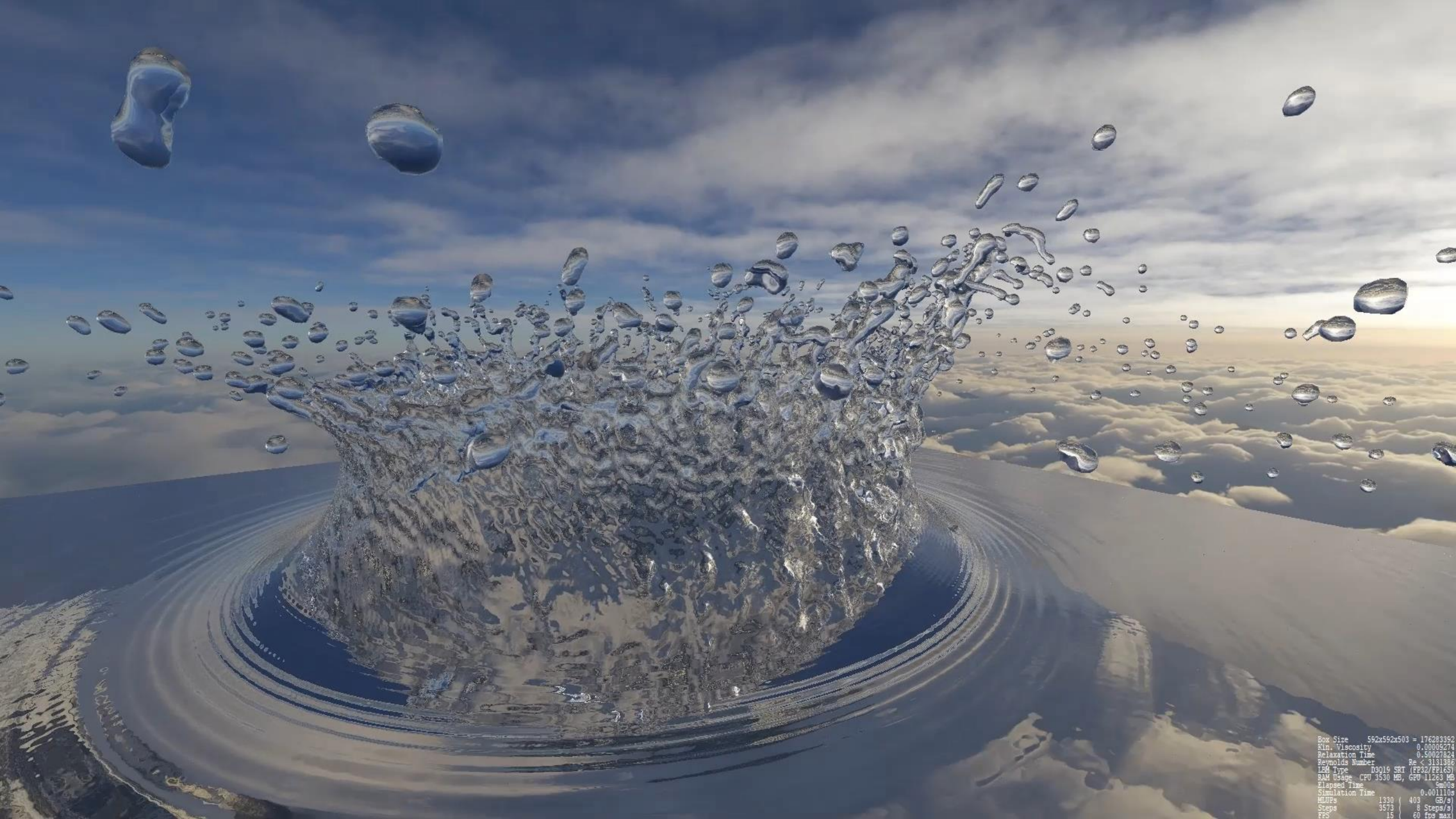
Combined scientific CFD simulation and interactive raytracing with OpenCL

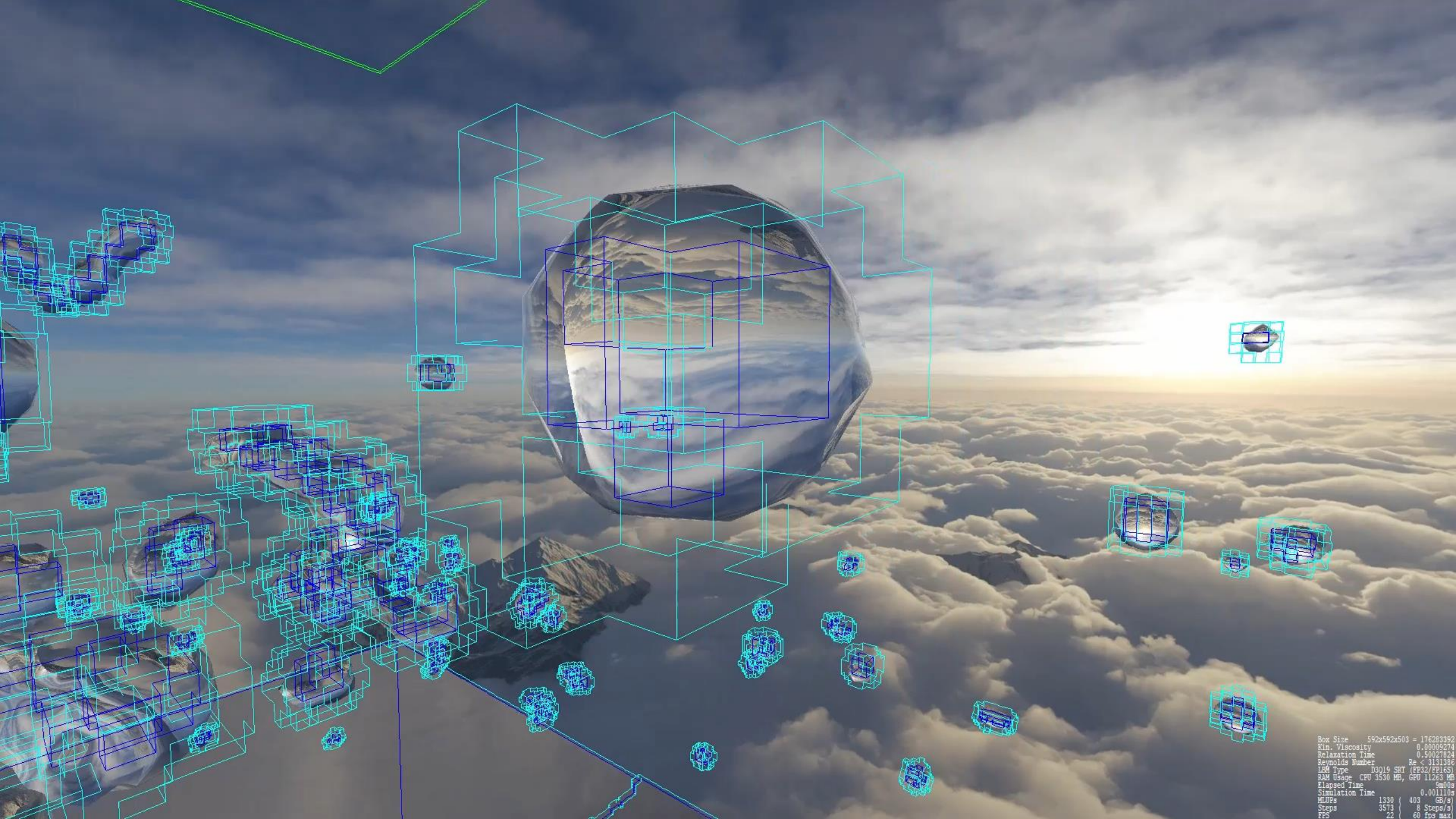
The FluidX3D Software

Moritz Lehmann, M.Sc.
moritz.lehmann@uni-bayreuth.de | @ProjectPhysX

SFB MIKROPLASTIK DFG Elitenetzwerk Bayern Biological Physics UNIVERSITÄT BAYREUTH


Box Size 96x352x96 = 3244032
Kin. Viscosity 0.00700000
Relaxation Time 0.52100004
Reynolds Number Re < 7918
LBM Type D3Q19 SRT (FP32/FP16S)
RAM Usage CPU 64 MB, GPU 207 MB
Elapsed Time 1m14s
Simulation Time 35525s
MLUPs 1605 (486 GB/s)
Steps 35525 (495 Steps/s)
FPS 84 (60 fps max)

