Fluid flow visualization in virtual reality

S. V. Kampani *

Environmental Flow Physics Lab (EFPL), Columbia University,
New York, NY 10027

September 1, 2023

1 Motivation and objectives

As urban populations grow worldwide, the impact of extreme heat and air pollution increasingly weighs on society. The situation is expected to worsen due to climate change. A targeted increase in green infrastructure has been identified as an important adaptation strategy for cities. Urban trees, an example of aforementioned green infrastructure, provide a plethora of ecosystem services including air and surface temperature mitigation and reduction of airborne particulate matter. Engineers and urban planners typically rely on results from numerical models to inform the optimal siting of urban trees, but findings from these models are often in the form of 1-D charts and hence difficult to relate to the spatial distribution of green infrastructure in 3-D urban environments. This makes it difficult to communicate findings to local authorities and stakeholders.

The objective of this project is to create an immersive virtual reality (VR) environment where high-fidelity numerical simulations of airflow over urban areas can be visualized clearly. To this end, we aim to develop interactive approaches to visualization (so called "primitives") that relate numerical simulation data to the three-dimensional spatial distribution of buildings and trees. In doing so, we will provide policy makers and urban planners with an understanding of how future green infrastructural developments may influence wind, temperature, humidity, and pollution patterns.

The report is organized as follows: in Sect. 2 focuses on the methodology for the project, Sect. 3 outlines visualization primitives, and Sect. 4 describes the design of the VR environment and interface. Finally, conclusions follow in Sect. 5.

*svk2118@columbia.edu

2 Methodology

Surface representation

Before performing simulations and creating a VR environment, a high fidelity representation of the urban area of interest is required. Such a surface representation should include the spatial distribution and three-dimensional form of buildings, trees, and other infrastructure. Preliminary simulations were performed on a simplified urban environment that models buildings as a grid of cuboids and trees using a scalar-field of leaf-area-density. This simplified surface representation, however, serves only as a proof-of-concept. In the future, surface representations of urban areas will be constructed from high precision airborne light detection and ranging (LiDAR) measurements. Algorithms will be employed to classify the scanned point-cloud into vegetation and infrastructure as well as reconstruct their exact shapes. Once a suitable high fidelity surface representation is obtained, flow simulations can be run.

Large-eddy simulation (LES)

The research group of PI Giometto at Columbia University has recently developed a hyperlocal (0.5 m spatial resolution) computational fluid dynamics software that makes use of surface data to describe the three-dimensional and time-varying structure of airflow, temperature, and humidity in urban areas. This software makes use of the large-eddy simulation (LES) technique, which is often used to obtain a detailed description of microscale processes in the atmospheric boundary layer over realistic built and natural environments. LES has enabled a step forward in our current understanding of turbulent flows across many disciplines, owing to its overall higher fidelity and to the lack of tuning parameters when compared to approaches based on the Reynolds Averaged Navier-Stokes equations. LES resolves the three-dimensional unsteady turbulent flow field on a computational grid, whose resolution is dictated by available computational resources, and uses a Subgrid-Scale (SGS) model to account for the impact of unresolved SGS motions on the resolved flow field. Data binaries from simulations on urban surface representations are then loaded into and visualized in VR.

Virtual reality with Unity 3D

The virtual reality environment for visualization is developed using the Unity 3D Engine. Visualization primitives are programmed in the C# and C programming languages. Using compute shader scripts, our VR application is able to take advantage of graphics processing units (GPU's). Parallel computation enables us to load data and render visualization primitives at interactive frame-rates (72-90 FPS). By interfacing with the Ultraleap API, we are able to perform hand-tracking using a forward-facing VR headset camera. Gesture-control algorithms enable users to control and interact with the VR environment in an intuitive manner. Parallelized algorithms are also used to construct detailed meshes for trees and buildings from urban surface representations. Our VR application was tested on a LambdaLabs dual-GPU server (NVIDIA GeForce RTX 3090s).

The headset used is the Varjo XR-3, the industry's highest-resolution mixed-reality head mounted display (HMD).

