Scalable Real-Time Animation of Rivers

Qizhi Yu, Fabrice Neyret, Eric Bruneton, Nicolas Holzschuch

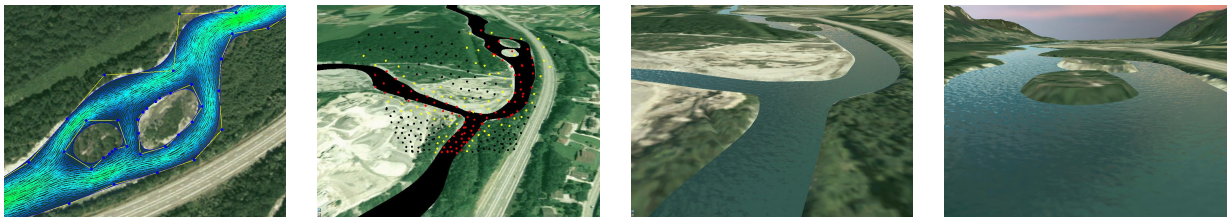
EUROGRAPHICS 2009 / P. Dutré and M. Stamminger
(Guest Editors)

Volume 28 (2009), Number 2

Scalable real-time animation of rivers

Qizhi Yu, Fabrice Neyret, Eric Bruneton and Nicolas Holzschuch

Laboratoire Jean Kuntzmann, INRIA Grenoble Rhône-Alpes, Université de Grenoble and CNRS



(a) (b) (c) (d)

Figure 1: *From left to right. Our procedural velocity model computes locally the velocity in a channel-confined flow with branches and obstacles (*a*). We use it to advect particles uniformly distributed in screen space, and hence limited to the view frustum (*b*). We reconstruct the fluid texture with wave texture sprites carried by the particles (*c,d*).*

Abstract

Many recent games and applications target the interactive exploration of realistic large scale worlds. These worlds consist mostly of static terrain models, as the simulation of animated fluids in these virtual worlds is computationally expensive. Adding flowing fluids, such as rivers, to these virtual worlds would greatly enhance their realism, but causes specific issues: as the user is usually observing the world at close range, small scale details such as waves and ripples are important. However, the large scale of the world makes classical methods impractical for simulating these effects. In this paper, we present an algorithm for the interactive simulation of realistic flowing fluids in large virtual worlds. Our method relies on two key contributions: the local computation of the velocity field of a steady flow given boundary conditions, and the advection of small scale details on a fluid, following the velocity field, and uniformly sampled in screen space.

1. Introduction

Many applications today are giving the user the ability to explore a virtual world of very large scale, possibly even unbounded. For practical reasons many of them (such as Google Earth) consist mostly of static terrain and geometry. The simulation of flowing fluids, such as rivers and lava flows would greatly improve the realism of these virtual worlds, but would also introduce scalability issues: in a typical situation, the observer is looking at the virtual world at close range, and thus paying attention to small scale details such as waves and ripples. Combined with the large scale of the world itself, this makes computational fluid dynamics solutions impractical, especially for interactive exploration. Most game engines, such as Crysis', use a constant flow, which has visible flaws, namely that the flow is going through obstacles.

In this paper, we present a method for the interactive simulation of running fluids in large virtual worlds. Our method creates the small scale details required for realism, such as waves, and makes them follow the velocity of the fluid. Our method is output-dependent: we perform our computations only in the visible portions of the flows at adapted resolution, making our algorithm well suited for large scale worlds.

Specifically, our contributions are twofold: first, a method for computing locally the velocity of a steady flow, given the boundary conditions, such as river banks and obstacles. Second, a method for advecting small scale details on a fluid, following the velocity field. Our small scale details are advected in world space, but we maintain uniform sampling in screen space, ensuring that our algorithm only performs computations on the visible parts of the flow.

Our paper is organized as follows: in the next section, we

review recent contributions on simulating fluids in virtual worlds. In Section 3, we present an overview of the overall algorithm. We then present the specific contributions: in Section 4, our method for local computation of the velocity of a steady flow, and in Section 5, our method for the advection of small details in world space, with constant sampling density in screen space. In Section 6, we present the results of our algorithm; we discuss these results and the limitations of our method in Section 7. Finally, we conclude in Section 8 and present directions for future work.

2. Previous work

2.1. Fluid velocity

Physically-based water simulation has been intensively researched. It is generally based on Eulerian grids in 2D [CdVL95], 2.5D [LvdP02] or 3D [Sta99,EMF02,LGF04], or on Lagrangian particles [MCG03, KW06, APKG07]. However these methods are reserved for off-line simulations or small domain real-time simulations.

Procedural methods can compute the velocity locally without doing a simulation in a whole domain. [PN01] extend Perlin noise for that purpose, but cannot handle boundaries nor global flowing. [Che04] propose a tiling of velocity tiles, but this is not adapted to complex boundaries and obstacles. [BHN07] propose a solution to impose boundary conditions to a velocity field based on procedural noise. But this solution does not work with complex channel confined flows with branching and obstacles. [Che04] and [BHN07] compute the velocity of the flow as the curl of a stream function, automatically ensuring that the divergence is null, a characteristic of incompressible flows. Our method is similar to theirs, with the main difference that we deal with a complex network of channels, including branching and obstacles.

2.2. Fluid surface

Wave models. Several models have been introduced for waves: analytical solutions for ocean waves [Pea86, HNC02], reproducing a wave spectrum by using a Fast Fourier Transform (FFT) [MWM87, Tes04]. The wave textures used by our texture advection method are generated by these methods. Compared to these papers, the main contribution of our algorithm is that we're dealing with flowing fluids, and the waves are advected by the flow.

