

Integrating Real-Time Fluid Simulation with a Voxel Engine

Johanne Zadick¹ · Benjamin Kenwright¹ ·
Kenny Mitchell¹

Received: 22 September 2015 / Accepted: 29 June 2016 / Published online: 29 July 2016
© The Author(s) 2016. This article is published with open access at Springerlink.com

Abstract We present a method of adding sophisticated physical simulations to voxel-based games such as the hugely popular *Minecraft* (2012. <http://minecraft.gamepedia.com/Liquid>), thus providing a dynamic and realistic fluid simulation in a voxel environment. An assessment of existing simulators and voxel engines is investigated, and an efficient real-time method to integrate optimized fluid simulations with voxel-based rasterisation on graphics hardware is demonstrated. We compare graphics processing unit (GPU) computer processing for a well-known incompressible fluid advection method with recent results on geometry shader-based voxel rendering. The rendering of visibility-culled voxels from fluid simulation results stored intermediately in CPU memory is compared with a novel, entirely GPU-resident algorithm.

Keywords Voxels · Fluid · Geometry shader · Real-time · Video games · Volume · Graphical processing unit (GPU)

1 Introduction

Voxel-based games such as *Minecraft* (2012) have become increasingly popular. However, the fluid in such games is usually simulated and rendered in simplistic fashion, and is seldom characterised by realistic dynamics. Fluids such as water are programmed in several different ways in voxel engines. The most commonly used method in voxel-based engines is cellular automaton, which consists of prescribed operations on a regular grid of cells. Each cell can adopt a finite number of states, and rules define the behavior of the fluid. The characteristics of these rules provide

✉ Kenny Mitchell
k.mitchell2@napier.ac.uk

¹ Edinburgh Napier University, Room C48, Merchiston Campus, 10 Colinton Road, Edinburgh EH10 5DT, UK

the major differences between such game simulations, but these are often very simple and not based on mathematical models, and consequently, the fluid does not appear to react realistically in many cases. For example, in *Minecraft*, water can be observed to spread in a single direction without any obstacles (Fig. 1).

By contrast, a dynamic fluid simulation fully rendered with voxels can provide an interesting source of engagement in video games and movies. The *Lego Movie* (©Warner Bros.) (Fig. 2) is a good example, with many compelling voxel-based ocean scenes.

However, dynamic fluid simulations are expensive, as a large amount of volumetric information must be updated for every frame, ideally in real-time. Furthermore, voxel engines are rather resource-intensive, due to the number of voxels that need to be displayed for an entire scene. Thus, the concepts of dynamic fluid and voxel engine are hard to merge. This challenge is somewhat alleviated by recent results in both GPU-accelerated fluid simulation (Crane et al. 2007 and Harris 2004) and voxel-based rendering, but to our knowledge, no work has been reported on the particular hurdles of integrating these efficiently.

We therefore propose an efficient integration of GPU voxel-based rendering (Miller et al. 2014) with a well-known fluid simulation advection, based on the Navier–Stokes differential equations (Stam 2003).

2 Related Work

2.1 Voxel-Based Rasterisation

There exist several online tutorials regarding the development of voxel engines, and the main challenges that arise when attempting efficient rendering. A major consideration is the platform technology upon which these are built, which is conducive to GPU fluid simulation. For example, an open source OpenGL fluid

