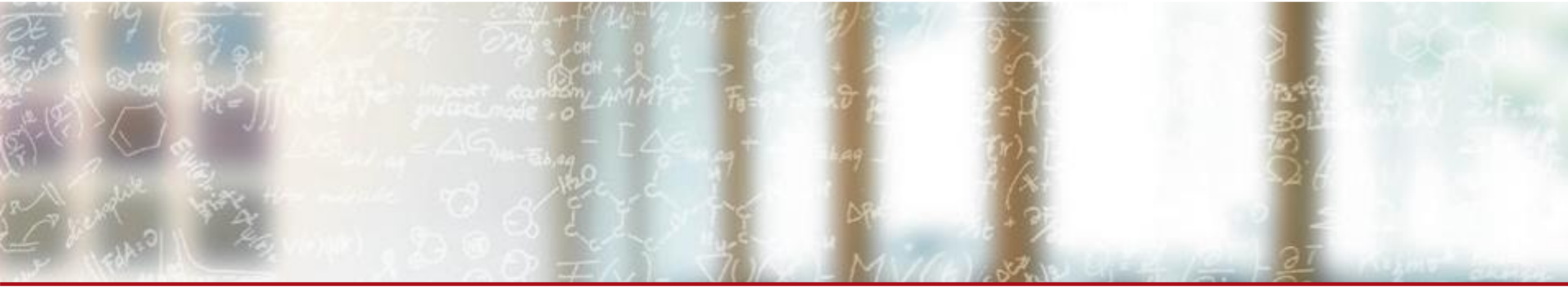
CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich



# ParaView Advanced Rendering for Data Visualization

Jean M. Favre, CSCS
October 12, 2020

# Motivations

- We have often heard "*Scientific Visualization is not about making pretty pictures*"

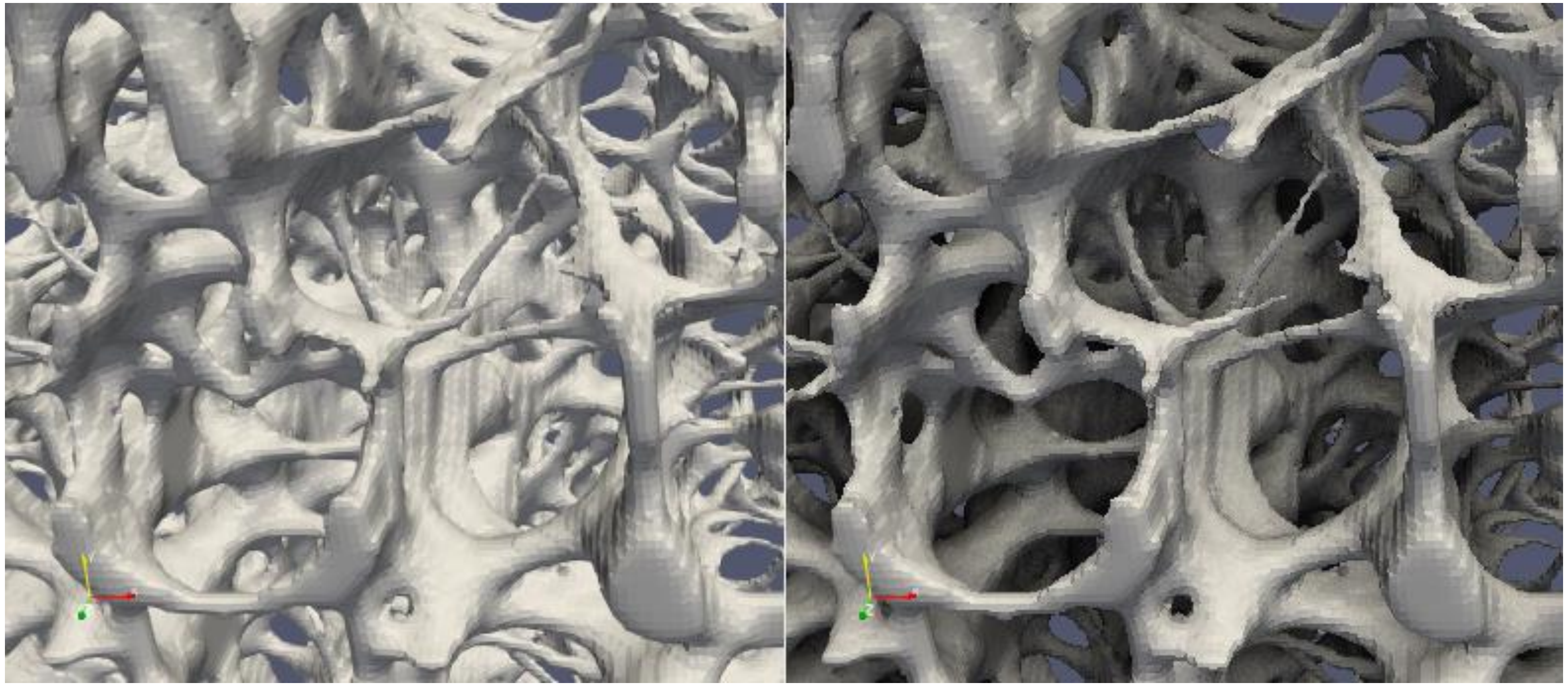
CFD != CFD

In other words Computational Fluid Dynamics != Colorful Fluid Dynamics

- Visualization's first goals remain:
  - Debugging a simulation
  - Understanding a phenomenon
  - Query for properties
- But a more appealing visualization can also:
  - Provide enhanced perception
  - Be closer to what Mother Nature has brought to us
  - Stimulate discussion, and provide a Front Cover in a famous publication

# Example: Screen-Space Ambient Occlusion (SSAO)

*“Our depth perception is highly sensitive to ambient occlusion. Holes, creases, concave surfaces are occluded by the surrounding geometry. This has to be taken into account when computing illumination”* Michael Migliore, Kitware



[https://blog.kitware.com/ssao/](https://blog.kitware.com/ssao/)

# Open ParaView Python Shell

```
import vtk
ogl = vtk.vtkRenderStepsPass()
ssao = vtk.vtkSSAOPass()
sceneSize=20.
ssao.SetRadius(0.1 * sceneSize) # comparison radius
#ssao.SetBias(0.001 * sceneSize) # comparison bias
ssao.SetKernelSize(256) # number of samples used
ssao.BlurOff() # do not blur occlusion
ssao.SetDelegatePass(ogl)

v = GetRenderView()
renderer  = v.GetRenderer()
renderer.SetPass(ssao)
Render()
```

```
# to turn if off

>>> ogl = vtk.vtkRenderStepsPass()
>>> renderer.SetPass(ogl)
```

**Try out**

paraview --script=pvSSAOTest.py

# From classic OpenGL to modern raytracing-based rendering

# Motivational material

- A rendering combining engineering data with some advanced rendering for an industrial showroom…
- https://blog.kitware.com/virtual-tour-and-high-quality-visualization-with-paraview-5-6-ospray/

# Two methods

## Use jupyter.cscs.ch

RayTracing.ipynb

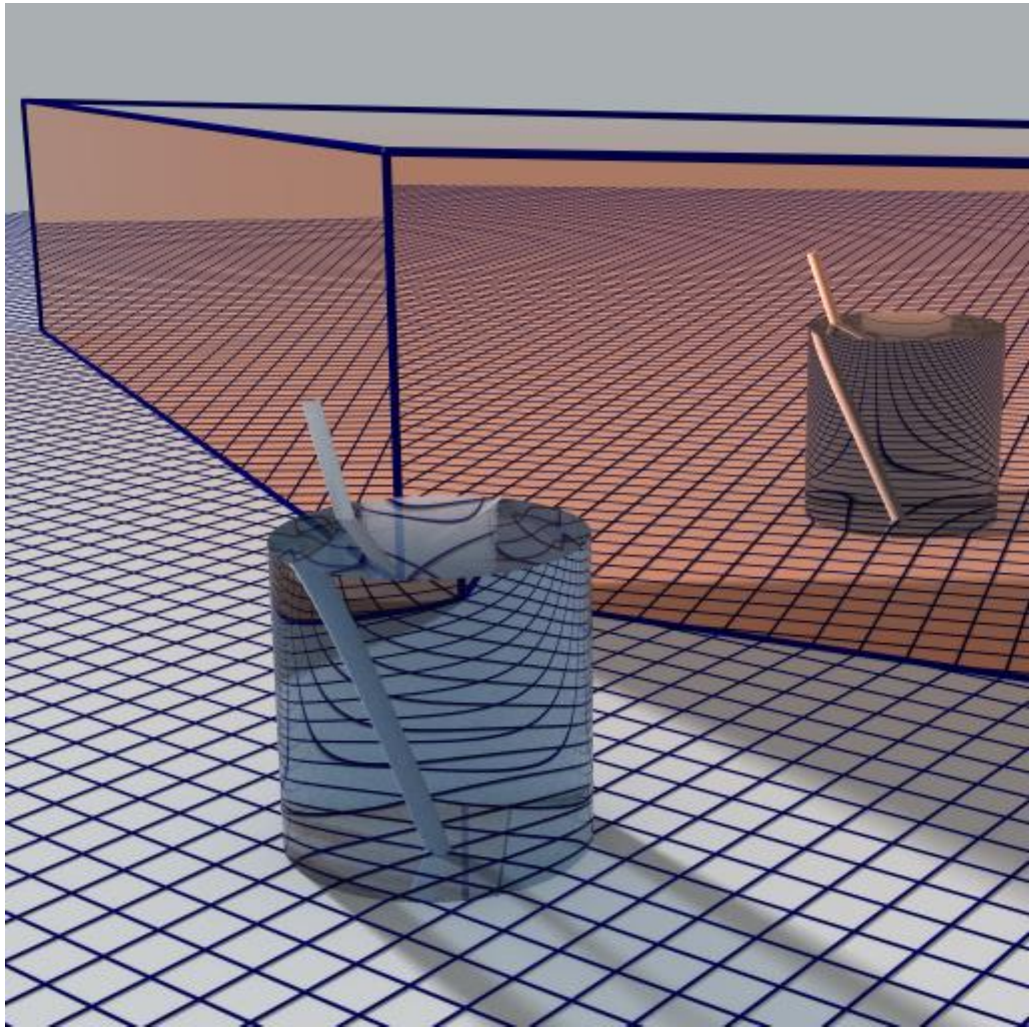
## Use ParaView

paraview --enable-streaming
 --script=pvRayTracing.py

# Rendering in 8 steps

- Review the old OpenGL shading
- Introduce raytracing-based rendering

- What do we see in this picture?
  - Reflection
  - Shadows
  - Refraction
  - Materials which behave differently

