Real-time GPU-based River Surface Simulation

Philip Scales
UGA, Grenoble, France
philip.scales@etu.univ-grenoble-alpes.fr

MoSIG supervisor: Ylies Falcone. MAVERICK (LJK/INRIA) supervisor: Fabrice Neyret

Abstract

Rivers are a part of many real-time graphics applications such as games and simulators. Stationary shockwaves are seen near obstacles in real rivers, as in Figure 1. These local surface phenomena are omitted by most existing methods for simulating rivers in real-time. Those that include them use CPU-based methods for computing velocity fields and generating surface details, limiting their scalability and applicable rendering techniques. Accurately representing surface details such as shockwaves contributes to making the scene more believable and rich. Our focus is on phenomena that occur in non-turbulent rivers or brooks. We provide a method to simulate them in real-time on a GPU.

1 Introduction

Many applications in the field of computer graphics require the representation of large explorable natural scenes. Many techniques exist to accurately reproduce the general aspect of water, its interaction with light, and to capture its movement. However most techniques used to represent water flows fail to represent certain types of phenomena that affect the water's surface. In most applications we usually see either waves without flow (for oceans and lakes), or flow in rivers with some drawbacks like water passing through obstacles, and not interacting with them. When attempting a realistic depiction of natural phenomena, any unnatural behaviour will stand out to the viewer. Given that these effects are part of our landscape, taking them into account will create a more plausible scene.
Some local surface modifications could be manually captured with the help of artists, however this is time consuming and not necessarily physically accurate. More realistic fluids are often animated with the help of physics simulations, using methods such as those described in part V of [Nguyen, 2007]. Fluid simulations remain computationally heavy workloads even if accelerated on a GPU, especially if performed in the 3D case where real-time framerates can only be achieved with very small simulation domains, or by reducing details.
Computing physically plausible 2D velocity fields in real-time at fine resolution is now achievable, and would yield a more lively and realistic water flow taking obstacles into account.
A few works have studied the simulation of local surface phenomena, with some remaining limitations related to precomputation of static velocity fields and mesh-based surface representations which are detailed in section 2.2. Alternatives to mesh-based techniques exist, such as computing per-pixel surface normals in order to apply techniques reviewed in section 2.3. In this paper, we focus on the case of shallow water, with calm rivers or brooks. We base ourselves on models and methods from previous works [Neyret and Praizelin, 2001] and [Yu *et al.*, 2011] to propose a meshless GPU-based approach for simulating surface details from a dynamic 2D velocity field.

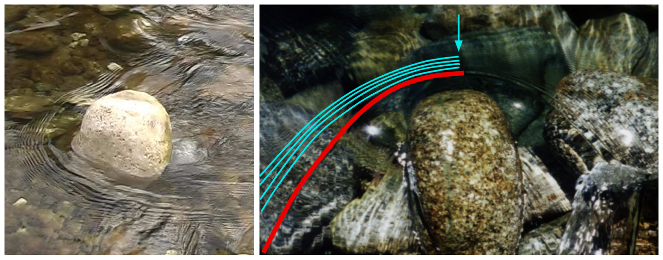
Figure 1: *Reference images for stationary shockwaves. Left:flow from left to right. Right: flow from top to bottom. The elements of the stationary shockwave are highlighted. Red: primary (gravity) wave. Blue: resulting (capillary) waves.*

2 Related Work

2.1 Water simulations from 3D flow

Physics-based approaches for simulating bodies of water have become more popular. Many methods to perform full 3D fluid simulations exist, for example using Solid Particle Hydrodynamics [Kipfer and Westermann,] or 3D Navier-Stokes equations [Takahashi *et al.*, 2003]. The performance of these methods depends heavily on the resolution of the simulation grid. The animation of rivers with changing viewpoints requires handling both small details at small viewing distances,

