LGURJCSIT
Volume No. 1, Issue No. 1 (Jan-March 2017) pp. 22-36

Algorithm and Technique for Animation

Syeda Binish Zahra[1]

Abstract: - Fluids simulation particularly water courses such as rivers are an important element to achieve realistic simulations in real-time applications like video games. This work presents a new approach called **SiViFlow** that simulates watercourses in real-time. The algorithm is flexible enough to be used in any type of environment and allows a river to be dynamically generated given any riverbed. The component that manages the flow is responsible for the water animation and allows the use of various techniques to simulate visual features. As all the information is dynamically generated, **SiViFlow** also reacts to dynamic objects that come in contact with the river, properly adjusting the course of the flow. This work helps accelerate and improve the methods of creating realistic rivers so that they can be used in video games.

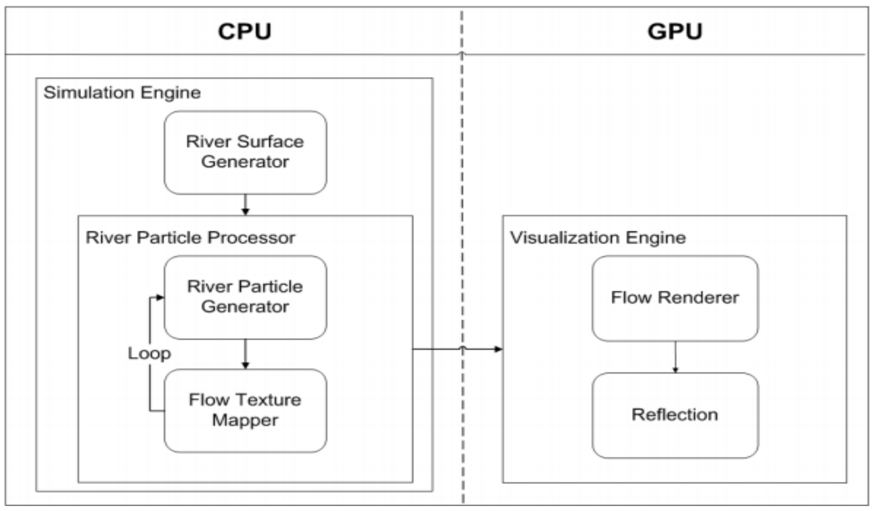
1. INTRODUCITON

SiViFlow is composed by two main elements: The Simulation Engine and the Visualization Engine. The Simulation Engine is where all the calculations related with 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. The description of VisiFlow is depicted in Figure 1 with all of its elements.

In the River Surface Generator, we start by generating the river surface mesh that will be used to apply the material and where the water animation algorithm will be rendered. At this stage, we have to calculate several features that will be needed in later stages such as river width, which vertices define the shore, the flow in each vertex, amongst others. The next stages are River Particle Generator and the filling of the flow and auxiliary textures in the Flow Texture Mapper. These two stages make up the application loop that runs in the CPU. In this loop we generate randomly distributed points that cover as much as possible of our domain in screen space and from those points we create a concept called river particles. These textures are lled with river particles so we can send their features to the GPU, which in turn allows us to update these river particles every frame. In the end of SiViFlow we have the rendering of the material which uses the textures that were sent from the CPU. At this stage we use the Visualization Engine to render all visual and physical effects such as flow and reflections.

Syeda Binish Zahra
Department of Computer Science,
Lahore Garrison University Lahore.
binishzahra@lgu.edu.pk

Figure 1: CPU Verses GPU



2. RIVER SURFACE GENERATOR

The first stage of all is the River Surface Generator. At this stage a river surface mesh needs to be created, which can either be done using an external modeling application or generated in real-time. Both options are viable and don't interfere with the next phase as long as we have access to the river mesh vertices. In both cases all we require is a mesh which will describe the river surface. The meshes we used assumed that the first vertex was one of the corner vertices of the river surface mesh.

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 fag 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. 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). This means the last vertex we processed belongs to the opposite shore. This idea is very similar to the one we used in Algorithm 1 to calculate the river width and works in a similar way.

We didn't consider different widths across the river sections as they don't affect any of the other modules of the algorithm and it only means that if one wants to introduce them, all that's required is to create a more sophisticated way to calculate the river width for each section in order to find out the amount of vertices of the mesh that go from one shore to the other. These ways later stage of the algorithm will know for each section what the correct width of the river is.
Algorithm 1 sums up all the steps taken during this pre-processing phase. The only input information required is 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 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. As we've mentioned before, if different river widths were a requirement, all that would be needed to do was to create a more sophisticated algorithm that would be able to know when a certain river section had ended and use that information to store for each river section its width. At this stage we know the river width at each section as 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.

