Integrating server-based simulations into web-based geo-applications

Pascal Bormann, Ralf Gutbell, Johannes Sebastian Mueller-Roemer

TU Darmstadt & Fraunhofer IGD, Germany

Abstract

In this work, we present a novel approach for combining fluid simulations running on a GPU server with terrain rendered by a web-based 3D GIS system. We introduce a hybrid rendering approach, combining server-side and client-side rendering, to interactively display the results of a shallow water simulation on client devices using web technology. To display water and terrain in unison, we utilize image merging based on depth values. We extend it to deal with numerical and compression artifacts as well as Level-of-detail rendering and use Depth Image Based Rendering to counteract network latency.

1. Introduction

Nowadays, the simulation of flood scenarios becomes increasingly important due to the rising number of weather phenomena, in particular intense rainfall. To aid in high water analysis and the creation of strategies for preemptive actions, these simulations should run interactively and be visualized in a 3D Geographic Information System (GIS) rendering terrain data. Interactive fluid simulations are possible using GPGPU (general-purpose computing on the GPU) approaches. This conflicts with the combined visualization of the simulation and terrain data, as GIS systems are more and more web-based and lightweight. While there has been valuable research in both areas - fluid simulation [BSA12] as well as geodata visualization [KG15] - to the best of our knowledge the two areas have seen no significant overlap. To alleviate this problem, we introduce an approach for combining fluid simulations and 3D GIS systems into a single web-application based on the smallest denominator of visual applications: pixels.

Our contributions are as follows: We present a novel hybrid-rendering approach for the combination of interactive server-based simulations with web-based rendering applications. This is achieved through a combination of server-side and client-side rendering. We demonstrate a showcase application that utilizes hybrid rendering to interactively display results of a GPU-based fluid simulation within a web-based 3D visualization of static terrain data. Our approach allows access to sophisticated simulations on ordinary client devices, utilizing the benefits of server-side rendering, while providing full interactivity through usage of Depth Image Based Rendering (DIBR) and perspectively correct integration of simulation data into the client visualization. We overcome a series of technical hurdles in this process:

- Correct image compositing based on depth values while dealing with numerical inaccuracies.
- Mixing Level-of-detail-based (LOD) rendering of terrain with a rendering of simulated data with full detail.
- Dynamic scaling of projected pixels to fill gaps between neighboring pixels after DIBR.

In Section 2, we cover the architecture of the system and introduce client and server implementations. Section 3 introduces the necessary algorithms for our image merging approach. Our results are evaluated in Section 4. Finally, Section 5 gives a conclusion and an outlook on future work.

2. Approach

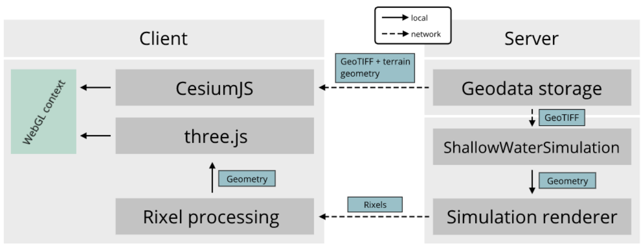

Figure 1: *Overview of the system architecture*

Our system is built upon a typical server-client architecture as depicted in Fig. 1. There are two server instances running in the cloud, a GPU instance running the shallow water simulation and a database that hosts GeoTIFF [RRG*00] files and corresponding terrain meshes. On the client side, two separate rendering systems are running in the same web-application: A globe-rendering-component built upon CesiumJS [CES] and a three.js-based [THR] rendering component responsible for rendering the data received from the simulation server. The workflow is as follows:

© 2019 The Author(s)
Eurographics Proceedings © 2019 The Eurographics Association.

delivered by EUROGRAPHICS DIGITAL LIBRARY www.eg.org diglib.eg.org

DOI: 10.2312/egs.20191012

1. The client application connects to the simulation server.
2. The client application loads a GeoTIFF file from the database and sends a message to the server instructing it to load the same GeoTIFF file.
3. The client application sends simulation-specific parameters, such as water height and the water starting area, to the simulation server.
4. The server starts running the simulation and periodically sends new data to the client, while simultaneously receiving camera matrices from the client.
5. The simulation data is continuously integrated into the client visualization using our image merging algorithm.

The next two sections first cover the server and client implementations before the image merging algorithm is explained in Section 3.