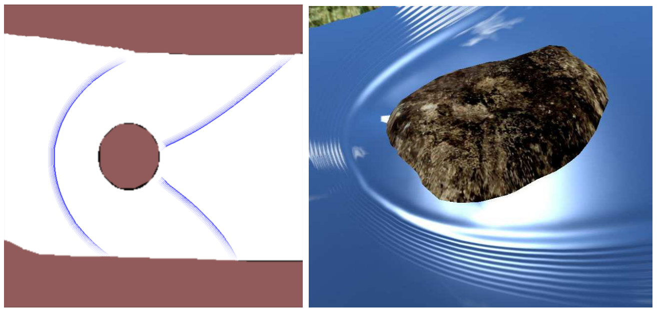

Figure 2: *Results from previous works. Left: front and rear shockwaves drawn from [Neyret and Praizelin, 2001] Right: basic surface rendering from citevecsim.*

and large amounts of water at large viewing distances. Using such methods to solve our problem would require a precision in the order of millimeters, making real-time performance impossible. Additionally, complex phenomena such as surface tension also play a role in the appearance of surface details at such a small scale, and taking this into account would lead to an even heavier workload. Additionally, simulation of the non-visible parts of the fluid are not useful for our purposes, and only serve to waste resources.

2.2 Surface simulation from 2D flow

Very few works have attempted to reproduce all of the surface details we are interested in. Here we present two previous works that model stationary shockwaves caused by obstacles. A procedural method is proposed in [Neyret and Praizelin, 2001] to generate the surface details we seek to represent. They suggest that those details' visual manifestations are local wave phenomena that mostly affect the surface of the river, despite their complex underlying causes. Their method does not rely on a 3D simulation, requiring only a static 2D velocity field. Based on shallow water and wave physics, they derived simple criteria to pinpoint feature occurrences, and describe their geometric properties. Interactions between floating objects and features are also handled. The main drawback is that their method does not include rendering of the water surface.
A later work [Yu *et al.*, 2011] used the former as a base, adding methods to render the surface details. Their method allows a convincing rendering for the targeted type of flow. This is achieved by representing the waves with triangle meshes that are stitched together with the original water mesh, enabling a classical rendering pipeline to be applied. While mesh representations are widely used in computer graphics, their use for representing intricate surface details poses several problems. Heavy computation is needed to determine intersections of meshes for overlapping features, and doing this for each frame quickly leads to performance issues. Adapting the meshes for efficient rendering at variable viewing distances and angles is also a complex task.
Both methods employ static precomputed velocity fields as their base. Computing the field and storing it uses a lot of time and memory resources, as well as limiting the realism of the water flow.

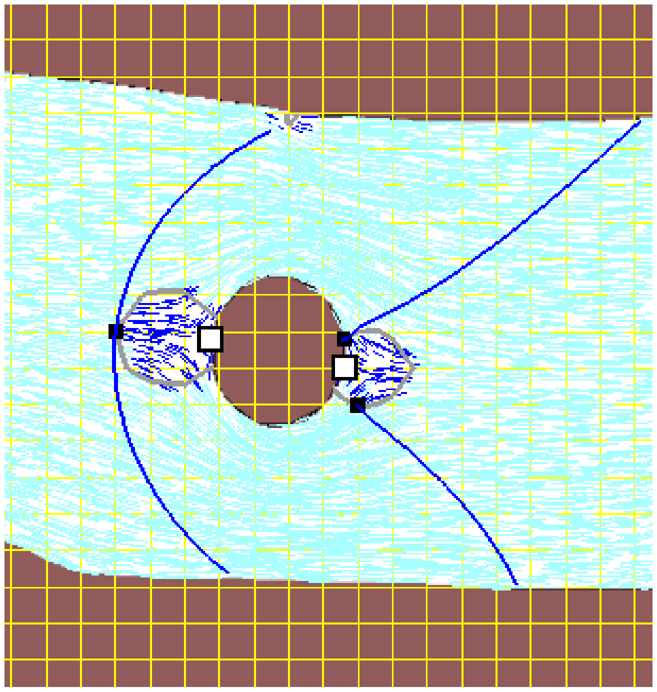

Figure 3: *Stationary shockwaves (blue curves) and their start points (black), $iso_{Fr=1}$ curve (grey lines), subcritical $Fr < 1$ areas (dark blue).*

2.3 Meshless surface rendering

The small scale surface details we aim to simulate are mostly visible due to the way they deform light by reflecting and refracting it. These properties can be captured by using the surface's normal vectors. These are fed into illumination models such as the Blinn-Phong model [Blinn, 1977] for opaque and diffuse objects which can be augmented to handle refractions [Rodgman and Chen, 2001].

