Physics-based Water Interaction and Shading: The SiViFlow Algorithm

David Sena
INESC-ID
Rua Alves Redol 9,
1000-029 Lisboa
davidsena@ist.utl.pt

Joao Pereira
INESC-ID
Rua Alves Redol 9,
1000-029 Lisboa
jap@inesc-id.pt

Vasco Costa
INESC-ID
Rua Alves Redol 9,
1000-029 Lisboa
vasco.costa@ist.utl.pt

ABSTRACT

Current real-time applications feature rivers that are pre-calculated off-line and present static animations and behaviours. These pre-calculated approaches have several limitations when used in real-time applications such as video games as they usually do not react to changes performed by the user. Due to the continuous pursue for better realism, the techniques used to simulate rivers have not only to improve the appearance of rivers but also allow them to adapt to dynamic changes performed in real-time. The approach presented in this work allows the dynamic generation of the river given any riverbed. The algorithm is also flexible enough to adapt the river flow in real-time. This approach not only accelerates the creation of realistic rivers but also increases the realism as the river is able to react to dynamic objects that come in contact with the flow, by properly adjusting its course.

Keywords

Water, Real-Time, River Animation, Flow Simulation.

1 INTRODUCTION

With the introduction of faster hardware and increasing demand for more realistic nature effects, researchers have been trying to create feasible nature models that are computationally viable and meet the constraints imposed by real-time applications. Nowadays applications such as video games try to simulate fully featured worlds with weather effects, large rivers and oceans, realistic animation systems among many other traits common in the real world. Due to the tight restrictions of real-time applications, an approach to simulate this type of phenomena would have to contain only the minimum amount of physical features necessary to make a river behave correctly and still leave enough computational resources available to draw a convincing visual representation of the fluid being simulated. The objective of the presented work is to create a new approach that simulates watercourses with any width, that flow correctly and are dynamic enough to be able to adapt to the features of their surroundings. A visually appealing representation of the flow being simulated is also included in order to be able to recreate with fidelity the watercourses from a visual standpoint. Our focus will reside mainly on the architecture description of the algorithm and less on implementation details or specific optimization issues. In order to focus the objectives of our work inside a broad subject such as fluid dynamics and as this work will be used in the context of video games, we decided to use real-time rendering techniques that allow the use of this approach in highly complex scenes. The final result had to be easily configurable both in terms of visual appearance and physical parameters in order to allow this approach to be used in any setting. This would allow not only to change the visual features but also the behaviour of the river according to its surrounding, making it more flexible to adapt to different surroundings (e.g. it should be flexible enough to able portray both a tropical or a sea like environments). Regarding the dynamic flow simulation two main contributions were done in our work. First the automatic generation of a velocity vector field given an arbitrary river surface mesh. Given the mesh as input, the algorithm analyses and generates enough data to be able to create a vector field that describes not only the direction of the flow but also its velocity at any point. Second once we've calculated the vector field, we'll generate a realistic and adaptive flow behaviour which allows us to portray any amount of turns in a given river network and even take into account changes performed to the river channel such as dynamic objects altering the flow. This contribution takes into account the fact that the river surface mesh might have any width, have a complex river shape and that all the flow information drawn on screen is updated accordingly.

2 RELATED WORK

2.1 Navier-Stokes equations

The basis of most fluid simulation models both in Computational Fluid Dynamics and Computer Graphics are the Navier-Stokes equations. These equations allow us to represent a fluid by its velocity field and a pressure

 ISBN 978-80-86943-75-6

field, varying both in time. If both fields are known at the initial time then we can describe the state of the fluid over time using:

$$\frac{\partial u}{\partial t} = -(u \cdot \nabla)u - \frac{1}{\rho}\nabla p + v\nabla^2 u + f \quad (1)$$

$$\nabla \cdot u = 0 \quad (2)$$

where $\cdot$ denotes a dot product between vectors, ∇ is the vector of spatial derivatives, u and p are the velocity and pressure fields of the fluid, ρ is the density and v is the kinematic viscosity. f is a vector representing external forces. Equation 2 is called the continuity equation and means that fluids conserve mass[Sta99]. The right-hand side of the equation 1 consists of four parts:

- Advection : $-(u \cdot \nabla)u$ which represents the process by which a fluid's velocity transports itself and other quantities in the fluid. In most simulations this represents the force that the surrounding fluid particles exert on a particle and causes it to transport itself along the velocity field.
- Pressure : $-\frac{1}{\rho}\nabla p$ causes regions with a higher pressure to accelerate the molecules away from that area.
- Diffusion : $v\nabla^2 u$ represents the force caused by the viscosity of the fluid.
- External forces : f represents forces that act on the fluid like gravity.

2.2 Approaches to Fluids Simulation

Physically-based water simulation has been an active research field for the last 30 years. Several different approaches have been proposed but usually they can be grouped into smaller distinct categories. In Figure 1 a schematic[GH06] is shown where the main types of water simulation are depicted.

