Eurographics/ACM SIGGRAPH Symposium on Computer Animation (2004)
R. Boulic, D. K. Pai (Editors)

Flow Tiles

Stephen Chenney

University of Wisconsin – Madison

Abstract

We present flow tiles*, a novel technique for representing and designing velocity fields. Unlike existing procedural flow generators, tiling offers a natural user interface for field design. Tilings can be constructed to meet a wide variety of external and internal boundary conditions, making them suitable for inclusion in larger environments. Tiles offer memory savings through the re-use of prototypical elements. Each flow tile contains a small field and many tiles can be combined to produce large flows. The corners and edges of tiles are constructed to ensure continuity across boundaries between tiles. In addition, all our tiles and the resulting tiling are divergence-free and hence suitable for representing a range of effects. We discuss issues that arise in designing flow tiles, algorithms for creating tilings, and three applications: a crowd on city streets, a river flowing between banks, and swirling fog. The first two applications use stationary fields, while the latter demonstrates a dynamic field.*

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism

Keywords: tiles, tiling, fluid simulation, velocity field

1. INTRODUCTION

Flows are used to drive the motion of many natural phenomena, and their representation and creation is a vital part of many animation applications. Simulation of fluids and gases is the most obvious example but velocity fields are also used to guide agents [Rey99] and force the motion of natural features, such as grass [PC01] or clouds [DKY*00]. Applications of flows are limited, however, by two significant problems: flows are hard to design if specific flow features are desired, and the resolution required to represent detailed motion severely limits the scale of effects that can be generated. This paper presents flow tiles, an approach to creating velocity fields that addresses design and scalability.

Each flow tile defines a small, stationary region of velocity field. They can be pieced together to form large stationary fields and then used to drive fluids, crowds or other effects. Figure 1 shows one such field. Different stationary fields can be interpolated in time to generate dynamic flows. Flow tiles share two primary advantages with other tiling applications, such as texture or terrain tiling, due to their re-use of a small set of prototypes (the tiles). Firstly, tiles make design easier and more efficient. Not only is it fast and simple to piece tiles together to create something larger, but the outcome of placing a tile is obvious and does not depend on blending, interpolation or other inter-tile effects. Furthermore, while tiles may encode a design that is difficult to create, they can be replicated by an unskilled user. For example, terrain tiles in computer games [Pea01] empower novice level designers but also constrain them to a consistent style, making their designs useful to the wider community. The second major advantage is that the data contained in each tile need only be stored once, no matter how often it is used. Tileable texture images highlight this feature, and flow tiles share it.

As with any tile set, to create visually seamless tilings the tiles themselves must meet certain requirements. The corners and edges of tiles are constrained by continuity requirements where tiles meet. Aperiodic square tilings that avoid obvious repetition artifacts (such as Wang Tiles [Sta97, CSHD03]) require the creation of tiles with particular edge combinations. Flow tiles also have constraints imposed by conservation of mass and boundary conditions; for many applications whatever flows into a tile must flow out again. In Section 3 we detail the design requirements for flow tiles and present an interpolation-based method for generating tile sets.

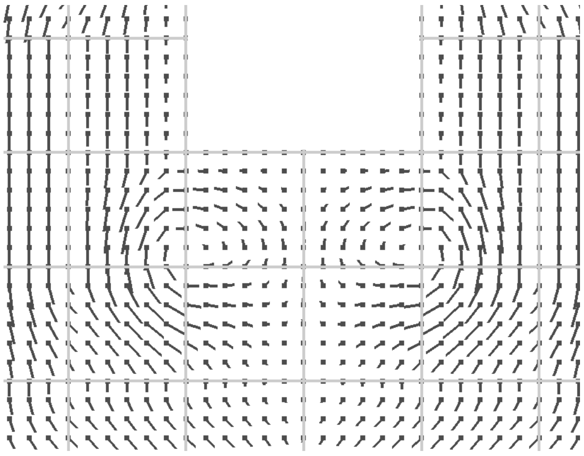
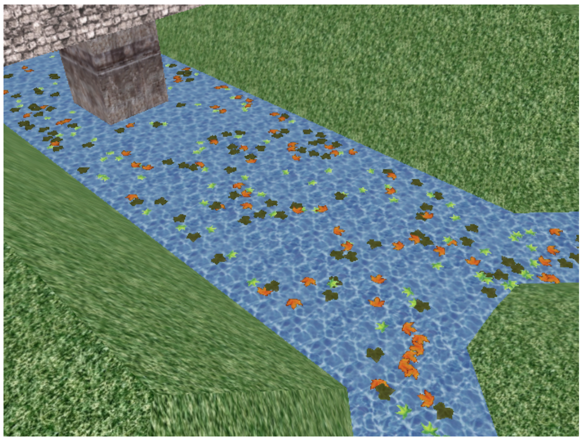



Figure 1: *Flow tiles drive a river flow. Each of the tiles in the zoomed grid at the top contain small examples of divergence-free velocity fields. A user interactively placed the tiles, including interior and exterior boundary conditions. In the section of tiling shown, boundary conditions were used to force the flow around a bridge pylon. Below is the flow embedded in an application, used to drive leaves in the river.*

Environmental conditions typically impose boundary conditions on flows; a river stays within its banks and flows to the sea, crowds do not pass through walls, and so on. Flow tiles support interior and exterior boundary conditions on their tiling, and can be conveniently inserted into applications using a texture-mapping like approach. Boundary conditions on the entire tiling impose constraints on the placement of individual tiles within the field, not just those that abut a boundary. For instance, if a flow that conserves mass is used to fill a dead-end alley, the net flow must be zero across the opening into the alley, even if there are many tiles placed within the alley. In Section 4 we characterize the constraints on tile placement and present algorithms for laying out tiles.