3 Background

In this section we review in more detail the necessary theory and methods from previous works [Neyret and Praizelin, 2001] and [Yu *et al.*, 2011].

3.1 Froude Number and shockwave origin

The Froude number is used to describe the characteristics of a water flow, much like the Mach number for air. It is written as $Fr = \frac{\|\vec{v}\|}{c}$ with $\vec{v}$ the local flow velocity, and c the wave propagation speed. In our case of shallow water, we have $c = \sqrt{gh}$ with g gravitational acceleration, and h the local water depth. $Fr > 1$ and $Fr < 1$ describe supercritical and subcritical flows respectively and $Fr = 1$ characterizes the $iso_{Fr=1}$ line. The physical model established in [Neyret and Praizelin, 2001] assumes that a stationary shockwave originates at the point where the local flow velocity is orthogonal to the $iso_{Fr=1}$ line.
They compute start points for each frame by iterating over terrain and obstacle boundary points until finding the end of a $iso_{Fr=1}$ line. The $iso_{Fr=1}$ line is then constructed as a list of

points, which they iterate over. A point is a startpoint if $\vec{v}_n \cdot \vec{t}$ and $\vec{v}_{n+1} \cdot \vec{t}$ have opposite signs, where $\vec{v}_n$ is the velocity at the considered point, and $\vec{v}_{n+1}$ is the velocity at the next point along the velocity isovalue curve.
A simplified approach is employed in [Yu *et al.*, 2011] by precomputing the start point and moving it relative to its original position at each frame, based on the local change in velocity.

3.2 Shockwave shape

The general shape of the shockwave caused by obstacles in a river can be compared to the mach cone observed when a flying object breaks the sound barrier. The main difference is in the fact that the fluid's velocity varies along the wave. [Neyret and Praizelin, 2001] conclude that the wave crest lies at an angle $\alpha = \arcsin\left(\frac{c}{\|\vec{v}\|}\right)$ relative to the local flow velocity $\vec{v}$. Their method to represent shockwave crests is to use a list of line segments, with each segment lying at an angle α to the flow velocity $\vec{v}$ at its upstream point. The length of the segment is a fraction of $\|\vec{v}\|$. Figure **??** shows shockwave crests constructed from user-placed start points.
The waves such as those in Figure 1 consist of a larger gravitational wave in front of which are several smaller capillary waves or ripples. To capture the detailed shape of the shockwave, [Yu *et al.*, 2011] proposes a wave profile derived from wave theory. It gives wave height as $z(d) = z_g(d) + z_c(d)$ where d is the distance to the wave crest, and z_g and z_c denote contributions from the gravity and capillary waves respectively. Any C^2 wave profile can be specified by the artist, and they provide an example profile.

4 Our work

Our aim is to adapt the methods from previous works so that all computation is done on the GPU. We aim to avoid the use of meshes, and to allow integration of dynamically computed 2D velocity fields.

4.1 Motivation

Precomputing the base flow velocity limits the scalability of the previous works. It also limits the kinds of rivers that can be represented by their methods. We also wish to avoid the mesh-based representation of [Yu *et al.*, 2011].
Performing all computation on the GPU allows for different workflows for surface detail generation and representation, as well as velocity field computation. GPU computation of surface details and features allows us to compute per-pixel surface information, most notably normals. Moving these computations to the GPU also allows us to use it to compute the velocity field without costly CPU-GPU communication. GPU computation of the velocity field can allow use of a dynamic fluid solver to add more realism to the flow.

4.2 Method

Our method relies on building a heightmap to represent the river surface. The river is represented in a 2D grid, aligned with the river's surface. Using the velocity field, we identify areas affected by stationary shockwaves. We compute their effect on local water height to construct the heightmap of the river surface. From the heightmap we derive the surface normal vectors which are used to render it.