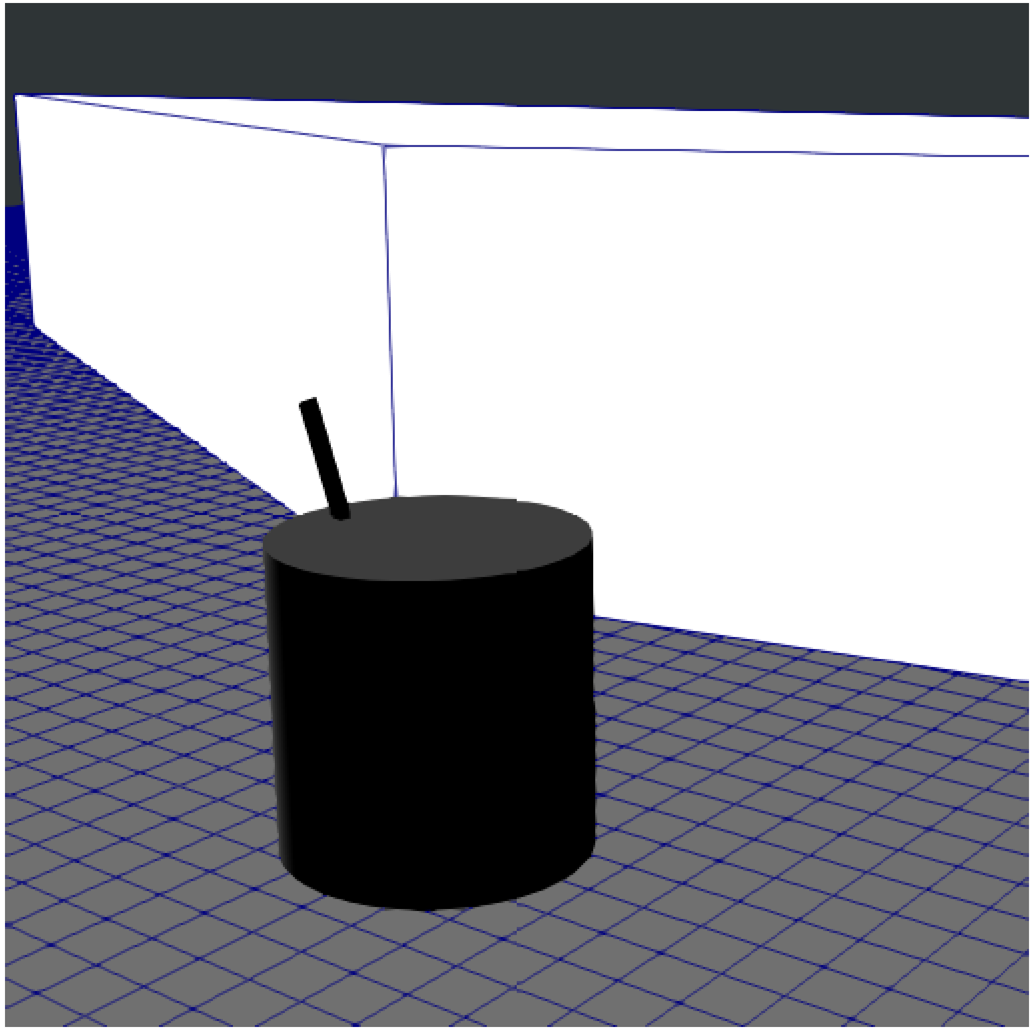


# OpenGL step 0

For every pixel, we have a hit or a miss. Multiple hits (multiple objects projecting onto the same pixel) are classified with the classic Z-buffer technique:

- The object closest to the viewer wins.
- A standard color is assigned to the pixel, regardless of orientation.

“**Ambient lighting**”

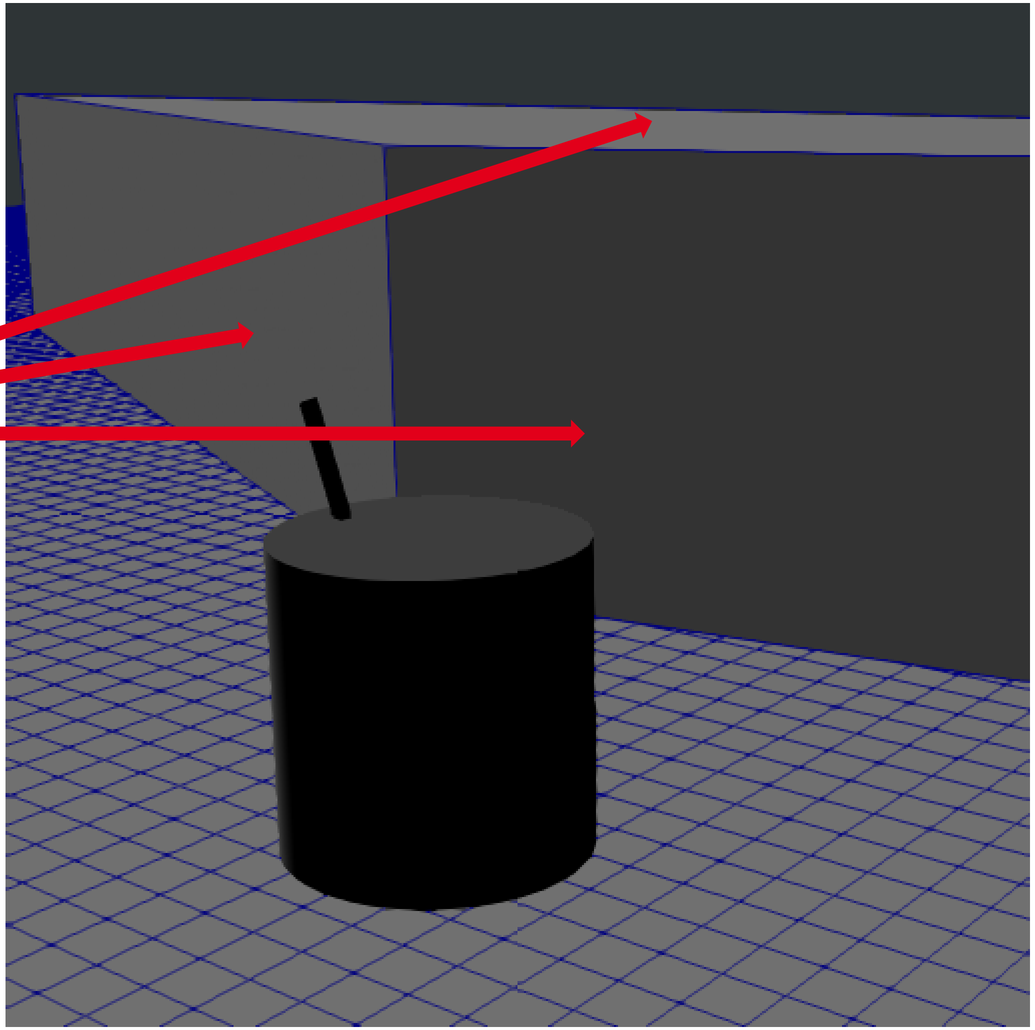


# OpenGL step 1

Adding a diffuse component

- Lighting depends on the orientation of the surface
- Need surface normals!

“**Ambient + Diffuse lighting**”

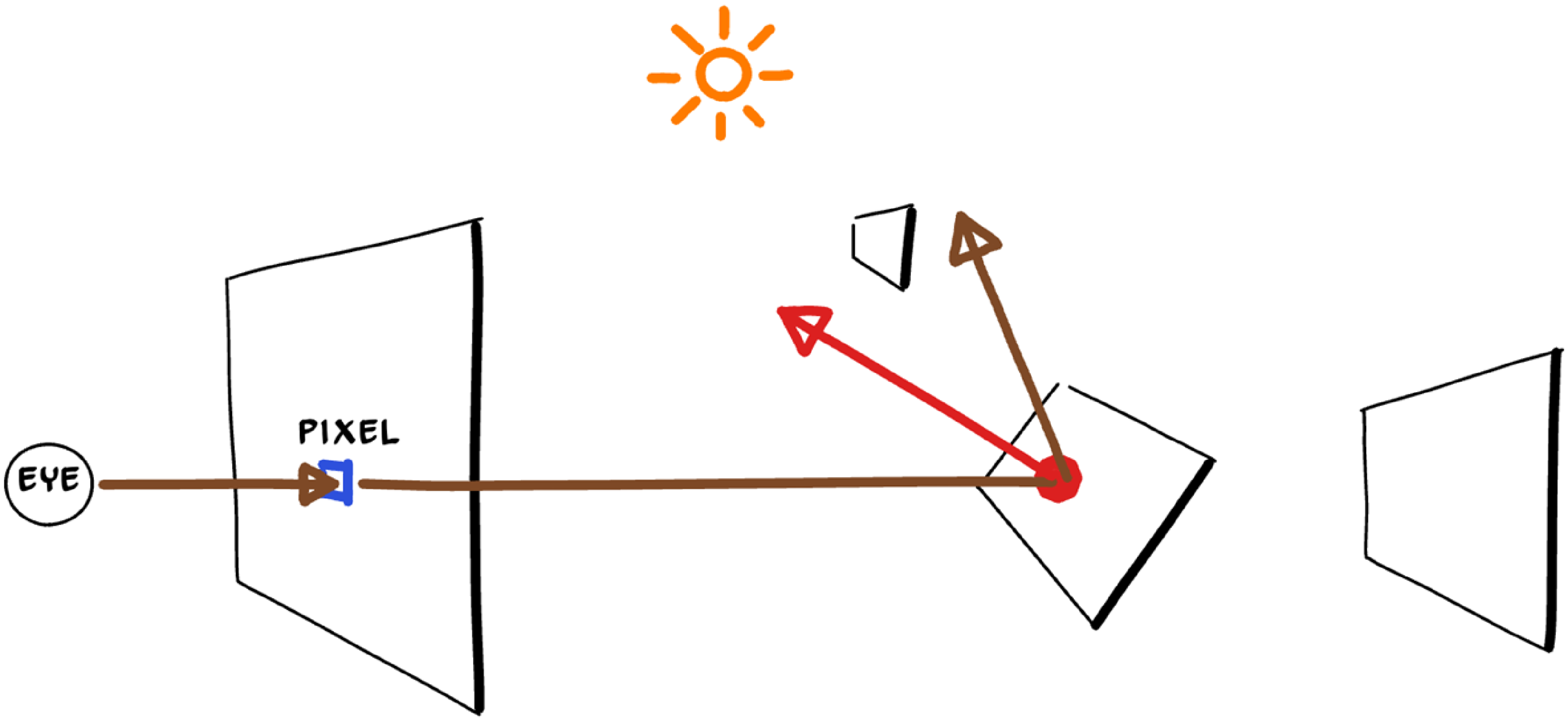


# Ray Tracing Background

**Conventional** 3D rendering is based on a process called **rasterization**. Rasterization uses objects created from a mesh of triangles or polygons to represent a 3D model of an object. The rendering pipeline converts each triangle of the 3D models into pixels on a 2D screen.

**Ray tracing** … provides realistic lighting by simulating the physical behavior of light. Ray tracing calculates the color of pixels by tracing the path that light would take if it were to travel from the eye of the viewer through the virtual 3D scene.

As it traverses the scene, the light may reflect from one object to another (reflections), be blocked by objects (shadows), or pass through transparent or semi-transparent objects (refractions).

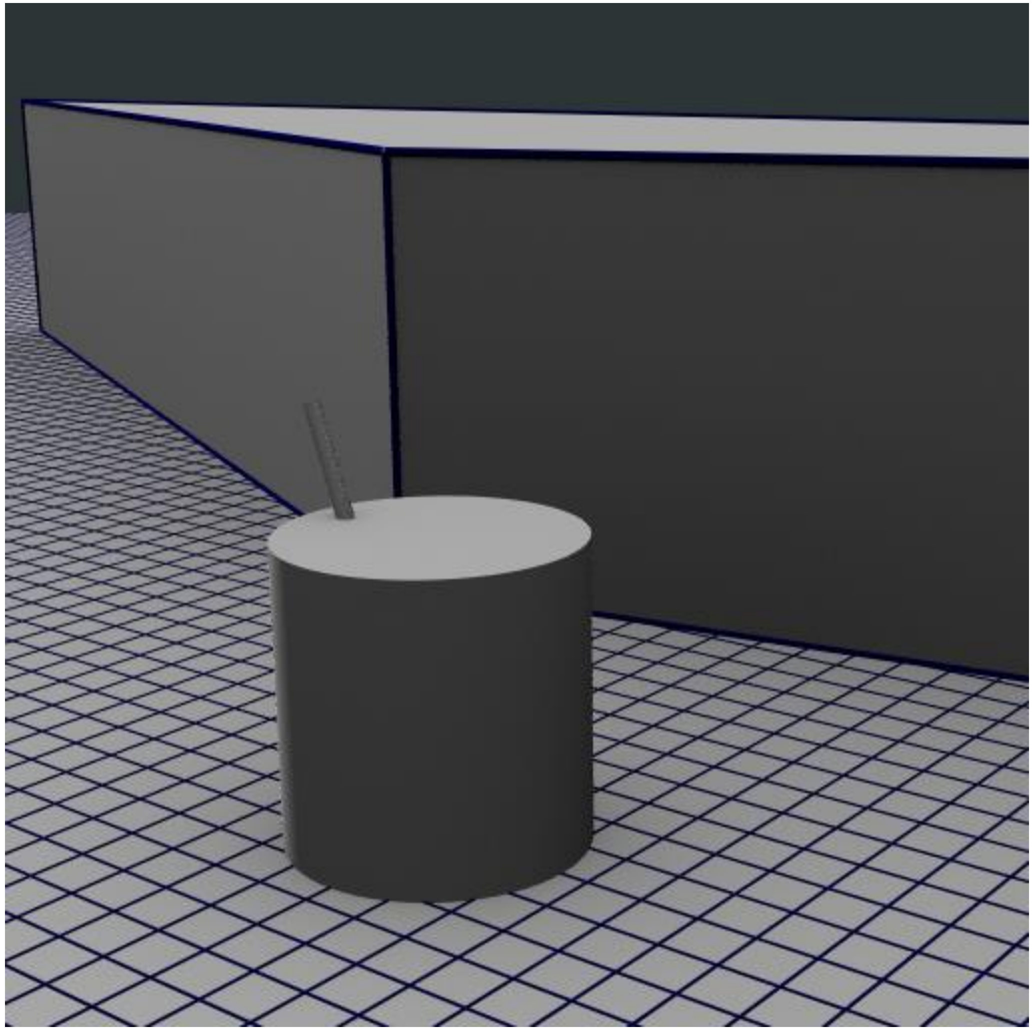



**Credits:** **Introduction to Real-Time Ray Tracing, Peter Shirley**
**Chris Wyman**, **Morgan McGuire**, NVIDIA

ETHzürich

# Switching from OpenGL to RayTracing

```
view = GetRenderView()
view.EnableRayTracing = 1
view.BackEnd = 'OSPRay raycaster‘
view.Shadows = 0
```