Texture advection. Texturing is an effective way to add small scale details on a surface. For fluids we need a way to generate details that look like waves and ripples, and a method to advect them with the fluid. These requirements can be in conflict, unless the surface details are continuously regenerated. [MB95] advects texture coordinates and periodically reset them globally to avoid excessive stretching. [Ney03] improves this by using multiple layers of advected textures in order to adapt the reset period to the local fluid stretching. These methods are fast but are limited to non-structured textures. Recently, methods based on texture synthesis have been proposed for texturing dynamic flows [BSM*06,KAK*07]. These methods can handle structured textures but are computationally expensive and do not ensure that particles always follow the flow motion.

All these methods are Eulerian: they update a fixed texture grid at each frame. There are also Lagrangian methods that combine a particle system with sprites to reconstruct the fluid appearance. They are mainly used for splashing water [Won, IC00, BSHK04]. This sprite-based texturing has also been used to simulate running drops on a surface [NHS02], and GPU adaptive structures have been proposed to efficiently render such textures, even with overlapping sprites [LHN05]. Our second contribution extends the Lagrangian advection methods with a method to advect particles in world space, while maintaining a constant density in screen space.

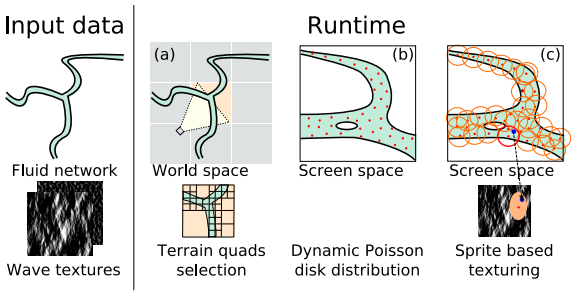


Figure 2: *Overview of our data structures and algorithms.*

3. Overview

Input data. The input data of our algorithm is made of raster data for the terrain elevation and texture, of vector data for the fluid boundaries, and of wave textures (see Figure 2, left). The vector data describes a network of *channels* connected by *junctions*, as well as *obstacles* such as islands. Each channel has two boundary curves, a flow rate and a flow direction. This data could come from a GIS database, be simulated from an erosion model, or be generated interactively or procedurally. The wave textures can come from any source (Perlin noise, FFT waves, etc.).

Runtime data. At run-time the terrain data is subdivided in a dynamic quadtree, based on the viewer position and distance. Each quad contains raster data and clipped vector data for its corresponding terrain part (see Figure 2a). This quadtree is computed as described in [BN08]. When a new quad becomes visible we compute on the fly the stream function values at channel boundaries (see Section 4.1). We also create an acceleration structure to quickly compute distances

to channel boundaries (see Section 4.4). This data remains in memory as long as the quad is visible.

Texture advection. In order to advect small scale details on the fluids we rely on particles that carry wave sprite textures. We distribute particles with a uniform and constant density in screen space, to simulate only the visible parts of moving fluids and to automatically adapt the sampling density in world space to the viewing distance (Figure 2b). However we advect the particles in world space, using our procedural velocity method to evaluate locally the velocity of each particle. This procedural velocity depends on the distances of the particle to the channel and obstacles boundaries (see Section 4.2). After advection we insert and delete particles in order to keep a uniform sampling (see Section 5.1).

Rendering. We render the fluids by rendering meshes on top of the terrain. The meshes are created from the channel boundary curves. They are rendered with a shader that recovers and blends together the wave sprites that overlap each pixel (see Section 5.2).

Algorithm 1 Scalable real-time animation of rivers

```
1:  loop
2:      for all new visible terrain quads do
3:          Compute the quad's channels network.
4:          Compute the stream function boundary values.
5:          Build a structure for fast distance evaluations.
6:      end for
7:      Advect particles with the flow in world space.
8:      Resample particles to keep uniform screen density.
9:      Render wave sprites associated with particles.
10: end loop
```

4. Computing flow velocities

Whenever a new terrain quad becomes visible, we take as input the constraints and boundary conditions on the flow for this quad (including a margin, as in [BN08]), such as river banks, and we produce as output a data structure that allows us to compute the velocity of the flow at any point, very quickly. Since the creation of this structure is very fast we can edit the constraints (*e.g.,* river shape) interactively.

Our input is the set of conditions imposed on the flow for the entire virtual terrain. It can be *e.g.,* a hydrographic network with the river banks, or ocean currents. Typically, a hydrographic network is stored as a directed graph, expressing the connections between rivers and channels. This graph is mapped on the terrain. Volumetric flow rates for each branch of the graph can be included as part of the original data set, or we can reconstruct an approximate version using the vertical cross-sections of each branch (see Section 4.1).

We assume that our running fluids are incompressible 2D flows. Hence $\nabla \cdot \boldsymbol{v} = 0$, and velocities can be expressed as the curl of a stream function, ψ [Whi01]:

$$\boldsymbol{v} = \nabla \times \psi$$

The stream function is related to the volumetric flow rate inside each branch of the hydrographic network: ψ must be constant along every connected boundary of the system, and the volumetric flow rate Q inside a given channel is equal to the difference between the values of ψ on each bank (see Figure 3):

$$Q = \psi_{left} - \psi_{right}$$

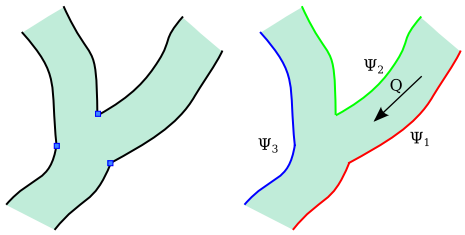


Figure 3: *The stream function, ψ is constant along every connected boundary (we merge connected boundaries). The volume flow rate of each channel is given by $Q = \psi_{left} - \psi_{right}$ ($Q = \psi_1 - \psi_2$ in the example).*

Given a connected graph for the hydrographic network, the values of the volumetric flow rate Q_i for each branch of the graph yields directly the values of the stream function ψ on each of the channel boundaries, up to a constant. The constant is meaningless since we are only interested in the curl of ψ.