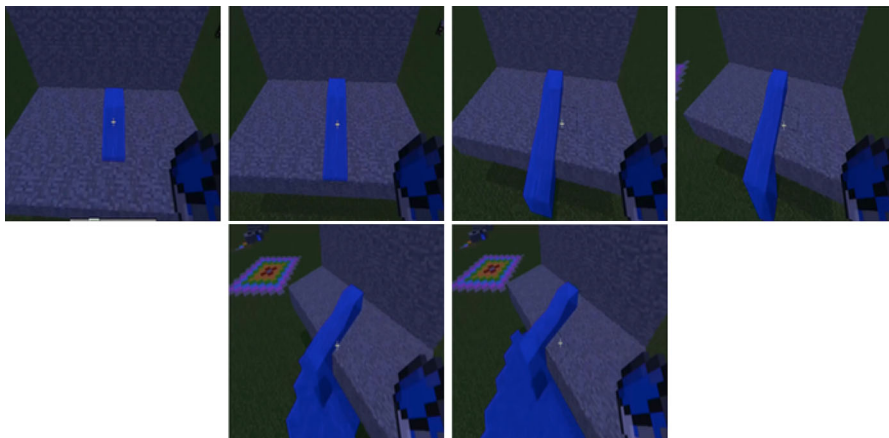


Fig. 1 Unrealistic behavior of water in *Minecraft*



Fig. 2 Sample image of a voxel-based ocean in *The Lego Movie* ©Warner Bros

simulation is not compatible with a Direct3D voxel-based rendering engine, and cannot be employed without considerable reworking of the code.

Barrett's (2015) Obbg engine was developed to illustrate the use of a highly efficient voxel render library, and it illustrates large *Minecraft*-styled voxel landscapes that are implemented with OpenGL. With an OpenGL voxel engine, CUDA, OpenCL, and OpenGL fluid simulations are valid candidates for integration. However, many of the optimizations applied in this framework are applied for large static open world rendering, and have proved unsuitable for our dynamic voxel rendering requirements, in which each voxel may change from frame to frame.

The Poxels rendering framework (Miller et al. 2014), implemented in Direct3D, focuses on rendering sparse 3D voxel environments with data amplification using Geometry Shaders on GPU. This approach reduces the CPU-to-GPU bandwidth costs of updating the volumetric grid data, and makes it more suitable for dynamic voxel-based environment rendering (Fig. 3). Miller et al. (2014) further justify the

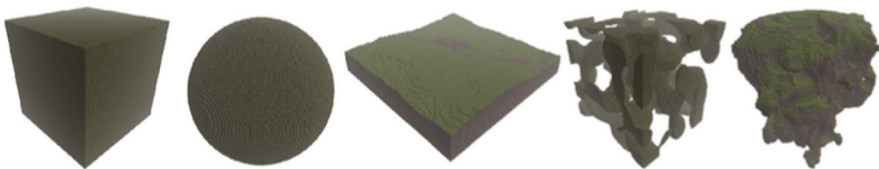


Fig. 3 Examples of the Poxels (Miller et al. 2014) sparse voxel-based rendering engine, which employs GPU data amplification suitable for efficient dynamic voxel environments

GPU rasterisation approach (also used in *Minecraft* 2012), as opposed to ray-casting oriented approaches.

2.2 Real-Time Fluid Simulation

The Liquid Voxels project (Cabello 2015) developed using Unity 3D allows more realistic fluid behaviors in a voxel-based engine. It uses C# CPU-resident 3D arrays to store fluid information, and a cellular automaton method to define the behavior. However, in our method, since we are working on GPU, we apply 3D textures to store the fluid data.

Stam's (2003) description of fast fluid simulation for games covers the simulation principles and details highly efficient GPU computation methods with which to apply them in 2D. Karoly's (2012) work details an open source real time fluid implementation and control method which is derived from Stam's work and which performs well in OpenGL; however, this was incompatible with our choice of the Direct3D voxel rendering method. Vlietinck (2009) uses a similar system, which has been expanded to support 3-dimensional fluid simulations. This work is demonstrated with a fire and smoke simulation (Fig. 4), but equally, we can show the method being applied to water. Since Vlietinck made use of DirectCompute for GPU compute advection, pressure acceleration and correction processing, this very much suits compatibility with our choice of a Direct3D voxel rendering engine.

In order to provide a brief background and the context of our Navier–Stokes processing framework, we define a number of processing stages:

- the advection of the velocity field of the fluid;
- the acceleration caused by the pressure in the fluid;
- the diffusion of the momentum resulting from the resistance of the fluid; and,
- external forces of static or dynamic bodies interacting with the fluid.

To apply the Navier–Stokes equations, it is necessary to perform three computations to update the velocity at each time step: advection, diffusion, and force application. We can then compute the pressure and subtract the pressure gradient. We want our simulation to realize those calculations on the GPU.