# Raycasting step 3

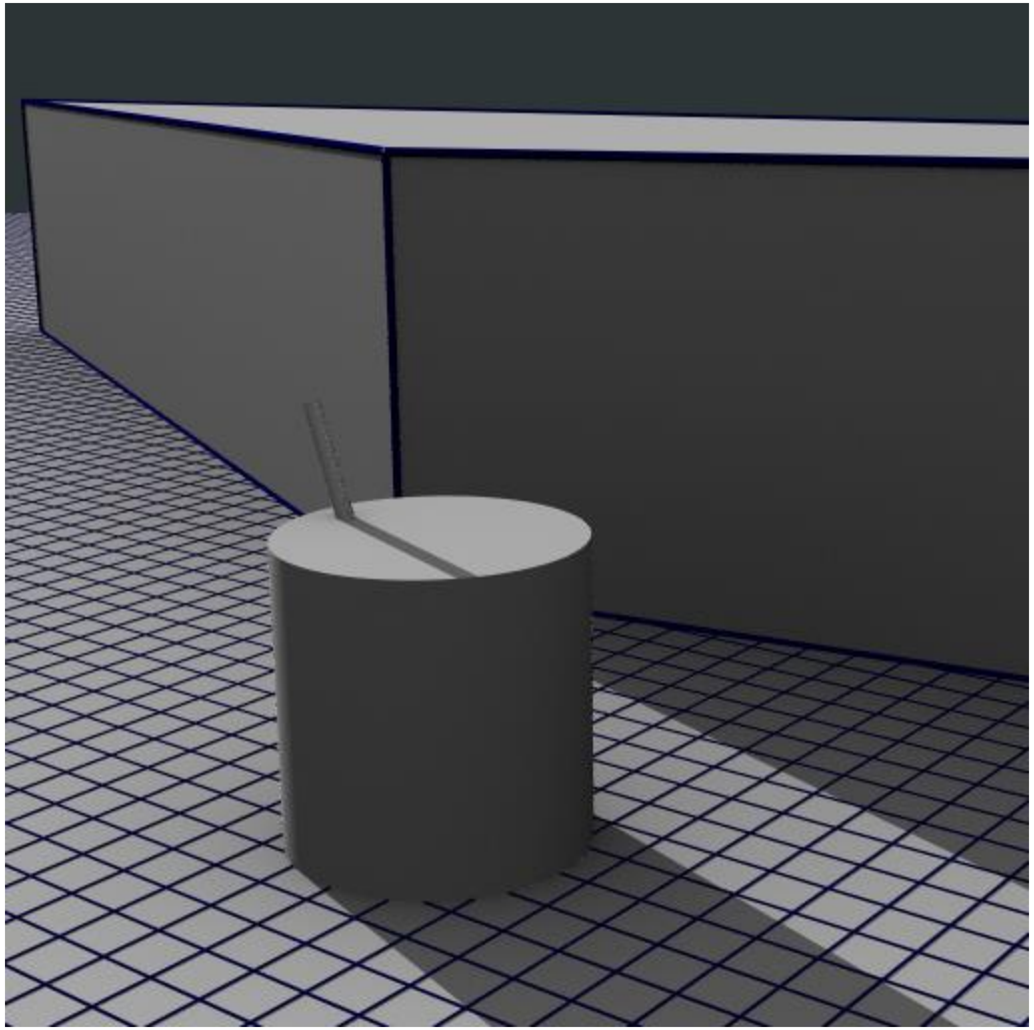
## Shadows

- We need to define lights

No lights, no shadows!

*view.Shadows = 1*

N.B. **Hard** shadow edges.



# Raycasting step 4

## Soft shadows

In real-life, shadow boundaries are always a little fuzzy. It is not *light diffraction* but rather the fact that the sun is not a point light source, but an extended light source.

N.B. Open the Light Inspector

*view.BackEnd = 'OSPRay pathtracer'*

*light1.Radius = 5*

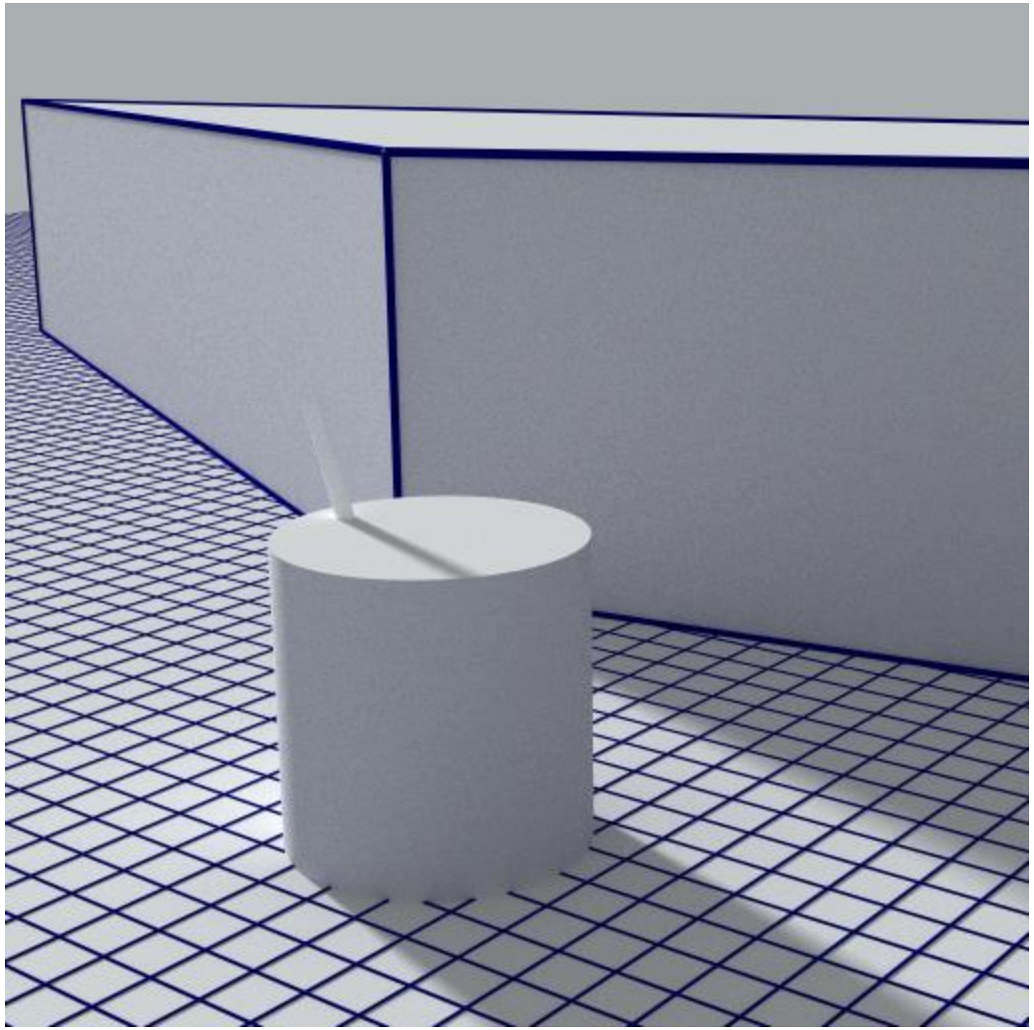
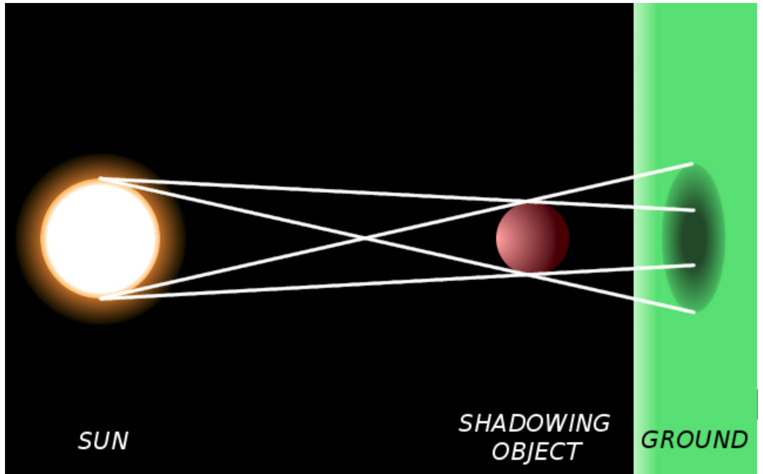


Image link



# Raycasting step 5

## Reflection

One surface reflects (mirrors) light, another does not.

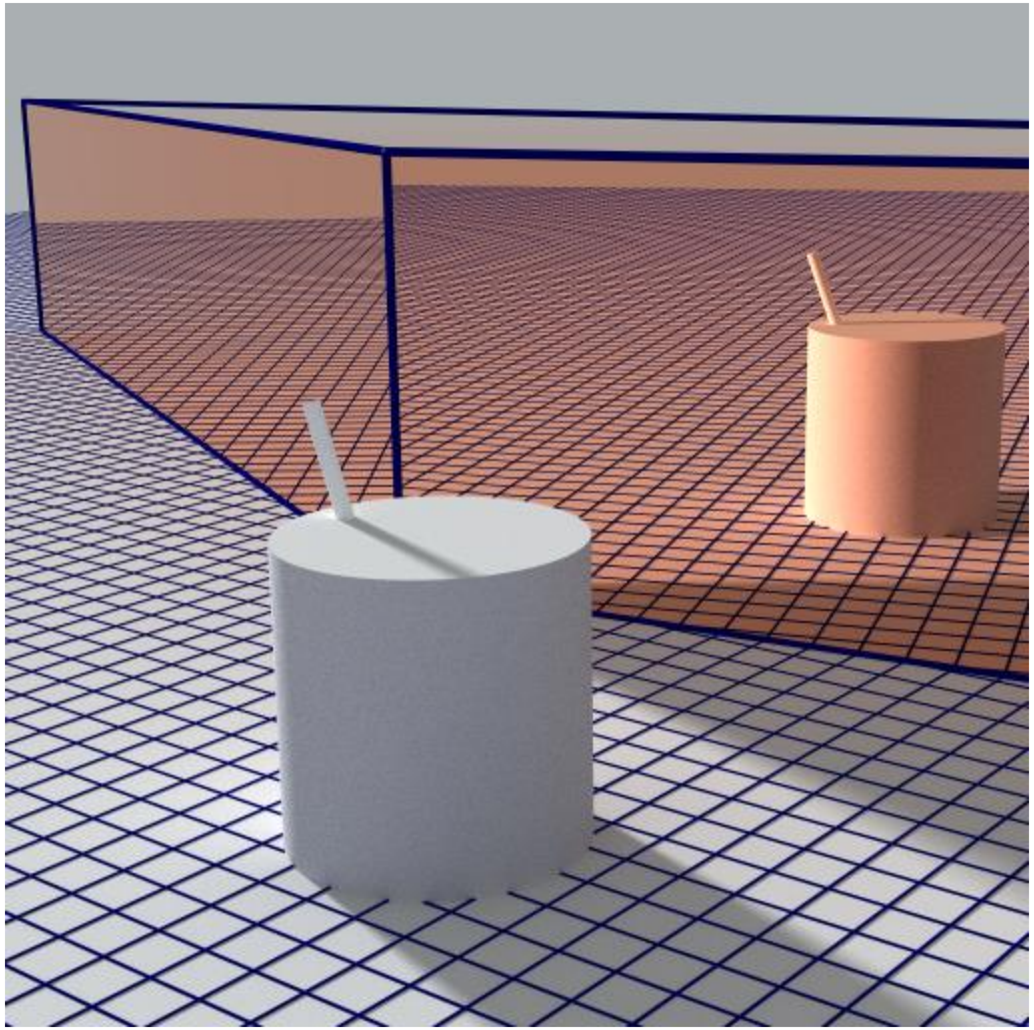
⇒ We need material definitions and we assign material names to each surface