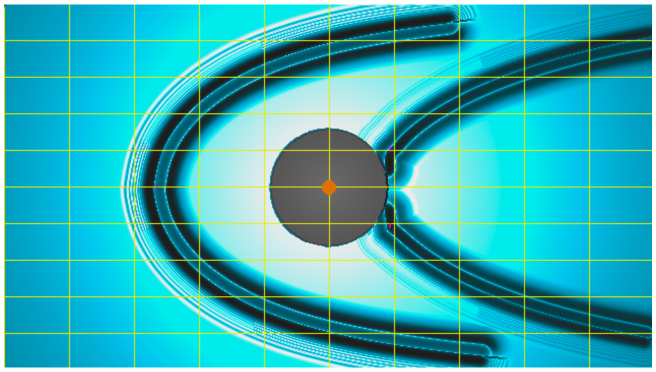

Figure 4: *Image rendered by our simulator, with a 100 region grid overlay in yellow.*

The rest of our method assumes velocity values are available at each frame, for each cell of the simulation grid, and does not depend on the exact method used to compute them. We also assume a description of the river banks and obstacles as input.

To locate the areas affected by the shockwaves, we first must find the shockwave starting points. The simplifications made in [Yu *et al.*, 2011] may result in inaccuracies in the position of the shockwave, especially if combined with a more complex and changing velocity field. We suggest to compute the start point criterion at each frame, for every cell representing water, using the same orthogonality criteria as [Neyret and Praizelin, 2001]. Given we do not construct $iso_{Fr=1}$ lines, we also explicitly check if the cell satisfies $Fr_{upstream} > 1$ and $Fr_{downstream} \leq 1$. On a CPU this would require a costly iteration over the whole grid, but exploiting the massively parallel architecture of the GPU renders this feasible, either through CUDA style programming or taking advantage of the per-fragment execution of fragment shaders.
Once start points are marked, we construct a list of start point coordinates to link each of them to their corresponding stationary shockwave. We search for start points over the entire simulation domain by dividing it into regions, each of which is searched in parallel.

To determine the shape of the shockwave crest originating from a given start point, we use the same method as [Neyret and Praizelin, 2001]. Shockwaves are represented as lists of segments whose endpoints are cells of the simulation domain. They are constructed by computing the next shockwave segment point's position from the local velocity field and previous point's position at each frame. The independence of shockwave list updates allows them to be performed in parallel.

To capture the 3D shockwave geometry, we store a height value for each cell of the grid to construct a heightmap. The

height values of cells affected by a shockwave are modified according to the profile given in [Yu *et al.*, 2011]. We make one modification consisting in using the capillary part z_c only for the upstream part of the shockwave, as ripples are normally observed in front of the shockwave as in Figure 1.
To render the surface we compute its normal vectors using the heightmap, thus avoiding the use of meshes. For this we use the approach proposed in [Yu *et al.*, 2011], offsetting the original surface normals along the direction of the normal to the wave crest.

In order to render the scene, rendering techniques such as those presented in section 2.3 are applied to the surface normal vectors.

4.3 Implementation Details

Our implementation [Scales and Neyret, 2018] consists of four fragment shaders and frame buffers, with a fifth fragment shader rendering the final output image. The shaders were developed and run with Shadertoy, a webGL tool allowing to share, view and program fragment shaders in GLSL ES 3.0. We will denote them as shaders and buffers A through D.
A fragment shader is a program executed by the GPU at each frame, for each fragment, which in our case is equivalent to once per pixel of the output image. A buffer stores the output values for each execution of it's respective fragment shader. The inputs of the fragment shader are the fragment's coordinates, and the sole output is a vector of four floating point values, stored in the shader's attributed frame buffer. For the final shader that generates the image, the output corresponds to the pixel values. A fragment shader has read-only access to all buffers.

Data structures

We have two types of data to store:

- data needed for each cell of the simulation grid (type of terrain, velocities, height values, normals)
- data describing features (list of shockwave start points, lists representing shockwave crests)

Given the available representations, per-cell information is computed by fragment shaders performing the same operation for all fragments, effectively mapping the simulation grid to output buffer, with one fragment representing one cell. All other data is computed by a "manager" fragment shader (shader A) with multiple execution paths depending on the processed fragment's screen coordinates. Fragments in a given region of the screen are allocated to a given task, resulting in different areas of the output buffer being used for different purposes.

Initialization and scene geometry

We require a description of the scene geometry, consisting of wall positions, obstacle positions and radii, and stream values at wall and obstacle borders. In the context of shadertoy, we simply define scene geometry directly in shader A. In practice this data would be uploaded from the CPU, either to uniform variables or to a buffer. Initialization of data structures and parameters are also performed by shader A.

