Procedural Generation and Realtime Rendering of Waterbodies on Height-Map based Terrains

Richard Wolf, Tom Uhlmann, Guido Brunnett

Abstract: Over the past few years, virtual outdoor scenes have the propensity to grow in size, occasionally spanning several kilometers in the horizontal plane. The amount of time it takes artists to manually place landmarks and vegetation has also increased. As a result, tools for procedural generation have become crucial for projects with a broader scope. This paper will describe the procedural geometry generation and placement process for lakes and rivers, assuming that a particular landscape has already been expressed through a height map. A brief overview of how to display and animate realistic-looking water surfaces while working within the limitations of real-time rendering will also be given. The hydrology generation algorithm described in this paper is built on top of a particle hydraulic erosion system.

Keywords: Procedural Hydrology Generation, Realtime Rendering, Height-Maps

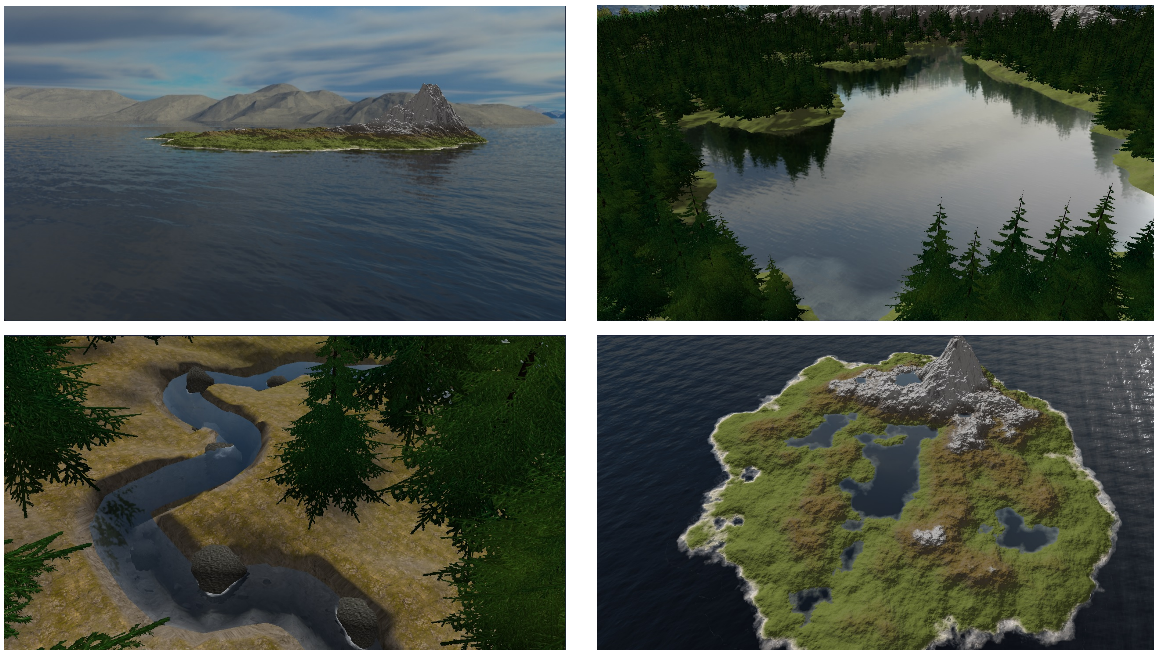


Figure 1: We show how to procedurally generate and dynamically animate different types of waterbodies, such as an ocean (top-left), lakes (top-right), and rivers (bottom-left). The different waterbodies are embedded in an existing height-map based terrain and enhance its appearance (bottom-right).

1 Introduction

Since water covers about 71% of the earth, the quality of the displayed water bodies has a significant impact on how the viewer perceives the visualized artificial outdoor scene. Since rivers, lakes, and oceans account for the vast majority of all naturally occurring bodies of water, these three water types will be the specific focus of this paper. Because all the rendering methods discussed are intended for real-time rendering, there are some limitations that must be taken into account. The algorithms for color calculation and animation, for instance, must have adjustable complexity while still maintaining a natural and as close to realistic look as possible due to the variation in hardware capabilities when displayed on different devices. The aspects of color calculation for rendering real-time water surfaces as well as insights into some animation techniques for water surfaces will be stated in section 2. Section 4 will assess the results attained, display performance benchmarks, and provide a brief analysis of how the methods presented might be improved. Although C/C++ and OpenGL were used to implement each of the methods, they are not unique to this programming language or graphics API. In contrast to the rendering of the water bodies, the procedural generation is not designed to be used in real-time, but is rather a pre-processing step performed offline. The intended viewing distance lies roughly between one meter to three kilometers. This papers only considers water surfaces viewed from above and does not cover any aspects of underwater rendering.