Once we have ψ values for the boundaries of the hydrographic system, we interpolate them to get the stream function inside each channel. This interpolation will be discussed in Section 4.2. Finally, we deduce the velocity from the interpolated values of ψ through finite differentiation.

4.1. Finding the volumetric flow rate from the hydro geometry

If our dataset does not include the volumetric flow rate for each channel, we reconstruct plausible values based on the hydrographic network and its geometry.

The hydrographic network is a directed graph, expressing the connections between channels, and flowing from the source to the ocean. In the case of streams joining into rivers, themselves joining into larger rivers, the graph is a tree. At the delta of a river, the graph is an inverted tree. If there are no islands, we have an acyclic graph; each obstacle or island results in a cycle in the graph. Islands and obstacles can either be treated at this level, or be treated as a special case in the interpolation method.

Each edge of the graph corresponds to a channel. We

know its cross-section, w_i, and we want to compute its volumetric flow rate, Q_i. This can be solved through a graph traversal, using the following laws:

1. At each junction, the sum of the volumetric flow rate of the incoming channels is equal to the sum of the volumetric flow rate of the outgoing channels (conservation of mass).
2. At each junction, the repartition of the volumetric flow rate between the incoming or outgoing channels is proportional to their relative cross-sections.

For an acyclic graph, these two conditions are sufficient to get the volumetric flow rate values for each edge of the graph in a single traversal, up to a multiplicative constant: we select an edge of the graph, for example one where the volumetric flow rate, Q_0 is known. All flow rate values for the edges of the graph will be proportional to Q_0. If we don't know Q for any edge, we simply pick an edge that is convenient for the user interface in controlling the flow rate, for example at the mouth of the river.

For each edge i connected to the first edge, the second rule gives us the value for its flow rate:

$$Q_i = \frac{w_i}{\sum_j w_j} Q_0$$

where w_i is the cross section of the channel i. Because we're using the relative cross-sections, the conservation of mass is a direct consequence:

$$\sum_i Q_i = \frac{\sum_i w_i}{\sum_j w_j} Q_0 = Q_0$$

Through a graph traversal, we get Q values for each connected edge of the graph.

Cycles in an hydrographic network graph come, mostly, from islands and obstacles in the flow (see Figure 7). For the computation of volumetric flow rate Q, they cause the problem to become over-constrained, and it is impossible to compute Q for the network. The solution is to move the problem to the stream function, ψ: as ψ must remain constant on the boundaries of the channel, the only free parameter is the value of ψ on the contour of the island. We use the interpolated value of the stream function at the center of the island.

4.2. Interpolation of stream function

Once we know the values of ψ at the boundaries of the channels, we interpolate between these values to get a continuous representation of ψ. Let's assume that we want to evaluate the stream function at a point P. We choose the following method: we fix a search radius s, and find all the boundaries that are inside this search radius, B_i. Let d_i be the distance from point P to B_i, and let ψ_i be the value of ψ on B_i (see Figure 4). $\psi(P)$ is a linear interpolation of the ψ_i:

$$\psi(P) = \frac{\sum_i w(d_i)\psi_i}{\sum_i w(d_i)} \quad (1)$$

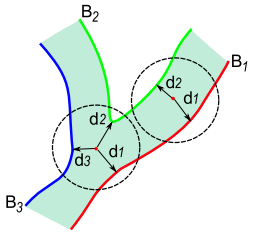


Figure 4: *To interpolate the stream function, we calculate the distances to the boundaries intersecting with the circular search region around the point of interest.*

where the weighing function w is defined as:

$$w(d) = \begin{cases} d^{-p} \cdot f(1 - d/s), & \text{if } 0 < d \leq s \\ 0, & \text{if } s < d \end{cases} \quad (2)$$

where p is a positive real number and f is a function that ensures C^2 continuity: $f(0) = 0$, $f(1) = 1$, with null first and second derivatives at 0 and 1. We used:

$$f(t) = 6t^5 - 15t^4 + 10t^3 \quad (3)$$

Our interpolation scheme has the desired properties: the function $\psi(P)$ is continuous (and Lipschitz continuous), the value of $\psi(P)$ depends on the boundaries that are closer to the sampling point. As point P gets closer to the boundary B_i, the value of $\psi(P)$ moves towards ψ_i:

$$\lim_{P \to B_i} \psi(P) = \psi_i$$

The influence of boundaries decreases with their distance to the sampling point. Boundaries beyond our search radius s have no influence on $\psi(P)$, allowing for local computations.

The function f was chosen to ensure that ψ is continuous at the boundary. The parameter p gives us control over the differentiability of ψ at the boundary. For $p > 1$, the tangential velocity at the boundary is null (no-slip boundary), and for $0 < p \leq 1$, ψ is not differentiable at the boundaries (slip boundary — see Figure 8). We adapt the value of p depending on the kind of river we want to model.

s is another important parameter. Computation time is related to s: the smaller the value of s, the faster the computations. As s gets larger, we have to compute the distance from P to a larger number of boundaries, increasing the computation time. For meaningful results, the behavior of ψ on a neighborhood of P must at least be influenced by all the relevant boundaries: *i.e.,* for a point inside a channel, $\psi(P)$ should depend on the values on both boundaries of the channel. Similarly, for a point inside a branching region, $\psi(P)$ should be influenced by the three boundaries (see Figure 4). Thus s must be larger than the largest branching region. For efficiency, s can be adapted to the current terrain quad, *e.g.,* twice the maximum channel width in the current terrain quad.

4.3. Handling obstacles

Obstacles in the flow, such as islands, are treated in the same way as the other boundaries: they are included in the search for boundaries, and we compute the distance. The only *caveat* is that we must first compute ψ_i for the obstacle: let C_i be the center of obstacle O_i. We first compute $\psi(C_i)$ using Equation 1, using only the boundaries of the channel. We then use $\psi(C_i)$ as the value of ψ on the boundary of the obstacle.