2.1 Flow Generation

At this stage all shore vertices are identified and we need to generate the flow vectors that later will be passed to the river material. In order to calculate the flow, we pick two shore vertices in the same river section, and 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. With these three points we can 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. As the plane generated 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 as shown in Figure 4. One advantage of generating the flow this way is related with its edibility 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, as long as we know the collision vertices, SiViFlow is able to recomputed the flow of the river and immediately reflect the changes.

Algorithm 1: River surface generation algorithm

```
Input: Set of control vertices that define the river surface mesh1
RiverSurfaceGeneration(ControlVerticesSet )
        vertices ⟵ ControlV erticesSet
        forall the vertices do
        if vertex is a shore vertex then
         Flag it
         RiverDistance (vertices) // For each vertex store river width
        DistanceToMargins (vertices) // For each vertex store distance to each margin
        CalculateFlow (vertices) // For each vertex calculate and store flow
        RiverDistance(Vertices )
        iterator ⟵ 0 // iterator
        maxV ertices ⟵ V ertices:length // length of the Vertices vector
 dist ⟵0 // value to hold the maximum distance obtained
 while iterator < maxVertices do
 distCmp = distance(Vertices [0], Vertices[iterator])
 if distCmp < dist then
 dist = distCmp
iterator = iterator + 1
DistanceToMargins (Vertices)
 iterator ⟵ 0 // iterator
 maxV ertices ⟵ V ertices:length // length of the Vertices vector
while iterator < maxVertices do
if Vertices[iterator] not a shore vertex then
Calculate and store distance to left shore
Calculate and store distance to right shore
iterator = iterator + 1
CalculateFlow(ShoreVertices )
forall the Pair of shore vertices do
midpoint ⟵ CalculateMidpoint(lShoreV ertex; rShoreV ertex)
flowV ector ⟵CalculateP laneNormal(lShoreV ertex; rShoreV ertex; midpoint)
forall the vertices in this river section do
 Store flowVector
```

2.2 Flow Velocity

2.2.1. Stream Function

In order to calculate the interpolated value of the stream function (ψ), we use an interpolation scheme suggested by [2][3]. At this stage we have all the information required to calculate the following equations. We run for each vertex all the Equations 2.1 2.2 2.3 and store their values.

$$\Psi(P) = \frac{\sum_i w(d_i)\Psi_i}{\sum_i w(d_i)} \qquad 2.1 \qquad (3.1)$$

With P being the position of each river surface vertex, d*i* the distance from point P to the each ofthe boundaries and the weighting factor is **ω:**

$$w(d) = \begin{cases} d^{-p} \cdot f\left(1 - \frac{d}{s}\right), & \\ 0, & if \ s \end{cases} \qquad 2.2 \qquad (3.2)$$

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

$$f(t) = 6t^5 - 15t^4 + 1(\qquad 2.3 \qquad (3.3)$$

As we didn't change the interpolated stream function method created by [4], we must guarantee that at least two boundaries are inside every vertex search radius. This guarantee is very important as the initial premise that the flow rate between any two points in a flow field is equal to the numerical difference in boundary values of that channel would be false in case only one boundary was found, invalidating this scheme and returning undefined values.

2.2.2. River Particle Processor

In this section we'll introduce the concept of river particles. These river particles are used as way 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 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, analyzing this problem in screen space guarantees that these radius disks maintain a uniform radius, something that would not happen if they were projected in world space. 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 and wave sprites which present an analogous solution adapted to the context of those works. The following sections present how we generate river particles, how we store their information and how we prepare these particles to be efficiently sent to the GPU

2.2.3. RIVER PARTICLE GENERATOR

We start by generating several randomly distributed points, generating a Poisson-disk pattern using a modified boundary sampling algorithm.

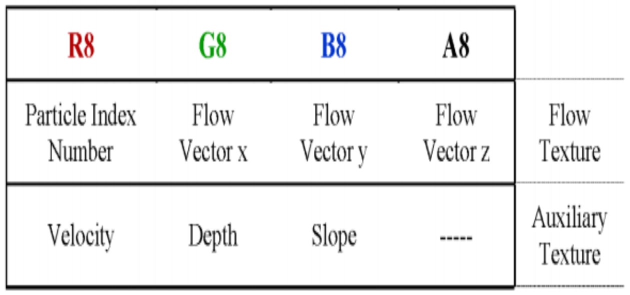
In the end of running this algorithm, we end up with a set of points that we'll convert to river particles. In order to generate a 3D world position for each of these points (after being generated we only have their 2D coordinates) we proceed. A ray is cast for each particle and we store the collision 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, velocity and flow.