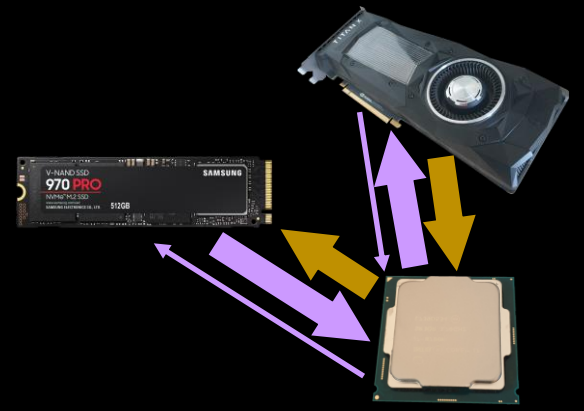

Box Size 592x592x503 = 176283392
Kin. Viscosity 0.00005274
Relaxation Time 0.50027824
Reynolds Number Re < 3131386
LBM Type D3Q19 SRT (FP32/FP16S)
RAM Usage CPU 3530 MB, GPU 11263 MB
Elapsed Time 5m00s
Simulation Time 0.001110s
MLUPs 1330 (403 GB/s)
Steps 3573 (8 Steps/s)
FPS 15 (60 fps max)


Box Size 592x592x503 = 176283392
Kin. Viscosity 0.00009274
Relaxation Time 0.50027824
Reynolds Number Re < 3131386
LBM Type D3Q19 SRT (FP32/FP16S)
RAM Usage CPU 3530 MB, GPU 11263 MB
Elapsed Time 9m00s
Simulation Time 0.001110s
MLUPs 1330 (403 GB/s)
Steps 3573 (8 Steps/s)
FPS 22 (60 fps max)

Both Simulation and Graphics with OpenCL

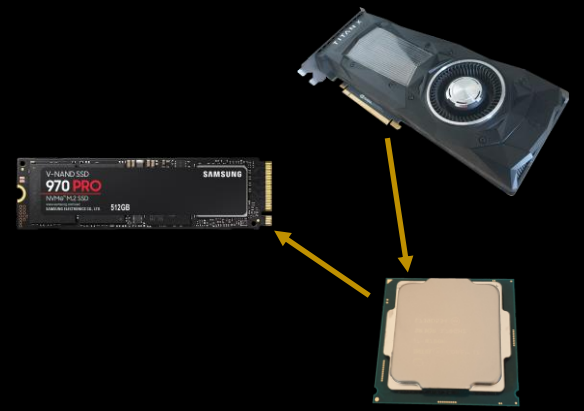
- traditionally
  - run simulation on GPU
  - store volumetric frames on hard disk
  - render with external software

    --> simulation would be fast, but IO takes forever



- much better alternative
  - run simulation on GPU
  - use GPU to render images from raw data in VRAM
  - use IO only for rendered images

    --> fast enough for interactive graphics



Moritz Lehmann, M.Sc.
moritz.lehmann@uni-bayreuth.de | @ProjectPhysX

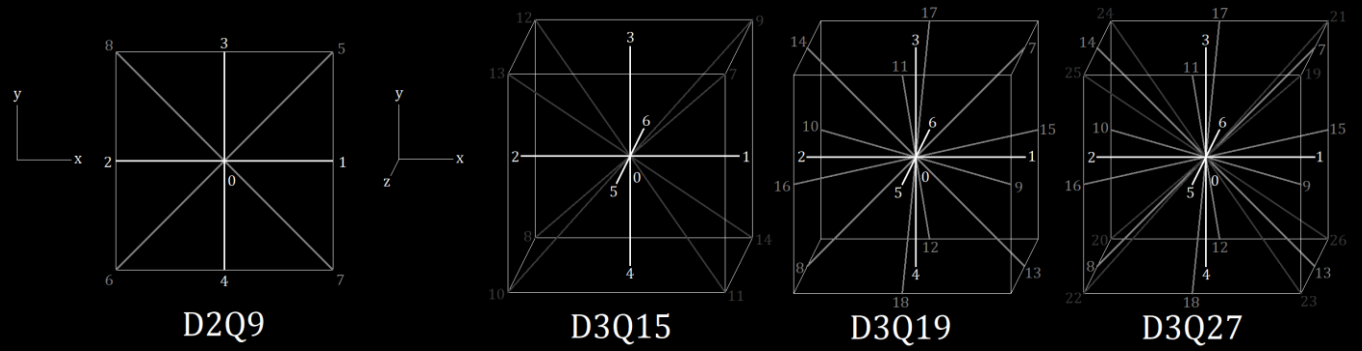
SFB MIKROPLASTIK
DFG
Elitenetzwerk Bayern
Biological Physics
UNIVERSITÄT BAYREUTH

Why OpenCL™?

- OpenCL 1.2 runs on any device since ~2012
  - gaming GPUs
  - „professional“ GPUs
  - datacenter GPUs
  - Xeon Phi
  - CPUs
  - my phone
- lots of math/vector functionality already built-in
- 100% efficiency on some GPUs (A100/V100, GeForce Ampere/Turing)

Physically Accurate CFD Simulation

- lattice Boltzmann method



- to reduce memory demand + increase performance
  - in-place streaming with implicit bounce-back
  - optional FP32/FP16 mixed precision for memory compression
    --> together: 55 Bytes/node (LBM), 67 Bytes/node (FSLBM)

- Free Surface LBM
  - Volume-of-Fluid model
  - surface tension via analytic PLIC
  - improved mass conservation

arXiv:2112.08926v2 [physics.comp-ph] 31 Jan 2022

On the accuracy and performance of the lattice Boltzmann method with 64-bit, 32-bit and novel 16-bit number formats

Moritz Lehmann[1*], Mathias J. Krause[2], Giorgio Amati[3], Marcello Sega[4], Jens Harting[4,5] and Stephan Gekle[1]

February 1, 2022

*Correspondence: moritz.lehmann@uni-bayreuth.de
[1]Biofluid Simulation and Modeling – Theoretische Physik VI, University of Bayreuth
[2]Institute of Mechanical Process Engineering and Mechanics, Karlsruhe Institute of Technology
[3]SCAI – SuperComputing Applications and Innovation Department, CINECA
[4]Helmholtz Institute Erlangen-Nürnberg for Renewable Energy, Forschungszentrum Jülich
[5]Department of Chemical and Biological Engineering and Department of Physics, Friedrich-Alexander-Universität Erlangen-Nürnberg

Abstract

Fluid dynamics simulations with the lattice Boltzmann method (LBM) are very memory-intensive. Alongside reduction in memory footprint, significant performance benefits can be achieved by using FP32 (single) precision compared to FP64 (double) precision, especially on GPUs. Here, we evaluate the possibility to use even FP16 and Posit16 (half) precision for storing fluid populations, while still carrying arithmetic operations in FP32. For this, we first show that the commonly occurring number range in the LBM is a lot smaller than the FP16 number range. Based on this observation, we develop novel 16-bit formats – based on a modified IEEE-754 and on a modified Posit standard – that are specifically tailored to the needs of the LBM. We then carry out an in-depth characterization of LBM accuracy for six different test systems with increasing complexity: Poiseuille flow, Taylor-Green vortices, Karman vortex streets, lid-driven cavity, a microcapsule in shear flow (utilizing the immersed-boundary method) and finally the impact of a raindrop (based on a Volume-of-Fluid approach). We find that the difference in accuracy between FP64 and FP32 is negligible in almost all cases, and that for a large number of cases even 16-bit is sufficient. Finally, we provide a detailed performance analysis of all precision levels on a large number of hardware microarchitectures and show that significant speedup is achieved with mixed FP32/16-bit.