2.1. Simulation engine and data transmission

For the simulation server, a GPU-based shallow water solver was implemented based on the work introduced in [BSA12] and [VPM14] . We use a second order in time and space discretization on a regular grid, using an explicit, adaptive time step integration method. While the state vector and all intermediates are allocated at the full size of the domain, only a small percentage of the domain contains water at any point in time. As an optimization, only 16×16 subdomains that contain water, or are adjacent to cells that contain water, are considered in the simulation kernels. Combined with the fast, GPU-parallelized solver, this allows us to achieve simulation rates well beyond real time suitable for prediction.

While the shallow water simulation is implemented in CUDA [NVI18], the server-side hybrid rendering component is implemented in OpenGL. However, CUDA offers OpenGL interoperability functions, therefore the results of the simulation can be used for rendering without copying data.

For data transmission, we build on the rich pixels (*rixels*) approach of [ADSF15] and add several enhancements:

- Instead of transmitting positions (96 bit per rixel), we only transmit post-projection z (32 bit per rixel) and a visibility mask (1 bit per viewport pixel). Besides reducing message size, renderer fill-rate requirements are reduced, as depth testing is always required but the additional render target can be omitted.
- Instead of transmitting colors after color mapping (24 bit per rixel), we transmit 16 bit fixed point values along with a 32 bit floating point minimum and maximum for the entire domain. In addition to further reducing message size, this approach allows for zero latency client-side changes to color mapping.
- Mask computation, stream compaction (removal of "empty" viewport pixels), and minimum/maximum computation are performed on the GPU to reduce PCIe bandwidth and latency to a minimum using CUDA and Thrust [BH15]. The CPU only adds a message ID and WebSocket framing.

A detailed description and analysis of the simulation server and rixel enhancements are out of the scope of this paper.

2.2. Client rendering implementation

One of the main challenges with our approach is the correct visual integration of two different render-sources—a terrain model rendered by CesiumJS and a water area rendered remotely by the server—into a coherent and correct final image. The combination of both render-sources is motivated by the depth-merging approach presented in [GPCK16] and is extended to support a constant stream of portrayals. In its most basic form, our algorithm utilizes depth values in image space to merge the renderings of terrain and water body. Given two color/depth image pairs $\mathcal{I}_1, \mathcal{D}_1$ and $\mathcal{I}_2, \mathcal{D}_2$ taken from the same vantage point, a merged image satisfying all occlusions can be computed using regular depth-testing:

$$\mathcal{I}_{\text{final}}(x,y) = \begin{cases} \mathcal{I}_1(x,y) & : \mathcal{D}_1(x,y) < \mathcal{D}_2(x,y) \\ \mathcal{I}_2(x,y) & : \mathcal{D}_1(x,y) \geq \mathcal{D}_2(x,y) \end{cases} \tag{1}$$

This so called "sort-last" method was historically used for parallelizing graphical computation [Mol91, BS01], we use the approach to mix distributed rendering setups. Implementation-wise, this assumes that both image pairs were generated using the same view and projection matrices, so that pixels with the same depth value coincide in world space. To guarantee that the merging process yields correct images, it is imperative that the CesiumJS client renderer and the simulation server use an identical terrain model. We guarantee this by generating terrain models directly from the GeoTIFF files upfront. However, some challenges remain that are explained in Section 3.3.

Instead of directly merging each newly received frame on the client, we use a DIBR algorithm to enable interactive viewing of the simulated water body. As mentioned in Section 2.1, the simulation data for each frame is sent as a set of rixels, with empty rixels being discarded, which can be reconstructed into $\mathcal{D}$, the depth image, as well as an additional image containing simulation-specific values, such as velocity or water height. For each pixel, its position in normalized device coordinates is reconstructed by combining its image-space position and its depth value in image space and then applying an inverse viewport transformation as illustrated in Eq. (2).

$$\vec{v}_{NDC_s} = \begin{bmatrix} 2*((x+0.5)/screenwidth) - 1 \\ 2*((y+0.5)/screenheight) - 1 \\ 2*\mathcal{D}(x,y) - 1 \end{bmatrix} \tag{2}$$

The converted positions are then uploaded to the GPU as a point-cloud geometry. Inside the vertex shader, the points are reprojected using the following DIBR equation, adapted from [MB16]:

$$\vec{v}_t = M_{P_t} * M_{V_t} * M_{V_s}^{-1} * M_{P_s}^{-1} * \vec{v}_{NDC_s} \tag{3}$$

The simulation values are uploaded as vertex attributes, which are colored using a color lookup table. The resulting image of this DIBR process is then merged with the terrain image rendered by CesiumJS into the final client-side visualization (see Fig. 2).