In the weighting function (Equation 2), we can assign a different search radius s for each obstacle, thus making the velocity pattern around an obstacle adapt to its size.

Moving obstacles In theory moving obstacles are out of the scope of our model as they result in a non-stationary velocity-field in world space. We can deal with them in various ways:

- *Slowly moving obstacles* are treated as if they were islands, with the boundary of the island being edited at each frame. This does not include the wake of the obstacle. For more accuracy, one could add a boat-frame local additive field, as in [WH91].
- *Transient obstacles*, such as objects falling in the river or an animal stepping in the river, are dealt with automatically by our model, using interactive edition. Transitional effects such as splashes can be added by a separate algorithm.
- *Advected objects*, such as objects floating on the river, are naturally dealt with by our algorithm, simply by adding sprites or geometry passively advected according to the velocity field.

4.4. Fast distance calculation

Our interpolation method for ψ makes intensive use of the distances to the boundaries; we need an accelerating data structure for efficient answer to those queries.

Whenever a terrain quad is created, we create a quadtree to store the segments of the river boundaries that fall within this quad. The segment quadtree allows us to quickly exclude boundaries that do not intersect with the search region of a given point. We then use the remaining boundaries for our computations.

For moving obstacles, such as boats, we use a separate data structure for each of them, to avoid frequent updates of the segment quadtree. If the obstacle can be approximated by a simple shape, we use analytical methods to compute the distance. Otherwise, we build a local adaptive distance field for each obstacle.

5. Adaptive texture advection

Now that we can compute the velocity at each point, we want to add small scale details to the flow, advected with the velocity field. For scalability reasons we want to advect these details only in the visible parts of the fluid, while ensuring a continuous flow when the camera moves. This is easier to do with methods based on particles and texture sprites, rather than with Eulerian texture advection methods. However, in order to correctly reconstruct the final fluid texture, the sprites must cover all the visible parts of the fluid, and must not overlap too much (to preserve the texture properties despite blending). It is also necessary to use more particles near the camera to get a constant apparent level of details on screen. Our solution to solve these two problems is to use a uniform sampling of disk particles in screen space: it ensures the absence of holes, a good overlapping, and a world space density inversely proportional to the viewer distance.

We use a Poisson-disk distribution with a minimal distance d between particles carrying sprites of radius r (in screen space). Then $r \geq d$ is sufficient to ensure the absence of holes between sprites. We advect the particles with the flow in order to convey the fluid motion, and we insert and delete particles when needed so as to maintain a Poisson-disk distribution.

5.1. Dynamic particle distribution

Particle advection. We advect each particle in two steps. We first update its world position $\mathbf{p}$ with $\mathbf{p} \leftarrow \mathbf{p} + \mathbf{v}(\mathbf{p})\mathrm{dt}$, where $\mathbf{v}$ is our procedural velocity function. We then compute its screen position by projecting it to screen.

Particle deletion. After advection some particles can get outside the view frustum, or be too close to other particles (in screen space). The first ones can be deleted because they are invisible. The second ones *must* be deleted to maintain a Poisson distribution. In order to avoid popping we fade out their sprites progressively (see Equation 7). We separate between *active* and *inactive* particles. Active particles maintain a Poisson distribution and become inactive when they get too close to other active particles. Inactive particles are deleted after the fading.

Particle insertion. After the previous steps some holes can appear between the remaining active particles. In order to fill them with new active particles we need a method for generating a Poisson-disk distribution from the existing particles which already respect the minimum distance criterion d. We adopt the boundary sampling algorithm presented in [DH06] because it is an incremental method and runs in $O(N)$ time.

New particles are generated in screen space, but we need their world position to advect them in the next frames. In order to get them we render the fluid surfaces to a buffer, using the vertices world positions as vertex color. We then read back on CPU the pixels of this buffer that correspond to the new particles. Note that we also fade in the sprites of new particles in order to avoid popping (see Equation 6).

Particle sampling domain. We can either use particles to sample the whole screen, or only the projected area of moving fluids. The latter makes insertion of new particles more complicated, especially if the apparent width of the channel is very small; the former is simple and more robust. It generates more particles, but particles outside the fluid are not advected nor rendered, and thus their cost is very small. We have elected to generate particles over the whole screen area.

Thus we have *inside* particles, whose center is inside the flow, and *outside* particles, whose center is outside. Inside particles have an associated texture sprite and are advected with the flow. Outside particles are not advected and do not have an associated sprite. This works well except for outside particles crossing boundaries. Indeed they prevent the presence of inside particles at the flow side of the boundary, and since they do not have an associated sprite, we may get holes in the reconstructed texture (see Figure 5). In order to solve this problem we inactivate these particles as soon as we detect them, in order to have more chances to create inside particles in the following frames (we detect them as the outside particles with at least one inside particle at distance $2d$ or less). As a byproduct, we get a correct sampling even for very narrow channels (see Figure 11).

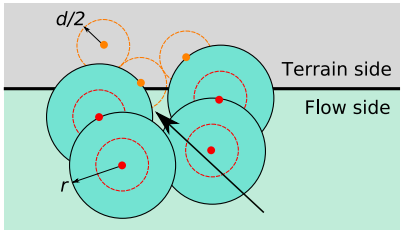


Figure 5: *An uncovered region appears near the boundary in the flow side due to the presences of near boundary particles in the terrain side. Here, the small disks are Poisson-disks of radius $d/2$, and the large disks are the circular footprint of sprites of radius $r = d$.*

Algorithm 2 Dynamic particle distribution

1: **loop**
2: Advect active and inactive inside particles.
3: Delete outside frustum or faded out particles.
4: Inactivate particles with active neighbors closer than d.
5: Inactivate outside particles with active inside neighbors closer than $2d$.
6: Insert new active particles to remove holes.
7: Compute world coordinates of new particles.
8: **end loop**

5.2. Sprite-based texturing

Wave patterns. In this work we focus on statistical wave patterns populating the surface and advected with the flow, such as simple flow fluctuations or wind ripples. We ignore individual waves such as stationary shockwaves or hydraulic jumps, which are not advected. In general, waves are not the same everywhere in the fluid. For instance wind ripples do not appear in wind shadowed regions. In order to reproduce this, our particles carry several kinds of waves simultaneously. During rendering we mix these waves with shaping and masking rules analog to the ones used in [BHN07]. We either use a user-defined map or a procedural rule (based on position, slope, etc) to locally modulate the amplitude of each kind of wave, or we use global parameters to control the wave appearance at the scene level (wavelength, wind direction, etc.).