Keywords: LBM, floating-point, FP16, Posit, mixed precision, customized precision, GPU, OpenCL

1 Introduction

The lattice Boltzmann method (LBM) [1–4] is a powerful tool to simulate fluid flow. The parallel nature of the underlying algorithm has led to (multi-)GPU implementations [5–56] becoming a popular choice as speedup can be up to two orders of magnitude compared to CPUs at similar power consumption. However, most GPUs have only poor FP64 (double precision) arithmetic capabilities and thus the vast majority of GPU codes has been implemented in FP32 (single precision), while most CPU codes are written in FP64. This difference, and in particular whether FP32 is sufficient for LBM simulations compared to FP64, has been a point of persistent discussion within the LBM community [13–18, 28–33, 49–55, 57–60]. Nevertheless, only few papers [17, 32, 33, 49, 57, 61] provide some comparison on how floating-point formats affect the accuracy of the LBM and mostly find only insignificant differences between FP64 and FP32 except at very low velocity and where floating-point round-off leads to spontaneous symmetry breaking. Besides the question of accuracy, a quantitative performance comparison across different hardware microarchitectures is missing as the vast majority of LBM software is either written only for CPUs [62–74] or only for Nvidia GPUs [27–53] or CPUs and Nvidia GPUs [16–26].

A second point of concern has been the amount of video memory on GPUs, which is in general smaller than standard memory on CPU systems and can thus lead to restrictions in domain size. LBM solely works on density distribution functions (DDFs) f_i (also called fluid populations) – floating-point numbers [75–78] – that

Moritz Lehmann, M.Sc.
moritz.lehmann@uni-bayreuth.de | @ProjectPhysX

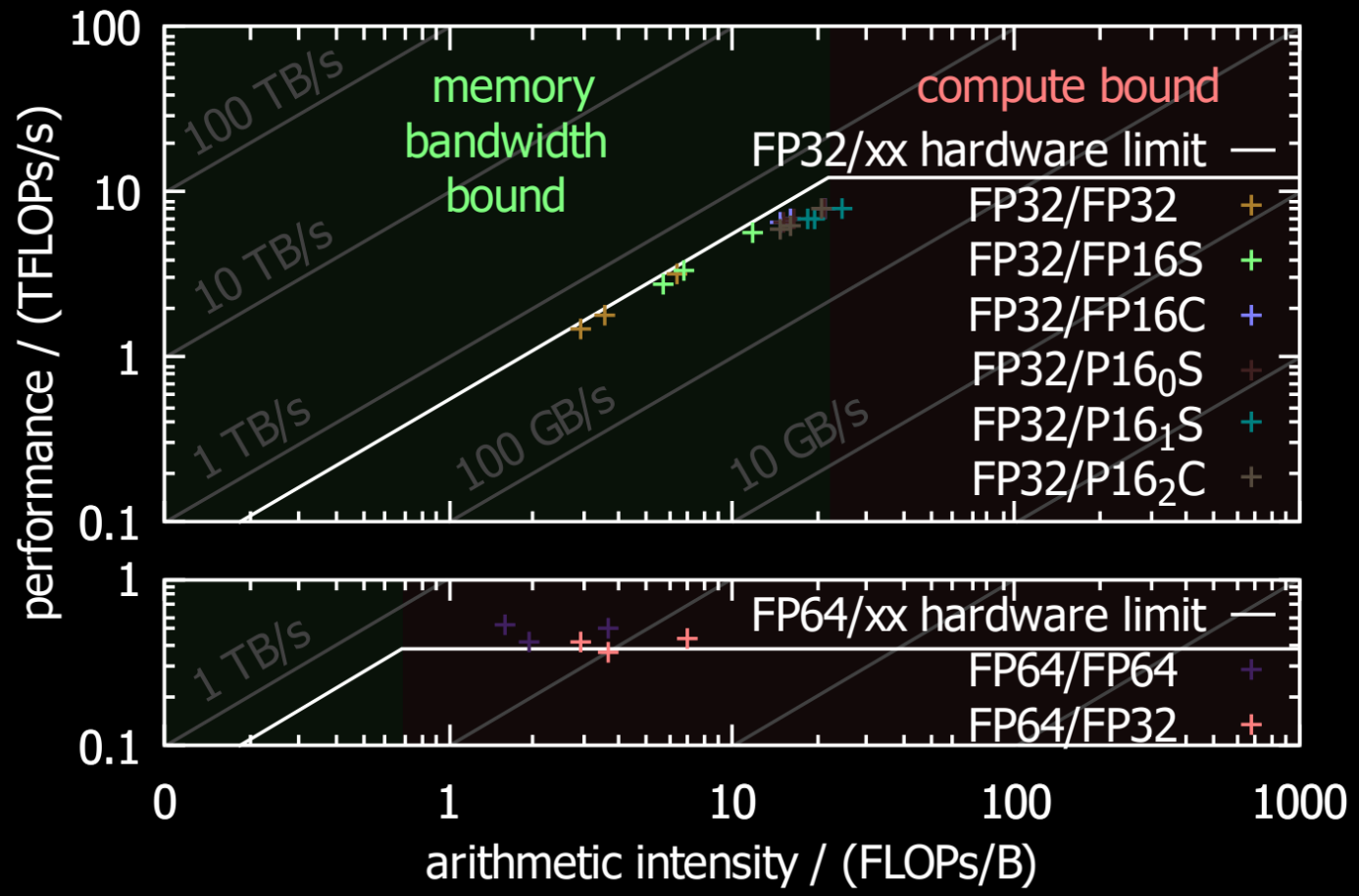
SFB MIKROPLASTIK DFG Elitenetzwerk Bayern Biological Physics UNIVERSITÄT BAYREUTH

Simulation Efficiency – Roofline Model

- Nvidia Titan Xp
- D3Q19, SRT/TRT/MRT (left to right)
- one-step-pull streaming
- FLOPs+IOPs counted combined

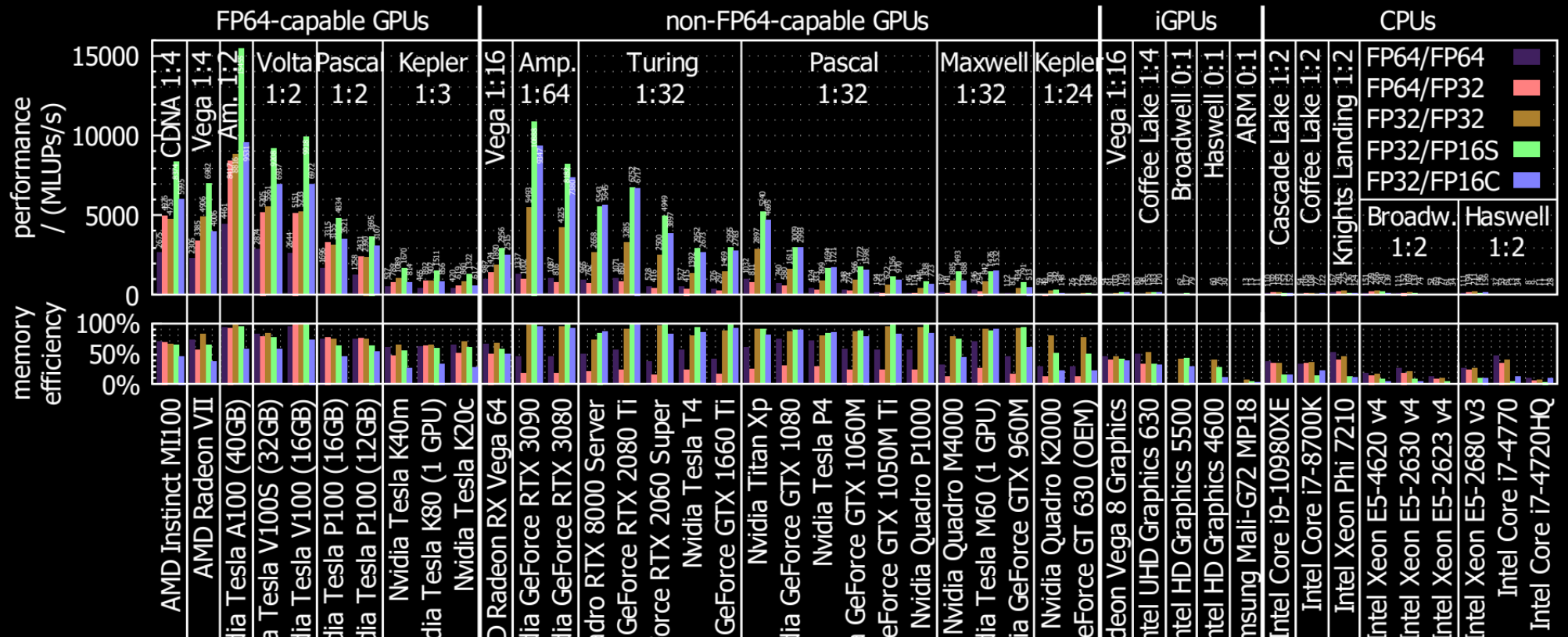
- mixed precision:
  arithmetic/memory

- memory precision formats
  - FP16S: shifted-range IEEE FP16
  - FP16C: custom FP16 format
  - P16: 16-bit Posit formats