3. Image merging

The introduction of DIBR complicates the trivial depth merging process as it was introduced in Section 2.2. The usage of DIBR in our implementation can be seen as a means for geometry compression: Geometry is compressed on the server by the process of rendering and sent to the client as a color/depth image pair. It

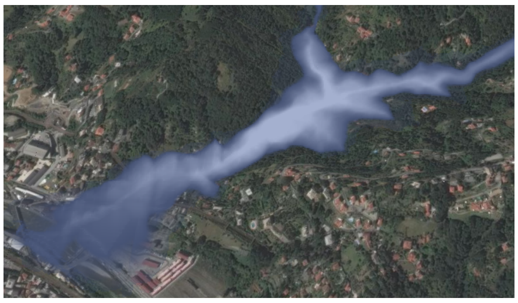

Figure 2: *Water body integrated into terrain rendering*

is then decompressed to yield geometry again, which can be rendered interactively. For a given vertex $\vec{v}$ on the server, the following equation—an extension to Eq. (3)—defines the chain of transformations applied to $\vec{v}$ resulting in a position in normalized device coordinates on the client device:

$$\vec{v}_t = \underbrace{M_{P_t} * M_{V_t} * M_{V_s}^{-1} * M_{P_s}^{-1} * vt^{-1}}_{\text{client-side DIBR}} \circ \underbrace{vt \circ M_{P_s} * M_{V_s} * \vec{v}}_{\text{server-side rendering}} \quad (4)$$

The application of this equation introduces several sources of numerical errors, which can lead to incorrect depth merging:

- The viewport transformation vt quantizes continuous normalized device coordinates into discrete pixel coordinates, applying vt^{-1} thus can yield incorrect positions.
- Depth precision is limited and non-uniform, leading to increasing quantization errors as camera distance increases.
- Calculation of inverse matrices as well as long multiplication chains exhibit numerical instabilities.

Apart from these numerical errors, depth merging may also fail due to the way terrain is rendered in 3D web-GIS applications. To enable rendering of large scale terrain in realtime, it is often rendered with Level-of-detail (LOD), which simplifies the terrain geometry the further away from the camera it is displayed. Our water simulation does not employ LOD but instead always operates on the most detailed terrain geometry level. This leads to a mismatch between server and client topologies and consequently to incorrect depth merging (see Fig. 5). The next three sections discuss our contributions to solve these depth merging errors.

3.1. Dealing with viewport quantization

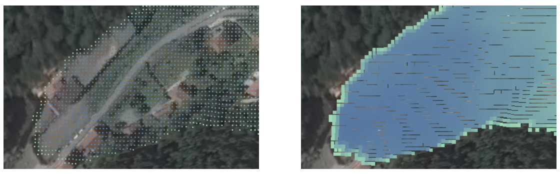

Figure 3: *Holes due to DIBR (left) fixed with adaptive point scaling (right)*

Viewport quantization introduces a maximum error of half the pixel size in world space. This error only becomes visible when applying DIBR to magnify the source geometry. Given the DIBR algorithm of Eq. (3), magnifying the source geometry introduces information gaps in the visualization [MB16]. As a result of these gaps, the viewport quantization errors become less noticeable due to a lack of visual cohesion in the geometry (Fig. 3, left). Scaling points dynamically based on the ratio between source viewspace distance and reprojected viewspace distance fixes most holes while also covering up the quantization errors (Fig. 3, right).

3.2. Dealing with depth imprecision

Depth buffer precision is a common problem in computer graphics, which is particularly relevant when rendering scenes covering very large areas, like in our use case. Adjusting of the clipping planes is not effective as the scenes in 3D GIS applications easily span dozens of kilometers. On these large scales, having less than 32 bit of precision in the depth buffer results in noticeable artifacts. Standard techniques for precision improvement, such as inverting the Z-values or using a logarithmic depth buffer, cannot be realized in our setting, both due to missing APIs in WebGL [Mar11] and because the DIBR Equation (3) requires linear depth values. Instead, as a general optimization to reduce numerical errors, we utilize the two-step vertex transform approach by Upchurch and Desbrun [UD12] when applying Eq. (3). We found that, given 32 bit depth values, the visualizations in our scenario work decently well, especially in conjunction with the depth probing algorithm of Section 3.3.