2 Related Work

This section gives a brief introduction to the topic of real-time water surface rendering by presenting a selection of methods for animating water surfaces, wave motion, and color calculation during the rasterization process. This is followed by a description of the hydrology generation approach which inspired the technique described in this paper.

2.1 Animating Water Surfaces

The exact movements of fluids depend on a wide range of variables, including the viscosity of the liquid, the gravitational force, the air pressure above the surface, the wind speed and direction, currents and turbulence currently present in the liquid, and many more. This makes the animation of waterbodies a complex topic. Thus, physical-based fluid simulations produce the most realistic results. However, this method is not appropriate for rendering waterbodies on a large scale in real-time because the computational cost quickly becomes unaffordable. Numerous approximations and water surface animation algorithms have been developed as a result, which, by being tailored for a specific arrangement of such factors and by removing liquid properties from the model that are unnecessary for their setup, imitate realistic water behavior while maintaining acceptable framerates even on low-end machines.

We will focus our explanations for each type of water body on the method most relevant

to our approach, but there are many other approaches that could be used to animate water bodies, each with their own advantages and disadvantages. The first technique is the fast Fourier transform [Flü17] and oceanographic spectra [T^+01] used to create and animate the ocean surface. This algorithm's main goal is to create a displacement and normal map that accurately represents the ocean's surface by combining many thousands of straightforward Gerstner waves [Fli18], each with its own direction, phase, and period. While the oceanographic spectra provide information about how these Gerstner waves relate to one another, the parameters for these waves can be slightly randomized in order to mimic the behavior of natural waves. The final displacement and normal map from these tens of thousands of waves can be computed in a fraction of the time using an optimized algorithm called the inverse fast Fourier transform. The main advantages of this approach include a high degree of realism, freely adjustable quality, and consequently cost. The algorithm also takes into account multiple parameters that determine how the ocean surface will ultimately appear, such as wind direction and speed, and makes them adjustable to fit particular scenes. By calculating the Jacobian, which shows how sharp the wave is at a specific point on the grid, FFT-based ocean animation algorithms also enable the calculation of the foam build-up on breaking waves. The main disadvantage of this strategy is that there is no physical interaction between objects floating in the water and the water's surface. Since most oceanographic spectra typically only take into account waves created by wind and gravitational forces, these physical interactions must be incorporated into the simulation separately.

Since the effect of vertex displacement would be practically undetectable when animating water surfaces with lower amplitude waves, such as lakes and rivers, the animation is typically only carried out in the fragment shader. As a result, simple meshes can be created that perform better than high-resolution ones. Although using a different wave spectrum, the FFT-based approach could also be used to animate lakes, the lake waves' wavelength and amplitude are much smaller than those of the ocean. Most applications forgo this higher level of realism in favor of a computationally less expensive approach. The method of scrolling textures used in [Wim15] comes in a wide range of variations. The basic idea is to simulate wave motion by moving precomputed wave textures across the water surface as opposed to real-time simulations like the FFT-based approach. Multiple layers of these textures are stacked on top of one another and moved in various directions and velocities to prevent the periodic appearance and produce a wave-colliding appearance.

The first two animation techniques are only practical for still waters with no water currents. However, the observed surface structure of rivers is not caused by the wind-produced waves but rather by the movement of water to a lower altitude and its interaction with obstacles on its way creating imperfections and turbulences. For water streams like rivers and brooks, for example, the amplitudes of the waves are also too low, so once again vertex displacement is not suited for most applications. Using flow maps to animate these streams is one option, as mentioned by Hayward [Hay10]. Similar to scrolling textures, a precalculated normal texture is used to render the surface structure. However, this normal

texture is not moved in an arbitrary direction, rather, a second texture is used, which contains information about the direction and speed of the water flow. The normal texture's texture samples are spaced apart in relation to this flow map and a time uniform. Despite the difference in flow velocities, the sampling algorithm is set up to give the user a realistic representation of moving water without causing the texture to overstretch. The physical interaction of the current with static objects that are within the stream can be computed in advance and recorded in the flow map. Foam can also be added to certain fragments to further enhance the realism of the scene where the difference in depth values between the water's surface and the background is below a predetermined threshold.