Moritz Lehmann, M.Sc.
moritz.lehmann@uni-bayreuth.de | @ProjectPhysX

Simulation Efficiency – Hardware Survey



Moritz Lehmann, M.Sc.
moritz.lehmann@uni-bayreuth.de | @ProjectPhysX

SFB MIKROPLASTIK DFG Elitenetzwerk Bayern Biological Physics UNIVERSITÄT BAYREUTH

Application: Microplastic Transfer

Research Article | Open Access | Published: 12 November 2021

Ejection of marine microplastics by raindrops: a computational and experimental study

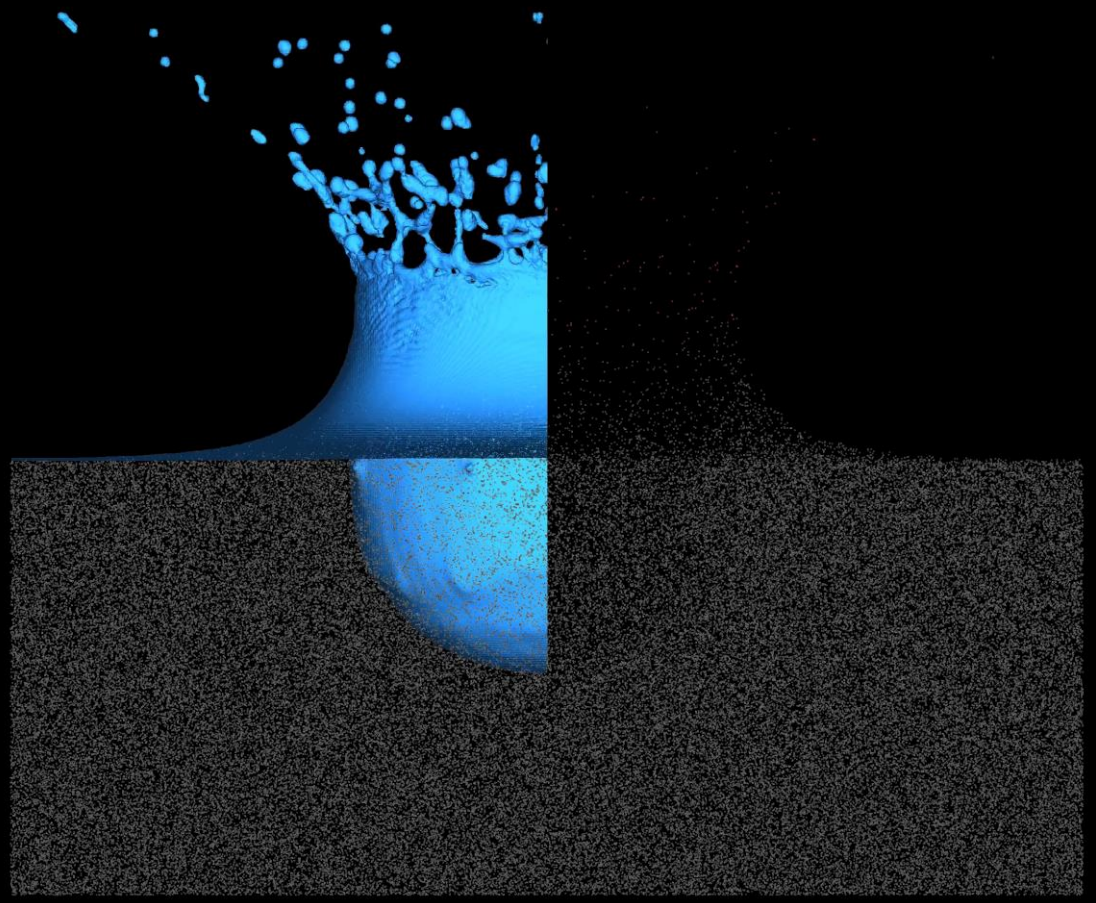
Moritz Lehmann, Lisa Marie Oehlschlägel, Fabian P. Häusl, Andreas Held & Stephan Gekle

Microplastics and Nanoplastics **1**, Article number: 18 (2021) | Cite this article

2153 Accesses | **2** Citations | **78** Altmetric | Metrics

Abstract

Raindrops impacting water surfaces such as lakes or oceans produce myriads of tiny droplets which are ejected into the atmosphere at very high speeds. Here we combine computer simulations and experimental measurements to investigate whether these droplets can serve as transport vehicles for the transition of microplastic particles with diameters of a few tens of μm from ocean water to the atmosphere. Using the Volume-of-Fluid lattice Boltzmann method, extended by the immersed-boundary method, we performed more than 1600 raindrop impact simulations and provide a detailed statistical analysis on the ejected droplets. Using typical sizes and velocities of real-world raindrops – parameter ranges that are very challenging for 3D simulations – we simulate straight impacts with various raindrop diameters as well as oblique impacts. We find that a 4mm diameter raindrop impact on average ejects more than 167 droplets. We show that these droplets indeed contain microplastic concentrations similar to the ocean water within a few millimeters below the surface. To further assess the plausibility of our simulation results, we conduct a series of laboratory experiments, where we find that microplastic particles are indeed contained in the spray. Based on our results and known data – assuming an average microplastic particle concentration of 2.9 particles per liter at the ocean surface – we estimate that, during rainfall, about 4800 microplastic particles transition into the atmosphere per square kilometer per hour for a typical rain rate of $10\frac{\mathrm{mm}}{\mathrm{h}}$ and vertical updraft velocity of $0.5\frac{\mathrm{m}}{\mathrm{s}}$.



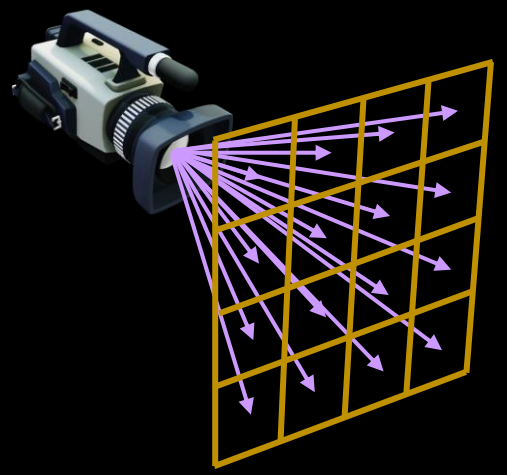
Moritz Lehmann, M.Sc.
moritz.lehmann@uni-bayreuth.de | @ProjectPhysX

SFB MIKROPLASTIK DFG Elitenetzwerk Bayern Biological Physics UNIVERSITÄT BAYREUTH

Graphics with OpenCL

- image = int array
- OpenCL can write to an array, so it can render; no OpenGL/Vulkan required

- rasterization
  - points/lines/triangles/circles… --> Bresenham
- raytracing
  - generate rays from camera
  - ray-triangle/rhombus intersection --> Möller-Trumbore
  - reflection/refraction --> Snell‘s law
  - UV-mapping for skybox



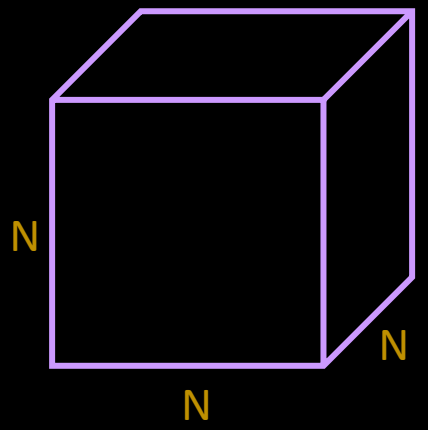
- ~700 lines for OpenCL rendering engine, plus ~300 lines for fluid rendering kernels