Wave sprites. In order to reconstruct the fluid texture we need a texture sprite for each particle. The size of sprites in world space is not constant, since it is proportional to the viewer distance to the particle. Hence it would not be easy, and inefficient, to use one texture per sprite. Instead we use a tileable *reference* texture (one per kind of wave), and associate with each sprite a *part* of this texture. This part is centered around a point selected at random when the particle is created, and its size is computed at each frame, depending on the viewer distance.

Texture reconstruction. At this point each projected fluid surface pixel on screen is covered by at least one sprite. In order to reconstruct a spatially and temporally continuous texture we blend these sprites together using blending coefficients $w(\mathbf{x},t) = w_x(\mathbf{x})w_t(t)$ defined as follows:

$$w_x(\mathbf{x}) = \|\mathbf{x}-\mathbf{x}_0\|/r \quad (4)$$

$$w_t(t) = \begin{cases} w_{in}(t) & \text{if } t < t_1 \\ w_{out}(t) & \text{otherwise} \end{cases} \quad (5)$$

$$w_{in}(t) = \min(t-t_0, T)/T \quad (6)$$

$$w_{out}(t) = w_{in}(t_1)\max(t_1 - t + T, 0)/T \quad (7)$$

where $\mathbf{x}_0$ is the sprite center in screen space, t_0 and t_1 are the creation and inactivation times, and T is a user defined fading period. The blending is applied to physical parameters such as vertical displacements, obtained from the reference textures. These blended parameters are then used to compute derived quantities such as normals or reflection and refraction terms, in order to compute a realistic fluid appearance (see Algorithm 3).

In order to find the sprites that overlap a given pixel we use an indirection grid (as in [LHN05], but in screen space). Each cell of the grid stores the precise location and parameters of the sprites that cover it. The grid is encoded into a texture, called *indirection texture*. This scheme allows us to treat our system of dynamic sprites as an ordinary material described by a fragment shader and applied to a mesh rendered as a simple geometry.

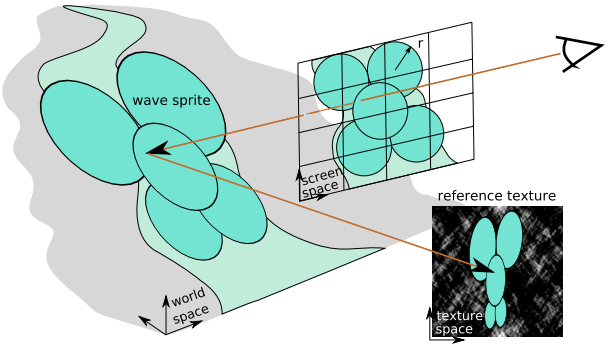


Figure 6: *The different frames considered for the sampling and rendering of sprites.*

Algorithm 3 Fragment shader for fluid meshes

1: $z_{sum} \leftarrow 0$ // *sum of surface heights*
2: $w_{sum} \leftarrow 0$ // *sum of blending coefficients*
3: Use indirection texture to find covering sprites.
4: **for all** sprites **do**
5: Compute location p of pixel in reference texture.
6: Get height z from reference texture at p.
7: Compute blending $w = w_x w_t$ for this sprite's pixel.
8: $z_{sum} \leftarrow z_{sum} + w \cdot z$
9: $w_{sum} \leftarrow w_{sum} + w$
10: **end for**
11: $z_{avg} \leftarrow z_{sum}/w_{sum}$ // *blended surface height*
12: Compute normal, Fresnel reflection and refraction, etc.

6. Implementation and results

In order to demonstrate the benefits of our method in real applications we tested it in a 25×25 km^2 scene with a river network, branchings and obstacles. We used a 800×600 window with $r = d = 20$ pixels. We used two kinds of waves: noise perturbations and wind ripples. For the former, we used a precomputed Perlin noise reference texture. For the latter, we used Fourier generation using analytical time evolution [Tes04] for wind waves. Both reference textures contain height fields, that are used by the water shader for bump mapping and environment mapping. The test was done on an AMD Athlon 3200 processor at 1.8 GHz with a GeForce 8800 GTS graphics board. The particles and the final rendering results are shown in Figure 11.

We first validated the procedural velocity generation by comparing the streamlines generated by our method against those generated by a potential flow solver (see Figure 7). The similarity between the two results shows that our method is a good approximation.

We then measured the performance of our method. Figure 9 shows that given a Poisson-disk radius, the running time of our method depends linearly on the projected area of river surfaces in the window. Thus our method does not depend on the complexity of the scene. In the test, we achieved real-time performance even in the worst case where the projected surfaces occupy the whole window. Certainly, the performance will decrease if we decrease the Poisson-disk radius. However, a moderate value as we used in this test is sufficient due to the adaptivity of the particles and the sprite-based rendering scheme.