This paper contributes a new approach to designing and storing flow fields that improves their controllability and efficiency. Flow tiles rely on a designer to create plausible fields, although the constraints built into the tiles and tiling algorithms aid the designer in satisfying important physical requirements; flow tiles will not let you lead a crowd into a blind alley. They sacrifice the guaranteed plausibility of other methods, such as simulation, but gain storage and computational efficiency in addition to direct control.

We envision the use of flow tiles primarily in real-time applications where large-scale, smooth flows are required to drive animation. In Section 5 we present two examples: the advection of agents to form a crowd flowing through city streets, and the creation of a river through complex banks and around obstacles. These applications use stationary flows (the field itself doesn't change over time). We also discuss time-varying flows, and show their application to swirling fog.

2. RELATED WORK

Flow tiles are a procedural method for generating velocity fields. Several approaches to this problem exist, based on superposition of flows, spectrum methods, and interpolation.

Sims [Sim90], Wejchert and Haumann [WH91], Barrero et. al. [BPC99], Rudolf and Raczkowski [RR00] and Perbet and Cani [PC01] all use superposition – the linear combination of basis flows – to generate wind fields that drive other motion. Their primitives include constant velocity linear fields, vorticity elements, moving waves, and random fields. While providing a convenient interface for designing flows over open regions, these superposition methods are not well suited to fixed boundary conditions because it is very difficult to select and weight the basis flows such that the combination results in the required conditions.

Spectral methods generate spatially and temporally periodic flow fields that have the advantage of being trivially tileable. Among spectral methods, Shinya and Fournier [SF92] use an observation based model while Stam and Fiume [SF93] used the Kolmogorov spectrum, a model that describes the amount of turbulence present at various spatial and temporal scales. While useful in cases where turbulent detail is required, there is little control over the details of the field and no way to meet anything other than periodic boundary conditions. Flow tiles offer similar memory savings through tiling, but with greater control over boundaries and without periodicity.

Interpolation based schemes require a user to specify a few velocity or potential field values and then interpolate to fill a larger region. Neyret and Praizelin [NP01] is one example of this approach. Interpolation suffers from difficulties in controlling the final result. For example, while Weimer and Warren's [WW99] subdivision scheme generates smooth flows that satisfy the physically-based PDEs, the final flow does not necessarily interpolate the original velocities. Nor does the subdivision maintain boundary conditions. Similarly, the projection onto a divergence free field of Neyret and Praizelin might change the flow significantly. Users have no direct way of controlling these effects, apart from specifying an overwhelming number of sample points.

Physically-based simulation approaches (see Stam [Sta03] for some recent results), face limitations on the size of the grid that can be solved in real-time (or even reasonable time). Total grid widths for real-time performance are limited to about two orders of magnitude more than the smallest resolvable feature size in each dimension. In three dimensions the situation is even worse, which led Rassmussen et. al. [RNGF03] to develop an approach for combining, via interpolation and superposition, 2D simulations and a 3D Kolmogorov field to obtain 3D results. To increase the effective period of the Kolmogorov field, they suggest using two fields with different periods. They generate temporal changes in the Kolmogorov field by interpolating in time between two fields – a technique that we also employ. While Rassmussen et. al.'s techniques build larger simulations by combining smaller pieces, they do not address the design issue nor support fixed boundary conditions.

From a design perspective, Treuille et. al. [TMPS03] control smoke simulations using a key-frame based interface and constrained optimization. Users set keyframes on the appearance of the smoke and the system solves for a simulation that passes through the keys. However, their approach is expensive, does not offer WYSIWYG design, and is only effective on small problems due to the underlying use of simulation and optimization.

Texture and terrain tilings have seen great use in computer graphics. Glassner [Gla98a, Gla98b] provides an overview of infinite tilings of the plane, including aperiodic tilings that are better than periodic at hiding repetition artifacts. Wang Tiles [GS87], first introduced to graphics by Stam [Sta97] and extended by Cohen et. al. [CSHD03], are square tiles that support aperiodic tilings. Stam provides a hierarchical tiling approach that guarantees aperiodicity, while Cohen et. al. use a randomized, scan-line order approach that requires fewer basis tiles. We also work with aperiodic, square tilings, but our tiling algorithms are designed to meet boundary conditions and support placing tiles in any order. Neyret and Cani [NC99] generate texture tilings on arbitrary surfaces using triangular tiles that support continuity. We share with their work an emphasis on the corners of tiles. In the area of terrain tiles for computer games, Peasley [Pea01] discusses many of the issues surrounding the creation of tiles.

3. DESIGNING FLOW TILES

Tiles must provide a set of flow fields that can be combined to form large tilings. In this paper we discuss only 2D tiles,

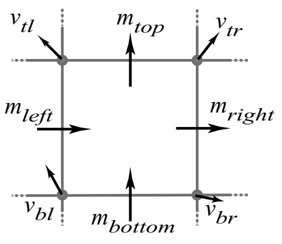


Figure 2: *Our method for indexing tiles required eight numbers: one for each corner indicating the velocity at that corner, and one for each edge indicating the flux across the edge.*

although most of the discussion extends naturally to 3D, space-filling tiles. We use square tiles due to their simplicity for tiling, similarity to texture maps, ease of representation and convenience for numerical computations. Square tiles somewhat restrict the domains we can tile, an issue we cover in Section 5. With applications in mind we focus on *divergence-free* tiles, or those that conserve mass. We allow flows to be warped when placed into applications which increases their usefulness, although at the expense of any guarantees about compressibility. Our choice of divergence-free tiles has many consequences for the representation and design of tiles, which we discuss below. The tiles we present are stationary (the field does not change over time) but by combining tilings over time we can generate dynamic flows. Finally, we assume that flow tilings are going to exist inside larger environments and will be required to meet specific boundary conditions.

3.1. Characterizing Flow Tile Sets

We choose to work with finite tile sets, as this is an explicit way of restricting memory usage and also simplifies the user interface. For efficient tiling, we require a way of indexing particular tiles in the tile set. For example, Stam [Sta97] indexes tiles with their edge "colors", Neyret and Cani [NC99] use the corners (they have only one edge per combination of endpoints), and Cohen et. al. [CSHD03] use both edges and corners. Tiling algorithms use rules to determine the types of edges and/or corners that will be used in the tiling, and then choose a tile from the set that matches the requirements.