2.2 Rendering Water Surfaces

As with most materials, correct physical illumination is achieved by ray tracing, which even with contemporary graphics hardware is only possible with some model simplifications, such as limited ray count and bounce. Therefore, this chapter focuses on methods that are more viable options for a wider range of devices.

The final rendering of our approach makes use of various approximations of the effects seen in nature. A Screen-Space-Reflection algorithm, or short SSR [Let19], is used in the rendering process to determine how the water surfaces reflect light. SSR uses the data that is already present in the buffers rather than rendering the scene from a different perspective, making it less computationally expensive than the majority of reflection algorithms. The buffers do not contain data about the entire scene, so it has some drawbacks like potential information gaps. The refraction is yet another natural phenomenon. An algorithm was presented by Sousa [Sou05] to ascertain the color of this effect. The fundamental concept is to render the background of the transparent object into a buffer first, and then, rather than directly sampling this buffer and multiplying it with the transparency of the water, shift the background sample into the direction of the surface normal, which projects the wave structure onto the ocean floor. The main advantage of this approach is that only two or three texture samples are required to run the entire refraction simulation. Depending on the depth buffer difference between the water's surface and the background, the resulting refraction color is then mixed with the hue of the water.

2.3 Particle-based hydraulic simulations

The most popular method for simulating a realistic formation of hydraulic surfaces on dry terrains involves running a full simulation of a large number of water droplet particles and calculating their movement and erosion behavior. Tcheblokov [KBKŠ09] and Cloward [McD20] are two examples of particle-based erosion systems. Although these systems mimic realistic water behavior and produce smooth transitions between the water mass and the terrain, they also have some more significant drawbacks that make them less suitable for some projects. The most obvious one is how expensive their computations are, especially when used with data from larger-scale height maps. For instance, Cloward [McD20] demonstrates how it

takes several seconds to simulate the hydrology of the terrain up until full rivers and lakes are formed, even on a small terrain size of about 256x256. Another characteristic of hydraulic particle-based erosion systems is their ability to significantly alter the shape of the terrain by smoothing out sharp edges, cutting trenches into mountain sides, and transporting the eroded material into valleys, where it increases the height of the latter. It is preferable to keep as much of the original height data as possible for some applications because altering a landscape's overall shape gives the terrain a more worn appearance.

3 Procedural Generation of Waterbodies

The hydrology simulation and waterbody generation process is separated from the initial generation of the terrain, so no further knowledge over the algorithm used to create the height map data is required. However, a full hydrology simulation is computationally too expensive to be run on larger-scale terrains. Despite that, our algorithm enables high customizability for the resulting hydrology structures by adjusting the parameters of the water particles, such as the curviness of the rivers, the rate of particle erosion, and the maximum carried mass for these simulated droplets. The following algorithms for lake and river generation are based on such particle-based erosion systems, but change some important aspects in order to maintain the benefits while also reducing the drawbacks. A lake or river can be created with just one particle rather than relying on the fact that thousands of particles at some point in the simulation added up their volume to form a single waterbody. This is done in order to significantly improve the performance of the algorithm. This performance improvement is particularly noticeable for larger terrains because, with the new method, the price for one lake of the same size is independent of the size of the landscape, whereas with the contemporary method, the cost for one lake of the same size is multiplied by the size of the landscape when the terrain dimensions are doubled, multiplying the number of particles to fill a lake by a factor of four. The second modification to the algorithm from the literature removes the droplets from the simulation of landmass erosion so that the terrain cannot be altered by the particles. There are two reasons for this change. On the one hand, this preserves the original shape of the terrain and only requires minor adjustments near the river beds, which is helpful for applications where real height map data is used and it is the objective to remain as close to the original material as possible. On the other hand, as the particle count is reduced, the erosion quality is also decreased.