  - transfer rendered frames to CPU
    - draw on screen / write to hard drive

Moritz Lehmann, M.Sc.
moritz.lehmann@uni-bayreuth.de | @ProjectPhysX

Runtime Scaling - LBM vs. Rasterizazion vs. Raytracing

- LBM: ~N^3
- trangles of isosurface: ~N^2
- surface rasterization: ~N^2



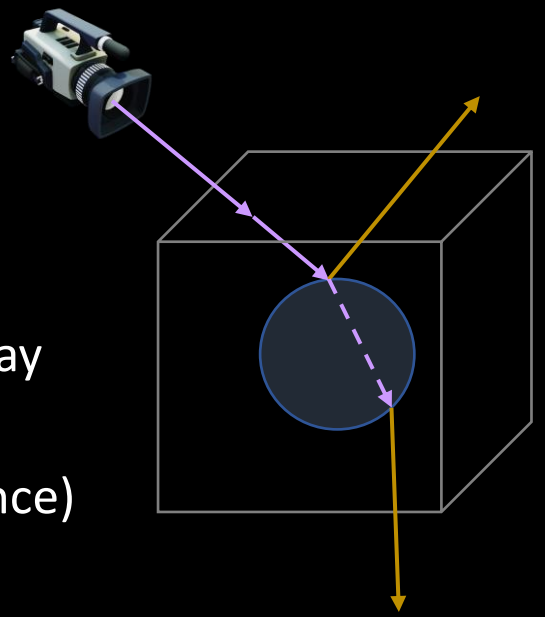
- surface raytracing - BVH: ~log(N), but generating the BVH tree is ~N^2
- surface raytracing - ray-grid traversal: ~N

--> benefit: runs on any hardware at peak performance, not just RTX/DXR

Moritz Lehmann, M.Sc.
moritz.lehmann@uni-bayreuth.de | @ProjectPhysX

Raytracing Kernel

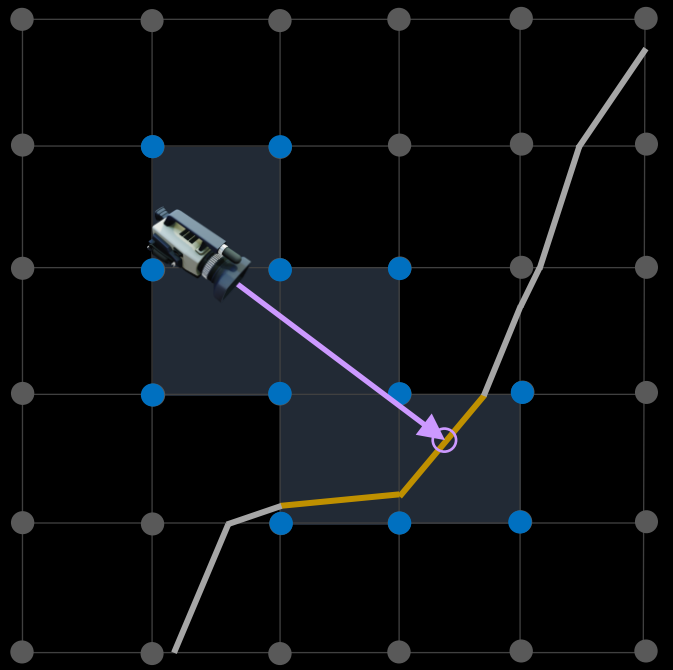
- for each pixel, shoot 1 ray out of the camera
- if camera is outside of grid, intersect with grid bounding box
- call ray-grid traversal on camera ray
  - no intersection --> skybox color of camera ray
  - intersection
    - reflect reflection ray & refract internal ray
    - call ray-grid traversal on internal ray & refract to refraction ray
    - get skybox colors of reflection ray & refraction ray
    - mix colors based on reflectivity (depends on angle of incidence)
    - optional: repeat with reflection and refraction ray



Moritz Lehmann, M.Sc.
moritz.lehmann@uni-bayreuth.de | @ProjectPhysX

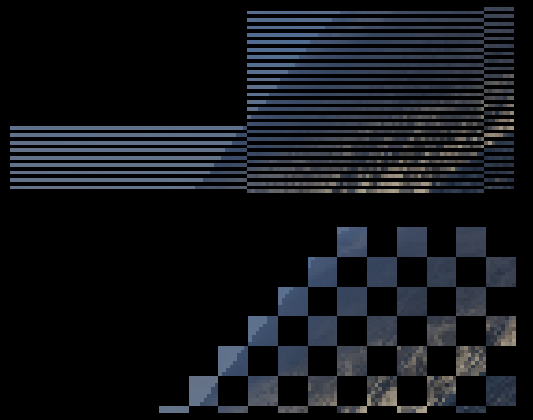
Ray-Grid Traversal

- idea: only consider grid cells through which a ray traverses
- for each traversed grid cell
  - check if cell contains isosurface
  - generate triangles with marching-cubes on-the-fly
  - check each of them for ray-triangle intersection
  - intersection
    - compute isosurface normal on all 8 grid points
    - interpolate surface normal at intersection point



Moritz Lehmann, M.Sc.
moritz.lehmann@uni-bayreuth.de | @ProjectPhysX

Ray-Grid Traversal - Optimizations

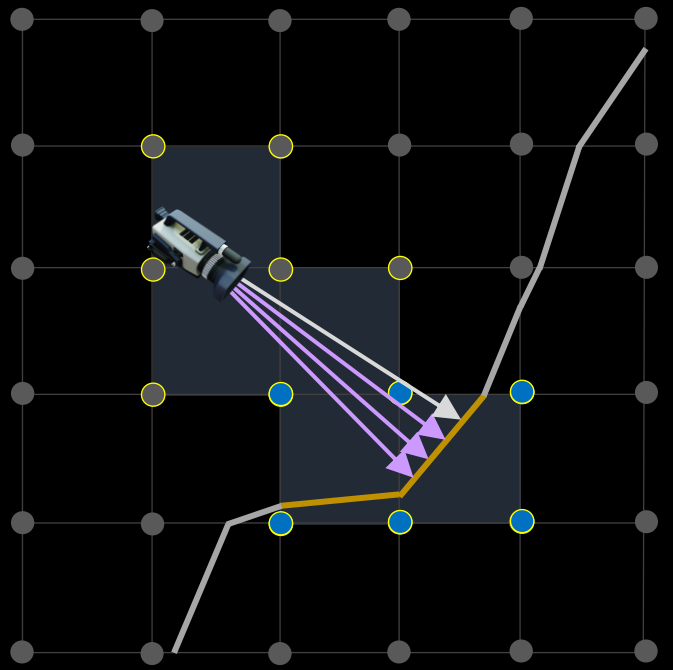


- workgroup alignment is critical
  - make workgrops not horizontal stripes of pixels,
  - but 8x8 pixel tiles (50% faster)

- while traversing the grid
  - instead of LBM isovalues (4 Byte), load LBM flags (1 Byte)
  - only if cell contains isosurface, load isovalues (15% faster)

- only load unknown isovalues when traversing into the next cell

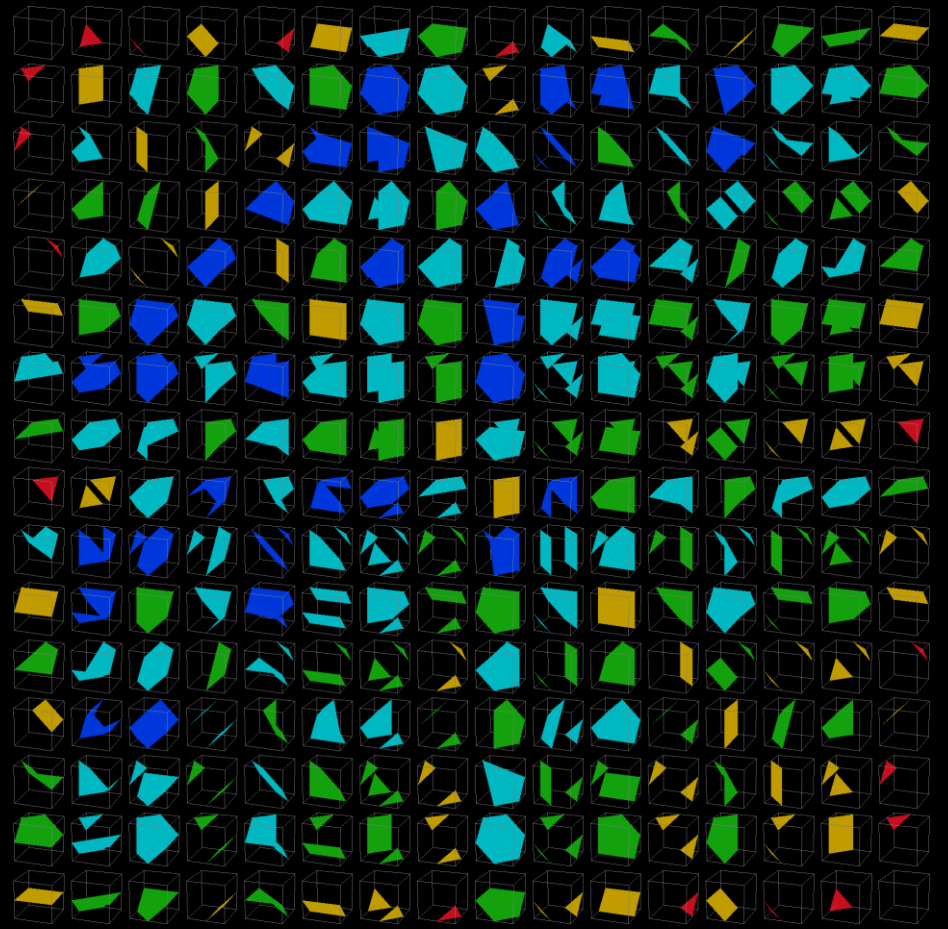
  --> breaks memory coalescence and leads to slowdown, despite less memory access