The flow tile indexing strategy is based upon the four velocities at the corner of each tile, and the net flow, or *flux* across each edge of the tile. Flux is also referred to as the volume flow rate in the fluids literature. We focus on corners because they are the places where four tiles overlap, and can be used to determine edges [NC99]. We explicitly track the flux across each edge because it is essential in controlling the divergence-free properties of tiles. The labeling scheme is shown in Figure 2.

Each corner index for a tile is a positive integer that indexes an array of possible corner velocities. The set of velocity options are provided by a user and there are no constraints on the choices, although including the zero velocity is very useful for meeting boundary conditions.

Each edge index is an integer that multiplies a base unit of flux to determine the total flux across the edge. Flux is defined as the integral of the normal component of the velocity at each point along the edge. In our diagrams we indicate the flux with an arrow normal to the tile edge, but note that the flux depends on the velocity at every point along the edge. There may also be velocity components tangential to the edge but they do not influence the flux. Fluxes are positive for flow toward the right and up, and negative for flow toward the left and down. We require a finite set of flux quantities to define horizontal tile edges, and another set to define vertical edges; only tiles with matching corners and fluxes can be placed together.

Our flux definition scheme, using a base multiplied by an integer, arises from a desire for divergence-free tiles. Such tiles have the property that whatever flows in also flows out, which is important for most applications. The divergence-free condition requires

$$f_{left} + f_{bottom} = f_{right} + f_{top} \tag{1}$$

where f_{left} is the flux across the left edge, and so on. We must choose edge fluxes that can be combined to form tiles that meet this condition. Furthermore, we would like to maximize the ability to combine tiles in a tiling, so given a set of edges and corners, we want as many valid tiles as possible. We hence choose fluxes that are integer multiples of a fixed unit of flux. The designer chooses the *base flux*, f_{base}, and the minimum and maximum multiple for each direction, $m_{x,min}$, $m_{x,max}$, $m_{y,min}$, $m_{y,max}$. We then have:

$$\begin{array}{rl}
f_{left} = m_{left} f_{base} & m_{x,min} \le m_{left} \le m_{x,max} \\
f_{right} = m_{right} f_{base} & m_{x,min} \le m_{right} \le m_{x,max} \\
f_{top} = m_{top} f_{base} & m_{y,min} \le m_{top} \le m_{y,max} \\
f_{bottom} = m_{bottom} f_{base} & m_{y,min} \le m_{bottom} \le m_{y,max}
\end{array}$$

A particular tile's edge is indexed by the integer multiple used to compute its flux. The f_{base} terms cancel out of many operations related to fluxes, allowing us to work with the indices themselves. In particular, the divergence-free condition can be re-written as:

$$m_{left} + m_{bottom} = m_{right} + m_{top} \tag{2}$$

3.2. Counting Flow Tiles

The combinatorics of flow tiles are important for assessing the cost of storing a tile set. Roughly speaking, the number of tiles in a complete set grows with $c^4(M_x + M_y)^3$, where c is the number of possible corner velocities and $M_x = m_{x,max} - m_{x,min}$ and $M_y = m_{y,max} - m_{y,min}$ are the number of choices offered for fluxes across the vertical and horizontal edges (we omit the proof for space reasons). We will refer to a tile set using the triple (M_x, M_y, c). To give a flavor for the numbers, selecting one corner velocity and three possible fluxes across each edge, a (3,3,1) set, contains 19 tiles. A (5,5,1) set increases the number to 85. Providing 5 corner options and three flux options, a (3,3,5), increases the number to 11,875.

In practice we have found that tile sets created from a small number of velocity and flux options produce very complex flows when tiled. This is because the variety in appearance comes from the combination of tiles, not the complexity of the tiles themselves. The situation is analogous to building a complex mosaic out of plain colored tiles. We have also found that applications use only a small subset of the tile set. The potentially large set is required to provide designers options when laying out tiles interactively, but unused tiles need not be created or stored. Furthermore, boundary conditions and constraints on tiling tend to force the re-use of only a small subset of possible tiles, again freeing us from the need to create and store all the possibilities. As a specific example, the river example of Figure 1 uses only 21 distinct tiles, out of a potential set measured in the tens of thousands. If larger tile sets are required, we can exploit rotational symmetry to reduce the storage cost. With symmetry, and assuming the only corner velocity is 0, the (3,3,1) set is reduced to only 6 tiles, shown in Figure 4.

The flux is an integral quantity, and there are an infinite number of velocity fields along an edge that all have the same corner velocities and integrate to the same flux. We could use an additional index for each edge to specify a particular edge velocity field, but we use only one edge velocity field uniquely defined by the corner velocities and the flux (see Section 3.4). This reduces the number of tiles in a set at the expense of greater repetition of particular edges in the tiling.

Tilings are created by placing tiles without overlap, so a single velocity vector is enough to define each corner. Greater overlap, and hence greater continuity in the flow across tiling seams, could be incorporated by specifying a small piece of corner velocity field. For example, tiles with one unit of overlap would require a 2×2 velocity field for each corner. Our tile filling and tiling algorithms can be easily adapted to support this.