3.1 Procedural Generation of Lakes

The general design of the Lake creation algorithm is depicted in figure Figure 2. It involves finding the valleys in the height map data in the first step, then performing a stepwise floodfill where the fill height is increased with each iteration. There are two new parameters that must be used in place of the water volume and evaporation rate because, unlike the particle hydrology simulation, this algorithm does not use multiple particles with known

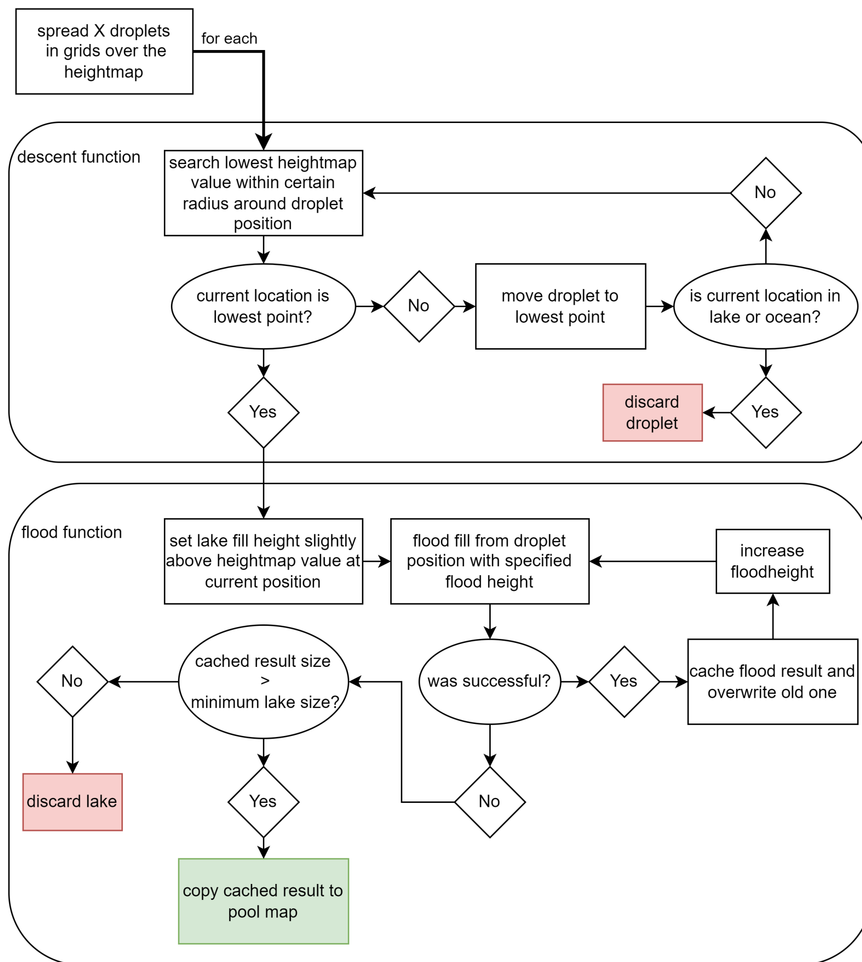


Figure 2: Overview of the lake generation process.

water volumes and evaporation rates. A lake generation is given boundaries by setting a minimum and maximum lake size, preventing generations of smaller puddles in place of lakes, as well as preventing a single lake from expanding to improbable sizes. A user-specified number of droplets are roughly evenly distributed over the terrain to begin the lake generation. The droplets may be distributed randomly or in grids, depending on the user's preference. The algorithm moves these particles directly downhill into the nearest valley rather than simulating actual water behavior for each of them. A droplet is moved to the location with the lowest height value after repeatedly reading the values of the height map in the box surrounding its current location to determine where the valley is. This is repeated until there is no lower location nearby on the height map. The presence and height of a lake relative to the terrain is then indicated using a second two-dimensional array, which is akin to the height map.