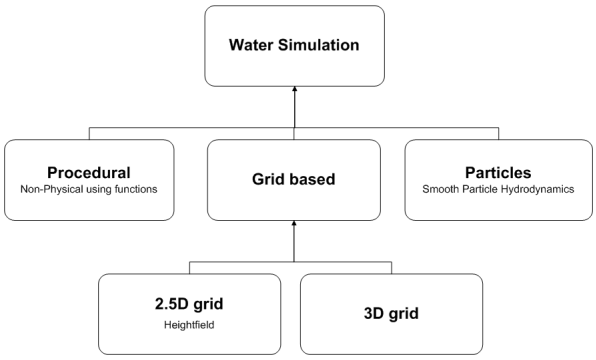

Figure 1: Water modeling techniques

The widest classification that can be made is a division between surface-based and volume-based techniques. The latter apply the Navier-Stokes equations to model the liquid's physical flow properties. Amongst the volume-based techniques, we can find many different approaches. One of those categories is the Eulerian approach. This approach looks at fixed points in space, discretizing the domain in regular grids, either in 2D [Sta99][Fos96][WLL04] or 3D [Ngu07][CTG10]. Each grid cell stores both scalar quantities (such as pressure and temperature) and vector quantities such as velocity. In this approach the computational elements are fixed in space throughout the simulation and a finite difference method is used to solve the equations numerically. The major advantage of this method is the possibility to allow adaptive time steps and the inherent smooth liquid surface that it allows. On the other hand, this method suffers from a lengthy computational time and grid resolution limitations allied with aliasing in the boundary discretization. It also suffers from poor scalability in terms of computational power and memory consumption. Another approach is the Lagrangian, where the fluid is approximated by several discrete particles and their respective properties. Each point in the fluid is considered as a single particle, with a position x and a velocity u. In order to solve several problems regarding the discretization of the continuum using the Navier-Stokes equations, the method most commonly used are Smoothed Particles Hydrodynamics (SPH)[CBL+09][HKK07][DG96]. The approach taken by SPH is to define a smoothing kernel to interpolate physical properties (velocities, densities, etc) at an arbitrary position from the neighbouring particles, instead of defining each particle and their physical properties individually. This approach has two major drawbacks. First the smoothing kernel should be designed carefully because the stability, accuracy and speed of the SPH method largely depends on the choice of those kernels. Second there is quite a complex step in the Lagrangian method that is constructing a smooth surface for rendering. Many research works have presented possible solutions [vdLGS09] but up till now, the quality of liquid surfaces constructed from the whole bunch of particles is not as compelling as its Eulerian counterpart.

Among surface-based techniques, there are procedural methods which despite the fact that they don't model the whole fluid domain or some fluid quantities (e.g. pressure), usually represent the fluid in terms of velocity fields. These approaches don't start from the equations but pick a way to describe the state of the system (usually through a velocity field of the fluid), evaluating and updating it anywhere in space and time. Even nowadays this kind of approach is preferable because it provides an extremely simple approach to efficiently generate a fluid-like behaviour in a body. It also allows to control the animation of a body of water, something that is not as easy to obtain when using volume-based methods as in those approaches we would have to deal

 ISBN 978-80-86943-75-6

with the discretization of partial differential equations, grids and solving systems of equations. Additionally most of previous methods rely on data that was computed with a fixed resolution, something that doesn't take into account a freedom of movement present inside most real-time applications and not present in movies or non-interactive demonstrations. One last advantage is the possibility to control several visual features of the fluid without having to recalculate the whole system, set the initial values and make sure that all the boundary conditions are well defined.

Method	Advantages	Disadvantages
Eulerian	Smooth Surface Adaptive time step	Memory usage Scalability Grid Resolution limitation
Lagrangian	More intuitive Irregular boundary	Smoothing Kernel Surface reconstruction
Procedural	Easy integration Extensible	Difficult to model some fluid values

Table 1: Water modelling techniques comparison

In Table 1 we show a summary of all the advantages and disadvantages of each technique.
For this work we chose the procedural approach because of the advantages described above and also due to the fact that it suits better the requirements of real-time applications.

2.3 Water Rendering

Fluids rendering is one of the most active fields inside Computer Graphics. As most of the physical behaviour of water couldn't be modelled at interactive frame rates inside real-time applications, developers and researchers focused most of their attention in getting as much visual fidelity as possible when rendering water. Reflection and refraction are elements that have been widely used in the simulation of water since the beginning of Computer Graphics [EMF02][GH06][PF05][Tes99]. Their use allows the user to see through the water and at the same time see the environment reflected on the water surface. This apparently trivial contribution fools the eye so much that most commercial products that include water algorithms sometimes only have these elements plus a wave generator. The most common way to describe reflection and refraction phenomena are the Fresnel equations[SJ09]. These equations allow us to describe the behaviour of light when moving between media with different refractive indices.

2.4 River Simulation and Rendering

A situation where fluid simulation is commonly applied to is when water flows between two or more boundaries, moving from a source into a sink. An example of that can be a river flowing where we have at least two river boundaries and the water flows to the river mouth or estuary. A river simulation can be decomposed in two main components: a simulation component where the physical behaviour is simulated and a visual component where the looks of the fluid are created. The work "Scalable Real-Time Animation of Rivers"[YN08][YNBH09] was able to simulate large scale rivers with realistic flow, yielding very appealing results. This work depicted a very realistic flow behaviour thanks to their new texture advection method, allowed real-time editing of the river channel with the respective flow adaptation to the new river boundaries and best of all it didn't depend on the scene complexity. Despite all these advantages there were still a couple of drawbacks. First the computational cost of the algorithm was linearly dependent with the projected river surface being rendered. Second the amount of data transferred between the Central Processing Unit (CPU) to the Graphics Processing Unit (GPU) is directly related with the Poisson-disk radius which increases linearly and quickly becomes prohibitive even with recent hardware. A final disadvantage was the need for the advection step to run on the CPU and the fact that this work assumed completely flat world profiles, excluding potential effects related with slopes of the terrain.
On the visual component there's a very visually appealing algorithm called Tiled Directional Flow[vH11]. This new algorithm offered several advantages over other flow simulation algorithms, was very cheap in terms of resources and yielded visually appealing results. They achieve a very realistic flow animation through the decomposition of the river surface in tiles, generating overlapping tiles all over the river channel (like a chess board on top of the river surface). Each tile has its own flow, local speed, direction and size of waves. By combining several normal maps together, the final result doesn't resemble sliding normal maps anymore and portrays a very pleasant appearance and animation. Even though the results of this algorithm were very satisfactory the fact that the authors have relied on the use of static flow maps limited the usage of this algorithm for big sized domains as it would require to either load a very large flow map or have some kind of spatial division algorithm to load the flow maps on the fly. Another disadvantage related with the use of static flow maps is that they can't take into account the influence of dynamic objects interacting with the river in real-time, which was something that had already been solved [YN08].