⇒ See materials/ospray_mats.json

⇒ https://www.ospray.org/documentation.html

```
 "copper" : {
      "type" : "Metal",
      "doubles" : {
                  "roughness" : [0.0],
          "reflectance" : [0.7843, 0.4588, 0.2],
}}
```

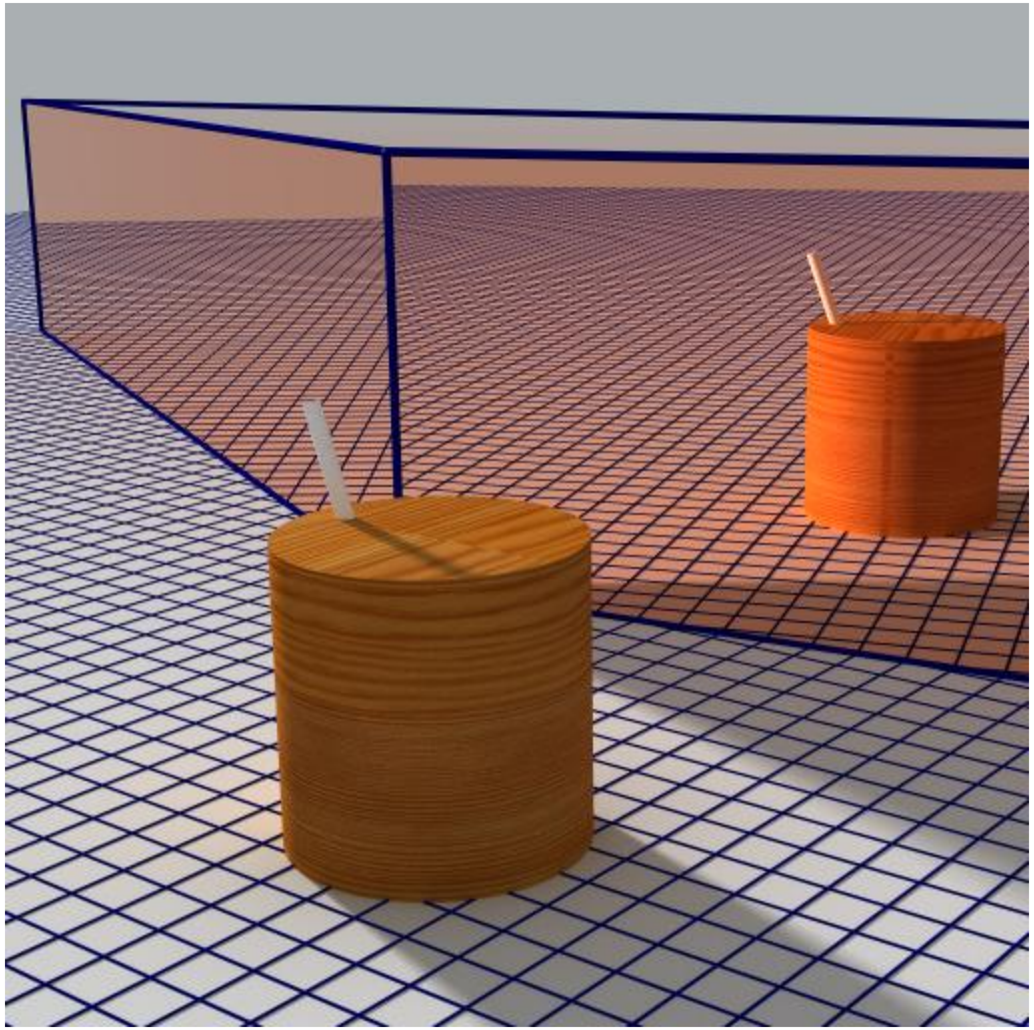
*rep.OSPRayMaterial = 'copper'*



# Raycasting step 6

Another material…

```
"wood" : {
    "type" : "OBJMaterial",
    "textures" : {
      "map_kd" : "wood.jpg"
    }
  }
```