A stepwise floodfill is started from the last position of the particle with a fill height that is just a little bit above the height value of the current terrain location. This floodfill differs from a traditional floodfill in that it has several distinctive features as opposed to simply replacing tiles with new colors. First, the result of the floodfill is cached rather than directly

coloring the final Lake array. Additionally, since there is no discrete color palette but rather float values, all adjacent tiles with a height level lower than the fill height are designated as being a part of the lake that is currently generating, as opposed to simply covering up an old color. The particle and an invalid lake are thrown away as a result of the flood fill's additional termination requirements. There is a check to see if the maximum lake size has been reached in addition to simply relying on the algorithm running out of suitable neighboring tiles to signal the end of the recursion. The attempt to floodfill is also terminated if the tiles that are currently filled overlap with a waterbody that has already been generated. The fill height is then slightly increased and a new floodfill is started only if a floodfill is naturally terminated by running out of valid neighbors. Until the maximum lake size is reached or an overlap occurs, this process is repeated. At that point, the last valid floodfill result is checked to see if it meets the minimum size requirement before being passed into the lake map. The final step in the lake generation process is to extract the geometry from the lake map, which is subsequently used for rendering. This is done after repeating this process for each droplet.

The obvious solution for the rendering step is to place a quad for each lake tile at its corresponding water level, but this method produces an excessive amount of redundant vertices that overlap on the edges of the quad or are simply unnecessary because they are in the middle of the lake plane without adding any additional detail. To reduce the number of vertices, only a single plane is used to render each lake. This plane has a vertical position equal to the water level of the lake and the horizontal dimensions represent the two-dimensional bounding box of the specific lake on the lake map that can be generated within the flood fill algorithm. Without any further changes to the rendering, this approach to generating lake geometry would result in the edges of the plane being visible in places on the terrain where there should be no water. To prevent this, each plane geometry is given a unique lake identifier, and a lake map-like texture is passed to the lake's fragment shader. Instead of the float values representing the lake's water levels, this texture contains information about which tile of the lake map belongs to which lake, represented by its ID. By sampling this texture and comparing it to the lake ID passed from the vertex shader, the fragment shader discards any fragments that are outside the region of the lake. By rendering only one plane with dimensions equal to the bounding box of the lakes, this overdrawing and fragment discarding is reduced to a minimum. Even outside of the lake surface rendering process, it is beneficial to keep the lake map after the generation algorithm, as it can later be used in other processes such as procedural object placement to indicate whether a particular location in the landscape is submerged or dry, to prevent vegetation such as trees from being placed underwater.

3.2 Procedural Generation of Rivers

The river generation algorithm uses two distinct data structures to store the information for the river course. Although this redundancy has a higher memory cost, it significantly boosts the performance of the generation process. The first data structure is similar to the lake map,

as it is another two-dimensional array that holds information about the presence and absence of a river in a discrete location on the terrain, which has the benefit that lookup operations at a given terrain coordinate only have a complexity of $O(1)$. However, if a process must follow a single river from beginning to end using only the river map, it must first check, in the worst case, every tile to locate the river's start, which results in a complexity of $O(N \times M)$, with N, M the dimensions of the height map. It must then iteratively search the surrounding area to locate the river's next tile. Another issue with simply saving the river data in an array is that it is very inaccurate to convert the location of a river tile from the array indices back into world space because all floating point information has been lost. As a result, a second data structure is used, which comprises a list of all the generated river objects. Each river object holds an array with coordinates of points through which the river is flowing, as well as the width and depth of the river at these positions. Additionally, a cubic spline [Klu21] is formed through these data points once a river course is known, allowing for a smooth, curvy interpolation between the data points. It is much faster to use the spline data structure rather than the river map for tasks where a process needs more detailed information over a specific river at, for example, 5 meters beginning from its source. Evaluating the value of such a cubic spline has a complexity of $O(log(P))$, making it suitable for tasks where a process wants more detailed information over a specific river at a particular location.

Similar to the lake generation algorithm, a set of particles is randomly distributed across the landscape to start the river generation process. For river generation, as opposed to lake generation, the particles use acceleration-based movement over the terrain, which means that they continue to move into valleys at a downward angle but with a modified turn radius to match the slope of the terrain. The particle adds its current position and water volume to the river course position array of its corresponding river object during each movement-update iteration so that it can later be used to generate the spline. In order to determine when the movement loop should end, the droplet must also look for collisions with lakes or the ocean. If a particle returns to a spot it has previously visited, this typically means the particle is trapped inside a valley with no other water body nearby. The particle and its associated river object are deleted because keeping it would either produce a river with a dry end or a river that feeds itself, neither of which are accurate representations of water behavior found in nature. The river that was crossed is shortened to the point of collision if a particle collides with another river because the stream of the currently simulated river is interfering with its own flow of water current. The properties of the currently simulated droplet are then changed to mimic the water flow of these interacting water streams. First, its flow direction is determined by extrapolating between the two colliding streams' flow directions based on their respective water volumes. This means that, compared to a smaller current, the direction of a wider river with more water flow has a greater influence over the new direction of the future stream. Additionally, the combined water volume of the two rivers gives the riverbed new dimensions. Over the course of a particle's lifetime, this merging process may occur more than once, altering the paths of numerous rivers in its

