, pp. 106-113

[12] Kelly, R., Bugarín, E. and Campa, R., Application of Velocity Field Control to Visual Navigation of Robots, *IFAC Proceedings Volumes*, vol. 37, iss. 8., 2004, pp. 537-542.

[13] Dixon, W.E., Galluzo, W.E., Hu, G. and Crane, C., Adaptive velocity field control of a wheeled mobile robot, *Proceedings of the Fifth International Workshop on Robot Motion and Control, 2005. RoMoCo '05.*, pp. 145-150.

[14] Kundu, P.K., Cohen, I.M., Hu, G. and Dowling, D.R.(2015), *Fluid Mechanics* 6th ed., Waltham, MA, USA: Academic Press.

[15] Sauer, T., (2013), *Numerical Analysis: Pearson New International Edition* 2d ed., Boston: Pearson International.

[16] Golas, A., Narain, R., Lin, M.C., Continuum modeling of crowd turbulence, *Phys. Rev. E.* **90**, Published 2014-10-28.

[17] Golas, A., Narain, R., Lin, M.C., A Continuum Model for Simulating Crowd Turbulence, *ACM SIGGRAPH 2014 Technical Talks* .