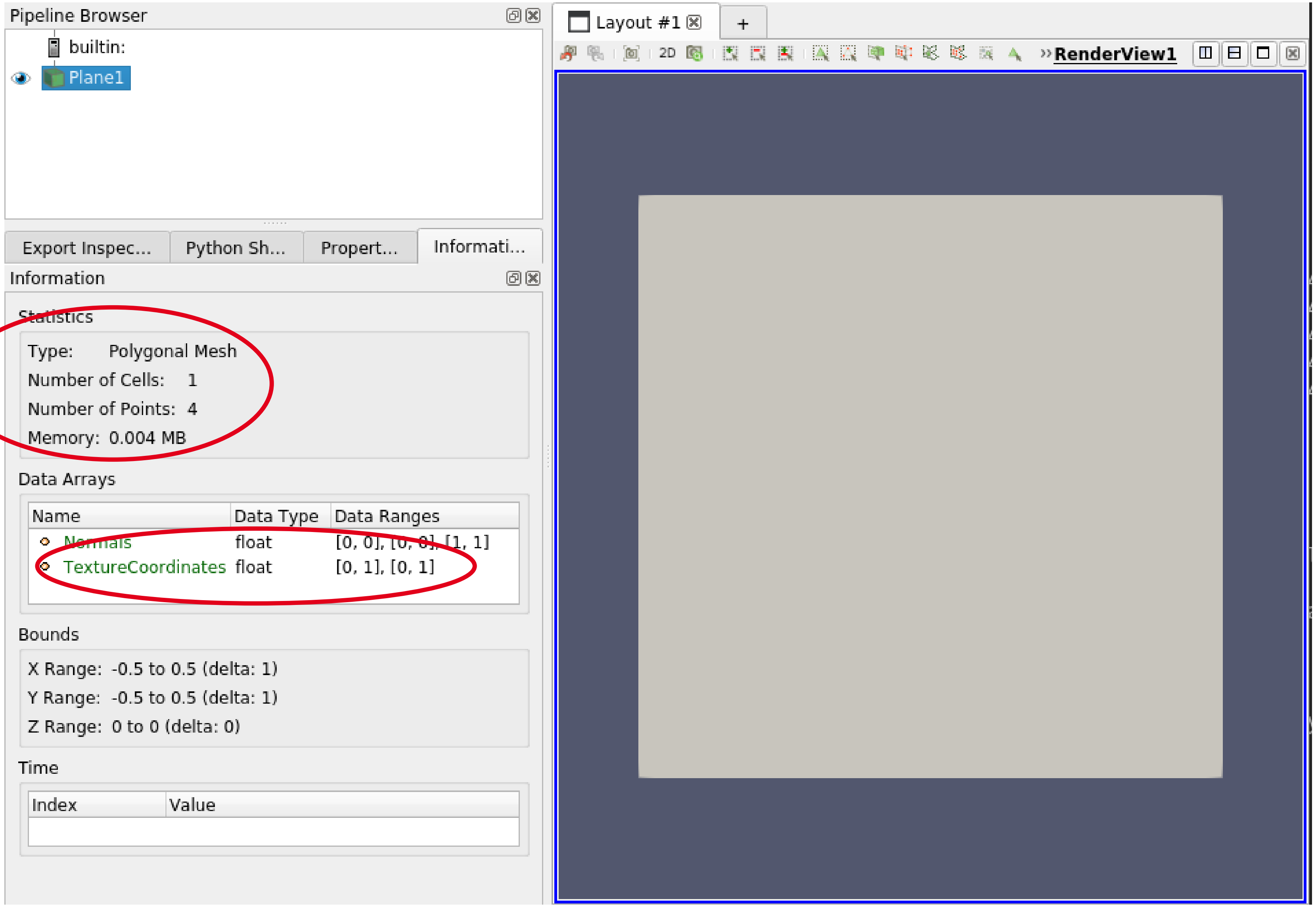


N.B. We need texture coordinates

# What are texture coordinates?

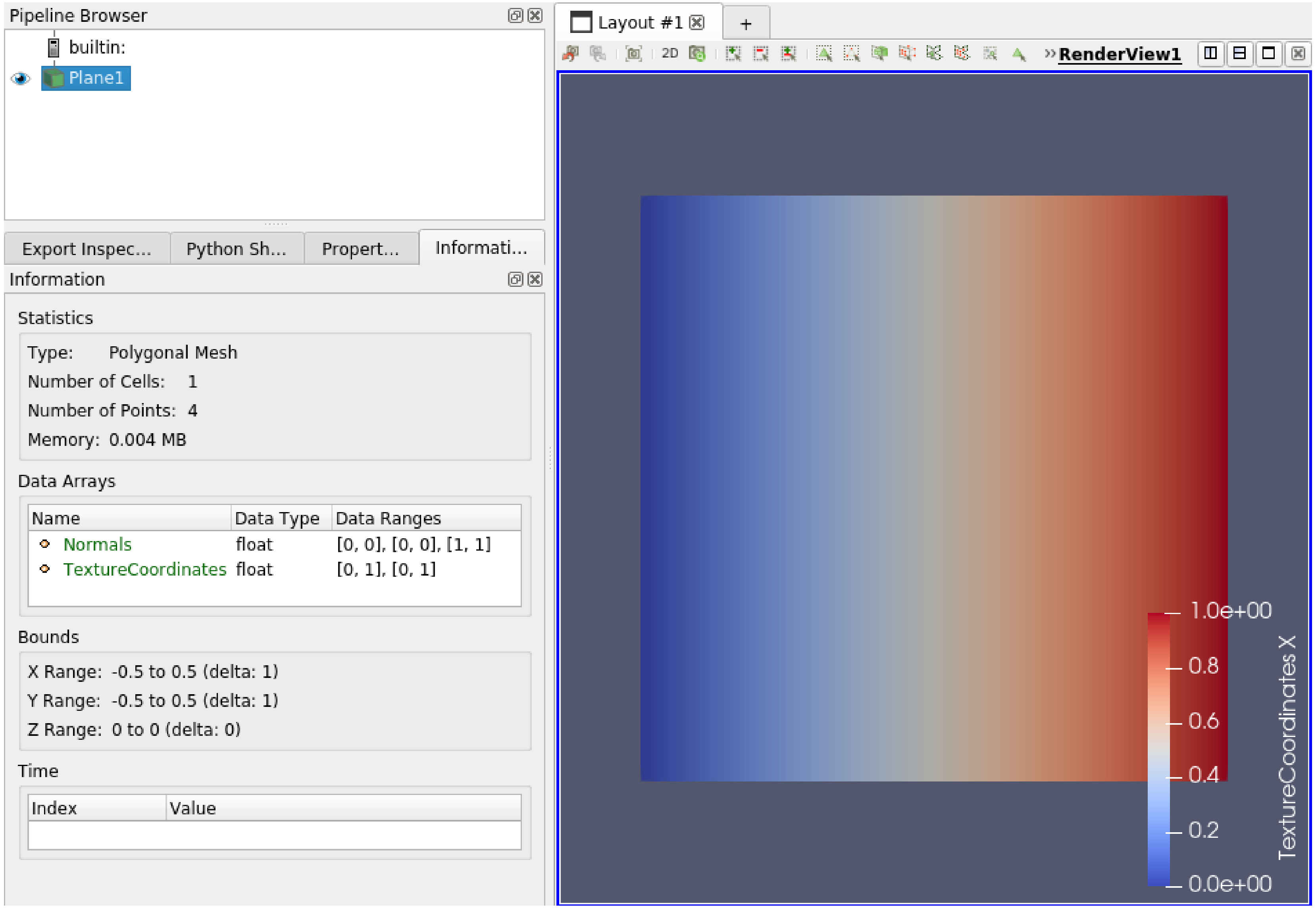
- Start paraview
- Create a Plane

- A single cell is enough



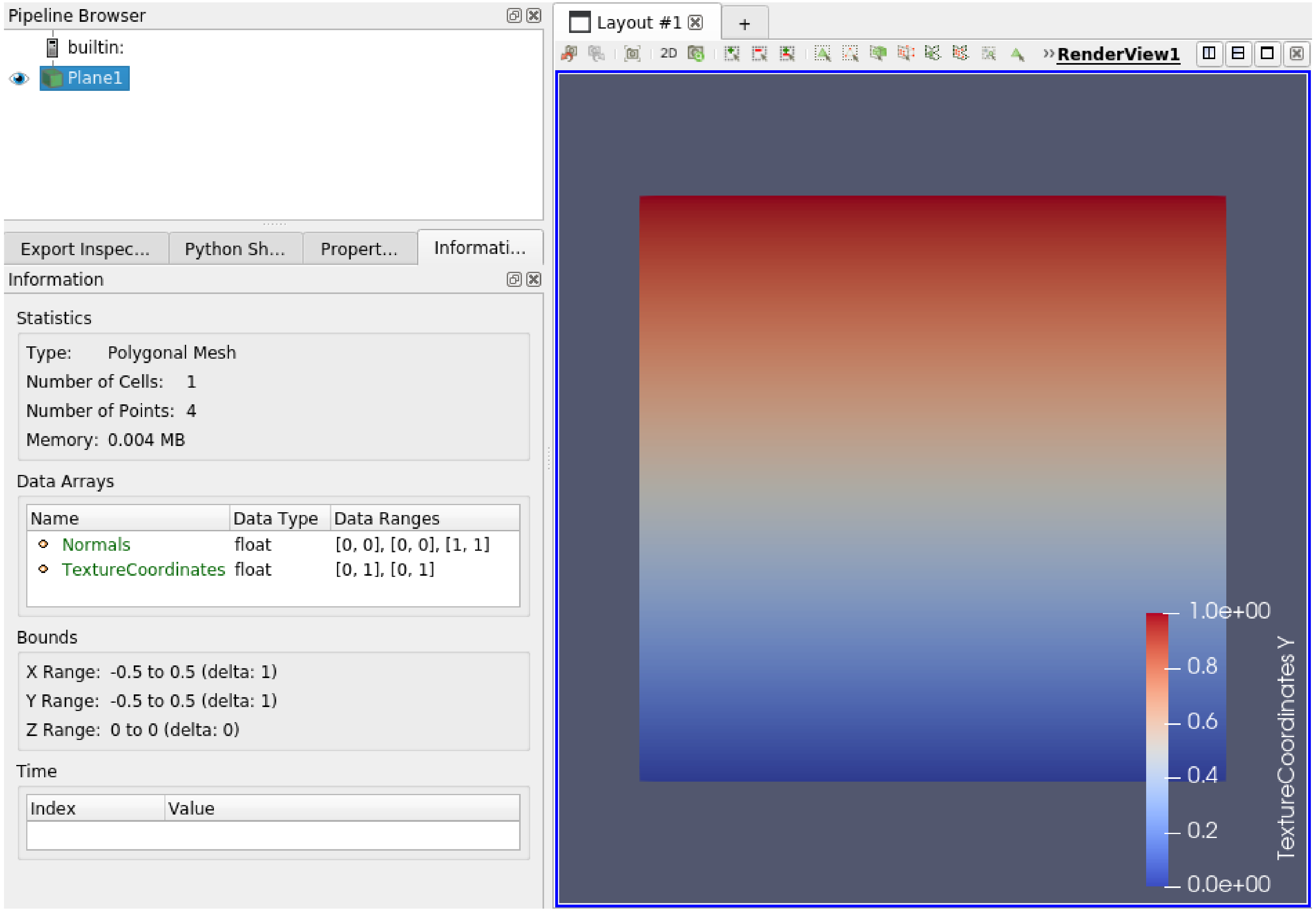
# What are texture coordinates?

- tx (a.k.a *u*) in [0, 1]



# What are texture coordinates?

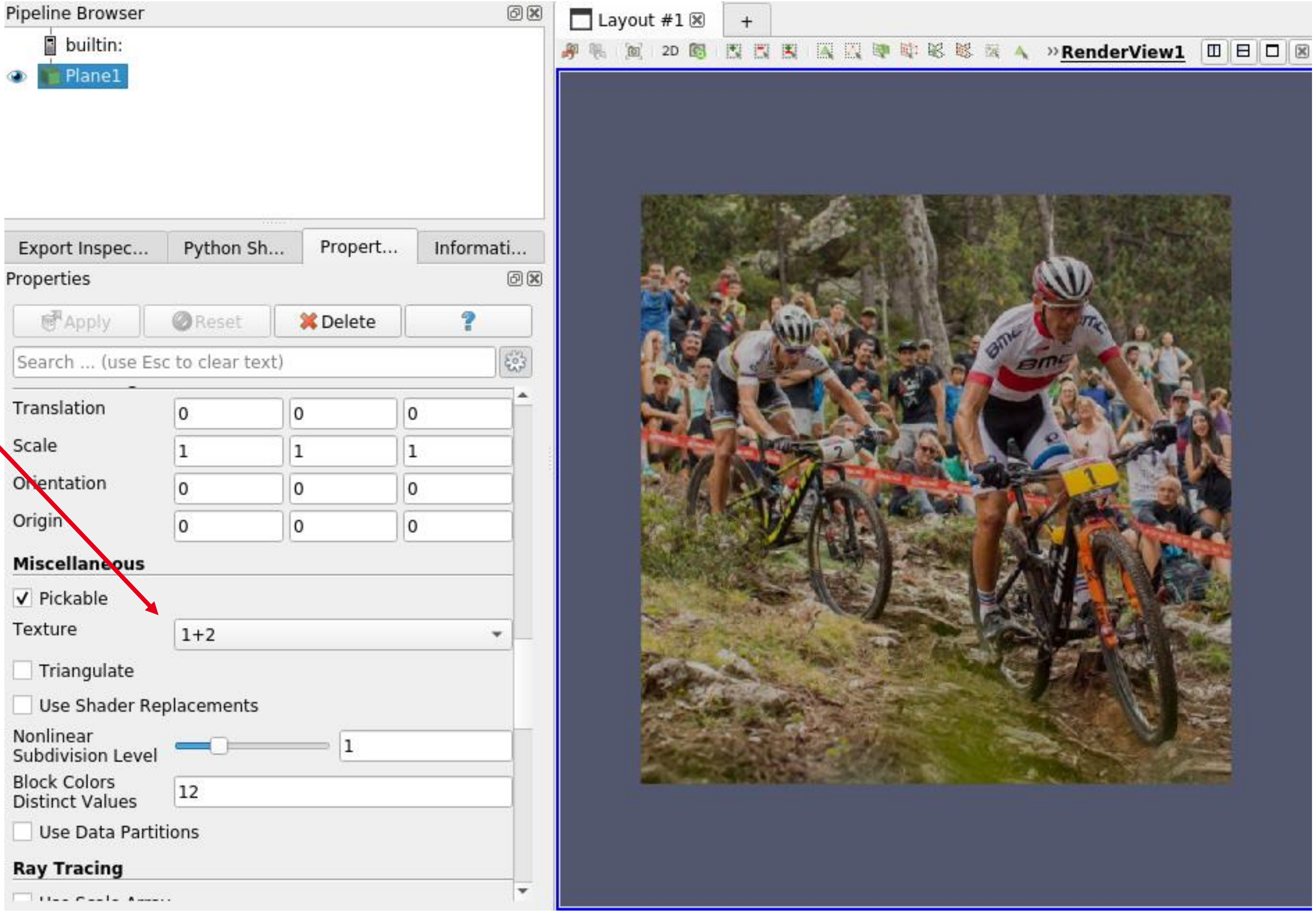
- ty (a.k.a *v*) in [0, 1]



# What are texture coordinates?

- Select an image filename

- Picture gets mapped to the [0-1][0-1] area



# Raycasting step 7

## Refraction

Use a material with appropriate definitions

*"water" : {*

*"type": "Glass",*

*"doubles" : {*

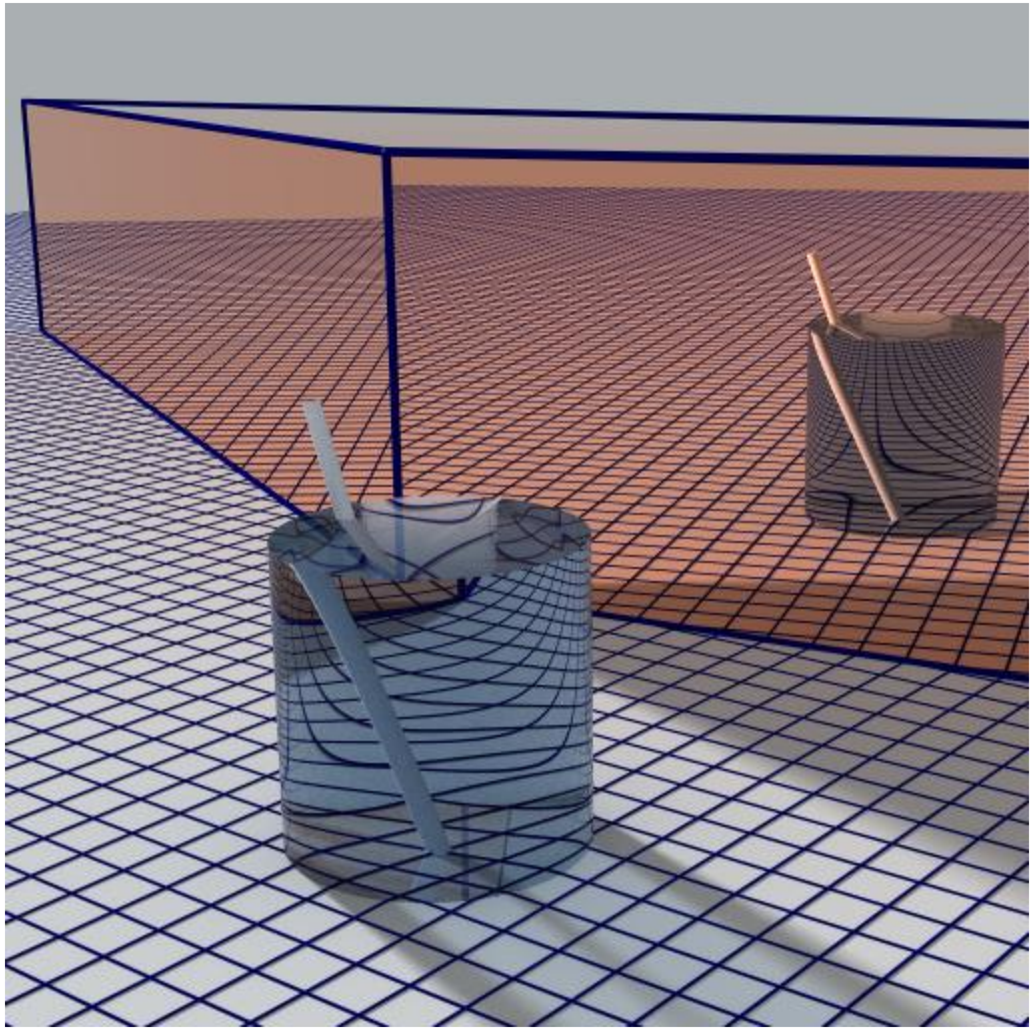
*"attenuationColor" : [0.22, 0.34, 0.47],*