Moritz Lehmann, M.Sc.
moritz.lehmann@uni-bayreuth.de | @ProjectPhysX

Generating Triangles: Marching-Cubes

- implementation from Paul Bourke
  - 0-5 triangles/cell
- optimizations
  - reduce table size to 1/8
    - edge table: mirror symmetry + ushort, 1024B -> 256B
    - triangle table: bit-packing + uchar, 16384B -> 1920B
  - compute cubeindex branchless (bit-shifting)
- interesting:
    - edge table is not required and causes branching, but leads to ~30% speedup
    - tables must be in “constant” OpenCL memory space



Moritz Lehmann, M.Sc.
moritz.lehmann@uni-bayreuth.de | @ProjectPhysX

Generating Triangles: Marching-Cubes

- implementation from Paul Bourke
  - 0-5 triangles/cell
- optimizations
  - reduce table size to 1/8
    - edge table: mirror symmetry + ushort, 1024B -> 256B
    - triangle table: bit-packing + uchar, 16384B -> 1920B
  - compute cubeindex branchless (bit-shifting)
- interesting:
  - edge table is not required and causes branching, but leads to ~30% speedup
  - tables must be in “constant” OpenCL memory space

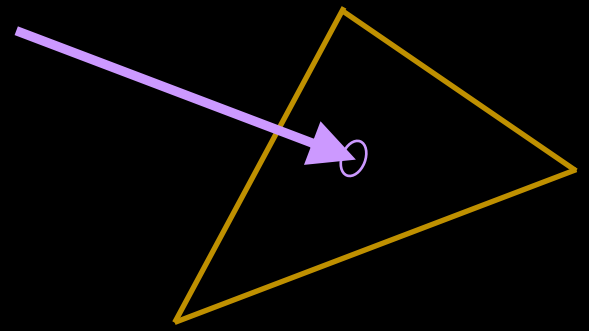
```
constant ushort edge_table_data[128] = { // // source: http://paulbourke.net/geometry/polygonise/, mirror symmetry applied, makes marching-cubes 31% faster
    0x000, 0x109, 0x203, 0x30A, 0x406, 0x50F, 0x605, 0x70C, 0x80C, 0x905, 0xA0F, 0xB06, 0xC0A, 0xD03, 0xE09, 0xF00,
    0x190, 0x099, 0x393, 0x29A, 0x596, 0x49F, 0x795, 0x69C, 0x99C, 0x895, 0xB9F, 0xA96, 0xD9A, 0xC93, 0xF99, 0xE90,
    0x230, 0x339, 0x033, 0x13A, 0x636, 0x73F, 0x435, 0x53C, 0xA3C, 0xB35, 0x83F, 0x936, 0xE3A, 0xF33, 0xC39, 0xD30,
    0x3A0, 0x2A9, 0x1A3, 0x0AA, 0x7A6, 0x6AF, 0x5A5, 0x4AC, 0xBAC, 0xAA5, 0x9AF, 0x8A6, 0xFAA, 0xEA3, 0xDA9, 0xCA0,
    0x460, 0x569, 0x663, 0x76A, 0x066, 0x16F, 0x265, 0x36C, 0xC6C, 0xD65, 0xE6F, 0xF66, 0x86A, 0x963, 0xA69, 0xB60,
    0x5F0, 0x4F9, 0x7F3, 0x6FA, 0x1F6, 0x0FF, 0x3F5, 0x2FC, 0xDFC, 0xCF5, 0xFFF, 0xEF6, 0x9FA, 0x8F3, 0xBF9, 0xAF0,
    0x650, 0x759, 0x453, 0x55A, 0x256, 0x35F, 0x055, 0x15C, 0xE5C, 0xF55, 0xC5F, 0xD56, 0x65A, 0x753, 0x459, 0x550,
    0x7C0, 0x6C9, 0x5C3, 0x4CA, 0x3C6, 0x2CF, 0x1C5, 0x0CC, 0xFCC, 0xEC5, 0xDCF, 0xCC6, 0x8CA, 0x9C3, 0xAC9, 0xBC0
};
constant uchar triangle_table_data[1920] = { // source: http://paulbourke.net/geometry/polygonise/, termination value 15, bit packed
    255,255,255,255,255,255,255,15, 15, 56,255,255,255,255,255,255, 16,249,255,255,255,255,255, 31, 56,137,241,255,255,255, 33,250,255,255,255,255,255, 41, 10,146,
    255,255,255,255,255, 47, 56,162,168,137,255,255,255,179,242,255,255,255,255,255, 15, 43,184,240,255,255,255,255,145, 32,179,255,255,255,255, 31, 43,145,155,184,255,255,255,163,177, 58,255,255,255,
    255, 15, 26,128,138,171,255,255,255,147, 48,155,171,249,255,255,159,168,138,251,255,255,255,255,116,248,255,255,255,255,255, 79, 3, 55,244,255,255,255,255, 16,137,116,255,255,255,255, 79,145,
    116,113, 19,255,255,255, 33,138,116,255,255,255,255,255, 63,110, 3, 20,162,255,255,255, 41,154, 32, 72,247,255,255, 47,154,146, 39, 55,151,244,255, 72, 55, 43,255,255,255,255,191,116, 43, 36, 64,
    255,255,255, 9,129,116, 50,251,255,255, 79,183, 73,155, 43, 41,241,255,163, 49,171,135,244,255,255, 31,171, 65, 27, 64,183,244,255,116,152,176,185,186, 48,255, 79,183,180,153,171,255,255,255,
    89,244,255,255,255,255,255,159, 69,128,243,255,255,255,255, 80, 20, 5,255,255,255,255,143, 69, 56, 53, 81,255,255,255, 33,154, 69,255,255,255,255, 63,128, 33, 74, 89,255,255,255, 37, 90, 36,
    4,242,255,255, 47, 90, 35, 53, 69, 67,248,255, 89, 36,179,255,255,255, 15, 43,128, 75, 89,255,255,255, 80, 4, 81, 50,251,255,255, 47, 81, 82, 48,184,132,246,255, 58,171, 49, 89,244,255,
    255, 79, 89,128,129, 26,184,250,255, 69, 80,176,181,186, 48,255, 95,132,133,170,184,255,255,255,121, 88,151,255,255,255,255,159, 3, 89, 83, 55,255,255,255,112, 8,113, 81,247,255,255, 31, 53,
    83,247,255,255,255,255,121,152,117, 26,242,255,255,175, 33, 89, 80, 3,117,243,255, 8,130, 82, 88,167, 37,255, 47, 90, 82, 51,117,255,255,255,151,117,152,179,242,255,255,159,117,121,146, 2,
    114,251,255, 50, 11,129,113, 24,117,255,191, 18, 27,119, 81,255,255,255, 89,136,117, 26,163,179,255, 95, 7, 5,121, 11, 1,186, 10,171,176, 48, 90,128,112,117,176, 90,183,245,255,255,255,255,
    106,245,255,255,255,255,255, 15, 56,165,246,255,255,255,255, 9, 81,106,255,255,255,255, 31, 56,145, 88,106,255,255,255, 97, 37, 22,255,255,255,255, 31, 86, 33, 54,128,255,255,255,105,149, 96,
    32,246,255,255, 95,137,133, 82, 98, 35,248,255, 50,171, 86,255,255,255,191,128, 43,160, 86,255,255,255, 16, 41,179,165,246,255,255, 95,106,145,146, 43,137,251,255, 54,107, 53, 21,243,255,
    255, 15,184,176, 5, 21,181,246,255,179, 6, 89, 96, 5,249,255,111,149,150,187,137,255,255,255,165, 70,135,255,255,255, 79, 3,116, 99,165,255,255,255,145, 80,186, 72,247,255,255,175, 86,
    145, 23, 55,151,244,255, 22, 98, 21,116,248,255,255, 31, 82, 37, 54, 64, 67,247,255, 72,151, 80, 96, 5, 98,255,127,147,151, 52,146,149, 38,150,179,114, 72,186,245,255,255, 95,106,116, 66, 2,
    114,251,255, 16, 73,135, 50, 91,106,255,159, 18,185,146,180,183, 84,186, 72, 55, 91, 83, 81,107,255, 95,177,181, 22,176,183, 4,180, 80, 9, 86, 48,182, 54, 72,183,149,150, 75,151,183,249,255,
    74,186,164,255,255,255, 79,186,148, 38,166,255,255,255, 10,161, 6, 70,240,255,255,143, 10, 74,134, 70, 22,250,255, 65, 25, 66, 98,244,255,255, 63,128, 33, 41,148, 98,244,255, 32, 68, 98,
    255,255,255,255,143, 35, 40, 68, 98,255,255,255, 74,169, 70, 43,243,255,255, 15, 40,130, 75,169,164,246,255,179, 2, 97, 96,180,161,255,111, 20, 22, 74, 24, 18,139, 27,105,148, 99, 25,179, 54,
    255,143, 27, 24,176, 22, 25,180, 20,179, 54, 6, 96,244,255,255,111,132,107,248,255,255,255,167,118,168,152,250,255,255, 15, 55,160, 7,169,118,250,255,106, 23,122,113, 24, 8,255,175,118,
    122, 17, 55,255,255,255, 33, 22,134,129,137,118,255, 47,150,146, 97,151,144,115,147,135,112, 96, 6,242,255,255,127, 35,118,242,255,255,255,255, 50,171,134,138,137,118,255, 47,112,114, 11,121,
    118,164,122,129,135,181,183,167, 50,187,176, 6,255,127,167, 22,118,241,255,152,134,118, 25,182, 54, 49, 6, 25,187,247,255,255,255,255,133,112, 98,179,176, 6,255,127,187,255,255,255,255,255,
    103,251,255,255,255,255,255, 63,128,123,246,255,255,255,255, 16,185,103,255,255,255,255,143,145, 56,177,103,255,255,255, 26, 98,123,255,255,255,255, 31,162, 3,104,123,255,255,255,104, 32,154,