3 SIVIFLOW

SiViFlow is composed by two main elements: the Simulation Engine and the Visualization Engine. Figure 2

 ISBN 978-80-86943-75-6

illustrates the block architecture of the SiViFlow algorithm. The Simulation Engine is where all the calculations related to physics of the river take place. This engine is divided in three main modules: the River Surface Generator, the River Particle Generator and the Flow Texture Mapper. From the programming point of view, the River Particle Generator and the Flow Texture Mapper make up a larger block called the River Particle Processor which will be described later in detail. The Visualization Engine is responsible for receiving the simulation data from the Simulation Engine and to output a graphical representation. This engine is divided in two main modules: the Flow Renderer and the Reflection.

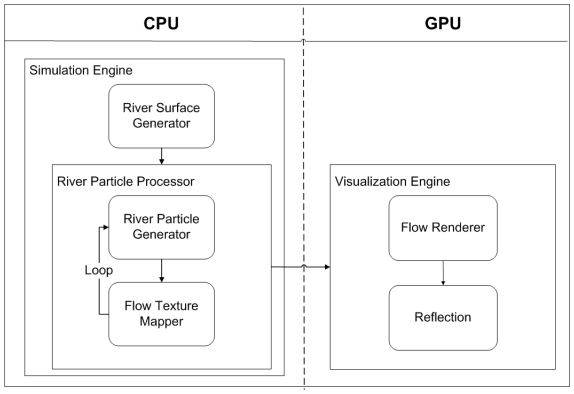

Figure 2: Block Architecture of SiViFlow

3.1 River Surface Generator

At this stage a river surface mesh needs to be created, which can either be done using an external modelling application or by generating it in real-time. At the beginning we don't know how many vertices go from one shore to the other in one single section of the river, so we start by calculating the river width and flag which vertices can be considered shore vertices. A river section is a set of vertices that are placed between two shore vertices and form a line that is perpendicular with both river shores as shown in Figure 3. In order to find out which vertices are shore vertices, we start by identifying the first vertex from the river mesh and calculate differences in distance between this vertex and all the other vertices that follow. When we reach the end of the river section we're processing, the difference stops increasing and it means we've reached the vertex which is on the same shore as our first vertex (the shore vertex right next to the one we're processing), thus the last vertex we processed belongs to the opposite shore. With that we calculate the river width (see Algorithm 1).

Algorithm 1 sums up all the steps taken during this preprocessing phase. The only input information required are the river mesh vertices. The algorithm starts looping from the first vertex which we know it's a shore vertex as it's located in a corner of the river mesh. We compare

for all vertices **do**
  if vertex is a shore vertex **then**
    Flag vertex
  $RiverWidth(vertices)$
  $DistanceToMargins(vertices)$
  $CalculateFlow(vertices)$

Algorithm 1: River surface generation algorithm

the width between this first vertex and the following vertices, making sure to always store a new width if the value is larger than what was previously stored. When the section of the river ends and we're processing the shore vertex which is on the same shore and right next to the first one, the distance between both vertices will be smaller than the full width of the river. We store the current width value and the amount of vertices that go from one shore to the other. At this stage we know the river width at each section as we have looped through all the river sections that compose the river surface. We also know the amount of vertices that go from one shore to another, allowing us to flag the vertices that belong to the river shore. These vertices need to be handled differently because they'll be used for calculating the flow. Now for each vertex in the river mesh, we store its distances to each of the river shore vertices at their river section. This information will later be used to calculate the flow velocity. Lastly we calculate the river flow at each river section, storing the information in every vertex. Both the flow velocity and flow generation will be described in more detail in the following sections.

3.1.1 Flow Generation

In order to calculate the flow we pick two shore vertices in the same river section, then we calculate their midpoint and translate in the positive up axis, as shown in Figure 3 where the up vector used is aligned with the y axis.

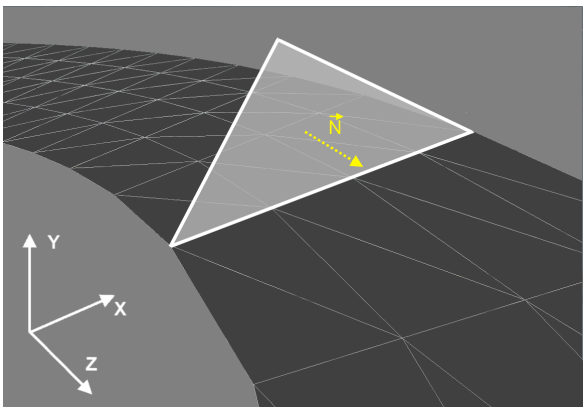

Figure 3: Flow vector created from a plane defined by two shore vertices and their midpoint translated in the +y axis

 ISBN 978-80-86943-75-6

With these three points we generate a plane that will allow us to create a vector that is perpendicular with the river section being processed. As the flow is constant for each river section and is parallel to the margins, the normal vector of the plane describes correctly the flow direction of that section as shown in Figure 3. Since the generated plane has two possible normal vectors, the normal generation procedure must take into account this direction and return the correct normal vector. In the end we have a flow field that is as detailed as the mesh of the river surface and where each vertex contains its own flow vector stored.

One advantage of generating the flow this way regards its flexibility to dynamically recalculate the flow when an object interacts with the river. In case a dynamic object alters the course of the flow, the boundaries of the object will be used to recalculate the new flow and will substitute the shore vertices that were previously used.

As the values are tied to the river mesh and the collision vertices are known, SiViFlow is able to recompute the flow of the river and immediately reflect the changes.