*"attenuationDistance" : [3.0],*

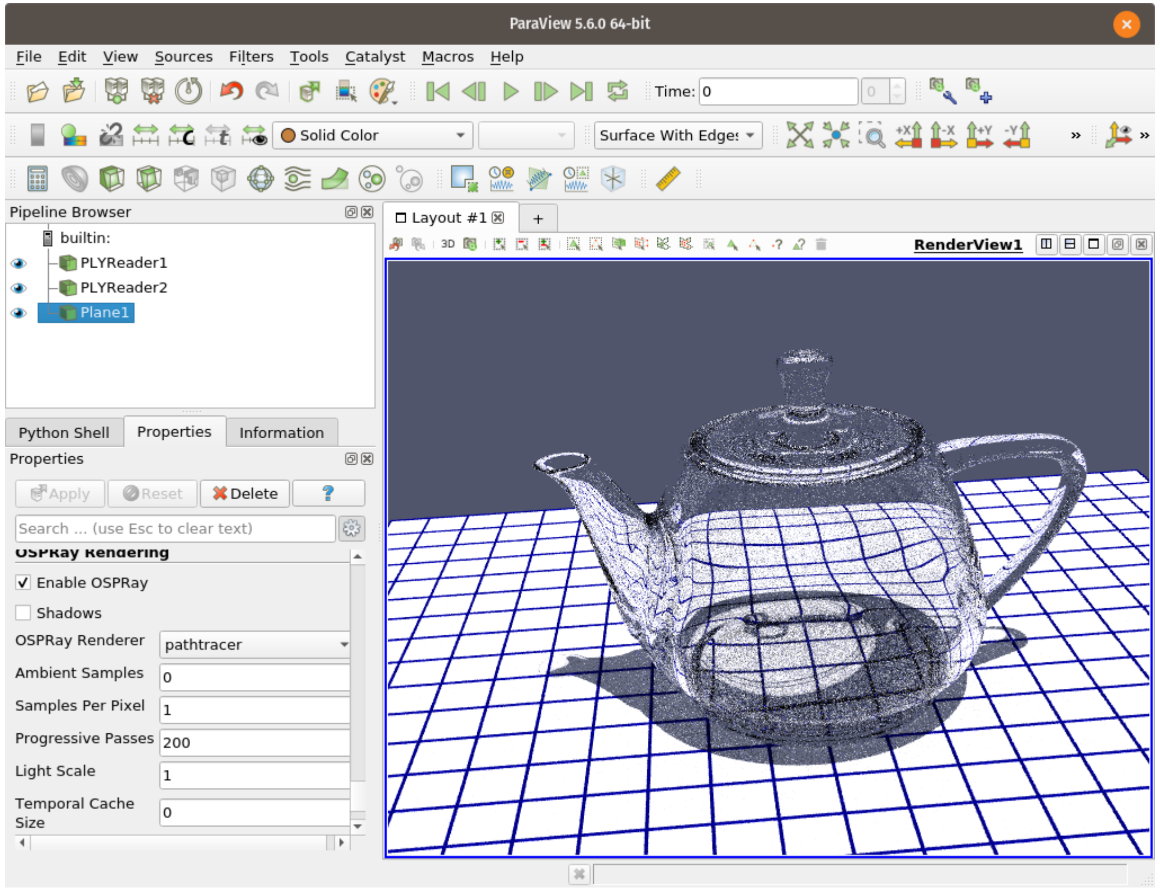
*"eta" : [1.33]*

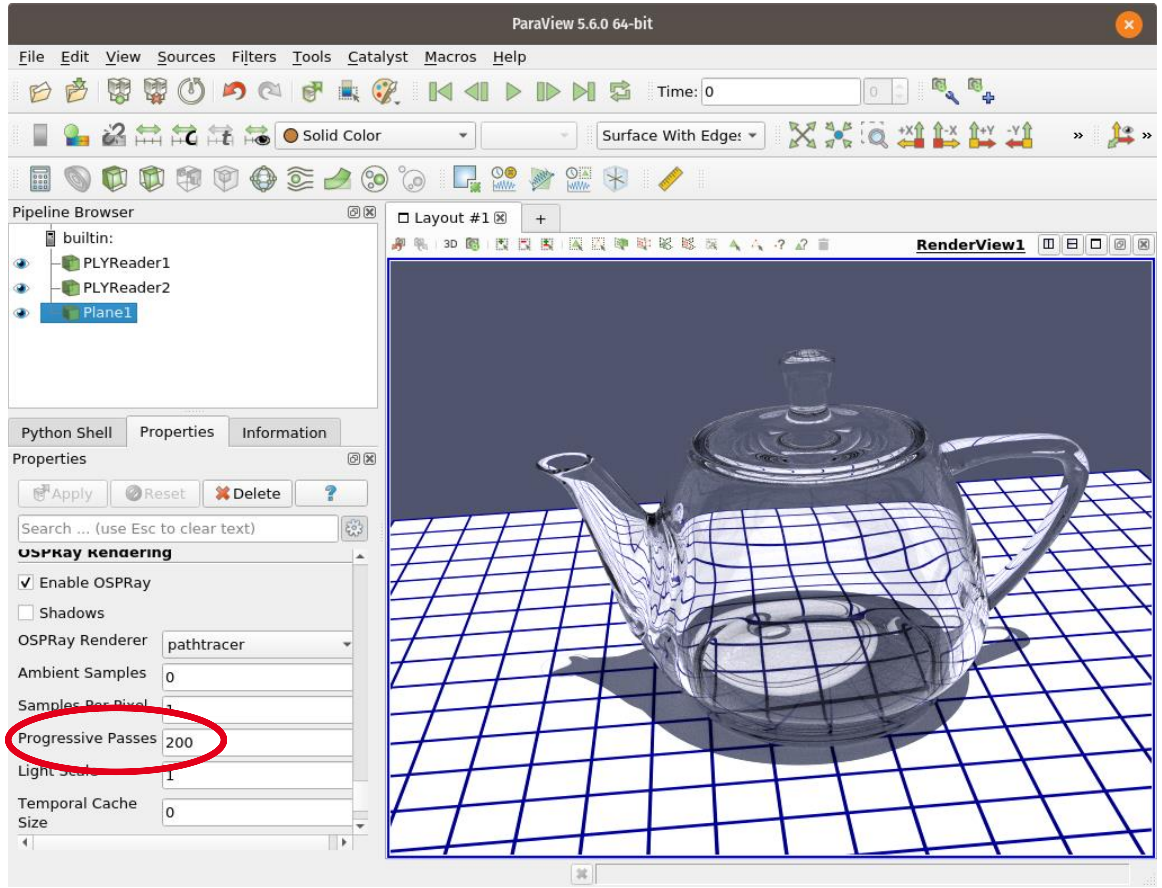
*}*

*}*



# Interactivity (progressive rendering)




Progressive rendering is accomplished by enabling streaming ( --enable-streaming) on the client side, and setting the number of progressive passes.

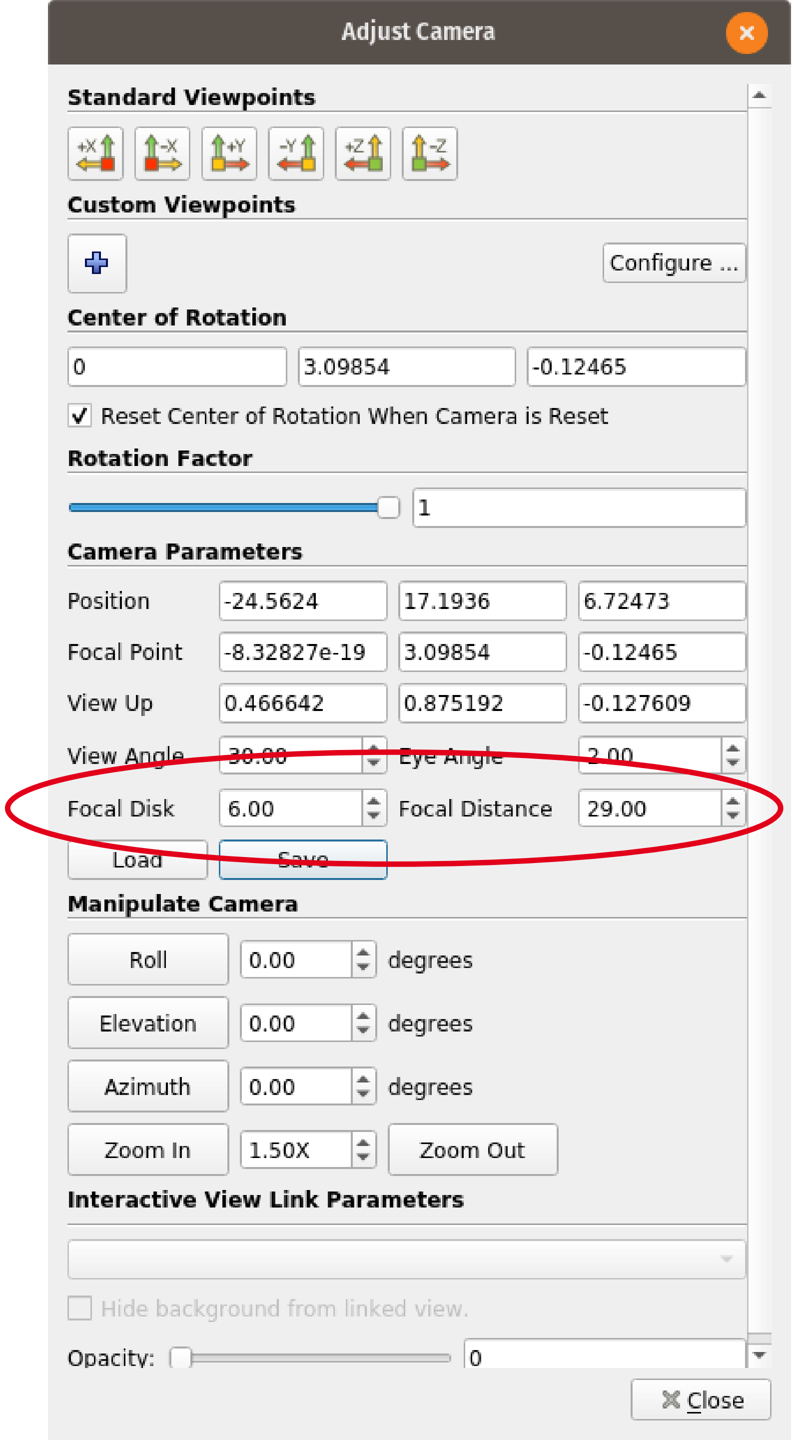
# Depth-of-Field

- To increase attention to a particular area, we can blur out data *in front-of*, and *behind* the object of interest.