Velocity Field

We will point out some necessary adaptations for using a grid-based fluid solver, and detail the use of a functional representation of velocity.
A grid-based method allows for solving 2D Navier Stokes equations, which can be achieved on a GPU by exploiting multipass rendering (*implementation example[Schuetze, 2017]*). Unlike traditional Computational Fluid Dynamics applications, the area we wish to simulate changes with the scene, requiring the shifting of boundary conditions which can lead to instability. Additionally, phenomena originating outside the scene can still impact its appearance, as is the case for a shockwave originating further upstream. For these reasons, the solver grid must extend further upstream and downstream than the visible simulation domain. Lastly, the grid can be refined or coarsened depending on viewing distance in order to improve performance.
A functional method allows for velocity values to be computed for any position in the simulation domain, and we will focus on the stream function method presented in [Yu *et al.*, 2009]. Stream values are specified as input for terrain and obstacle cells, and are interpolated to give the values at each cell of the simulation. The velocity at each point is determined by a finite differences operation on the stream values.
The velocity field is computed for each fragment by shader B. A call is made to the stream function with world coordinates (derived from the fragment coordinates), and stream values at river banks and obstacles as inputs.

Locating waves

Shader C checks each cell of the simulation with the start point criteria. It also computes the gradient of the velocity field, which we use to deduce tangents to velocity isovalue lines such as the $iso_{Fr=1}$ line. We require access to the current point's value as well as neighbouring values for velocity and gradient which are read from buffers B and C respectively. One of the output values of shader C is used to designate whether the fragment is a start point.
In order to build a list of start points in buffer A, the domain is divided into r regions of equal size, where r is a compile-time parameter. Shader A allocates r fragments to search the regions, which output the coordinates of the first start point found, creating an r by r matrix of potential start points.

Computing wave shape and profile

Shockwaves are separated into two lists representing the left and right hand side of the wave. Shader A allocates $2 * r^2$ lines of fragments to handle each list. The first fragment of each list reads the output value of the region it is allocated to, and uses it to compute the next point in the shockwave. The remaining fragments of a given list read their preceding fragment's output, and compute the next shockwave point. A noteworthy consequence of this is that it takes n frames for a modification to propagate to showckwave segment n. This emulates the time taken for information to propagate along the wave crest when it is perturbed.

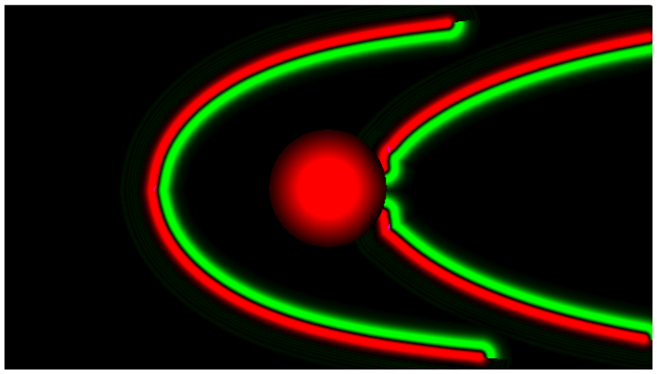

Figure 5: *A heightmap generated by our simulator. Black areas are the original surface height. Positive offsets in red, negative in green.*