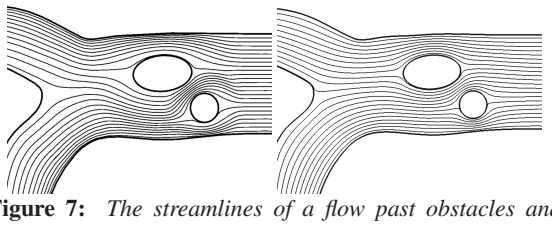


Figure 7: *The streamlines of a flow past obstacles and through a junction.* Left: *Generated by our procedural method with distance power* $p = 1.0$. Right: *Generated by the potential flow solver in* OpenFoam*, a CFD software.*

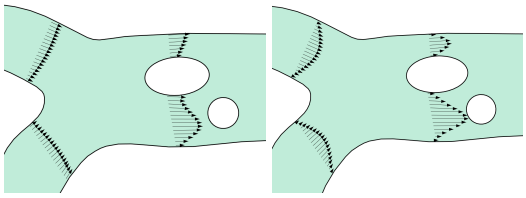


Figure 8: *Velocity profiles. Slip and no-slip boundary conditions can be simulated with* $p = 0.9$ (left) *or* $p = 1.2$ (right)*, respectively.*

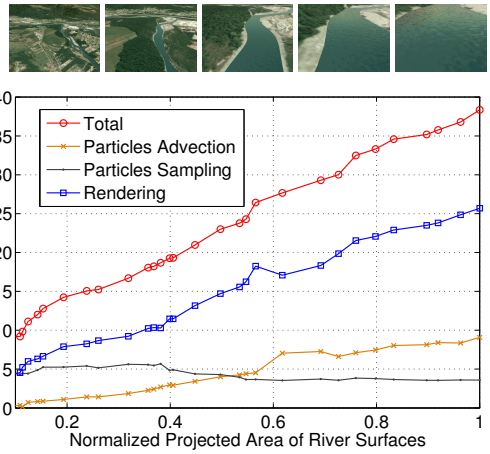


Figure 9: Top: *Typical views with increasing projected areas of river surfaces.* Bottom: *The running time of our method depending on the projected fluid surface area.*

We also demonstrate the controllability of our method in the accompanying video[†]. Our system allows users to edit

[†] http://www-evasion.imag.fr/Membres/Qizhi.Yu/

channels without interrupting the animation, which is due to our procedural velocity generation. In addition, the river appearance can be easily modified using the reference wave textures.

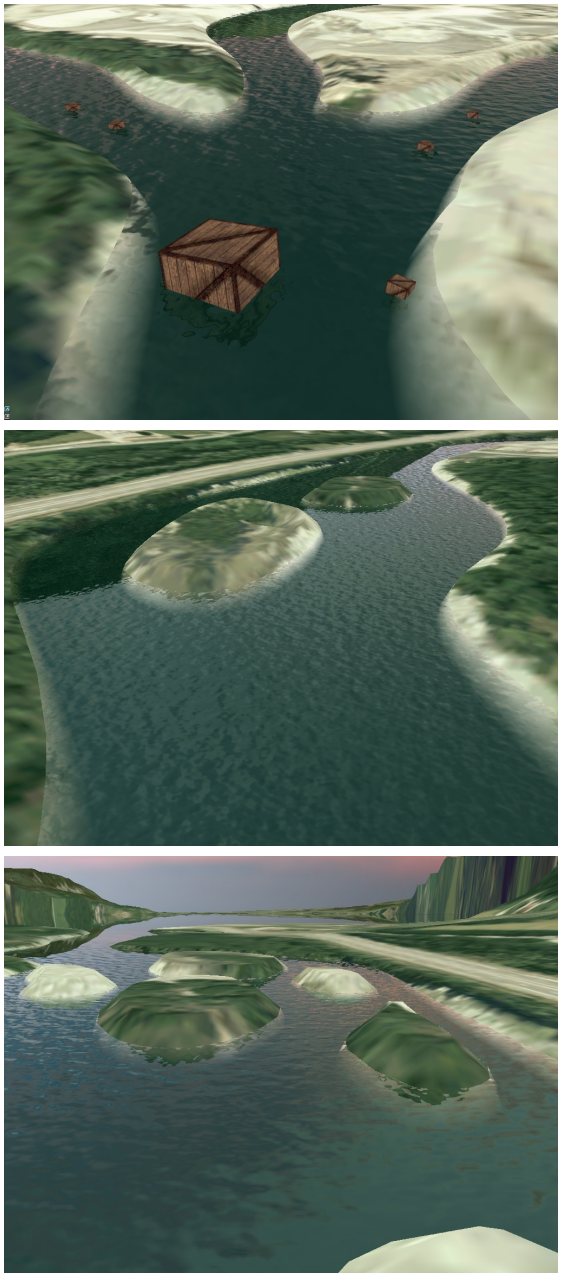


Figure 10: Top: *a trifurcating junction and moving obstacles advected with the flow, with realistic refractions and reflections.* Middle: *a close view showing islands.* Bottom: *a very close view with a grazing angle.*

Finally, to demonstrate that considering channel-confined flow is necessary for river animation, we compared our results against the animation of non-flowing water and uniform flow which can be handled by previous methods. The results demonstrate that our method brings considerable improvements (see the accompanying video): with non-flowing water, the waves stay in the same position, giving the impression of a static river; with uniform flow, the waves go through the obstacles and boundaries, breaking the assumptions of the model.

7. Discussion and limitations

2D flow hypothesis and terrain slope Our 2D flow hypothesis is valid for constant water depth (and homogeneous velocity profile along each water column). To account for depth $h(x,y)$ variations we should simply conserve $q(x,y) = v(x,y)h(x,y)$ instead of v: $\nabla \cdot q = 0$, $q = \nabla \times \psi$, $v = q/h$. This supposes either to know h or to deduce it from the terrain elevation $z(x,y)$. In our static case, the Chézy law provides a convenient approximation: $v = C\sqrt{Rs}$ with C the Chézy constant, s the slope, $v = Q/S$ the average velocity in a vertical section of surface S, perimeter P, and hydraulic radius $R = S/P$. Assuming the section has a known shape, e.g., a rectangle of known length l and height h, this yields h as a function of s, l, Q.