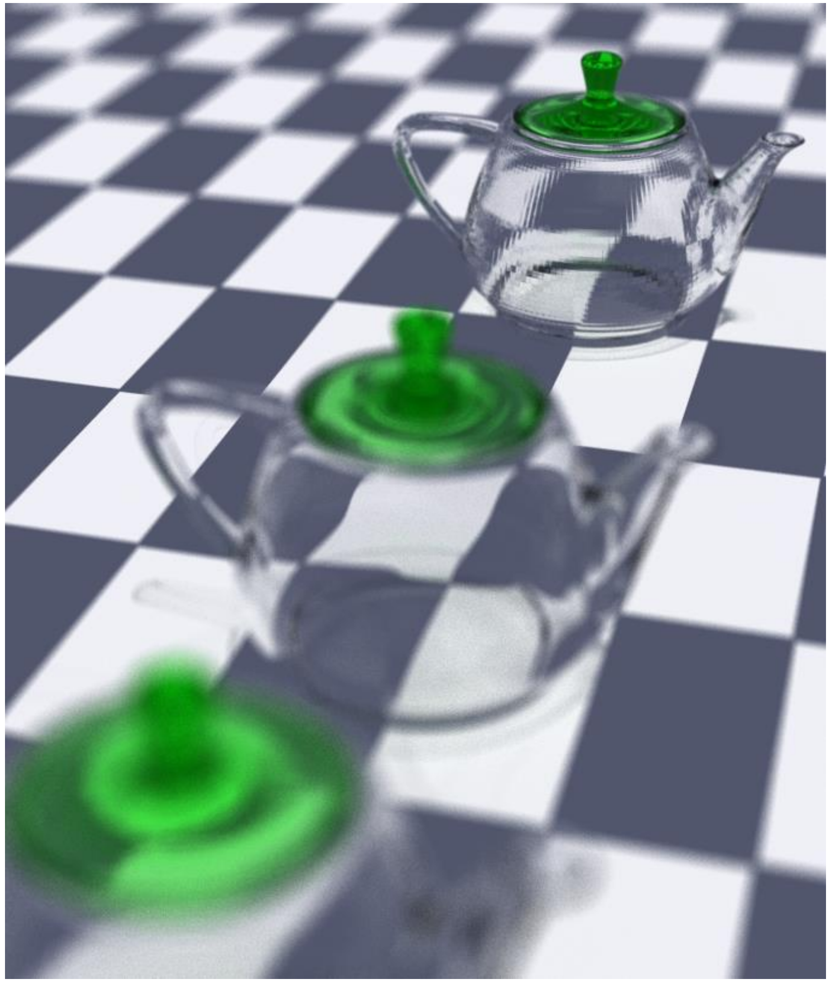
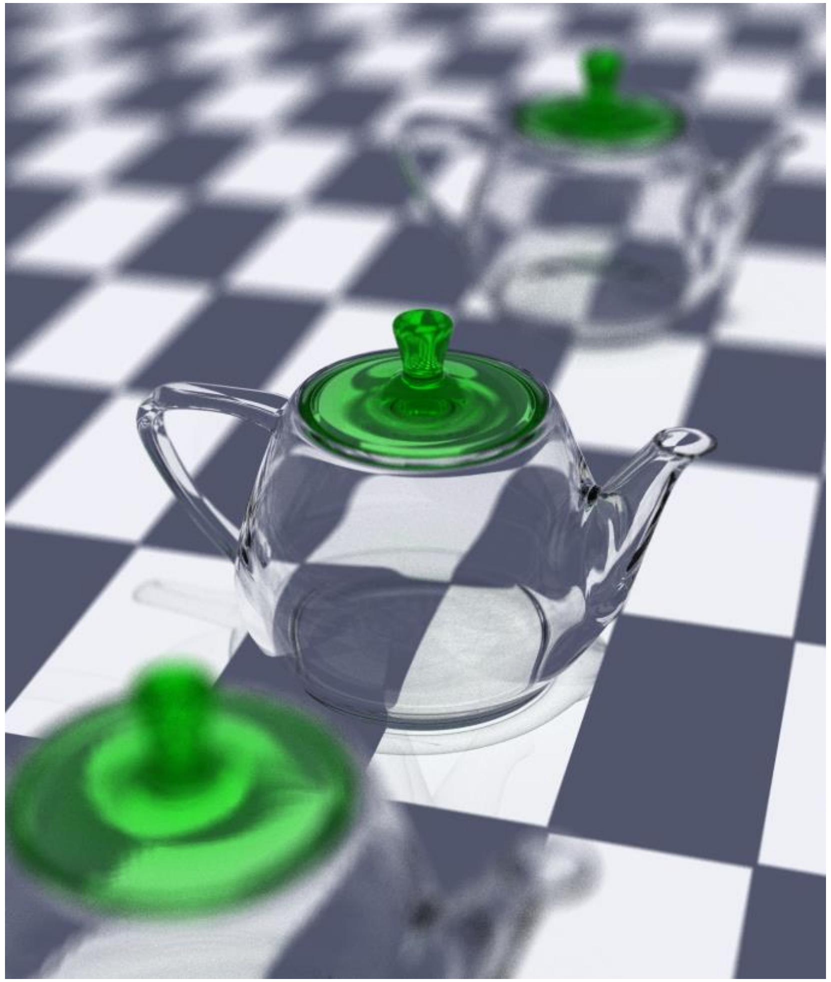
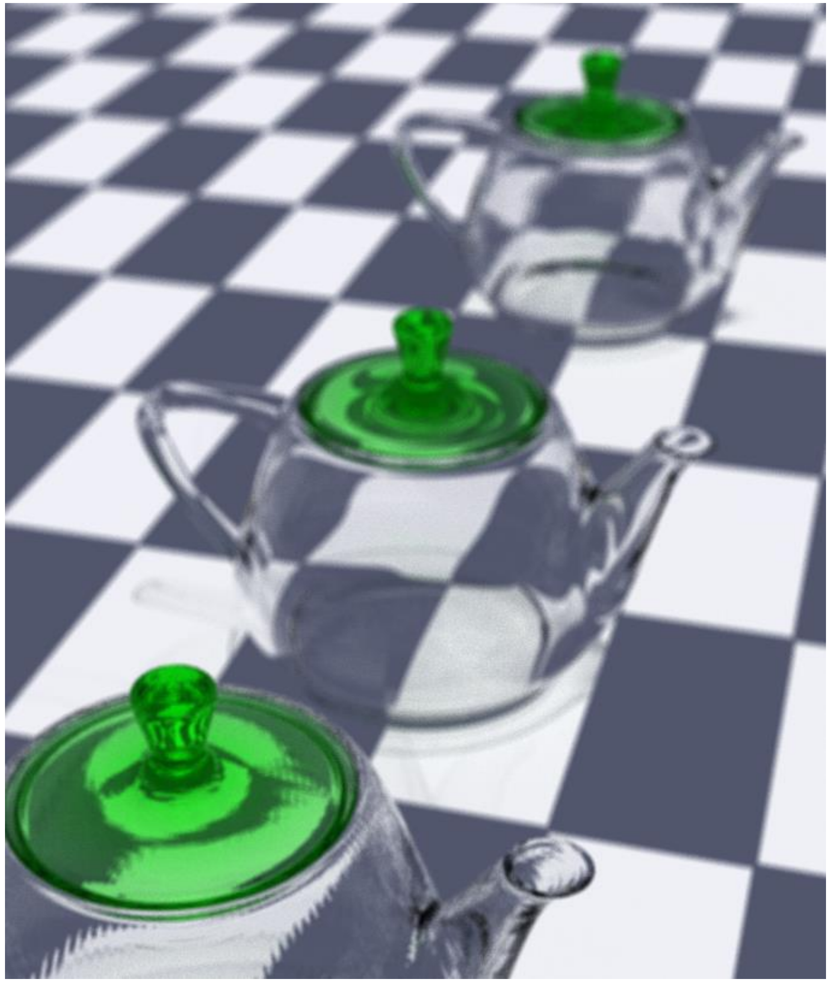
camera = GetActiveCamera()

camera.SetFocalDistance(..)

camera.SetFocalDisk(..)  # greater than 0



# Depth-of-Field

# Demonstration

Try out the jupyter notebook

- ThreeTeapots.Depth-of-field.ipynb

# Geometric Issues

RAY TRACING GEMS

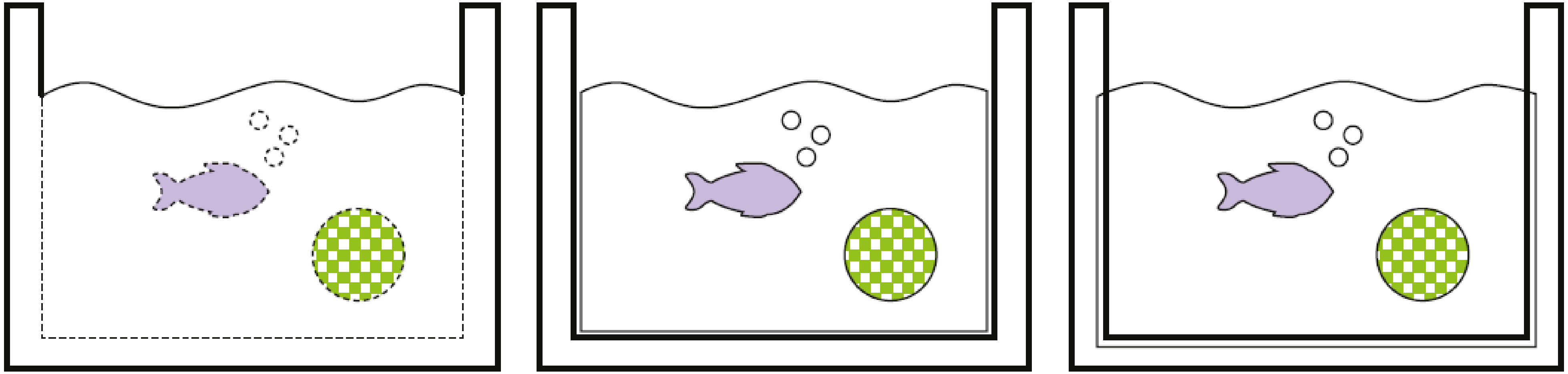


**Figure 11-1.** *Left: explicit boundary crossing of volumes marked with dashed lines. Center: air gap to avoid numerical problems. Right: overlapping volumes.*

- Unique borders ?
- Additional air gap ?
- Overlapping hulls ?

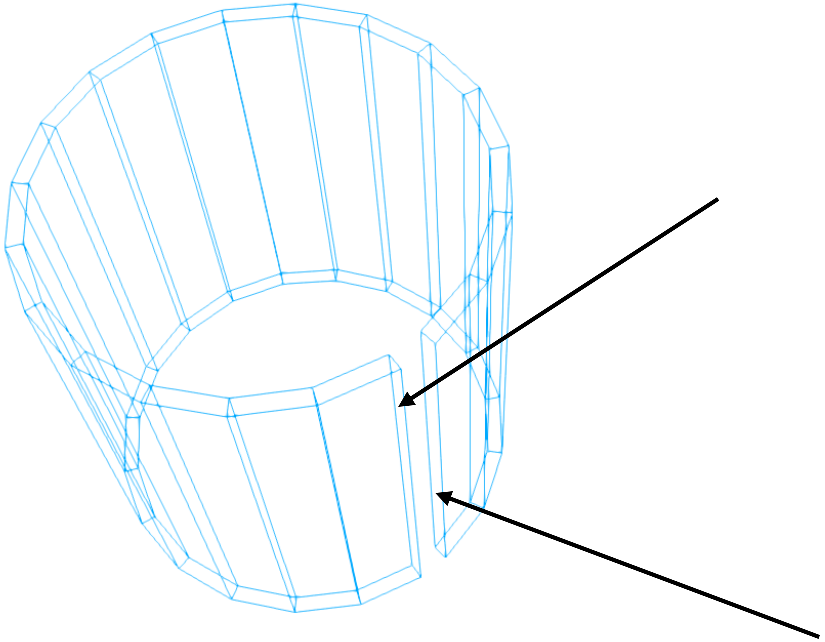
Ray tracing is always affected by the limits of the floating-point precision implementation.

Handling volume transitions robustly requires careful modelling of the volumes and their surrounding hull geometry.

# Geometric Issues

1) The cylindrical container made out of a warped thick plate

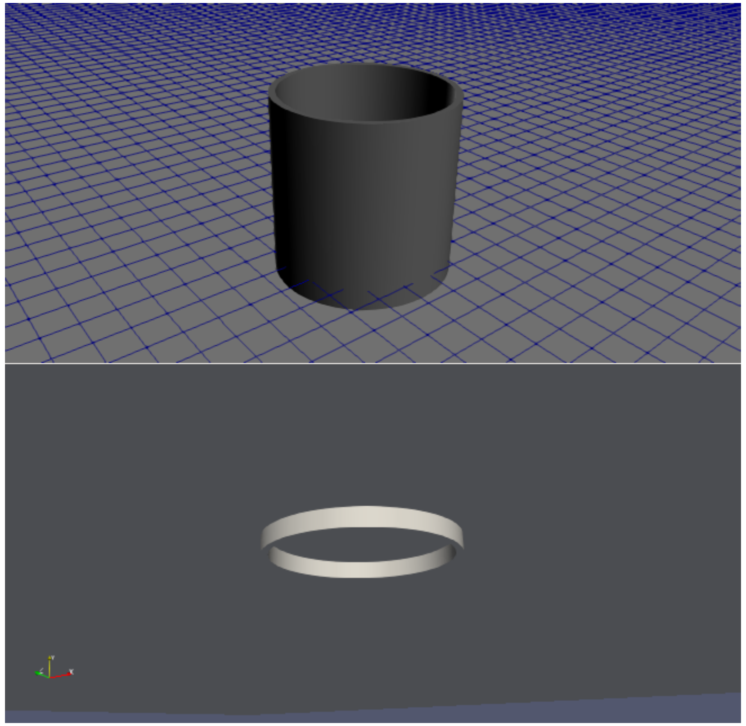
- The internal solid surfaces create artifacts and must be avoided.



2) The cup on a flat table

Advice:

Push the cup a little bit below the plate.