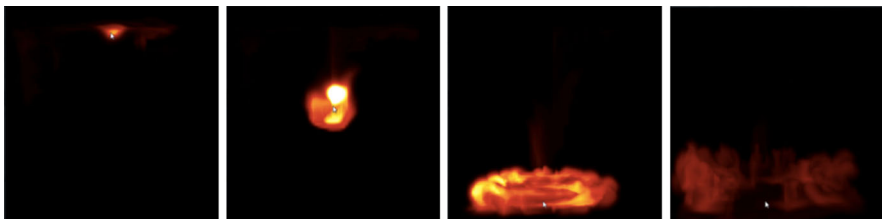


Fig. 4 Vlietinck's simulation (Vlietinck 2009)

3 Method and Implementation

In the GPU compute fluid simulation process, several textures are associated with the compute shaders that can be either be the input or the output. Textures are defined for the speed, pressure and divergence. In detail, there are three velocity textures and two pressure textures, which deal with the temporary values that must be stored during each stage of the process. Those textures are composed of single precision float 4 types. The final result of the five GPU compute stages is output in a 3D speed grid texture filled with single precision floats. This texture provides the 3D source fluid field for transfer into the voxel-based rendering engine.

In the Poxels voxel engine (Miller et al. 2014), it is on the CPU that we must define which voxels of the grid are active (to be displayed). Therefore, to retrieve the result in the initial approach, we read the speed grid texture back on the CPU. Since a texture passed to the polygonal voxel-generating Geometry Shader cannot be read directly from the CPU, we create a staging texture in which we transmit the data from the resulting speed grid texture into the GPU memory. This is performed by mapping the staging texture in CPU memory, and copying the result from our 3D float array. It is important that the size used for the fluid simulation is the same as the size of the 3D voxel staging texture, and that the CPU memory buffer has an appropriate memory address alignment. Once the values in the float array are on the CPU, we can process them in the voxel engine. We include a color argument to the voxel that highlights the spread of the fluid concentration through the simulation. For the purpose of a locally contained fluid simulation within the limits of in-core GPU memory constraints, we define the concept of a volumetric in the class, ‘Chunk’. In this CPU approach, we implement a method in the Chunk class, which links between the speed map array that we have retrieved from the fluid simulation, and the input voxel array to be displayed. This approach activates a voxel if the corresponding value in the dense 3D array is not zero. To deal with the

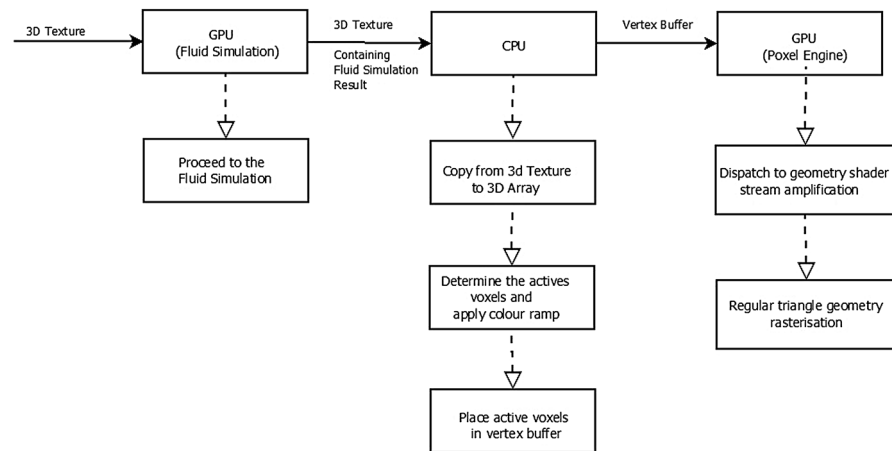


Fig. 5 Flow of memory data between the GPU and the CPU

concentration, we produce a color ramp by applying a function that takes a float between 0 and 1 and returns a RGB color that is then passed to the geometry shader of the Poxel simulation. In a game environment, either textures or an adapted colours range would give a more realistic result. The Poxel-rendering approach (Miller et al. 2014) follows 2 main steps: (1) dispatch to geometry shader stream amplification with stream-out to the cached vertex buffer; followed by (2) regular triangle geometry rasterisation with the generated vertex buffer.