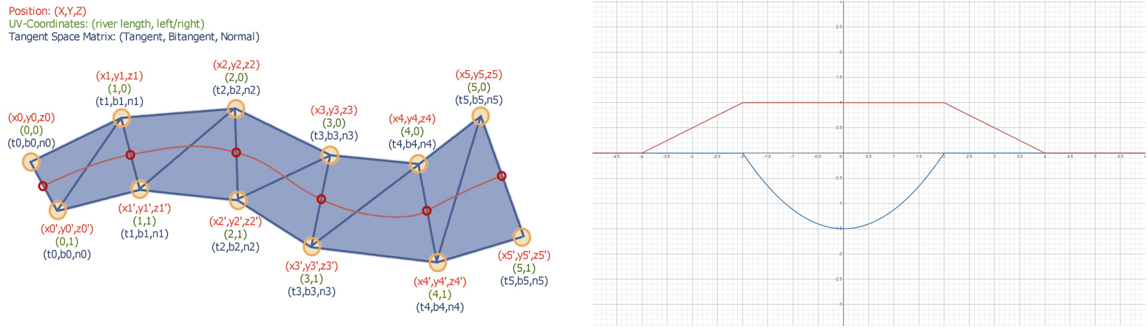


Figure 3: Left: Triangle strip with positions (red), texture coordinates (green), and tangents (red) that form a river triangle mesh. The guiding cubic spline is depicted in dark red. Right: Example of the riverbed function with depth-function in blue and blend-function in red.

path. Due to the course change of the river they have already merged with, such changes to already established river routes frequently result in previously valid rivers having a dry end. Each generated river is therefore verified once more to have its end connected to another water body after simulating all the river particles up to their termination. In the event that this validation is unsuccessful, the particle is reactivated and its simulation is carried out in order to reconnect the river to a legitimate end or to remove the invalid river object. If the reactivated particles have merged with new rivers or have been deleted, this validation process must be repeated.

The next step is to generate the geometry that can subsequently be used for rendering once all generated river objects have successfully undergone validation. A straightforward triangle strip is sufficient to depict the river since the rendering techniques listed in subsection subsection 3.2 do not use vertex offsets for the animation of the river surface. The algorithm first creates a cubic spline [Klu21] based on the accumulated position points and the width and depth values kept in the river object in order to provide flexibility for the number of subdivisions in this triangle strip. As a result, it is possible to improve the quality of the river geometry without being constrained by the resolution of the simulation in the previous step thanks to the spline's ability to sample river positions between data points while still maintaining the curvy nature of water currents. The generated vertices also require the normal, tangent, and bitangent in order for the normal texture to be applied correctly within the tangent space of the generated geometry because the resulting surface triangles, unlike the mesh generated for the lakes, are tilted in different directions. The fragment shader can determine whether a fragment is located in the center of the river or on one of its sides, which is crucial when animating the flow speed, as well as the distance of the fragment from the river's source, thanks to the addition of UV coordinates to the vertices. The structure of the resulting mesh is shown in figure Figure 3 on the left side. In order to simulate the natural erosion caused by river currents, the river generation algorithm's final step involves cutting the river bed into the terrain's height map. The algorithm does this by

using two functions depicted on the right side in figure Figure 3. The first function, which is highlighted in red, is the height of the riverbed with respect to the positions on the spline, and the second function represents the blend value between the original value of the terrain height map and the new value after the cut has been applied so that there are no sharp cuts at the edge of the river bed.