3.1.2 Flow Velocity

In order to obtain the flow velocity we calculate a stream function field(Ψ) for the river channel flow using an existent interpolation scheme [YN08][YNBH09]. At this stage we have all the information required to calculate the following equations. We run for each vertex all the Equations 3, 4 and 5 [YN08][YNBH09] and store their values.

$$\Psi(P) = \frac{\sum_i w(d_i)\Psi_i}{\sum_i w(d_i)} \tag{3}$$

with P being the position of each river surface vertex, d_i the distance from point P to the each of the boundaries, Ψ_i the stream function value of a margin and the weighting factor w is:

$$w(d) = \begin{cases} d^{-p} \cdot f(1-\frac{d}{s}), \; if \;\; 0 < d \leq s, \\ 0, \qquad\qquad\quad if \;\; s < d, \end{cases} \tag{4}$$

where s is the radius used to search for boundaries, p is a positive real number and f is defined as:

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

3.2 River Particle Processor

River particles are a concept we created in order to sample information from our domain and retrieve its values. As we want to be able to handle large watercourses, it's not feasible to rely on loading all the river surface information to Video RAM (VRAM) every frame. In our case we're interested in getting only the visible river mesh values so we can retrieve and send them to be rendered on the GPU. One of the main features of the river particles is that they're created in screen space in order to guarantee a uniform distribution of the particles over the visible domain at each frame. The reason for generating these points in screen space is that as each particle contains a defined radius to make sure no two particles are too close to each other, analysing this problem in screen space guarantees that these radius disks maintain a uniform radius. In world space these disks would be ellipses which would make the detection of overlapping particles harder. Another advantage of this scheme is that we only process visible information as we eliminate all non-visible particles which minimizes the waste of resources. There are some similar approaches to ours such as texture sprites [Ney03] and wave sprites [YN08][YNBH09].

3.2.1 River Particle Generator

We start by generating several randomly distributed points, generating a Poisson-disk pattern using a modified boundary sampling algorithm [Bri07][DH06]. We've adapted this algorithm to start from a fixed set of points instead of a random point. An advantage of this algorithm is that it guarantees that all points are equally distributed over the given domain, which in this case as we're aiming to generate particles in screen space, means they're all equally distributed over the screen.

In the end of running this algorithm, we end up with a set of points that we'll convert to river particles. In order to generate a 3D world position for each of these points (after being generated we only have their 2D coordinates) we proceed as Figure 4 shows. A ray is cast for each particle and we store the collision point between the ray and the 3D world. Using this method we can compute at each frame, for each point, its 3D world position. Besides calculating the world position we also calculate other features such as global identifiers to be able to identify each of the particles, velocity and flow. Unlike other algorithms [YN08][YNBH09], we don't advect our particles during our CPU update loop. The reason for this is due to the fact that our particles aren't concerned with the fluid's motion, they're simply a way to sample the necessary information in screen space and send it from the CPU to the GPU. An inherent advantage of not having to advect particles during the update loop is that it allows us to offload the work from the CPU to the GPU.

All of this information will allow us to find out in the next stage what's the nearest flow data to load into the flow texture. We just search inside a radius r for the closest vertex and assign that flow information to the

 ISBN 978-80-86943-75-6

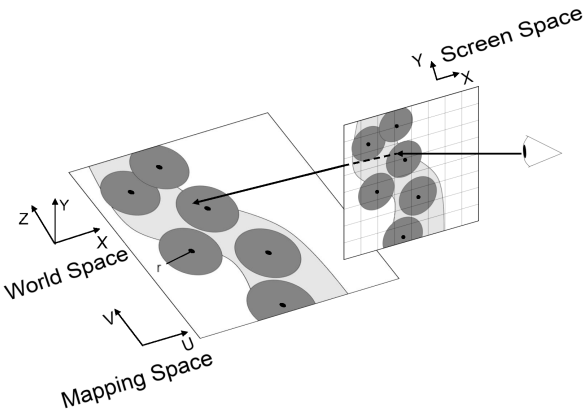

Figure 4: Ray cast performed from camera position and mapped into world space to obtain each particle's world position

river particle. The value of the radius r used for each situation was obtained through trial and error, although more sophisticated approaches can be used. This step differs from [YN08][YNBH09] as they first render the river surface to a buffer inside the GPU, find out which particles are inside the river surface and then query each individual pixel to find out which particle sits inside. Our approach despite being a bit more computationally intensive, doesn't have the inherent problems that might arise from relying in performing constant transfers between the CPU and GPU.

3.2.2 *Flow Texture Mapper*

In order to feed the GPU with the information required to render the flow, we used a flow texture and an auxiliary texture. Similar ideas have been explored by other authors [YN08][YNBH09][PF05] to achieve other objectives. In our approach we store all the information we need inside each color channel and read it back when it reaches the GPU. In Figure 5 we can see the distribution of each of the components in both the flow texture and auxiliary texture.

R8	G8	B8	A8	
Particle Index Number	Flow Vector x	Flow Vector y	Flow Vector z	Flow Texture
Velocity	Depth	Slope	-----	Auxiliary Texture

Figure 5: How each component is stored inside each of the 8 bit size texture channels

These textures will store the river particles previously generated using each of the color channels of the texture.