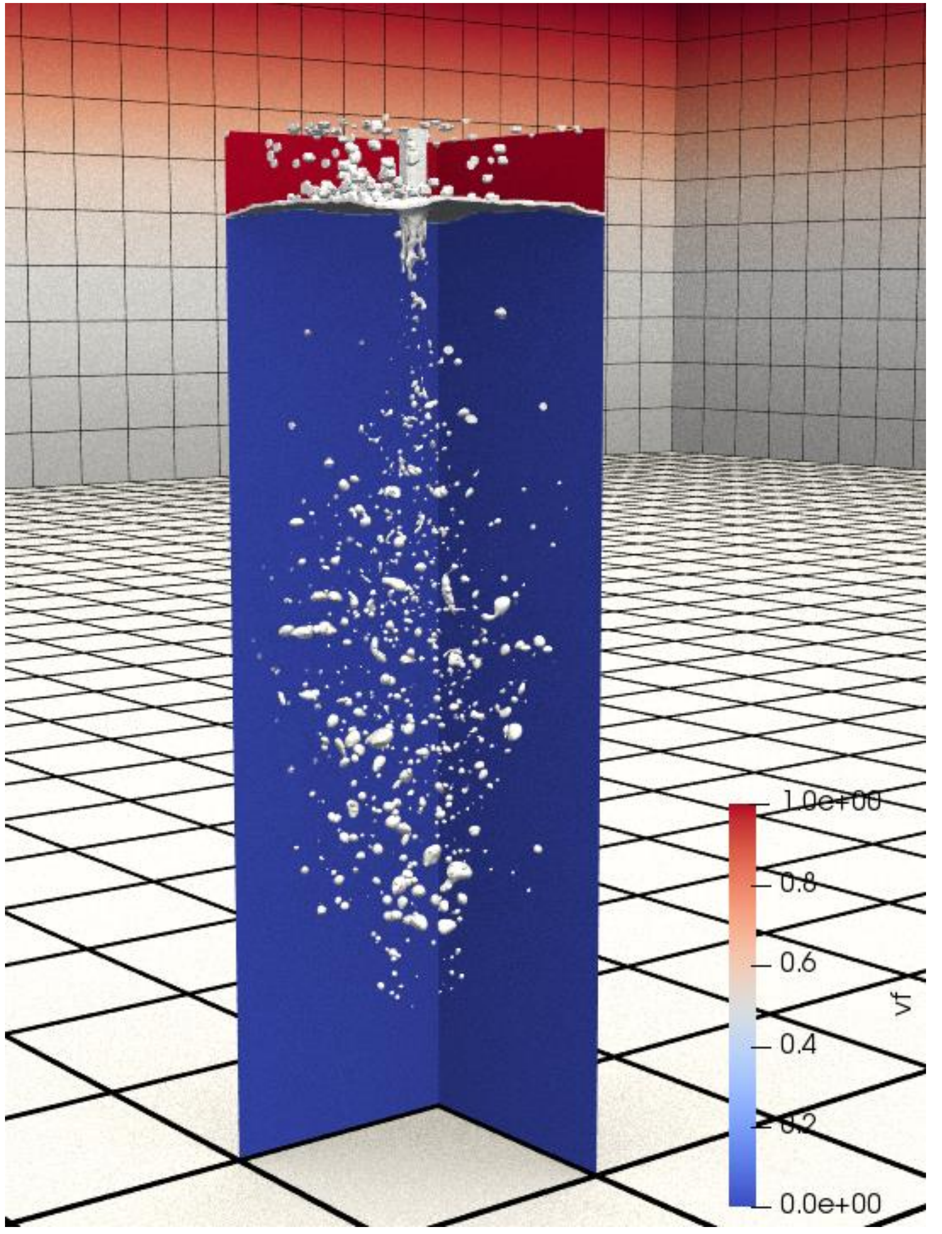
# Application: Air entrainment by a plunging jet

# Air entrainment by a plunging jet

## A simulation by Petr Karnakov, Sergey Litvinov and Petros Koumoutsakos

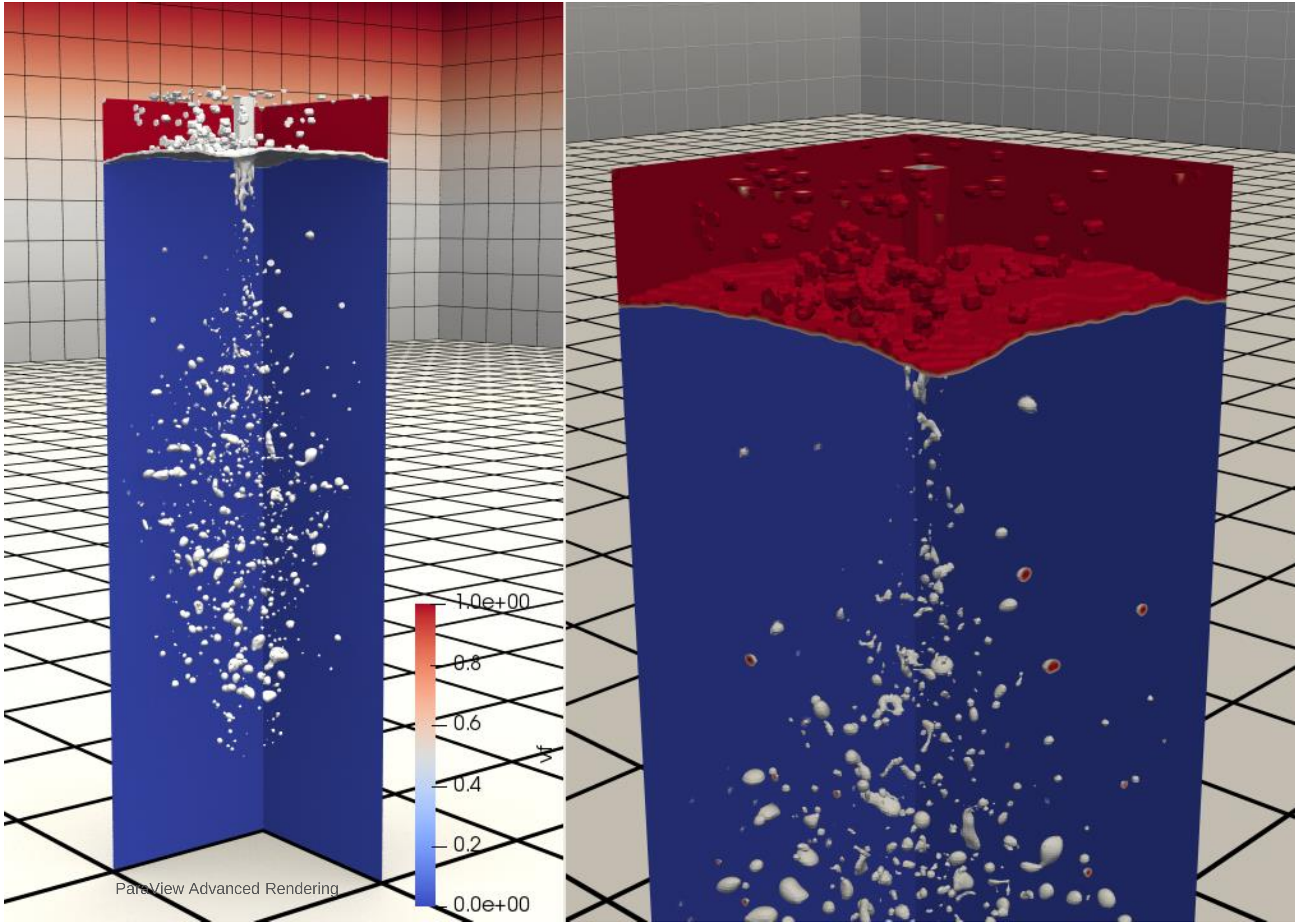
## Chair of Computational Science, ETH Zurich

***Air bubbles in a column of water***



Air bubbles are computed as *iso-volumes* of volume fraction = 0.5

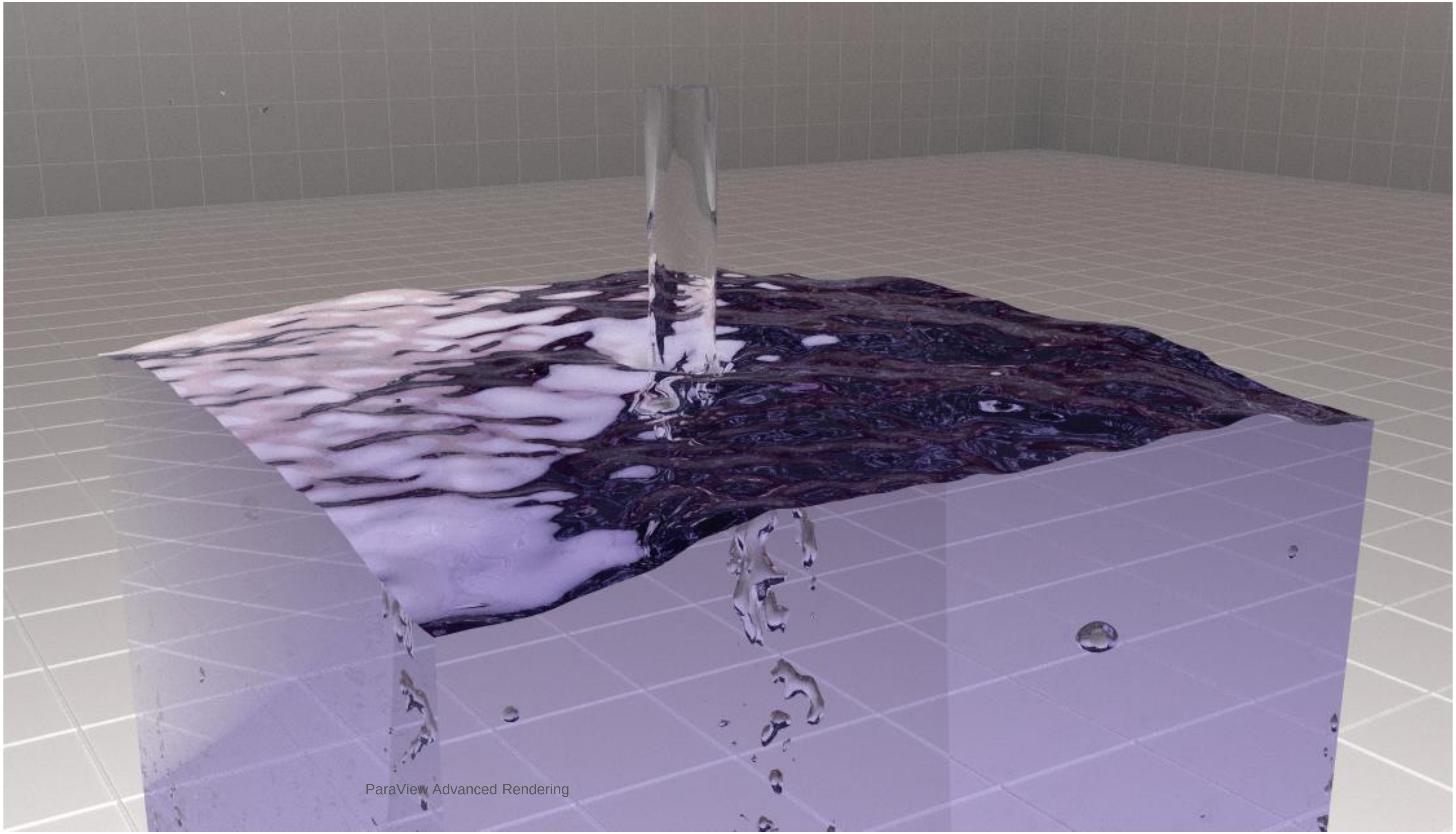
The external surface of the volume is extracted, surface normal are computed, and a “water” material is assigned for path-tracing



# The movie

Movie Link

Local link



# In practice, on Piz Daint

- Allocate a single paraview task, and the multi-threading (TBB) will do the rest.
- 72 compute threads on the nodes of the daint-mc partition

- SLURM options:

#SBATCH --cpus-per-task=72
#SBATCH --ntasks-per-core=2
#SBATCH --hint=multithread

See also the winner of the 2019 American Physics of Fluid's Gallery of Fluid Motion

# Physically based rendering (PBR)

# Introducing Physically Based Rendering with VTK

- https://blog.kitware.com/vtk-pbr/

- Replacing the classic Phong Reflectance model, with something more intuitive. I mean, how intuitive is it to specify the ambient color (RGB), diffuse color (RGB), the specular color (RGB), and the specular power (positive floating value) of an object?

- The computer-graphics experts (fans, or would-be experts) in the room will consult:
- https://learnopengl.com/PBR/Theory

- **Base Color (RGB)**: also called *albedo*, this is the perceived color of the object, the diffuse color for non-metallic objects or the specular color for metallic objects. This is set using the usual `vtkProperty::SetColor()` .
- **Metallic (float)**: in the real world, common objects are either metallic or non-metallic (called *dielectric*) and the shading computation is different depending on this parameter. For most materials, the value is either 0.0 or 1.0 but any value in between is valid. This is set using `vtkProperty::SetMetallic(value)` .
- **Roughness (float)**: parameter used to specify how an object is glossy. This is set using `vtkProperty::SetRoughness(value)`

# Image-based Lighting

- The environment is made of a cubemap texture which is a texture consisting of 6 seamless images for the 6 different directions of the 3D space

- A demo is worth 100 words…

- Try out PBR.ipynb

# End