Moving boundaries The fact that we can efficiently recompute the acceleration structure for distance computations, and that we can combine it with individual distance fields for moving objects yields various advantages. First, it allows our model to fit in a precomputation-free environment where visible tiles are generated on demand. This is a key condition to make our model amenable to very large terrains. Second, we can deal with interactive changes, including falling objects, or moving obstacles. Third, this property could make it possible to deal with flows with moving boundaries, such as flooding rivers, mud or lava flows...

Limitations of texture frequency range For very close views the sprite diameter in world space is smaller than the maximum wavelength of the reference wave texture. Hence our sprites cannot reproduce the low frequencies of this texture in this case. The solution is to represent these low frequencies with a scalar value attached to each sprite, sampled from a low pass filtered version of the wave texture.

Extension to 3D river surfaces In our implementation we render rivers as flat surfaces with bump mapping using fake reflection and refraction, like in most game engines. This very common technique has known limitations, especially at grazing view angles (wave elevation is not visible, especially along banks and obstacles, the back side of waves is not masked, etc.). Still, our model can be used with rendering methods taking parallax into account. For instance we could render the water surface with a coarse 3D mesh as in [HNC02], the height of the vertices being generated using

our 2D texture. We could also use recent works such as parallax map, displacement map, inverse displacement map and relief textures.

8. Conclusion and future work

We have presented a high performance framework to render animated rivers on very large terrains, allowing close views as well as large views. Our method fits well with real-time navigation of a large-scale virtual environment (Google Earth, simulators, games — although there is only limited interaction with the water), and is also controllable by designers.

For this, we proposed a stream-function based procedural velocity scheme conforming efficiently to complex rivers, and an efficient dynamic particle sampling scheme ensuring at the same time the adaptation to the viewing condition, the respect of the simulated flow, and an homogeneous distribution in screen space.

In future work, we want to link together the different parameters, depending on the required balance between accuracy and performance. We have only adapted the particles sampling to the viewing distance. We could also adapt them to the stretching of the flow to better represent regions of high variation.

Acknowledgments Qizhi Yu was supported by a Marie Curie PhD grant through the VISITOR project. This work was also supported in part by the French National Research Agency, reference ANR-05-MDMSA-004 “Natsim” and by the GRAVIT consortium, reference “GVTR”.

References

[APKG07] ADAMS B., PAULY M., KEISER R., GUIBAS L. J.: Adaptively sampled particle fluids. *ACM Transactions on Graphics 26*, 3 (2007).

[BHN07] BRIDSON R., HOURIHAM J., NORDENSTAM M.: Curl-noise for procedural fluid flow. *ACM Transactions on Graphics 26*, 3 (2007).

[BN08] BRUNETON E., NEYRET F.: Real-time rendering and editing of vector-based terrains. *Computer Graphics Forum 27*, 2 (2008).

[BSHK04] BHAT K. S., SEITZ S. M., HODGINS J. K., KHOSLA P. K.: Flow-based video synthesis and editing. *ACM Transactions on Graphics 23*, 3 (2004), 360–363.

[BSM*06] BARGTEIL A. W., SIN F., MICHAELS J. E., GOKTEKIN T. G., O’BRIEN J. F.: A texture synthesis method for liquid animations. In *Symposium on Computer animation* (2006), pp. 345–351.

[CdVL95] CHEN J. X., DA VITORIA LOBO N.: Toward interactive-rate simulation of fluids with moving obstacles using navier-stokes equations. *Graphics Models Image Processing 57*, 2 (1995), 107–116.

[Che04] CHENNEY S.: Flow tiles. In *Symposium on Computer Animation* (2004), pp. 233–242.

[DH06] DUNBAR D., HUMPHREYS G.: A spatial data structure for fast poisson-disk sample generation. *ACM Transactions on Graphics 25*, 3 (2006), 503–508.

[EMF02] ENRIGHT D., MARSCHNER S., FEDKIW R.: Animation and rendering of complex water surfaces. *ACM Transactions on Graphics 21*, 3 (2002), 736–744.

[HNC02] HINSINGER D., NEYRET F., CANI M.-P.: Interactive animation of ocean waves. In *Symposium on Computer Animation* (2002).

[IC00] IKELER D., COHEN J.: The use of Spryticle in the visual FX for "The Road to El Dorado". In *ACM SIGGRAPH 2000 sketches* (2000).

[KAK*07] KWATRA V., ADALSTEINSSON D., KIM T., KWATRA N., CARLSON M., LIN M.: Texturing fluids. *IEEE Transactions on Visualization and Computer Graphics 13*, 5 (2007), 939–952.

[KW06] KIPFER P., WESTERMANN R.: Realistic and interactive simulation of rivers. In *GI ’06: Proceedings of Graphics Interface 2006* (2006), pp. 41–48.

[LGF04] LOSASSO F., GIBOU F., FEDKIW R.: Simulating water and smoke with an octree data structure. *ACM Transactions on Graphics 23*, 3 (2004), 457–462.

[LHN05] LEFEBVRE S., HORNUS S., NEYRET F.: Texture sprites: Texture elements splatted on surfaces. In *Symposium on Interactive 3D Graphics (I3D)* (April 2005).

[LvdP02] LAYTON A. T., VAN DE PANNE M.: A numerically efficient and stable algorithm for animating water waves. *The Visual Computer 18*, 1 (2002), 41–53.

[MB95] MAX N., BECKER B.: Flow visualization using moving textures. In *Proceedings of the ICASW/LaRC Symposium on Visualizing Time-Varying Data* (1995), pp. 77–87.

[MCG03] MÜLLER M., CHARYPAR D., GROSS M.: Particle-based fluid simulation for interactive applications. In *Symposium on Computer Animation* (2003), pp. 154–159.