3.3. Dealing with terrain LOD

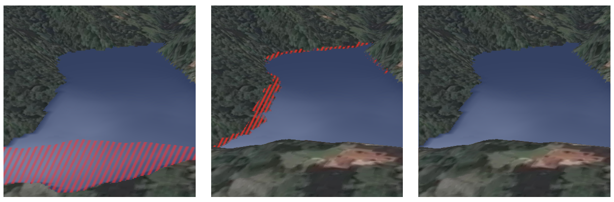

Figure 4: *No depth test (left, falsely visible regions in red), depth test using Eq. (1) (middle, falsely clipped regions in red), depth test using depth probing (right)*

Correctly integrating water and terrain is crucial for our use case, as occlusions enable correct localization of the water body within the terrain (compare Fig. 4, left and middle). In the presence of LOD, there is no trivially correct approach for this integration. When a coarse-grained LOD terrain segment occludes the fine-grained simulation geometry (like in Fig. 5, bottom left), clipping the simulated geometry is logically correct but does not yield satisfactory visual results. Users can get confused by the water body suddenly disappearing when zooming out. As such, any approach that fixes this issue will trade correctness for visual quality and will be prone to false positives—showing water where it should actually be occluded. Switching off LOD altogether is not feasible due to performance considerations. For the use-case presented herein, we found the following depth probing approach to work sufficiently well to solve incorrect, LOD-induced occlusion.

Let $\mathcal{I}_1, \mathcal{D}_1$ be the terrain image pair and $\mathcal{I}_2, \mathcal{D}_2$ the water image pair. For each pixel (x,y), if $\mathcal{D}_2(x,y) \leq \mathcal{D}_1(x,y)$, let $\mathcal{I}_2(x,y)$ be visible. Otherwise, let z_{v_i} be the viewspace distance of $\mathcal{D}_i(x,y)$.

Set $\mathcal{I}_1(x,y)$ visible only if $(z_{v_2} - z_{v_1}) \geq \mathcal{T}$, where $\mathcal{T}$ is an empirical threshold value set to 10 units in our implementation. Fig. 5 (right) illustrates this algorithm visually, Fig. 4 shows the result of the algorithm as compared to the trivial depth merging approach.

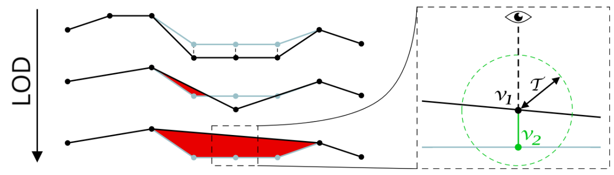

Figure 5: *Faulty depth merging due to terrain LOD (left, terrain in black, water in blue, wrong occlusion in red) solved with depth probing (right)*

4. Results

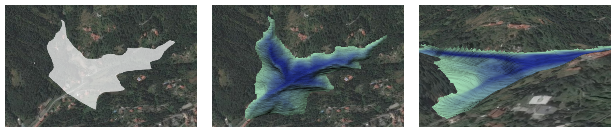

Figure 6: *Overview of the combined application. From left to right: Selection of water origin; water spread after some time; integration of water body into terrain*

Fig. 6 gives an overview of an exemplary use of our system. The combined visualization of terrain data and simulated water makes the simulation results easy to understand and interact with. Through the usage of DIBR, interactive viewing of the water spread within the browser is possible. A demonstration of the system is shown in the supplementary video. A limiting factor of our system is the size of the transmitted rixels. The larger the visible area of the water body, the more rixels have to be transmitted, which reduces the perceived update rate of the simulation on the client depending on the network throughput. On less capable client devices, we found that the processing of the rixel data — decoding and uploading to the GPU — can also reduce the interactivity of the system.