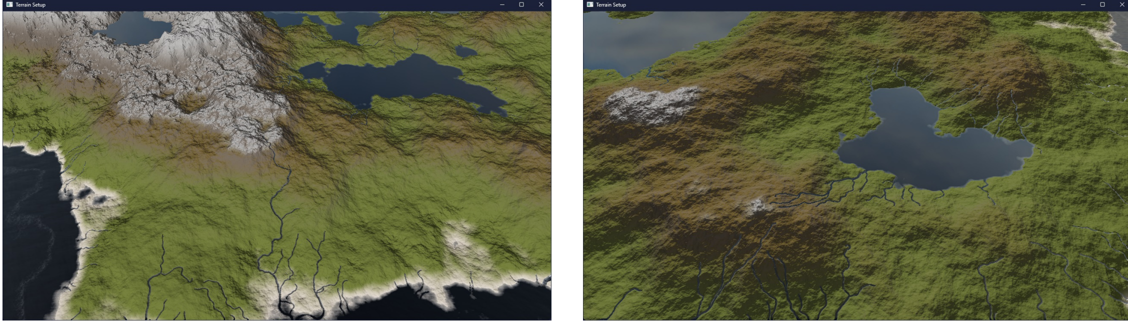


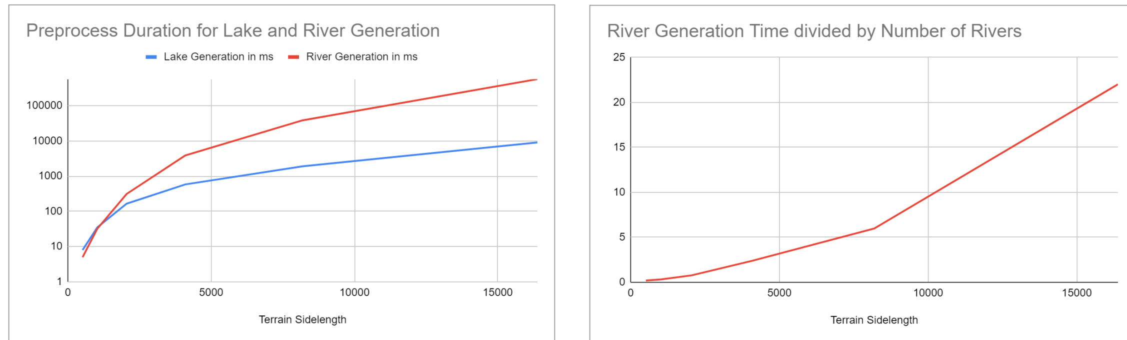
Figure 4: Renderings of some examples of the output from the procedural river generation.

4 Evaluation and Benchmark

One advantage of this method for creating rivers and lakes is that it can be used on any height-based terrain without requiring any additional knowledge about how the original terrain generation algorithm works. It may even work on height maps that have been specifically created or that have been derived from geographic data. Shorter generation times are achieved by the method's drastic reduction of the required particle count, but this also implies that results, particularly for rivers, vary greatly depending on the initial particle distribution. The result was calculated by averaging the paths of many thousands of particles, which would produce more consistent results when run multiple times on the same height map data. The original algorithm from [McD20] was still nondeterministic. Additionally, the algorithm minimizes terraforming by only changing the height map values near the paths of rivers, preserving the general shape of the terrain.

We implemented the presented techniques with C/C++ and OpenGL using the CrossForge graphics framework [UC23] as basis. The achieved hydrology-generation times in correlation to a landscape with increasing heightmap sizes are shown in figure 5 on a logarithmic scale. The Benchmark was performed by taking the average of multiple test runs on a system with the following specifications: Intel® Core™ i7-8700K Processor, 64GB of DDR4 Memory, and an NVIDIA GeForce RTX 3080 Ti graphics card. The generation times of the lake and river both exhibit exponential growth. There are several reasons for this. The area of the terrain doubles every time the heightmap's side lengths are doubled, increasing by a factor of four. Four times as many rivers and lakes must be created in order to maintain the same percentage of the land covered by water. Additionally, to match the larger landscape, the maximum generated lake size is increased, which raises the cost of the flood fill algorithm. The number of data points generated by rivers increases the cost of creating

and evaluating splines, and the size and number of rivers have a direct impact on how much riverbed carving is done. Rivers also tend to travel farther before joining with another water body, which can be observed in Figure 5, which shows that not only the number of rivers but also the generation time of a single river is dependent on the size of the height-map. Examples of generated hydrology can be seen in Figure 6 and Figure 4.