The flow of memory data between the GPU and the CPU for each frame is illustrated in Fig. 5.

4 Results

4.1 System Specifications

To generate the results shown in this paper, we ran the simulation on an Asus N751JX-T4180H. It has a GeForce GTX 950M video card with 2.0 GB of dedicated video RAM. The CPU is an Intel(R) Core(TM) i7-4720HQ with 2.6 GHz Dual-Core 64-bit. The OS is Microsoft Windows 10 Famille, 64-bit, with 8 GB of RAM.

4.2 Performances

In our CPU approach, the time used to copy the data back to the CPU is more significant than the time needed for the all other steps, and so this issue had to be addressed (Fig. 6; Tables 1, 2).

For the previous screenshots (Fig. 7) we inject fluid every 400 frames at the top left corner of the chunk and then apply a downward force. A full loop, from the

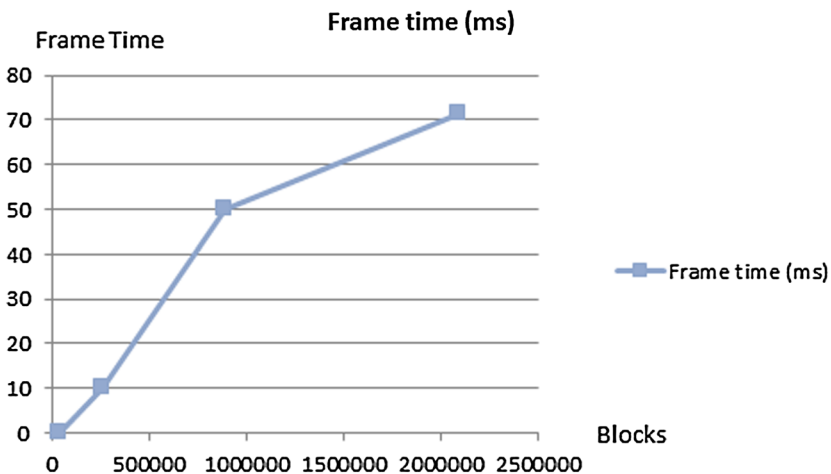


Fig. 6 Graph of frame time (ms) of combined voxel fluid simulation and rendering showing linear scaling and the number of voxels

Table 1 FPS and frame time on NVIDIA GeForce GTX950M with different simulation chunk sizes

Chunk size	Blocks	FPS	Frame time (ms)
$32 \times 32 \times 32$	32,768	401	2.494
$64 \times 64 \times 64$	262,144	100	10.000
$96 \times 96 \times 96$	884,736	20	50.000
$128 \times 128 \times 128$	2,097,152	14	71.429

Table 2 Major steps—execution times on $64 \times 64 \times 64$ chunk

Step	Execution time (ns)
Advect	5986
Backward advect and pre correct	1710
Second order correction	34,637
Draw source	855
Calculate speed divergence	1282
Jaccobi	14,539
Project	855
Copy in staging texture	2993
Copy from staging texture to CPU	23,088,969
Activate voxels and render	976,276

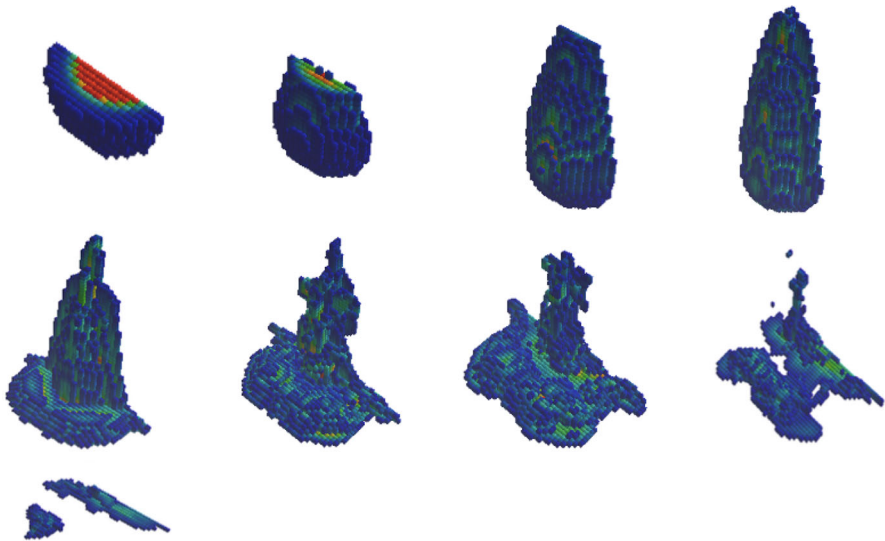


Fig. 7 Results of simulation running on $64 \times 64 \times 64$ chunk every 20 frames