3.3. Representing Flows

Flow tiles must store a continuous velocity field. As with grid based fluid simulation algorithms we store the field using a discrete set of samples on an axis aligned grid. Most fluid solvers for graphics store face-centered velocities [FM97] because this arrangement makes it simpler to handle boundary conditions and projection onto a divergence-free field. Each edge of the 2D grid (face in 3D) stores the velocity component normal to the edge at its midpoint. Such a representation requires $2n(n+1)$ values for a

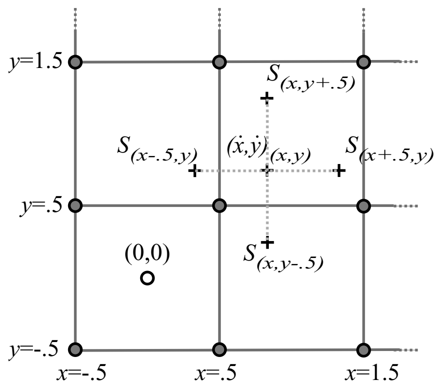


Figure 3: *The stream function method for representing divergence-free velocity fields. We show a small portion of the bottom left of a grid. Values of a scalar stream function, S, are stored at vertices of the grid. To evaluate a velocity, $(\dot{x}, \dot{y})$, we first find four surrounding stream function values by interpolating the vertex values. The velocity is then obtained by taking the curl of the stream function.*

2D grid with $n \times n$ cells. In a divergence-free field, however, there are constraints upon the velocity values. Exploiting this allows us to halve the storage cost of a field.

Every 2D divergence-free incompressible field can be represented as the curl of a scalar function, the *stream function* $S(x,y)$, multiplied by a vector in the third dimension, $\mathbf{z}$:

$$\mathbf{v}(x,y) = \nabla \times (S(x,y)\mathbf{z})$$

Discretizing the curl operator and storing the stream function, instead of the velocity field, gives us the arrangement shown in Figure 3, with stream function values stored at the vertices of the grid. The velocity, $(\dot{x}, \dot{y})$, at a point, (x,y), is evaluated by interpolating the stream function to find $S(x-.5,y)$, $S(x+0.5,y)$, $S(x,y-.5)$ and $S(x,y+.5)$. Then:

$$\begin{aligned} \dot{x} &= S_{x,y-.5} - S_{x,y+.5} \\ \dot{y} &= S_{x+.5,y} - S_{x-.5,y} \end{aligned} \tag{3}$$

Evaluating these expressions is no more expensive than interpolating the face-centered scheme, and the stream function representation has two advantages: the memory cost is lower, at $(n+1)^2$ for an $n \times n$ cell grid, and it is simpler to construct tiles with specific fluxes across the edge. The total flux across a tile edge is arrived at by adding (or subtracting) the stream function values at the *endpoints* of the edge. All the intermediate stream function values cancel out in a summation of the velocities across the edge. We can therefore ensure a particular flux across an edge by setting the corner stream function values appropriately, with no concern about the field at interior points along the edge. We exploit this in creating flow tiles.

The discussion above assumes square grid cells in the derivation of the velocity from the stream function. Scaling terms would be required for non-square cells. We discuss this further in Section 5 in the context of mapping the tiling onto an environment.

3.4. Filling Tiles

We fill flow tiles using a simple interpolation scheme based on bi-cubic patches. Other alternatives for tile design include specifying the field by hand, extracting tiles from another flow-generation method, such as the Kolmogorov spectrum, or taking a snapshot of a simulation on the tile.

The goal of our interpolation scheme is to take the corner velocities and flux chosen for a tile and produce a stream function on a grid of a given size (the tile dimensions). We work only with the stream function when filling tiles because it frees us from divergence-free considerations; any stream function generates a divergence-free velocity field. Our interpolation technique works for tile sizes greater than or equal to 3×3 (4×4 stream samples).

The corner velocities and edge fluxes completely determine the stream function values in the corners of the tile (up to an additive constant). Consider filling a tile of size $n_x \times n_y$, with indices into the stream function in the domain $(-.5,\ldots,n_x+.5) \times (-.5,\ldots,n_y+.5)$. Using the fluxes assigned to this tile, we set the corner values:

$$\begin{aligned} S_{-.5,-.5} &= 0 \\ S_{n_x+.5,-.5} &= S_{-.5,-.5} + f_{bottom} \\ S_{-.5,n_y+.5} &= S_{-.5,-.5} - f_{left} \\ S_{n_x+.5,n_y+.5} &= S_{-.5,n_y+.5} + f_{top} \end{aligned}$$

Using the velocities and Equation 3 we can set the three stream function values surrounding each corner. For example:

$$\begin{aligned} S_{.5,-.5} &= S_{-.5,-.5} + \dot{y}_{0,0} \\ S_{-.5,.5} &= S_{-.5,-.5} - \dot{x}_{0,0} \\ S_{.5,.5} &= S_{-.5,.5} + \dot{y}_{0,0} \end{aligned}$$

where $\dot{x}_{0,0}$ and $\dot{y}_{0,0}$ are the horizontal and vertical components of the velocity at the bottom left corner. Values around the other corners are set similarly.

The above construction around each corner results in 16 known stream values. We use a bi-cubic Bezier patch to fill the remainder of the grid. We solve a linear system to find the control points for the patch that interpolates the values around the corners, and then evaluate the patch at other grid locations to determine the remaining stream function values. The result is a smooth, continuous flow within the tile. The (3,3,1) tile set, suppressing rotationally symmetric tiles, is shown in Figure 4.

4. TILING

Tiling is the process of laying out tiles to meet continuity and boundary conditions. The tiling scheme meets two goals: to