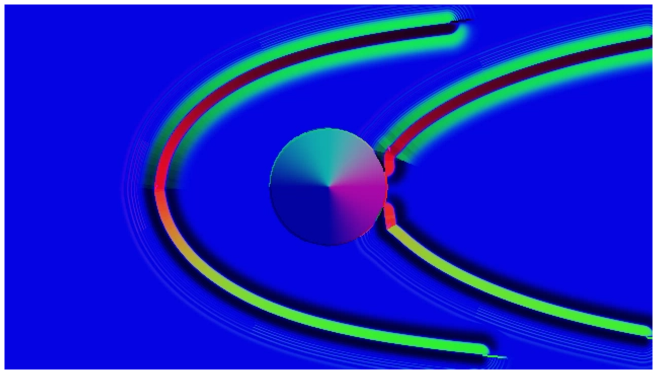

Figure 6: *A normal map generated by our simulator.*

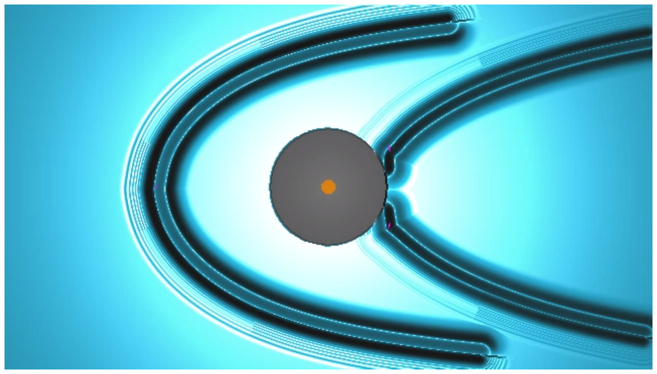

Figure 7: *Surface rendered by our simulator.*

Heightmap construction and rendering

The heightmap is computed for each fragment by shader D. For each fragment, we search for the closest of all shockwave segments. If the distance to that segment is less than half the wave width, we use it to sample the wave profile. This gives us the height offset to apply to that fragment, which we store as output (see Figure 5). The tangent to the wave crest is also stored. Normals are obtained in the final Image shader by sampling the heightmap at the next cell along the direction orthogonal to the shockwave crest (see Figure 6). The final Image shader computes lighting with a typical Phong illumination model in order to visualize the river surface.

5 Results and discussion

5.1 Results

The implementation of our method produces shockwaves with similar geometric properties to those generated by [Neyret and Praizelin, 2001], with both front and rear shockwaves. We also go further by computing detailed surface normals for the waves, resembling those in [Yu *et al.*, 2011]. A realistic rendering can be obtained from the surface normals using known methods for rendering water surfaces, which is outside of the scope of this work. Figures 7 and 2 show our surface and that of [Yu *et al.*, 2011] rendered with simple diffuse and specular lighting.

5.2 Discussion

Framerates for computing surface normals are real-time for small resolutions and simple scenes, but this is not the case at higher resolutions and with more complex scenes. We suspect the main cause for this is the naïve method used for computing the nearest shockwave segment for each point. This issue can be addressed by implementing acceleration structures to limit the number of comparisons needed, or by changing the method used to compute each point's position to the shockwaves.

One drawback of our implementation is the lack of robustness when changing the velocity field. Shockwave start points are still correctly identified, however the waves construction is interrupted, leading to gaps between segments in the waves. The order of updates of velocity, Froude number and shockwave points may lead to waves being terminated. Froude number values will be recomputed before the startpoint's position, potentially leading to the start of the wave being in a subcritical area where it should not exist. At the next pass, the new start point will once again be slightly upstream of the $iso_{Fr=1}$ line, and the line can be generated again. This could be improved either by limiting the speed at which the velocity field can change, or by modifying the order of updates.

Another drawback is the identification of start points. Instead of a single point being marked for each theoretical start point, multiple points are marked for each start point. Given that we currently generate shockwaves for each start point, this is inaccurate. Implementation of the start point check could be improved to take the local neighbourhood into account to avoid false positives. The marked points are still within a limited distance of the desired location, so alternatively we could apply an extra step to leave only one marked point from each cluster of points.

More realistic information propagation times could be obtained by weighting the displacement of the new point relative to the movement of it's predecessor with a time-dependant coefficient.

Our method focuses on stationary shockwaves, but more features could be taken into account using our heightmap representation. Given the height offsets caused by each feature, we can blend them together to obtain a single heightmap and easily determine the resulting surface normals.

6 Conclusion and future work

In this paper we presented a real-time GPU-based method for simulating shockwaves caused by obstacles in calm rivers and brooks where the shallow water case applies. Our method is more scalable than previous works, and provides comparable results in terms of accurate feature shapes and positions. We achieve this by combining the feature descriptions of previous works, with a dynamically computed velocity field. Our method is meshless, avoiding performance issues and allowing more flexibility regarding choices for the rendering pipeline of the water surface.
This work represents a small step in a large project to accurately depict the surfaces of rivers and brooks in complex scenes, and as such there are multiple aspects to explore. A first future work is to simulate the other features depicted in Figure 8. Hydraulic jumps due to discontinuities in riverbed height or swells caused by underwater sources simply depend on velocity and riverbed geometry. We can derive similar criteria allowing us to mark points located at riverbed discontinuities, or sources.
Another could be to study how to perform and distribute computations on the GPU in a more efficient manner, especially if more types of features are added.