    182,247,255,255,111,123,162,163, 56,154,248,255, 39, 99,114,255,255,255,255,127,128,103, 90, 2,255,255,255,114, 38,115, 16,249,255,255, 31, 38,129, 22,137,120,246,255,122,166,113, 49,247,255,
    255,175,183,113, 26,128, 1,248,255, 48, 7,167,160,185,122,255,127,166,167,136,154,255,255,255,154,180,104,255,255,255,255, 63,182, 3, 6,180,255,255,104,139,180, 9,241,255,255,255,159,180,
    105,147, 19, 59,246,255,134,180,139,162,241,255,255, 31,162, 3, 11,182, 64,246,255,180, 72,182, 32, 41,154,255,175, 57, 58,146, 52, 59, 70, 54, 40,131, 36,180,242,255,255, 15, 36,180,242,255,
    255,255,255,145, 32, 67, 66, 70,131,255, 31, 73, 65, 34,180,255,255,255, 24,131, 22, 72,182, 26,255,175, 1, 10,182, 64,255,255,255,180, 67,131,166, 3,147,154,163, 73,166,244,255,255,255,255,
    148,117,182,255,255,255,255, 15, 56,148,181,183,255,255,255, 5, 81, 4,181,251,255,255,191,103, 56, 32, 69, 19,245,255, 89,164, 33,183,253,255,255,111,123, 31, 10, 56,148,245,255,183, 91,164,
    36, 74, 32,255, 63,132, 83, 52, 82, 90,178,103, 39,115, 38, 69,249,255,255,159, 69,128, 6, 38,134,247,255, 99, 50,103, 81, 80, 4,255,111,130,134, 39,129,132, 21,133, 89,164, 97,113, 22,115,
    255, 31,166,113, 22,112,120,144, 69, 4, 74, 90, 48,186,122,115,122,166,167, 88,164,132,250,255,150,181,155,139,249,255,255, 63,182, 96, 3,181,144,245,255,176, 8,181, 16, 85,182,255,111, 59,
    54, 85, 19,255,255,255, 33,154,181,185,184,181,255, 15, 59, 96, 11,185,181, 25,162,139,181,181, 8,165, 37, 32,181, 59, 54, 37, 58, 90,243,255,133, 89,130,181, 58, 40,255,159,181,185, 0, 38,
    255,255,255, 81, 24, 8,181, 56, 40, 38, 24,181,184,255,255, 49, 22,166,131, 86,150,152,166, 1, 10,150, 5,101,240,255, 48, 88,166,255,255,255,175,101,255,255,255,255,255,
    91,122,181,255,255,255,255,191,165,123,133, 3,255,255,255,181, 87,186,145,240,255,255,175, 87,186,151, 24, 56,241,255, 27,178, 23, 87,241,255,255, 15, 56, 33, 23, 87, 39,251,255,121,149,114,
    9, 34,123,255,127, 37, 39, 91, 41, 95,152, 40, 82, 42, 83,115,245,255,255,143, 2, 88,130, 87, 42,245,255, 9, 81, 58, 53, 55, 42,255,159, 40, 41,129, 39, 42,117, 37, 49, 53, 87,255,255,255,
    255, 15,120,112, 17, 87,255,255,255, 9,147, 83, 53,247,255,255,159,120,149,247,255,255,255,133, 84,138,186,248,255,255, 95, 64,181, 80,186, 59,240,255, 16,137,164,168,171, 84,255,175, 75,
    74,181, 67, 73, 49, 65, 82, 33, 88,178, 72,133,255, 15,180,176, 67,181,178, 81,177, 32, 5,149,178, 69,133,139,149, 84,178,243,255,255,255,255, 82, 58, 37, 67, 53, 72,255, 95, 42, 37, 68, 2,
    255,255,255,163, 58,165,131, 69,133, 16, 89, 42, 37, 20, 41, 73,242,255, 72,133, 53, 83,241,255,255, 15, 84, 1,245,255,255,255, 72,133, 53, 9, 5, 83,255,159, 84,255,255,255,255,255,
    180, 71,185,169,251,255,255, 15, 56,148,151,123,169,251,255,161, 27, 75, 65,112,180,255, 65, 65, 67, 24, 74,171, 75,180,151, 75, 41,155, 33,255,159, 71,185,151,177,178, 1, 56,123,180, 36,
    66,240,255,255,191, 71, 75,130, 67, 35,244,255,146, 42,151, 50,119,148,255,159,122,121,164,114,120, 32,112,115, 58, 42, 71, 26, 10, 4, 26, 42,120,244,255,255,255,255,148, 65,113, 23,243,255,
    255, 79, 25, 20, 7, 24,120,241,255, 4,115, 52,255,255,255, 79,120,255,255,255,255,255,169,168,139,255,255,255, 63,144,147,187,169,255,255,255, 18, 10,138,168,251,255,255, 63,161,
    39,250,255,255,255, 31, 27,155,148,255,255, 63,144,147, 27,146,178,249,255, 32,139,176,255,255,255, 63,178,255,255,255,255,255, 50, 40,168,138,249,255,255,255, 42,144,242,255,
    255,255,255, 50, 40,168, 16, 24,138,255, 31, 42,255,255,255,255,255,255, 49,152,129,255,255,255,255, 15, 25,255,255,255,255,255,255, 48,248,255,255,255,255,255,255,255,255,255,255
};

ushort edge_table(const uint i) {
    return edge_table_data[i<128u?i:255u-i];
}
uchar triangle_table(const uint i) {
    return (triangle_table_data[i/2u]>>(4u*(i%2u)))&0xF;
}
float3 interpolate_vertex(const float3 p1, const float3 p2, const float v1, const float v2, const float iso) { // linearly interpolate position where isosurface cuts an edge between 2 vertices
    const float w = (iso-v1)/(v2-v1);
    return (1.0f-w)*p1+w*p2;
}
uint marching_cubes(const float* v, const float iso, float3* triangles) { // input: 8 values v, isovalue; output: returns number of triangles, 15 triangle vertices t
    uint cube = 0u; // determine index of which vertices are inside of the isosurface
    for(uint i=0u; i<8u; i++) cube |= (v[i]<iso)<<i;
    if(cube==0u || cube==255u) return 0u; // cube is entirely inside/outside of the isosurface
    float3 p[8]; // definition of unit cube corners
    p[0] = (float3)(0.0f, 0.0f, 0.0f);
    p[1] = (float3)(1.0f, 0.0f, 0.0f);
    p[2] = (float3)(1.0f, 0.0f, 1.0f);
    p[3] = (float3)(0.0f, 0.0f, 1.0f);
    p[4] = (float3)(0.0f, 1.0f, 0.0f);
    p[5] = (float3)(1.0f, 1.0f, 0.0f);
    p[6] = (float3)(1.0f, 1.0f, 1.0f);
    p[7] = (float3)(0.0f, 1.0f, 1.0f);
    const uint edges = edge_table(cube);
    float3 vertex[12]; // find the vertices where the surface intersects the cube
    if(edges& 1u) vertex[ 0] = interpolate_vertex(p[0], p[1], v[0], v[1], iso); // calculate vertices on all 12 edges
    if(edges& 2u) vertex[ 1] = interpolate_vertex(p[1], p[2], v[1], v[2], iso);
    if(edges& 4u) vertex[ 2] = interpolate_vertex(p[2], p[3], v[2], v[3], iso);
    if(edges& 8u) vertex[ 3] = interpolate_vertex(p[3], p[0], v[3], v[0], iso);
    if(edges& 16u) vertex[ 4] = interpolate_vertex(p[4], p[5], v[4], v[5], iso);
    if(edges& 32u) vertex[ 5] = interpolate_vertex(p[5], p[6], v[5], v[6], iso);
    if(edges& 64u) vertex[ 6] = interpolate_vertex(p[6], p[7], v[6], v[7], iso);
    if(edges& 128u) vertex[ 7] = interpolate_vertex(p[7], p[4], v[7], v[4], iso);
    if(edges& 256u) vertex[ 8] = interpolate_vertex(p[0], p[4], v[0], v[4], iso);
    if(edges& 512u) vertex[ 9] = interpolate_vertex(p[1], p[5], v[1], v[5], iso);
    if(edges&1024u) vertex[10] = interpolate_vertex(p[2], p[6], v[2], v[6], iso);
    if(edges&2048u) vertex[11] = interpolate_vertex(p[3], p[7], v[3], v[7], iso);
    cube *= 15u;
    uint i; // number of triangle vertices
    for(i=0u; i<15u&&triangle_table(cube+i)!=15u; i+=3u) { // create the triangles
        triangles[i+2u] = vertex[triangle_table(cube+i+2u)];
        triangles[i+1u] = vertex[triangle_table(cube+i+1u)];
        triangles[i   ] = vertex[triangle_table(cube+i   )];
    }
    return i/3u; // return number of triangles
}
```