In the flow texture we will store for every entry data such as the global identifier of the river particle and its respective flow. The identifier in this texture will be used as a way to look-up the remaining data from the auxiliary texture. For each entry of the flow texture, we

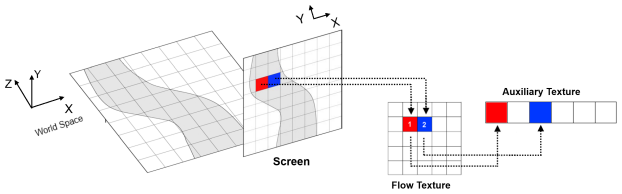

Figure 6: Storage scheme used in the flow and auxiliary textures

```
while true do
    for all particles do
        if Particle is outside of frustum then
            Delete Particle
        if Particle violates the minimum distance criterion in Screen Space then
            Delete Particle
        Insert new particles to keep the Poisson-disk
        for all new particles do
            Convert to river particles
        Write new data to the flow texture
        Write new data to the auxiliary texture
    Render
```

Algorithm 2: Application loop

store the flow information that covers that pixel. For performance reasons we used a flow texture that had a lower resolution than the screen resolution being used. The auxiliary texture will have other parameters such as velocity, river bed slope and river depth. In Figure 6 we can see how each river particle is stored in a smaller sized version of the flow texture and how the global identifier for each particle will be used to address the auxiliary texture.

In Algorithm 2 we can see that the whole update process is performed at every frame update. First we start by having to delete the particles that are not visible as they are wasting resources and won't affect the final result. Then we need to delete the particles that are too close to one another violating the initial Poisson-disk requirement that all particles must be no closer to each other more than a specified radius distance. In order to keep a reasonable number of particles in screen, after deleting all the unnecessary particles we generate new ones using the previously mentioned algorithm. After this, for all new particles, we have to convert them to river particles by calculating all their features. To end the algorithm we fill the flow and auxiliary textures with the current data from that frame and get them ready to be sent to the GPU.

3.3 Visualization Engine

The Visualization Engine is the last stage of SiViFlow and consists of mapping a material to the river surface

 ISBN 978-80-86943-75-6

Access flow texture to find covering sprite index and flow information
Access auxiliary texture to find velocity and depth
Use flow information for Tiled Directional Flow algorithm
Use new normal vector for reflection
Blend all the elements

Algorithm 3: Fragment Shader of the Visualization Engine

mesh. This stage is divided in two main elements: the Flow Renderer and the Reflection algorithm which are implemented in a fragment shader. We start by accessing the flow texture and consult the river particle identifier of this pixel. In order to optimize the texture look-up, the flow information is also saved during this operation. Now we can use the river particle identifier to look-up the rest of the parameters contained inside the auxiliary texture.

We also use the flow information to generate the normal which will be used to compute the scene's reflection. All the steps of the algorithm are summed up in Algorithm 3.

3.3.1 Flow Renderer

Our flow algorithm is based in the "Tiled Directional Flow" described in[vH11]. In our approach one of the main differences is that all the flow information being fed to the algorithm is not based on a fixed flow map but comes from our flow and auxiliary textures. This allows us to work with a much smaller amount of information at each render cycle because our flow texture only contains information that's visible during that frame. The fact that our flow texture is updated every frame, means that we can change the flow if any dynamic object changes river flow.

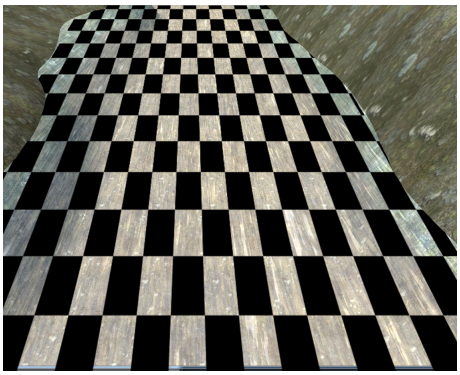

Figure 7: Example of the tiling division performed on top of the river surface for the flow algorithm

The way this approach works is by dividing a river channel in tiles, similar to a chess board. We show this division in Figure 7. Each tile is independent from its peers and its composed by several normal maps. In order to get a more convincing look, we used for each tile four normal maps that are combined and blended together. First the regular normal map is loaded for the tile being processed. Then we sample a normal map with half a tile shift in the x direction and we rotate it in order to have independent features from the previous normal map. These two tiles are blended together using a blending factor. The next two normal maps follow the same idea, the first one is sampled with a shift in the y direction and the second is shifted in the x and y direction. Both normal maps are rotated and combined together using the same blending factor. To get the final normal value, both normal maps are blended once more by using the same blending factor. To conclude this final blending step of normal maps a scaling operation has to be performed. This scaling operation avoids the problem of having a resulting normal closer to the actual average normal, which is common when several normal vectors are added together.

3.3.2 Reflection

In order to simulate dynamic reflections of objects on our river surface we used the well-known planar reflections algorithm [AMHH08][Eng03][PF05].

This approach has been widely used since the introduction of the programmable pipelines because of its ease of use and how inexpensive it is in terms of resources. An example of this technique can be seen in Figure 8 where it is visible the reflection of the house near the shore. This technique is based on the use of a texture called a reflection map, which is an inverted version of what it is visible above the water level and that we want to reflect. To obtain a reflection map, we start by defining a clipping plane, which has to be about the same height as the river surface.

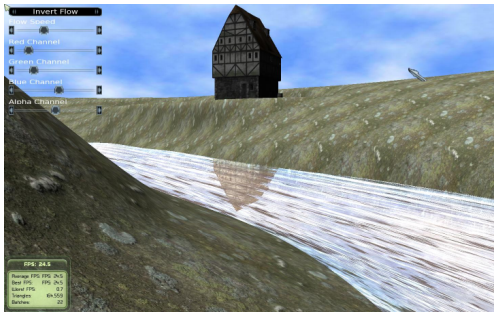

Figure 8: Example of the final scene appearance using planar reflections

This clipping plane will be useful to cut all the geometry below the river surface that we're not interested in rendering. If we didn't clip the contents below the river surface, we would reflect also the contents of the river which would break all illusion of reflection. After that we save an inverted copy of this clipped scene

 ISBN 978-80-86943-75-6

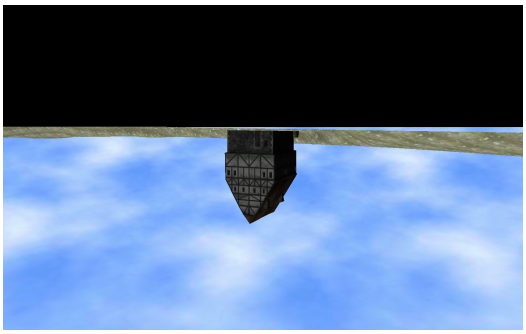

Figure 9: Example of a reflection map created clipping all the geometry below the river surface and reflecting the remaining contents

to a texture as in Figure 9 where we can see the contents of Figure 8 inverted and the whole river surface clipped. As the inverted copy is saved into a texture, we can send it to the GPU in order to be read inside our material. When we render our river material, we sample the correspondent pixel and blend the reflected information with the color we'll be outputting from the fragment shader.