3 Visualization primitives

Flow visualization is an important aspect of fluid mechanics; it makes certain characteristics of turbulent flow fields, such as coherent structures and particle transport, visually accessible. We have developed four real-time, interactive visualization primitives that make widely-used visualization techniques in fluid mechanics available in VR. The following sections discuss each primitive and its parallelized implementation.

3.1 Planar projection

A useful technique to visualize three-dimensional flow is to consider its projection onto a two-dimensional plane. Given a two-dimensional plane of arbitrary orientation and center position, the velocity of the flow is sampled at every point on the plane. The velocity is then projected onto the basis of the plane and a color based on its projected magnitude is assigned to that point on the plane. The direction of the two-dimensional projection is used to plot an arrow on the plane.

While assigning a color to every point on the plane is straightforward in the Unity Engine, plotting arrows to indicate direction is not trivial. Since the plane's color is set using a shader, we are only given control of individual pixels at a time, without being able to stamp "images" such as arrows onto the plane. To solve this problem, the signed distance field (SDF) technique was used. For each pixel, we calculate the nearest arrow to the current pixel and determine its rotation (depending on the two-dimensional planar projection of velocity). Then, we compute the signed distance to the closest point on the rotated arrow. We set the color of the pixel to black if the signed distance is negative (the pixel lies "within" the arrow) and to the color of the field if the signed distance is positive. By making a minor modification to this approach, a blur-effect (anti-aliasing technique) is also added to the edges of arrows to make them appear clearer.

Visualization in Unity Engine

The visualization algorithm described above was implemented in the Unity Engine using a shader applied to a planar triangle mesh. Rotation of the plane is left to the control of the user. Flow field data for the next timestep is loaded at the beginning of each frame and the algorithm is repeated. Users are able to view the flow projection change with real-time performance.

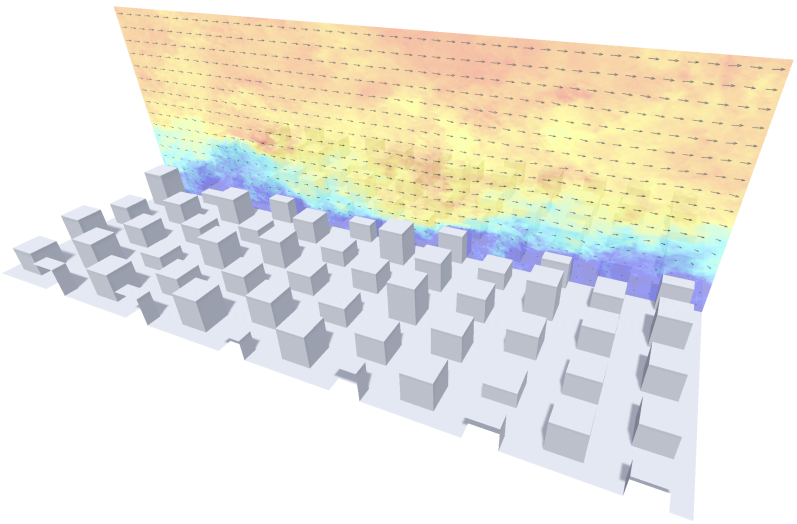

Figure 1: *Planar projection for flow over model environment*

3.2 Iso-surfaces

Iso-surfaces, or level-sets, of a scalar flow field variable are defined as the locus of all points where the scalar variable has the same value. More specifically, the k-surface of a scalar function $f(\mathbf{r}, t)$ is the solution to $f(\mathbf{r}, t) = k$. At a specific instant in time, the solution can be visualized as a surface in 3D space. Iso-surfaces of important scalar quantities, for example pressure, velocity magnitude, vorticity, may reveal flow field characteristics, and this visualization technique is helpful in identifying coherent structures in turbulent flow. We considered two methods of visualizing iso-surfaces: (i) GPU ray marching and (ii) marching cubes.

GPU ray marching

GPU ray marching is a popular rendering technique in computer graphics. Frames are rendered pixel-by-pixel and the color of each pixel is determined by a ray originating at the camera, broadcast in the direction of that pixel. Each GPU thread "marches" only a few rays. Thus, the process of rendering an entire frame is efficient, since many rays can be "marched" in parallel. When a ray corresponding to a specific pixel intersects a surface, it sets the pixel color to the color of that surface. Ray marching is then repeated for the next frame.