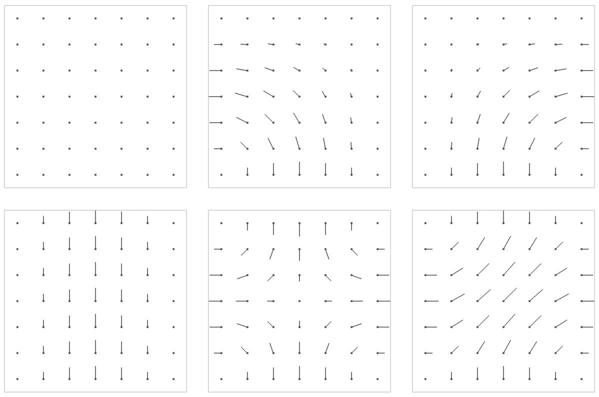


Figure 4: *The six tiles from a (3,3,1) set that remain when rotational symmetries are removed. There is only one corner option: zero velocity. Each edge has three flux options: -1, 0 or 1. Dots show the center of each grid cell, with a line drawn from the dot in the direction of the velocity, scaled according to the velocity's magnitude. The fluxes, (bottom,right,top,left), for each tile are, in clockwise order from the top left: (0,0,0,0), (1,0,0,-1), (1,1,0,0), (1,1,1,1), (1,1,-1,-1) and (1,0,1,0).*

support user interaction similar to that for terrain tiles or texture mapping, and to incorporate flows into larger environments. The latter necessitates support for exterior and interior boundary conditions. The challenges in achieving these goals arise from the divergence-free constraint, combined with the use of a finite tile set. For a sense of the problems, consider Figure 5, in which the aim was to fill a 3×3-tile region with zero boundary conditions. Even though the partial tiling shown is divergence-free, it cannot be completed because a tile is required with more flux across an edge than exists in the tile set. Hence, constraints imposed on tile placement by the edges of existing tiles are not sufficient to ensure valid complete tilings with a fixed tile set.

The tiling algorithm places tiles on a rectangular grid. Boundary conditions can be set on the flux across any edge in the tiling and the velocity at any corner. The specified flux or velocity must be one of those used to create the tile set. A user may also specify periodic external boundaries or completely free boundaries. If an application does not require all of the rectangular grid, the boundaries of the unwanted region can be set to have zero velocity and flux; the flow cannot cross such boundaries.

The basic operation for tiling is a selection function that returns a set of tiles given a partial tiling and an unfilled location. The selection operation guarantees that the tiling can be completed if one of the selected tiles is used in the given location. Tiling can thus proceed in any order, including those requested by a user in an interactive program.

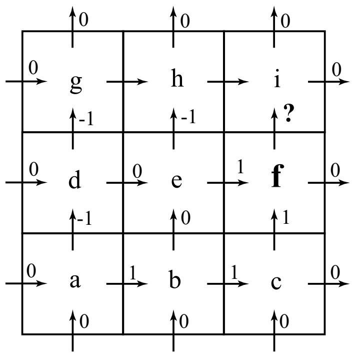
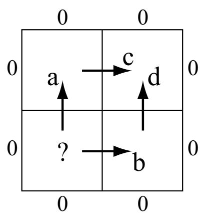


Figure 5: *A tiling that cannot be completed, despite the fact that all the tiles placed so far satisfy their neighbor conditions. In this example, tiles a-e from a (3,3,1) set have been placed. Next to be placed is f, but there exists no tile in the set with the required top edge flux of 2. Our tiling algorithm ensures that all partial tilings have a completion.*



$$
\begin{array}{rcccl}
m_{y,min} & \le & a & \le & m_{y,max} \\
m_{x,min} & \le & b & \le & m_{x,max} \\
m_{x,min} & \le & c & \le & m_{x,max} \\
m_{y,min} & \le & d & \le & m_{y,max} \\
& a+b & = & 0 & \\
& a-c & = & 0 & \\
& b-d & = & 0 & \\
& c+d & = & 0 &
\end{array}
$$

Figure 6: *An example tiling grid and the constraints in the integer program used to select tiles for the bottom left corner. The values a, b, c and d are the integral flux values across the edge. The inequalities enforce the limitations of the finite tile set, while the equalities enforce the divergence-free constraint for each tile.*

4.1. Tile Selection

The intuition behind tile selection is that every tile choice pushes advected mass into it neighbors, who push it on into other neighbors, and so on across the grid. At some point the tiling reaches a place where three edges are constrained, as in Figure 5, and we have no choice in the flux across an edge. We must be certain, at this point, that there is a tile with the required flux. Tile selection gives us this certainty, by ensuring every partial tiling can be completed.

Tile selection for a given position is based on identifying the range of possible values for the fluxes and corner velocities of candidate tiles. To identify valid corners, we first check if that corner is explicitly constrained, either by the user or and existing tile. If so, the velocity must match, otherwise any velocity is allowed. We identify valid edge fluxes one at a time, using two integer programs for each edge. Figure 6 is an example of the constraints for one such program: there is one constraint for every free edge in the tiling limiting its value to those available in the tile set, and one constraint per open tile slot to enforce the divergence-free constraint for that slot. The objective function for a given edge's first linear program maximizes its flux, while the second minimizes it. After solving the integer programs, we have bounds for the valid edge fluxes, and we search the tile set for tiles that meet the bounds.

Some tiles, however, are still not acceptable. For instance, minimizing and maximizing the right edge flux may give a range of $[0,2]$, and the top might also give $[0,2]$. But it may be the case that a right edge of 2 is only acceptable with a top edge of 0, and vice versa. We therefore filter the set of tiles by testing each one in turn for feasibility. In other words, we add additional constraints to the integer program, representing a candidate tile, and test if a solution exists that meets the constraints. If so, the tile is returned as one of the possibilities for the location.

Solving an integer program is a key step in our selection algorithm. Integer programming is appropriate because we have bounds on the flows across edges, determined by the possible tile edges, and constraints imposed by the divergence-free constraint and boundary conditions. The program in Figure 6 implicitly accounts for the zero boundary conditions on the tiling. In practice, we associate variables with each boundary edge and use constraints to enforce boundary conditions explicitly.

To solve the integer programs we use a standard simplex method *linear* program solver. The integer programs we are dealing with are *network flow* problems [Van01] and the Integrality Theorem for such problems ensures that a simplex solver returns integer results. This allows us to bypass expensive integer program solvers. While special purpose network flow solvers exist, we instead use the Osi/Clp linear program solver from the COIN package [LH03]. It supports hot-start solutions where an existing solution with modified bounds or a new objective is re-solved at significantly lower cost than if started from scratch. This is highly beneficial in our application, because when solving for a particular tile slot we change the objective function but not the constraints – they depend on the overall configuration of the tiling, not a particular edge.