5. Conclusion and future work

We presented an architecture and reference implementation which combines a web-based 3D-GIS application with a server-based fluid simulation. By utilizing the rich-pixels approach of [ADSF15], combining the concepts of [GPCK16] and [Mol91] and applying a DIBR algorithm, the simulation data can be displayed interactively within the browser. We illustrated problems with the depth merging process that is required to combine images generated through client-side and server-side rendering and proposed different mechanisms to mitigate these problems. We believe the approach illustrated in this paper is not limited to fluid simulations and extensible to other computational expensive domains. In the future, we aim to integrate different types of simulations using the proposed architecture to enable easy access to sophisticated simulations from within the browser. The limitations of this approach are those shared by most systems that use a form of server-side rendering, namely latency and throughput problems. We aim to analyze how predicting camera motion in combination with depth buffer compression can help mitigate the visualization delay introduced by these issues. Additionally, we want to further reduce the effects of LOD on the depth merge by utilizing screen-space error metrics. If the simulation is extended to support fields other than water height and velocity, the JIT-accelerated streaming queries introduced in [MA16] could be used to efficiently compute derived fields on the GPU before hybrid rendering.

References

[ADSF15] Altenhofen C., Dietrich A., Stork A., Fellner D.: Rixels: Towards secure interactive 3d graphics in engineering clouds. *The IPSI BgD Transactions on Internet Research 31* (2015). 2, 4

[BH15] Bell N., Hoberock J.: Thrust 1.8.1, 2015. URL: https://thrust.github.io/. 2

[BS01] Bartz D., Silva C.: Rendering and Visualization in Parallel Environments. In *Eurographics 2001 - Tutorials* (2001), Eurographics Association. doi:10.2312/egt.20011052. 2

[BSA12] Brodtkorb A. R., Sætra M. L., Altinakar M.: Efficient shallow water simulations on GPUs: Implementation, visualization, verification, and validation. *Computers & Fluids 55* (2012), 1–12. doi:10.1016/j.compfluid.2011.10.012. 1, 2

[CES] Cesium - WebGL Virtual Globe and Map Engine. https://cesiumjs.org. 1

[GPCK16] Gutbell R., Pandikow L., Coors V., Kammeyer Y.: A framework for server side rendering using ogc's 3d portrayal service. In *Proceedings of the 21st International Conference on Web3D Technology* (2016), ACM, pp. 137–146. 2, 4

[KG15] Krämer M., Gutbell R.: A case study on 3d geospatial applications in the web using state-of-the-art webgl frameworks. In *Proceedings of the 20th International Conference on 3D Web Technology* (2015), ACM, pp. 189–197. 1

[MA16] Mueller-Roemer J. S., Altenhofen C.: JIT-compilation for interactive scientific visualization. In *Short Papers Proceedings: 24th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision* (2016), WSCG '16, pp. 197–206. 4

[Mar11] Marrin C.: Webgl specification. *Khronos WebGL Working Group* (2011). 3

[MB16] Meder J., Brüderlin B.: Fast depth image based rendering for synthetic frame extrapolation. *Journal of Theoretical and Applied Computer Science 10*, 3 (2016), 3–18. 2, 3

[Mol91] Molnar S.: Combining z-buffer engines for higher-speed rendering. In *Proceedings of the Third Eurographics Conference on Advances in Computer Graphics Hardware* (Aire-la-Ville, Switzerland, Switzerland, 1991), EGGH'88, Eurographics Association, pp. 171–182. URL: http://dx.doi.org/10.2312/EGGH/EGGH88/171-182, doi:10.2312/EGGH/EGGH88/171-182. 2, 4

[NVI18] NVIDIA Corporation: *CUDA C Programming Guide*. Manual PG-02829-001_v10.0, 2018. URL: https://docs.nvidia.com/pdf/CUDA_C_Programming_Guide.pdf. 2

[RRG*00] Ritter N., Ruth M., Grissom B. B., Galang G., Haller J., Stephenson G., Covington S., Nagy T., Moyers J., Stickley J., et al.: Geotiff format specification geotiff revision 1.0. *SPOT Image Corp* (2000). 1

[THR] three.js - Javascript 3D library. https://threejs.org/. 1

[UD12] Upchurch P., Desbrun M.: Tightening the precision of perspective rendering. *Journal of Graphics Tools 16*, 1 (2012), 40–56. 3

[VPM14] Vacondio R., Palù A. D., Mignosa P.: GPU-enhanced finite volume shallow water solver for fast flood simulations. *Environmental Modelling & Software 57* (2014), 60–75. doi:10.1016/j.envsoft.2014.02.003. 2