4 RESULTS

This section provides the results and corresponding analysis for both the Simulation Engine and Visualization Engine. For the Simulation Engine we consider all the stages that deal with the creation, update and destruction of river particles and have to pack the required information in order to make it readable by the GPU. For the Visualization Engine we consider the Flow Renderer and Reflection stages which are comprised within the river material. We implemented our approach on top of the open-source game engine Ogre[1] version 1.7.3.(Cthuga). The algorithm was coded in C++ using the DirectX 9 API renderer provided by Ogre and the shaders were coded in HLSL. The platform used for testing is a computer with an Intel Core i7 running at 3 GHz with 8GB of RAM, a Nvidia GeForce GTX 480 with 1536 MB of VRAM and Microsoft Windows 7 x64 as the operating system. In order to measure the timings that each stage of our algorithm takes, we used Intel's VTune Amplifier[2] for the code that runs in the CPU and Intel's Graphics Performance Analyzer[3] to profile the timings in the GPU.

4.1 Simulation Engine

The Simulation Engine is composed of the River Surface Generator, the River Particle Generator and the Flow Texture Mapper. As the River Surface Generator only runs once to create the river surface mesh and it is not part of the application loop, all the measurements performed focused on the remaining components. This means that the application update loop can be divided in two main phases: the River Particle Generator and the Flow Texture Mapper. In Table 2 we can see how many particles were used in average to sample the whole screen.

Screen Resolution	Average Number of River Particles	Frames per second
800x600	336	32
1280x800	369	30
1440x900	407	29
1680x1050	384	28

Table 2: Average amount of river particles existent for different screen resolutions and average frames per second obtained throughout the tests.

We didn't use a fixed number of particles across all tests due to the nature of the sampling method we used. As the Poisson disk method randomly samples points across the domain, in order to minimize possible holes, some distributions might require more points than others. As shown when the screen resolution increases, the average frames per second decreases. This is due to the fact that as screen resolution increases, more particles are used and more pixels need to be processed in the CPU in order to map the best particle into the flow texture.

4.1.1 River Particle Generator

As mentioned in Section 3.2, the River Particle Generator is responsible for deleting river particles that are not visible, delete river particles that are too close to one another and generate new particles making sure they're converted to river particles.

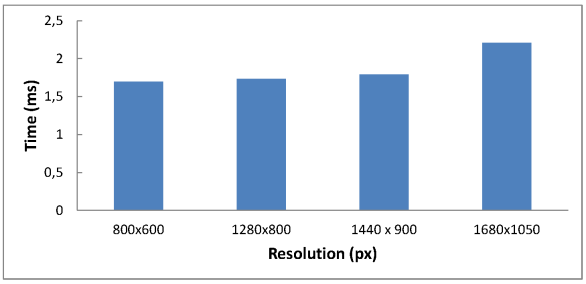

Figure 10: Time taken in milliseconds to update the river particles

In Figure 10 we can see that the time taken to update the river particles varies slightly across different resolutions. It's possible to see a slight increase in time taken to update the particles as the resolutions increase but the difference is less than 0.4 millisecond from the smaller resolution to the largest one.

[1] http://www.ogre3d.org/
[2] http://software.intel.com/en-us/intel-vtune-amplifier-xe
[3] http://software.intel.com/en-us/vcsource/tools/intel-gpa

 ISBN 978-80-86943-75-6

4.1.2 *Flow Texture Mapper*

The loading of new data into the flow and auxiliary textures is a step that must run at every frame and is performed in the Flow Texture Mapper. We'll start by analysing the time taken by the flow texture and after we'll analyse the auxiliary texture. As soon as we started profiling the application, we saw that the loading of data into the flow texture was the step in the whole algorithm that consumed more time. We used for all tests a flow texture with 64 by 64 pixels, meaning we had to map the screen resolution being used to the size of the flow texture and find the best particle that cover that section of the screen.

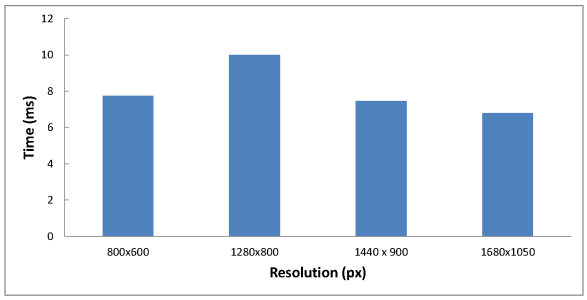

Figure 11: Time taken in milliseconds to load all the data into the Flow texture

We can see in Figure 11 that all the values tend to stay relatively close to one another. This is due to the fact that this step is not only our application's bottleneck but it's not directly influenced by the screen resolution as we always load a flow texture with the same dimensions. Upon closer look we noticed that the operations that were taking most of the time were finding the particle that better covers the largest amount of the pixels that are being processed and making sure that there were no sections of the texture without river particles. As the flow texture has a smaller size than our screen resolution, we map an amount of screen pixels that correspond to a single entry in the flow texture and process it. We retrieve all the river particles that cover this section and choose the one that covers the largest amount of the area being processed. The second costly operation is the second pass that we must perform in the flow texture to make sure that when one section without river particles is found, a suitable value is retrieved.
On the other hand, we have the auxiliary texture that contrary to the flow texture, is only affected by the amount of particles used as we load all the particles data into it.