[MWM87] MASTIN G. A., WATTERBERG P. A., MAREDA J. F.: Fourier synthesis of ocean scenes. *IEEE Computer Graphics and Applications 7*, 3 (Mar. 1987), 16–23.

[Ney03] NEYRET F.: Advected textures. In *Symposium on Computer Animation* (2003), pp. 147–153.

[NHS02] NEYRET F., HEISS R., SENEGAS F.: Realistic rendering of an organ surface in real-time for laparoscopic surgery simulation. *the Visual Computer 18*, 3 (May 2002), 135–149.

[Pea86] PEACHEY D. R.: Modeling waves and surf. *Computer Graphics (SIGGRAPH ’86 Proceedings) 20*, 3 (Aug. 1986), 65–74.

[PN01] PERLIN K., NEYRET F.: Flow noise: textural synthesis of animated flow using enhanced Perlin noise. In *SIGGRAPH 2001 Technical Sketches and Applications* (Aug. 2001).

[Sta99] STAM J.: Stable fluids. In *SIGGRAPH ’99* (1999), pp. 121–128.

[Tes04] TESSENDORF J.: Simulating ocean water. In *The elements of nature: interactive and realistic techniques*. 2004. SIGGRAPH 2004 Course Notes 31.

[WH91] WEJCHERT J., HAUMANN D.: Animation aerodynamics. In *Computer Graphics (SIGGRAPH ’91 Proceedings)* (July 1991), Sederberg T. W., (Ed.), vol. 25, pp. 19–22.

[Whi01] WHITE F. M.: *Fluid Mechanics*, fourth ed. McGraw-Hill, Columbus, OH, 2001, section 4.7, pp. 238–245.

[Won] Wonder touch. http://www.wondertouch.com/.

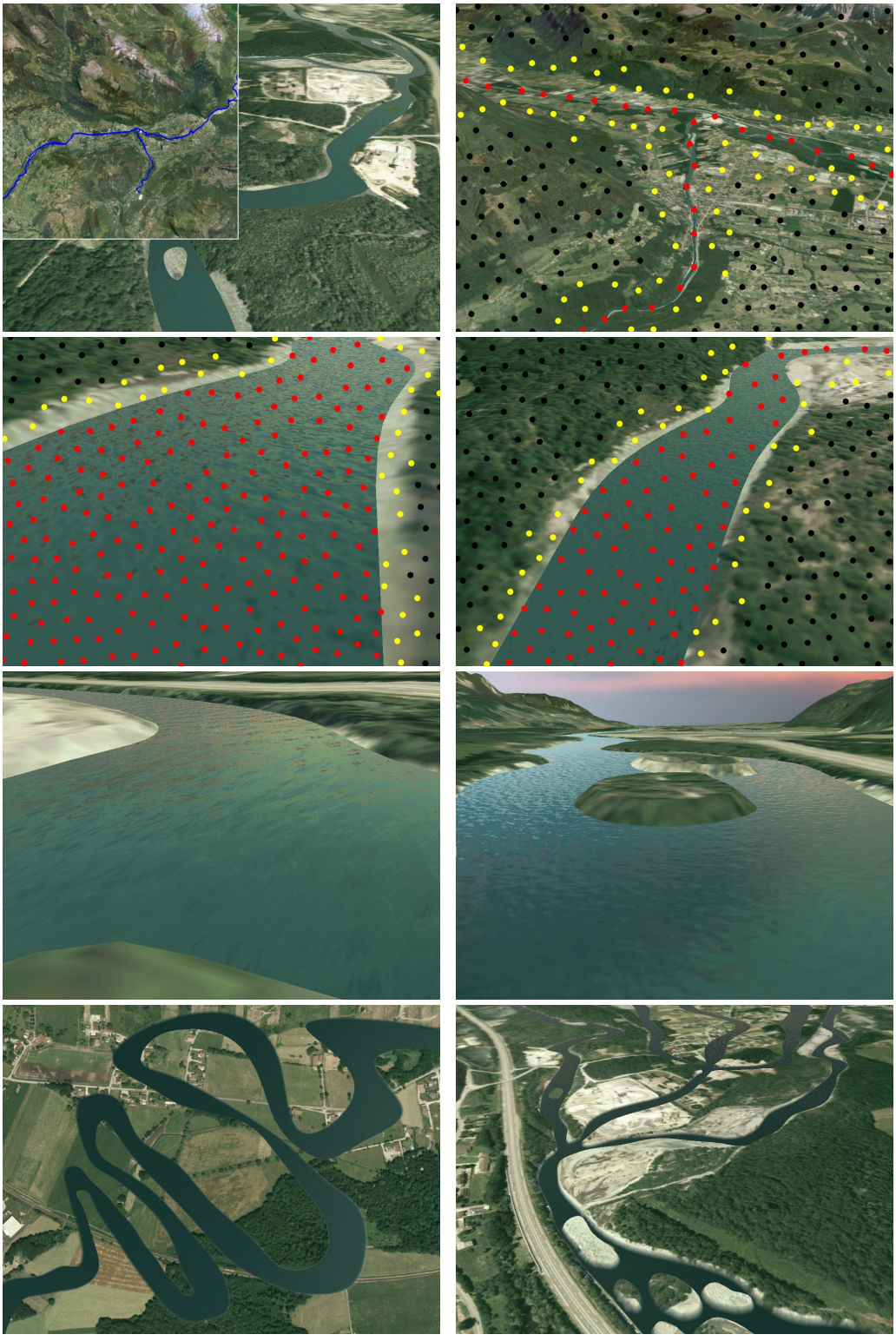


Figure 11: Top-left corner: *The river network (Source:* Régie de Gestion des Données des Pays de Savoie*) in a 25×25 km^2 terrain* . Top: *Particles maintain the Poisson-disk pattern in screen space. Here, the red are particles inside rivers, the black are particles outside rivers, and the yellow are particles outside rivers but near river boundaries.* Bottom: *Animation results (see the accompanied video for better demonstration).*