appearance of the source to the disappearance of the last voxel last 180 frames. This duration will change based on the chunk size, the source position and the force applied.

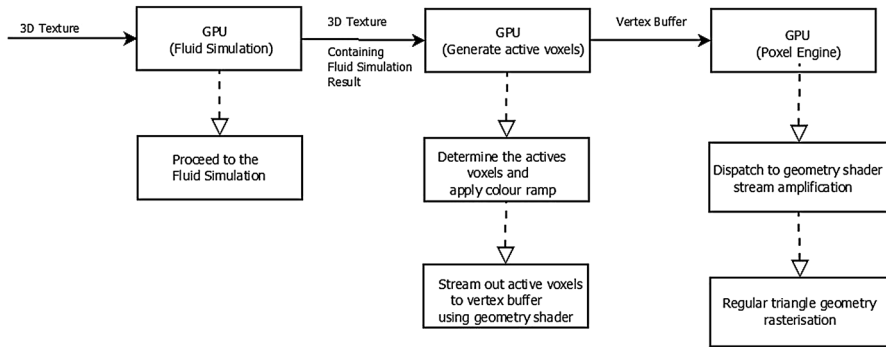


Fig. 8 Flow of memory data between the fluid simulation and the Poxel engine

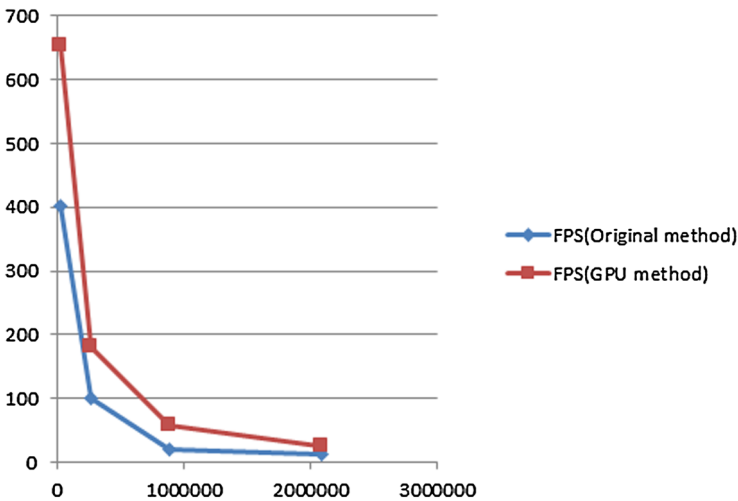


Fig. 9 Graph of FPS with GPU-only method (red) and original method (blue)

4.3 GPU Optimization

In order to improve performance, we needed to avoid copying back the data to the CPU. To prevent this from happening, it was necessary to move some of the Poxel voxel engine CPU's functionality to the geometry shader. As a result, the voxels are no longer activated by the CPU, since it is in the geometry shader that we use to determine which voxels should be rendered. Instead, in the fluid simulation compute shader, we add another texture (wherein only values that do not equal zero are placed), and we send that texture to the geometry shader. We then continuously output vertex data from the geometry-shader stage using stream-out, which guarantees the deterministic ordering of rendered primitives. The color ramp is then applied directly in the geometry shader (Fig. 8).