As the number of particles doesn't change abruptly across screen resolutions, we can see in Figure 12 that the difference in values is no bigger than 0.05 milliseconds. As the auxiliary texture only needs to go over all river particles and load their respective values in the texture, this operation can be seen as a linear copy of data

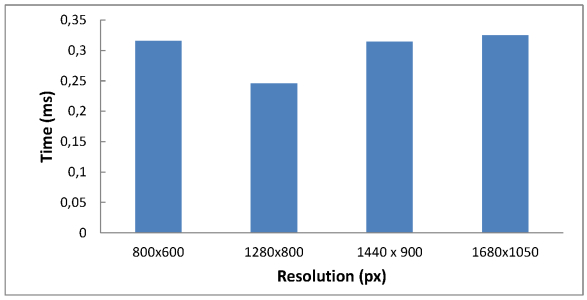

Figure 12: Time taken in milliseconds to load all the data into the auxiliary texture

from the river particles array into the texture, which can be performed quite fast.

4.2 Visualization Engine

As we've previously mentioned the components that make up the Visualization Engine are implemented as two distinct elements: the vertex shader and the fragment shader. Both the Flow Renderer and the Reflection make use of information existent in both of these elements.

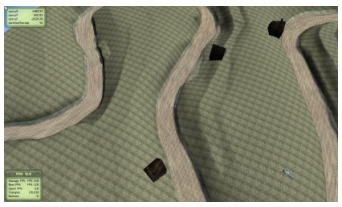

Figure 13: Camera far away from the river surface where little detail can be seen

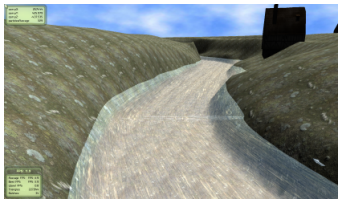

Figure 14: River surface sharing almost the same percentage of screen as all the other elements where several visual details are visible

Figure 15: River occupies almost the entire screen where details can be clearly seen

All the tests were performed with the same river mesh and the camera placed in the positions seen in Figures 13, 14 and 15. This way we can not only understand

 ISBN 978-80-86943-75-6

how the cost evolves across different resolutions but also how it varies according to different percentages of river mesh present on screen.

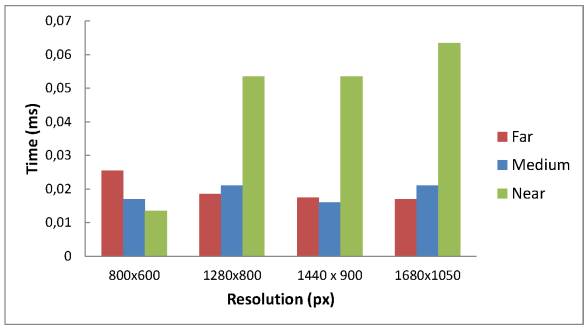

Figure 16: Time taken in milliseconds by the vertex shader to run at different resolutions and different cameras distances

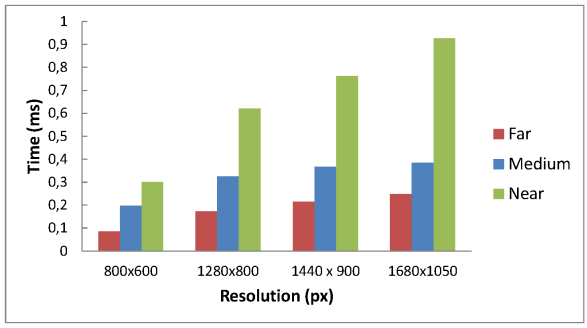

Figure 17: Time taken in milliseconds by the fragment shader to run at different resolutions and different camera distances

4.2.1 Vertex Shader

We have in Figure 16 the results of several measurements performed at different class distances (near, medium and far) and with different resolutions. As most of our computations are performed in the fragment shader, the vertex shader performs only very simple calculations such as transforming vertex positions from one space to another, calculate the camera direction and pass the vertex normal and texture mapping coordinates to the pixel shader. This means that all values are very small and despite the apparent increase in the near distance values when compared with the medium and far values, we see it never reaches differences higher than 0.05 milliseconds.

4.2.2 Fragment Shader

In Figure 17 we can see the time in milliseconds taken by the river fragment shader to complete.

As the resolution increases, the cost of performing the fragment shader increases along with the number of pixels to color. We can also see that the cost increases as we get closer to the river. As the far and medium distances have a smaller amount of river covering the screen, their costs are much smaller than the near distance which covers almost the entire screen. Despite doing several reads from textures, the cost of running the fragment shader even in the highest resolution is quite small. This is due to the fact that most of the operations we perform are based on reading the information provided by the textures created in the CPU and as far as new calculations go, we perform only the flow algorithm and the reflections which are not very expensive.

4.3 Conclusions and Future Work

We presented a new approach called SiViFlow which simulates realistic rivers in real-time. SiViFlow has two main components: the Simulation Engine and the Visualization Engine. Thanks to the Simulation Engine, SiViFlow is able to adapt to an arbitrary shaped river bed with any number of turns and dynamically calculate the necessary data based on the river surface mesh alone. It also utilizes a concept called river particles to retrieve flow information from the river surface mesh and send it to be drawn in the GPU. The Visualization Engine renders the river flow and is flexible enough to be combined with any visual technique used to simulated water, not being bounded only to the techniques presented in this work. SiViFlow also allows for dynamic objects to alter the course of the flow and change in real-time its behaviour through access to the flow information stored at the river surface mesh.While this approach fulfilled all of the objectives initially defined, there's still room for improvement. With all the advances in the computing capabilities of the new GPU's and respective API's that allow them to perform general computations, a future improvement would be to move the particle update, creation and destruction to the GPU, performing the whole update loop there. As the loading of new data to the flow texture does not have interdependencies among entries, this means that in the limit the whole process of filling the flow texture can be performed completely in parallel. As the approach presented does not have any limitation when it comes to the shading of the water, all visual techniques are compatible with the algorithm and are easily implementable within the Visualization Engine.