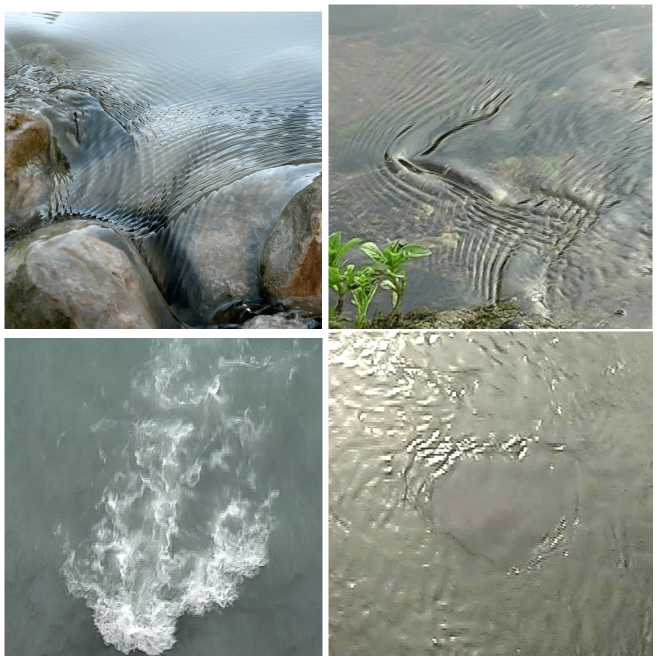
Figure 8: *Reference images. Intersection of waves (top left), hydraulic jump (top right), foam (bottom left), and boils (bottom right).*

References

[Blinn, 1977] James F. Blinn. Models of light reflection for computer synthesized pictures. *SIGGRAPH Comput. Graph.*, 11(2):192–198, July 1977.

[Kipfer and Westermann,] Peter Kipfer and Rüdiger Westermann. Realistic and interactive simulation of rivers. In *Proceedings of Graphics Interface 2006*, GI '06, pages 41–48. Canadian Information Processing Society.

[Neyret and Praizelin, 2001] Fabrice Neyret and Nathalie Praizelin. Phenomenological Simulation of Brooks. In *Eurographics Workshop on Computer Animation and Simulation (EGCAS)*, pages 53–64. Springer, 2001.

[Nguyen, 2007] Hubert Nguyen. *Gpu Gems 3*. Addison-Wesley Professional, first edition, 2007.

[Rodgman and Chen, 2001] David Rodgman and Min Chen. Refraction in discrete ray tracing. In Klaus Mueller and Arie E. Kaufman, editors, *Volume Graphics 2001*, pages 3–17, Vienna, 2001. Springer Vienna.

[Scales and Neyret, 2018] Philip Scales and Fabrice Neyret. River surface simulation. `https://www.shadertoy.com/view/MsGfz1`, 2018. [Online; accessed 2018-06-05].

[Schuetze, 2017] Robert Schuetze. Multistep fluid simulation. `https://www.shadertoy.com/view/MdSczK`, 2017. [Online; accessed 2018-06-01].

[Takahashi *et al.*, 2003] Tsunemi Takahashi, Hiroko Fujii, Atsushi Kunimatsu, Kazuhiro Hiwada, Takahiro Saito, Ken Tanaka, and Heihachi Ueki. Realistic Animation of Fluid with Splash and Foam. *Computer Graphics Forum*, 2003.

[Yu *et al.*, 2009] Qizhi Yu, Fabrice Neyret, Eric Bruneton, and Nicolas Holzschuch. Scalable Real-Time Animation of Rivers. *Computer Graphics Forum*, 28(2):239–248, 2009.

[Yu *et al.*, 2011] Qizhi Yu, Fabrice Neyret, and Anthony Steed. Feature-based vector simulation of water waves. 22:91–98, 04 2011.