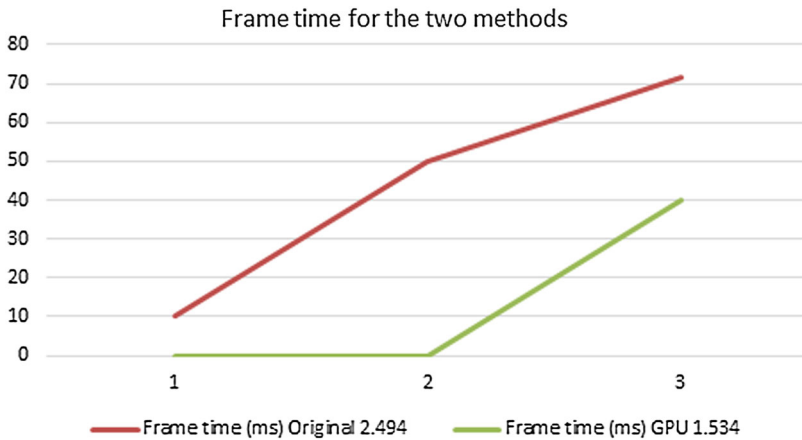


Fig. 10 Graph of frame time with GPU-only method (*green*) and original method (*red*)

Table 3 FPS and frame time on NVIDIA GeForce GT540M with different simulation chunk sizes (GPU approach)

Chunk size	Blocks	FPS	Frame time (ms)
32 × 32 × 32	32,768	652	1.534
64 × 64 × 64	262,144	180	5.556
96 × 96 × 96	884,736	58	17.241
128 × 128 × 128	2,097,152	25	40.000

Using that optimization, we have observed a huge improvement in performance, going from 30 fps for a 64 × 64 × 64 chunk, up to 1100 fps (Figs. 8, 9, 10; Table 3).

5 Conclusion

This method can be used to generate realistic fluid behaviors in voxel engines. It could be used for water, other fluids, fire, and smoke. This would result in more dynamic voxel works. However, several further optimizations would have to be made to improve performance and efficiency. The fluid simulation that we have described in this paper does is not programmed to detect or respond to obstacles. An extension of this work would be to add obstacles. In that way, the currently active voxels of the chunk would define the position of the obstacles. An interesting point would also be to deal with the way in which different fluids would interact with each other, e.g. water and lava. Likewise, the reaction between fluid-based voxels and a different kind of voxel would need to be tested further. For example, fire or lava can ignite wood. Moreover, the simulation is only based on a single chunk, and it needs to be extended to several chunks. A solution would be to associate each chunk with a fluid texture, and to add the source, based on the position and concentration of fluid at the borders of the other chunks.

Acknowledgments The authors would like to thank the reviewers, David Sinclair for his help, and Babis Koniaris for his coding expertise and the valuable assistance he has given us on the GPU-part of the project. This work was supported by the Erasmus programme.

Open Access This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

References

- Barett, S. (2015). Obbg—Open block building game. <https://github.com/nothings/obbg>
- Cabello, R. (2015). Liquid voxels. <http://forum.unity3d.com/threads/liquid-voxels.242821/>
- Crane, K., Llamas, I., & Tariq, S. (2007). Real-time simulation and rendering of 3D fluids. In H. Nguyen (Ed.), *Gpu Gems* (chap. 30) (Vol. 3, pp. 633–676). Addison-Wesley.
- Harris, M. J. (2004). Fast fluid dynamics simulation on the GPU. In R. Fernando (Ed.), *GPU Gems: Programming techniques, tips and tricks for real-time graphics* (chap. 38) (Vol. 1, pp. 637–665). Addison-Wesley.
- Karoly, Z. (2012). Real time fluid simulation and control using the Navier–Stokes equations. A thesis submitted in partial fulfilment of the Requirements of Budapest University of Technology and Economics for the M.Sc. in Computer Engineering.
- Miller, M., Cumming, A., Chalmers, K., Kenwright, B. & Mitchell, K. (2014). Poxels: Polygonal voxel environment rendering. In: *Proceedings of the 20th ACM symposium on virtual reality software and technology*. Edinburgh, Scotland, 2014.
- Minecraft Wiki. (2012). Liquid. Available from <http://minecraft.gamepedia.com/Liquid>
- Stam, J. (2003). Real-time fluid dynamics for games. In: *The game developer conference*. San Jose CA, 2002.
- Vlietinck, J. (2009). Fluid simulation (DX11/DirectCompute). <http://users.skynet.be/fquake/>