4.4 Acknowledgements

This work was supported by national funds through FCT - Fundacao para a Ciencia e a Tecnologia, under project PEst-OE/EEI/LA0021/2013.

5 REFERENCES

[AMHH08] Tomas Akenine-Möller, Eric Haines, and Natty Hoffman, *Real-time rendering 3rd edition*, ch. Reflections, pp. 386–391, A. K. Peters, Ltd., Natick, MA, USA, 2008.

 ISBN 978-80-86943-75-6

[Bri07] Robert Bridson, *Fast poisson disk sampling in arbitrary dimensions*, ACM SIGGRAPH 2007 sketches (New York, USA), SIGGRAPH '07, ACM, 2007.

[CBL+09] Yuanzhang Chang, Kai Bao, Youquan Liu, Jian Zhu, and Enhua Wu, *Particle importance based fluid simulation*, Proceedings of the 2009 Sixth International Conference on Computer Graphics, Imaging and Visualization (Washington, DC, USA), CGIV '09, IEEE Computer Society, 2009, pp. 38–43.

[CTG10] Jonathan M. Cohen, Sarah Tariq, and Simon Green, *Interactive fluid-particle simulation using translating eulerian grids.*, SI3D, ACM, 2010, pp. 15–22.

[DG96] Mathieu Desbrun and Marie-Paule Gascuel, *Smoothed particles: a new paradigm for animating highly deformable bodies*, Proceedings of the Eurographics workshop on Computer animation and simulation '96 (New York, USA), Springer-Verlag New York, Inc., 1996, pp. 61–76.

[DH06] Daniel Dunbar and Greg Humphreys, *A spatial data structure for fast poisson-disk sample generation*, ACM Transactions on Graphics **25** (2006), no. 3, 503–508.

[EMF02] Douglas Enright, Stephen Marschner, and Ronald Fedkiw, *Animation and rendering of complex water surfaces*, Proceedings of the 29th annual conference on Computer graphics and interactive techniques (New York, USA), SIGGRAPH '02, ACM, 2002, pp. 736–744.

[Eng03] Wolfgang Engel, *Shaderx shader programming tips and tricks with directx 9*, ch. Rippling Reflective and Refractive Water, pp. 357–362, Wordware Publishing, 2003.

[Fos96] Nick Foster, *Realistic animation of liquids*, Graphical Models and Image Processing **58** (1996), no. 5, 471–483.

[GH06] Jostein Gustavsen and Dan Lewi Harkestad, *Visualization of water surface using GPU*, Master's thesis, Norwegian University of Science and Technology, 2006.

[HKK07] Takahiro Harada, Seiichi Koshizuka, and Yoichiro Kawaguchi, *Smoothed Particle Hydrodynamics on GPUs*, Proceedings of Computer Graphics International, 2007, pp. 63–70.

[Ney03] Fabrice Neyret, *Advected textures*, Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation (Aire-la-Ville, Switzerland, Switzerland), SCA '03, Eurographics Association, 2003, pp. 147–153.

[Ngu07] Hubert Nguyen, *Gpu gems 3*, ch. Real-Time Simulation and Rendering of 3D Fluids, pp. 633–675, Addison-Wesley Professional, 2007.

[PF05] Matt Pharr and Randima Fernando, *Gpu gems 2 - programming techniques for high-performance graphics and general-purpose computation*, ch. Octree Textures on the GPU, pp. 595–613, Addison Wesley, 2005.

[SJ09] Raymon Serway and John Jewett, *Physics for scientists and engineers 8th edition*, ch. The Nature of Light and the Principles of Ray Optics, pp. 1010–1025, Brooks Cole, 2009.

[Sta99] Jos Stam, *Stable fluids*, Proceedings of the 26th annual conference on Computer graphics and interactive techniques (New York, NY, USA), SIGGRAPH '99, ACM Press/Addison-Wesley Publishing Co., 1999, pp. 121–128.

[Tes99] Jerry Tessendorf, *Simulating ocean water*, SIGGRAPH'99 Course Notes, vol. 2, ACM, 1999.

[vdLGS09] Wladimir J. van der Laan, Simon Green, and Miguel Sainz, *Screen space fluid rendering with curvature flow*, Proceedings of the 2009 symposium on Interactive 3D graphics and games (New York, NY, USA), I3D '09, ACM, 2009, pp. 91–98.

[vH11] Frans van Hoesel, *Tiled directional flow*, ACM SIGGRAPH 2011 Posters (New York, USA), SIGGRAPH '11, ACM, 2011, pp. 19:1–19:1.

[WLL04] Enhua Wu, Youquan Liu, and Xuehui Liu, *An improved study of real-time fluid simulation on gpu: Research articles*, Computer Animation and Virtual Worlds **15** (2004), no. 3-4, 139–146.

[YN08] Qizhi Yu and Fabrice Neyret, *Models of animated rivers for the interactive exploration of landscapes*, Ph.D. thesis, Institut National Polytechnique de Grenoble, November 2008.

[YNBH09] Qizhi Yu, Fabrice Neyret, Eric Bruneton, and Nicolas Holzschuch, *Scalable real-time animation of rivers*, Computer Graphics Forum (Proceedings of Eurographics), vol. 28 (2), March 2009.

 ISBN 978-80-86943-75-6