Unlike other algorithms, we don't advent 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 advent particles during the update loop is that it allows us to load 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 river particle. This step differs from 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 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 to achieve similar



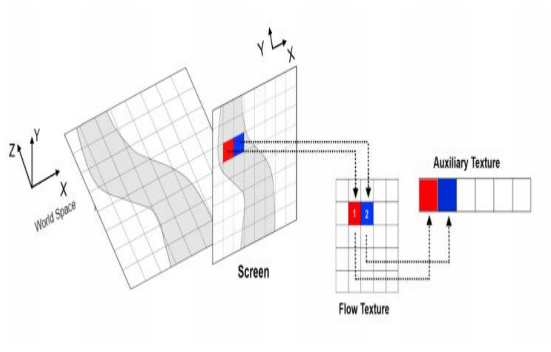
R8	G8	B8	A8	
Particle Index Number	Flow Vector x	Flow Vector y	Flow Vector z	Flow Texture
Velocity	Depth	Slope	-----	Auxiliary Texture

objectives. We store all the information we need inside each color channel and read it back when it reaches the GPU. This approach of using an auxiliary texture to carry data into the GPU allows us to update every frame the contents of these two textures, refreshing the particles and their respective values. One of the disadvantages is how their flow texture size must be as close as possible to the

Figure 2: Flow vectors

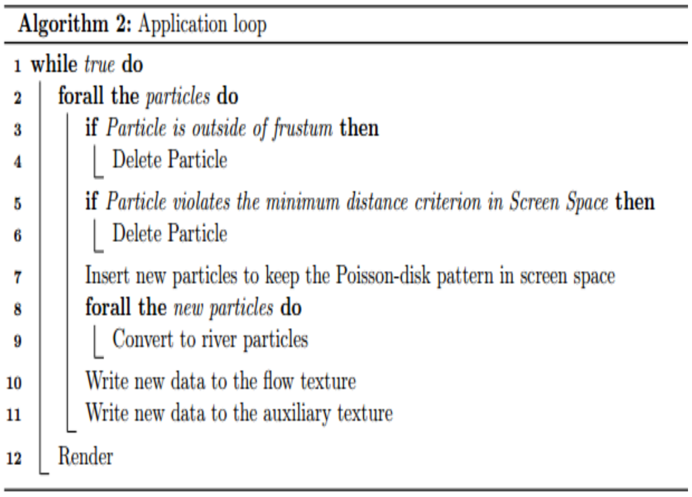
application resolution being used and with that the radius of the Poisson-disk needs be larger too. The increment in both these elements prevents their approach from being executed in high screen resolutions that are so common today. In our case, our flow texture and Poisson-disk may be much smaller than the screen size as we don't need a description of the domain. The reason behind this is due to the fact that both our river flow and speed don't change dramatically from one vertex to the next, meaning that if we want we can keep a much smaller copy of the flow texture when compared with the screen size.

Figure 3: Texture at screen



These textures will store the river particles previously generated using each of the color channels of the texture. In the flow texture we'll 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 store the flow information that covers that pixel. The auxiliary texture will have other parameters such as velocity, river bed slope and river depth. In Figure 3 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 a
etc. the final result. Then we need to delete the particles that are too close to one another