Moritz Lehmann, M.Sc.
moritz.lehmann@uni-bayreuth.de | @ProjectPhysX

SFB MIKROPLASTIK DFG Elitenetzwerk Bayern Biological Physics UNIVERSITÄT BAYREUTH

Ray-Triangle Intersection

- branchless version of Möller-Trumbore algorithm
- bidirectional intersection
- returns distance from ray origin to intersection point



```
float intersect_triangle_bidirectional(const ray r, const float3 p0, const float3 p1, const float3 p2) {
    const float3 u=p1-p0, v=p2-p0, w=r.origin-p0, h=cross(r.direction, v), q=cross(w, u);
    const float f=1.0f/dot(u, h), s=f*dot(w, h), t=f*dot(r.direction, q);
    return (s<0.0f||s>1.0f||t<0.0f||s+t>1.0f) ? -1.0f : f*dot(v, q);
}
```

Moritz Lehmann, M.Sc.
moritz.lehmann@uni-bayreuth.de | @ProjectPhysX

Normal Interpolation

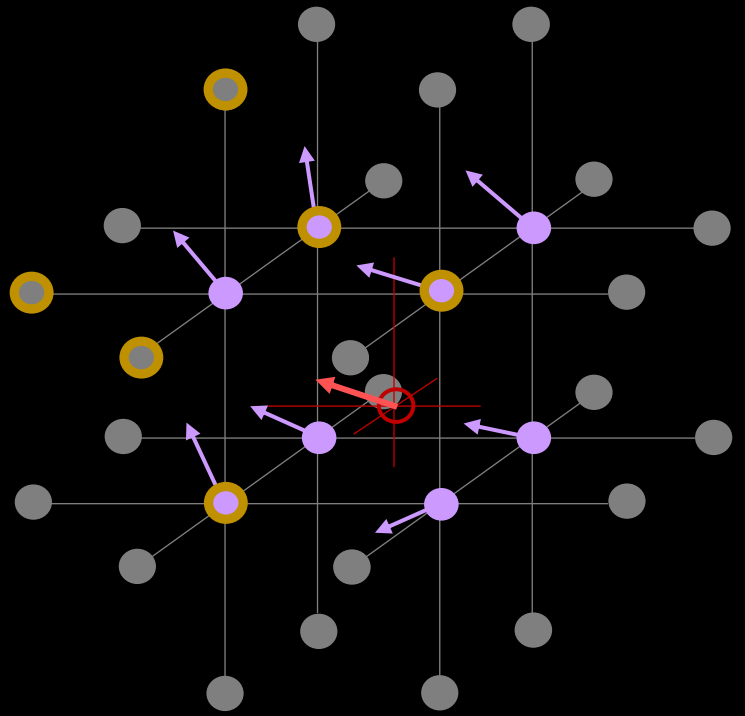
- calculate normals on grid points
- trilinear interpolation to hitpoint

  --> comes at almost no additional cost

Moritz Lehmann, M.Sc.
moritz.lehmann@uni-bayreuth.de | @ProjectPhysX

Normal Interpolation

- calculate normals on grid points
  - central difference stencil
  - reuse isovalues that have already been loaded
    --> only 24 new isovalues instead of 56
- trilinear interpolation to hitpoint



Moritz Lehmann, M.Sc.
moritz.lehmann@uni-bayreuth.de | @ProjectPhysX

SFB MIKROPLASTIK DFG Elitenetzwerk Bayern Biological Physics UNIVERSITÄT BAYREUTH

Skybox

- UV-mapping (3D vector to xy-coordinates)
- bilinear interpolation for pixel color

https://www.hdri-hub.com/hdri-skies-aviation-aerospace

```
uint skybox_color(const ray r, const global uint* skybox) {
    const float fu = (float)def_skybox_w*fma(atan2(r.direction.x, r.direction.y),  0.5f/3.1415927f, 0.5f);
    const float fv = (float)def_skybox_h*fma(asin (r.direction.z              ), -1.0f/3.1415927f, 0.5f);
    const int ua=clamp((int)fu, 0, (int)def_skybox_w-1), va=clamp((int)fv, 0, (int)def_skybox_h-1), ub=(ua+1)%def_skybox_w, vb=min(va+1, (int)def_skybox_h-1);
    const uint s00=skybox[ua+va*def_skybox_w], s01=skybox[ua+vb*def_skybox_w], s10=skybox[ub+va*def_skybox_w], s11=skybox[ub+vb*def_skybox_w];
    const float u1=fu-(float)ua, v1=fv-(float)va, u0=1.0f-u1, v0=1.0f-v1;
    return color_mix(color_mix(s00, s01, v0), color_mix(s10, s11, v0), u0);
}
```

Moritz Lehmann, M.Sc.
moritz.lehmann@uni-bayreuth.de | @ProjectPhysX

Conclusions

- if the simulation is limited by PCIe/IO, you have already lost
- much faster alternative: combine compute and graphics

- rendering is just another form of compute

- OpenCL enables simulations at peak efficiency
- OpenCL can render faster than anything else

- (micro-)optimization adds up

Moritz Lehmann, M.Sc.
moritz.lehmann@uni-bayreuth.de | @ProjectPhysX


148x48 11fps