It is difficult to produce tight bounds on the cost of tiling. While the number of variables for each program is known (an $n \times n$ tiling contains $2n(n+1)$ variables, some with fixed values), and the number of constraints depends on the free slots, the use of a simplex solver makes predicting performance a challenge. Furthermore, the number of programs solved depends on the set of potentially feasible tiles found in the first phase of the search. A scan-line order 10×10 random tiling algorithm requests the solutions to over 650 programs which are computed in a total under 1.5 seconds. A 4×4 random tiling takes 0.077 seconds on a 1.3GHz PC, while a 32×32 tiling takes 113 seconds.

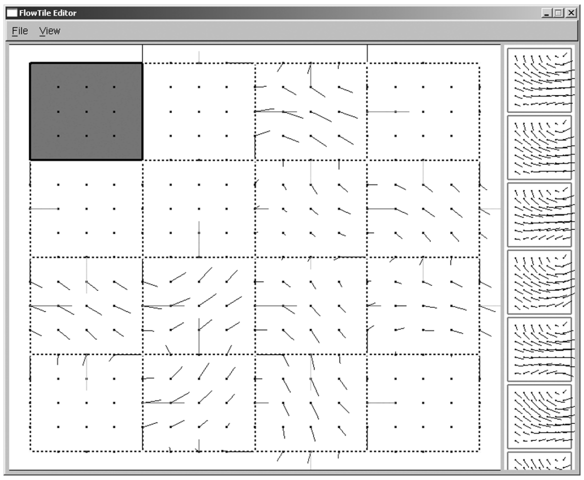


Figure 7: *A 4×4 tiling under construction in an editor. The user specifies constraints on the flux across edges and the velocities at the corners of tiles. When an empty location is selected, in this case the top left corner, the editor presents a set of tiles appropriate to that location, shown on the right. The user can select one of the options to place in the location. In the situation shown, the user is choosing the top-left tile, which has boundary conditions above and to the left, and velocity constraints on three of its corners. Hence all of the tiles offered have nearly identical left and top edges.*

5. APPLICATIONS

Flow tiles are suited to the generation of large, aperiodic velocity fields that meet specific boundary conditions. Tiling offers an WYSIWYG user interface for designing flows. We present three applications that can take advantage of flow tiles: the design of a river flowing between banks and around bridge pylons; the advection of a crowd of agents through city streets, and swirling fog over a pool. To create the first two environments, we incorporated flow tiles into the content creation pipeline for a student developed game engine.

Random tilings provide an automated method of producing velocity fields. To produce a random tiling, we proceed in scan-line order across the tile grid. For each position, we request the set of tiles that satisfy the constraints and choose one uniformly at random. Random tilings are interesting for some natural effects, such as turbulent wind fields, but frequently an animator has something else in mind.

We have built an interactive editor for tilings that is integrated with a tool set for game content creation. The editor, shown in Figure 7, allows a user to load a tile set, specify a region to be tiled and set boundary conditions. The user can then interactively create the tiling by clicking on a tile location and choosing one of the valid tiles provided by the system. Using the selection process of Section 4.1, the system can guarantee the user that their tiling can be completed, regardless of the order in which they place tiles. If desired,

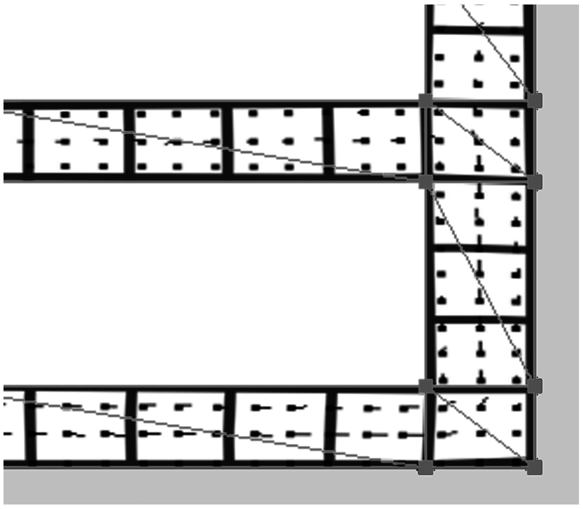


Figure 8: *A zoomed in view showing the mapping for the city example between a polygonal model (the gray triangles) and a tiling (the heavy black lines). Vertices in the model are associated with points in the tiling using flow coordinates. As with texture coordinates, these allow us to map points in the tiling, and hence velocities, back into the world.*

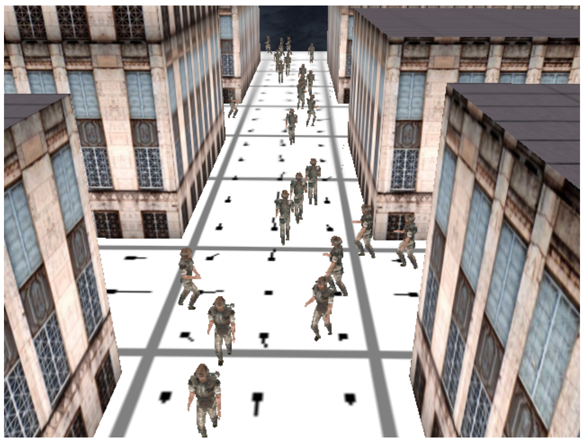


Figure 9: *A frame showing people moving through a city, driven by a flow tiling of the streets. Internal boundary conditions prevent people from walking through building walls.*

the system can automatically locate positions with only one valid option, and fill those for the user.

After a tiling is created, it is exported to another tool that binds the tiling into an environment (see Figure 8). This is done by identifying points in the environment with points in the tiling using *flow coordinates*, just as texture mapping uses texture coordinates to map points on surfaces to texture locations. Boundary conditions in the tiling are typically associated with obstacles in the environment, such as a river bank or the wall of a building. For good results, the flow coordinates must map the tiling grid into the environment without introducing flips, although tears in the grid are reasonable provided they are along tile boundaries. Figure 1 shows such a tear where the river forks. The tear prevents excess tiles in the region under the ground and allows sharp corners in the environment with minimal distortion of the tiling.