Algorithm 2: Application loop

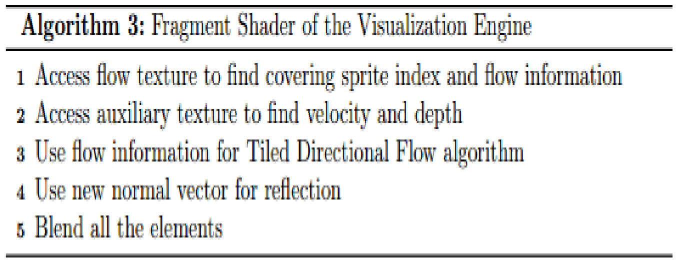
```
while true do
    forall the particles do
        if Particle is outside of frustum then
            Delete Particle
        if Particle violates the minimum distance criterion in Screen Space then
            Delete Particle
        Insert new particles to keep the Poisson-disk pattern in screen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

violating the initial

Poisson-disk requirement that all particles must be no closer to each other more than a specified radius distances. 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. VISUALIZATION ENGINE

The Visualization Engine is the last stage of SiViFlow and consists of mapping a material to the river surface mesh. This stage is divided in two main elements: The Flow Renderer and the Reflection algorithm. 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.



Algorithm 3: Fragment Shader of the Visualization Engine

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

We use the flow information to generate a new normal vector using the Tiled Directional Flow algorithm and use this new normal to compute the scene's reflection. In the end all the elements are blended together. All the steps of the algorithm are summed up in Algorithm 3.

4. FLOW RENDERER

The flow algorithm used is based in the approach proposed called "Tiled Directional Flow". Our approach uses a similar concept to render the flow. One of the main differences is that all the flow information being fed to the algorithm isn't 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.

4.1 Tiling of the water

The way the Tiled Directional Flow works is by dividing a river channel in tiles, similar to a chess board. We show this division where we painted some tiles with black color in order to make it easier to visualize what happens. Each tile is independent from its peers and it's composed by several normal maps. This tiling allows this algorithm to have several normal maps combined per region, that when seen as a whole don't resemble the usual texture scrolling seen in most video games implementation. This visual advantage combined with an adaptive flow system as ours, allow the river to behave in a realistic way and react to any interactions.

4.2 Normal maps composition

Normal mapping is a technique which modifies the per-pixel shading routine of a mesh in order to fake the lighting of bumps and dents [6][10]. Usually a normal map is created from a highly detailed mesh and used to fake details in a simplified mesh with much less polygons. 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. After that 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 these normal maps are rotated and combined together using the same blending factor. To get the final normal value, both normal maps that were combined using the blending factor are blended once more. 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.

4.3 Reflection

In order to simulate dynamic reflections of objects on our river surface we used a method commonly called planar reflections. 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. This technique is based on the use of a texture called a reflection map, which is an inverted version of what it's visible above the water level and that we want to reflect. To obtain a reflection map, we start by defining a clipping plan, which has to be about the same height as the river surface. This clipping plane will be useful to cut all the geometry below the river surface that we're not interested in having rendered. 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 to a texture.

5. CONCLUSION

This Document presented a new flow visualization algorithm called SiViFlow and explained each of its components. We start by generating a river surface mesh and calculate several attributes that will be useful for the next stages of the algorithm. These attributes include river width, width of each vertex to both margins and the flow of each vertex. We start the update loop of the algorithm where we first update the state of our river particles by creating, deleting and updating river particles and then we fill the flow and auxiliary textures with data. These textures are sent to the GPU where it reads them inside our Visualization Engine and outputs the appearance of a river flowing.

6. REFERENCES

[1]. Tomas Akenine-MUoller, Eric Haines, and Natty Ho man. Real-Time Rendering 3rd Edition, chapter Reections, pages 386{391. A. K. Peters, Ltd., Natick, MA, USA, 2008.

[2]. Robert Bridson. Fast poisson disk sampling in arbitrary dimensions. In ACM SIGGRAPH 2007 sketches, SIGGRAPH '07, New York, USA, 2007. ACM.

[3]. Robert Bridson, Ronald Fedkiw, and Matthias Muller-Fischer. Fluid simulation: Siggraph 2006 course notes. In ACM SIGGRAPH 2006 Courses, SIGGRAPH '06, pages 1{87, New York, USA, 2006. ACM.

[4]. Yuanzhang Chang, Kai Bao, Youquan Liu, Jian Zhu, and Enhua Wu. Particle importance based uid simulation. In Proceedings of the 2009 Sixth International Conference on Computer Graphics, Imaging and Visualization, CGIV '09, pages 38{43, Washington, DC, USA, 2009. IEEE Computer Society.

[5]. Nuttapong Chentanez and Matthias Muller. Real-time simulation of large bodies of water with small scale details. In Proceedings of the 2010 ACM SIGGRAPH/Euro graphics Symposium on Computer Animation, SCA '10, pages 197{206, Aire-la-Ville, Switzerland, 2010. Euro graphics Association.

[6]. Jonathan Cohen, Marc Olano, and Dinesh Manocha. Appearance-preserving implication. In Proceedings of the 25th annual conference on Computer graphics and interactive techniques, SIGGRAPH '98, pages 115{122, New York, USA, 1998. ACM.

[7]. Jonathan M. Cohen, Sarah Tariq, and Simon Green. Interactive uid-particle simulation using translating eulerian grids. In SI3D, pages 15{22. ACM, 2010.

[8]. Mathieu Desbrun and Marie-Paule Gascuel. Smoothed particles: a new paradigm for animating highly deformable bodies. In Proceedings of the Euro graphics workshop on Computer animation and simulation '96, pages 61{76, New York, USA, 1996. Springer-Verlag New York, Inc.

[9]. Daniel Dunbar and Greg Humphreys. A spatial data structure for fast poisson-disk sample generation. ACM Transactions on Graphics, 25(3):503{508, 2006.

[10]. Wolfgang Engel. ShaderX Shader Programming Tips and Tricks With DirectX 9, chapter Rippling Reective and Refractive Water, pages 357{362. Wordware Publishing, 2003