Terrain Sidelength	512	1,024	2,048	4,096	8,192	16,384
Max lake size	14,400	57,600	230,400	921,600	3,686,400	14,745,600
Lake Generation (ms)	8	35	165	581	1,895	9,019
Number of rivers	25	100	400	1,600	6,400	25,600
River Generation(ms)	5	32	311	3,844	38,309	563,630
Time per river (ms)	0.2	0.32	0.78	2.40	5.98	22.02

Figure 5: This figure shows the pre-processing times for lakes and rivers (top left) and the average generation times for individual rivers as the side length of the height map increases (top right). The table shows the respective times for common image resolutions.

5 Future Work

As a major performance improvement, the hydrology generation algorithm could be changed from its current single-threaded design to one in which the particle simulations are carried out simultaneously across multiple cores of modern CPUs, which would further reduce generation times. It might also be thought about moving some of the calculations to the GPU as the particle count approaches thousands. But this jump from sequential to parallel comes with its own share of new problems. When designing multithreaded processes, it is important to take into account the direct interactions between the generation algorithms of the water bodies, such as the merging of rivers when one of them is later determined to be invalid or when generating lakes, multiple simultaneous flood fill attempts from different corners of the same valley, where conflicts need to be resolved. According to the local topography, another modification to the hydrology generation would be to alter the shape of riverbeds. For example, erosion of rocky mountains typically results in thinner but deeper-cut riverbeds as opposed to wide but flat riverbeds that form in softer soil. But more details about a specific terrain would be needed, and those are not always available.

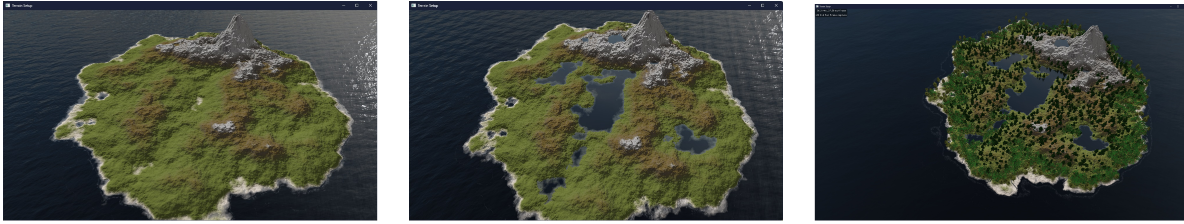


Figure 6: Procedurally generated landscape without foliage and hydrology (left), after lake generation (center), and after procedural foliage placement (right).

References

[Fli18] Jasper Flick. Waves, moving vertices. https://catlikecoding.com/unity/tutorials, 2018.

[Flü17] Fynn-Jorin Flügge. Realtime gpgpu fft ocean water simulation. Technical report, 2017.

[Hay10] Kyle Hayward. Animating water using flow maps, 2010.

[KBKŠ09] Peter Krištof, Bedrich Beneš, Jaroslav Křivánek, and Ondrej Št'ava. Hydraulic erosion using smoothed particle hydrodynamics. In *Computer Graphics Forum*, volume 28, pages 219–228. Wiley Online Library, 2009.

[Klu21] Tino Kluge. Cubic spline interpolation in c++, 2021.

[Let19] David Lettier. 3d game shaders for beginners, screen space reflection (ssr). https://lettier.github.io/3d-game-shaders-for-beginners/screen-space-reflection.html, 2019.

[McD20] Nicholas McDonald. Simple particle-based hydraulic erosion, 2020.

[Sou05] Tiago Sousa. Gpu gems 2, chapter 19. https://developer.nvidia.com/gpugems/gpugems2/part-ii-shading-lighting-and-shadows/chapter-19-generic-refraction-simulation, 2005.

[T+01] Jerry Tessendorf et al. Simulating ocean water. *Simulating nature: realistic and interactive techniques. SIGGRAPH*, 1(2):5, 2001.

[UC23] Tom Uhlmann and Contributors. Crossforge: A cross-platform 3d visualization and animation framework for research and education in computer graphics. https://github.com/Tachikoma87/CrossForge, 2020-2023.

[Wim15] Karl Wimble. Opengl water tutorials 7: Normal maps, 2015.