As with texture mapping, flow coordinates may be specified such that the resulting mapping deforms the grid. The physical interpretation of this is the that the advected material is compressible, which is reasonable for crowds – the agents move closer together or farther apart. For animations of incompressible fluids, such as water, a plausible visual interpretation is that mass is being pushed under or above the 2D surface. While our current system does not modify the third dimension, we could use the local deformation to add waves or other effects.

Given a set of flow coordinates, we triangulate the world space polygons. Each triangle in world space corresponds to a triangle in the flow tiling. To determine the velocity at a point in the world, (x,y), we first express the point in the barycentric coordinate system, (α,β,γ), defined by the world space triangulation. We transfer these barycentric coordinates to the tiling and use flow coordinates to define another coordinate system from which we can extract the point's (u,v) location in flow space. This is essentially equivalent to standard texture mapping, except instead of a color we have a velocity to transform back into world space.

The velocity in flow space, $(\dot{u},\dot{v})$ is transformed first into the barycentric coordinate system defined by the flow coordinates, $(\dot{\alpha},\dot{\beta},\dot{\gamma})$:

$$\dot{\alpha} = \frac{\partial\alpha}{\partial u}\dot{u} + \frac{\partial\alpha}{\partial v}\dot{v}$$

The equations for $\dot{\beta}$ and $\dot{\gamma}$ are similar. The terms $\partial\alpha/\partial u$ and so on are the partial derivatives of the barycentric coordinates with respect to the flow coordinate system. They are found by expressing α, β and γ as functions of u, v and the flow coordinates, and taking the partial derivatives. They can be stored with the triangle for fast look-up. The velocities in world space are computed from the barycentric velocities:

$$\dot{x} = \frac{\partial x}{\partial\alpha}\dot{\alpha} + \frac{\partial x}{\partial\beta}\dot{\beta} + \frac{\partial x}{\partial\gamma}\dot{\gamma}$$

with a similar equation for $\dot{y}$. The terms $\partial x/\partial\alpha$ are found by expressing the world coordinates in terms of α, β and γ and taking partial derivatives. They turn out to be components of the world coordinates of the triangle vertices.

The final part of our game development pipeline is a runtime engine that understands flow coordinates and the mapping into tilings. We have used it to create two demonstrations of flow tiles: a city model with tiles that define the flow of people through the city streets (Figure 9), and a river that flows within its banks, around a bridge pylon, and then forks

(Figure 1). The city example uses a tile set with five flux options for each edge and five corner velocities. The river uses five fluxes in the up/down stream direction and three options across the stream, along with five corner velocities.

Flow tiles drive the crowd using the velocity to define the direction of travel for each member. The use of divergence-free flows to define crowd motion ensures that, under reasonable conditions, the agents do not require any form of collision detection. Provided the agents start off non-intersecting and with some buffer space around them, and the field is close to incompressible, the flow will keep them apart from each other and the walls of building. Due to the fixed nature of the tiles, the crowd does not interact in any way, but for animations of an emergency evacuation or a panicking crowd this is not important. Interaction could be added by including local areas of simulation where agents ceased following the field and instead interacted using some other method. A limitation of flow tiles is that streamlines never cross, which means that characters can never, for instance, cross each other at the middle of an intersection. This could be addressed by overlaying multiple tiles in a single tiling space, with collision detection required in the overlay region.

The river example (Figure 1) advects leaves in the flow around a bridge pylon and a fork in the river. Flux and corner velocity constraints were used to add rotational motion to the flow in the wake of the bridge. The fork was achieved by cutting the tiling grid along an internal edge and pulling the two halves apart. In a relatively large flow application such as this, simulation would be too expensive for real-time performance, particularly given the other demands on processor time and memory in a computer game or interactive environment. Flow tiles are efficient to store (they use only 20% of the space required to store the field without repeating tiles) and fast to evaluate for a velocity.

Flow tiles can be extended to generate time-varying flows by interpolating between tilings. By the superposition principle, the linear combination of two divergence-free flows is also divergence free. A user could define a sequence of key-frame tilings, which would be looped to generate long running animations. An alternative, which we have implemented with random tilings, is to design a few key tilings and apply them in a random sequence generated an the fly. We have used such a field to drive a swirling fog animation, a frame of which is shown in Figure 10.

6. CONCLUSION

Flow tiles are particularly useful in applications where large, designed flows are required for use at interactive rates. However, there are several artifacts and limitations that arise from flow tiles, primarily due to the way that we specify and fill tiles, and the use of a regular grid.

Our linear interpolation technique for fill the interiors of tiles makes the generation of small scale turbulence difficult, and it is not physically motivated. Moreover, using interpolation based tiles is equivalent to simply specifying edge fluxes and corner velocities, and interpolating those directly, without going to the effort of designing tiles. We emphasize, however, that the techniques outlined for defining the constraints on tiles and tiling apply regardless of how tiles are filled. One superior approach would be to design more complex edge velocity profiles and use simulation or an adaption of a spectrum method to fill the tiles. The subdivision flow technique of Weimer and Warren [WW99] could also be extended to meet boundary constraints and hence fill tiles.

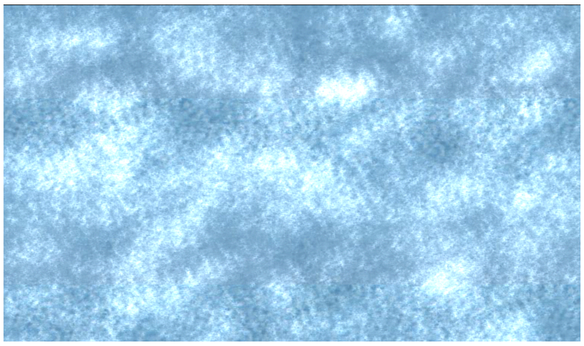


Figure 10: *A frame from a swirling fog animation generated using time-varying flow tilings.*

The number of tiles in a set grows rapidly as the number of corner options is increased. Tiles with only a couple of corners produce noticeable artifacts at the regularly spaced vertices of the tiling grid. The grid could be distorted to reduce the regularity of the artifacts, which are most obvious in randomly generated, incoherent fields. Symmetry could also be exploited to allow more corner options without larger sets. Large tile sets also pose a problem in interactive tiling because the number of potential tiles for a given position may be very large, although this can be alleviated by asking a user to specify more information about the desired tiles.

We use square tiles on a regular grid, which limits the topologies onto which tiles can be mapped with low distortion. Our tiling selection algorithm, however, works on any mesh of edges, thus allowing us to generate divergence-free flows for arbitrary polygonal tile shapes, including triangles. In particular, Neyret and Cani's [NC99] surface tiling approach could be adapted for triangular flow tiles. The greatest challenge is extending to other shapes the techniques for filling tiles and controlling divergence.

The primary advantage of flow tiles is the ability to meet boundary conditions – allowing our flows to be inserted into larger environments – and the provision of a WYSIWYG interface for creating fields. Our flows are divergence-free, making them suitable to drive a wide range of natural effects, such as fluid flow or crowd motion, and are efficient to store and access, allowing their use in interactive environments such as games.

ACKNOWLEDGMENTS

Thanks to Mark Zeller and Chris Olsen for integrating flow tiles into their game. Eric McDaniel and David Dynerman helped with video. Many thanks to James O'Brien and Jim Propp for providing insights at various stages of this work. Funded by NSF grant CCR-0204372.

References

[BPC99] BARERRO D., PAULIN M., CAUBET R.: A physics based multiresolution model for the simulation of turbulent gases and combustion. In *Eurographics Workshop on Animation and Simulation* (1999), pp. 177–188.

[CSHD03] COHEN M. F., SHADE J., HILLER S., DEUSSEN O.: Wang tiles for image and texture generation. *ACM Transaction on Graphics 22*, 3 (2003), 287–294.

[DKY*00] DOBASHI Y., KANEDA K., YAMASHITA H., OKITA T., NISHITA T.: A simple, efficient method for realistic animation of clouds. In *Proceedings of ACM SIGGRAPH 2000* (July 2000), pp. 19–28.

[FM97] FOSTER N., METAXAS D.: Modeling the motion of a hot, turbulent gas. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques* (1997), pp. 181–188.

[Gla98a] GLASSNER A.: Aperiodic tiling. *IEEE Computer Graphics and Applications 18*, 3 (May-June 1998), 83–90.

[Gla98b] GLASSNER A.: Penrose tiling. *IEEE Computer Graphics and Applications 18*, 4 (July-Aug 1998), 78–86.

[GS87] GRÜNBAUM, SHEPHARD: *Tiling and Patterns*. W.H. Freeman, 1987. ISBN 0716711931.

[LH03] LOUGEE-HEIMER R.: The Common Optimization INterface for operations research: Promoting open-source software in the operations research community. *IBM Journal of Research and Development 47*, 1 (2003), 57–66.

[NC99] NEYRET F., CANI M.-P.: Pattern-based texturing revisited. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques* (1999), pp. 235–242.

[NP01] NEYRET F., PRAIZELIN N.: Phenomenological simulation of brooks. In *Eurographics Workshop on Animation and Simulation* (2001), pp. 53–64.

[PC01] PERBET F., CANI M.-P.: Animating prairies in real-time. In *Proceedings of the 2001 symposium on Interactive 3D graphics* (2001), pp. 103–110.

[Pea01] PEASLEY M.: Tiled terrain. *Game Developer 8*, 8 (2001), 25–29.

[Rey99] REYNOLDS C. W.: Steering behaviors for autonomous characters. In *1999 Game Developers Conference* (1999), pp. 763–782.

[RNGF03] RASMUSSEN N., NGUYEN D. Q., GEIGER W., FEDKIW R.: Smoke simulation for large scale phenomena. *ACM Transactions on Graphics (TOG) 22*, 3 (2003), 703–707.

[RR00] RUDOLF M. J., RACZKOWSKI J.: Modeling the motion of dense smoke in the wind field. *Computer Graphics Forum (Proceedings EG 2000) 19*, 3 (2000), 21–29.

[SF92] SHINYA M., FOURNIER A.: Stochastic motion-motion under the influence of wind. In *Computer Graphics Forum: Proceedings of Eurographs'92* (1992), vol. 11(3), pp. 119–128.

[SF93] STAM J., FIUME E.: Turbulent wind fields for gaseous phenomena. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques* (1993), pp. 369–376.

[Sim90] SIMS K.: Particle animation and rendering using data parallel computation. In *Proceedings of the 17th annual conference on Computer graphics and interactive techniques* (1990), pp. 405–413.

[Sta97] STAM J.: *Aperiodic texture mapping*. Tech. Rep. R046, European Research Consortium for Informatics and Mathematics (ERCIM), 1997.

[Sta03] STAM J.: Flows on surfaces of arbitrary topology. *ACM Transactions on Graphics 22*, 3 (July 2003), 724–731.

[TMPS03] TREUILLE A., MCNAMARA A., POPOVIĆ Z., STAM J.: Keyframe control of smoke simulations. *ACM Transactions on Graphics (TOG) 22*, 3 (2003), 716–723.

[Van01] VANDERBEI R. J.: *Linear Programming: Foundations and Extensions*. Kluwer Academic Publishers, 2001. ISBN 0792373421.

[WH91] WEJCHERT J., HAUMANN D.: Animation aerodynamics. In *Proceedings of the 18th annual conference on Computer graphics and interactive techniques* (1991), pp. 19–22.

[WW99] WEIMER H., WARREN J.: Subdivision schemes for fluid flow. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques* (1999), pp. 111–120.