We can apply this technique to visualize iso-surfaces. Each ray is "marched" forwards in-space along its broadcast direction with constant steps. When the ray has intersected with an iso-surface of interest, we simply return the user-selected color of visualization and terminate the marching process. If a ray does not intersect with the iso-surface, we simply return the color of the scene.

Let f represent the scalar variable of interest at the instant during which the current ray marching operation is performed and $\mathbf{r}(s)$ represent the position of a specific ray after "step" s. We say that the ray intersects the k-surface of f at step s if:

$$f(\mathbf{r}(s-1)) \leq k \leq f(\mathbf{r}(s))$$

With our current description of ray marching, however, iso-surfaces appear uniformly colored. To indicate details of iso-surfaces, we need to add diffuse lighting and specular highlights. Let us represent colors as vectors $\in [0,1]^3$ corresponding to scaled RGB values. The updated color $\mathbf{c}_{\text{new}}$ of each pixel is given by the Phong lighting model:

$$\begin{aligned}
\mathbf{c}_{\text{new}} &= \mathbf{c}_{\text{ray}}(I_{\text{amb}} + I_{\text{diff}}) + \mathbf{c}_{\text{white}} I_{\text{spec}} \\
\mathbf{c}_{\text{ray}} &= \text{pixel color from ray marching} \\
\mathbf{c}_{\text{white}} &= (1,1,1) \\
I_{\text{amb}} &= k_a \quad \text{(constant ambient intensity)} \\
I_{\text{diff}} &= k_d(\hat{\mathbf{n}} \cdot \hat{\mathbf{l}}) \quad \text{(diffuse intensity)} \\
I_{\text{spec}} &= k_s((2(\hat{\mathbf{n}} \cdot \hat{\mathbf{l}})\,\hat{\mathbf{n}} - \hat{\mathbf{l}}) \cdot \hat{\mathbf{d}})^{\alpha} \\
k_a, k_d, k_s &\in [0,1] \quad \text{and} \quad k_a + k_d + k_s = 1
\end{aligned}$$

In the equation above, $\hat{\mathbf{l}}, \hat{\mathbf{n}}, \hat{\mathbf{d}}$ are unit vectors representing the light-direction, normal to the iso-surface at the ray's intersection point, and the ray direction respectively. While $\hat{\mathbf{l}}, \hat{\mathbf{d}}$ are known, $\hat{\mathbf{n}}$ must be calculated at intersection. Since all iso-surfaces of interest are level-sets of the scalar variable f, we have $\hat{\mathbf{n}} = \nabla f / |\nabla f|$. First-order derivatives of f are approximated by taking central-differences at the ray's point of intersection with the iso-surface. α is a specular reflection parameter that depends on the reflectivity of the surface ($\alpha \approx 10$ was chosen).

Ray marching is simple yet powerful. By making small modifications, we can produce complex visualizations. To visualize multiple, translucent iso-surfaces with limited additional computational cost, we can modify our algorithm to not terminate ray marching when we intersect with an iso-surface. Instead, we can blend the iso-surface visualization color with the current color of the pixel and continue marching. After the ray has travelled a pre-defined maximum distance, or has left the simulation domain, we can also blend the color of the scene with the pixel color. Alternatively, by accumulating color depending on the value of the scalar function itself, we can visualize "clouds" whose opacity corresponds to the scalar function value. Such a modification may be useful to visualize temperature, humidity, or concentration fields for pollutants.

A disadvantage of ray marching is the high computational cost associated with constant-step marching. Since the iso-surface of interest is not known *a priori*, each ray must propagate with small, constant steps so as to not neglect any features of the surface by overstepping. Moreover, for high-resolution VR headsets (e.g. Varjo XR-3) with high pixel-counts, the number of pixels for which ray marching needs to be performed far exceeds the thread-count of standard GPU's. In this situation, each thread is responsible

for marching several rays, which affects real-time performance. The following section discusses the marching cubes algorithm for iso-surface visualization whose computational cost does not depend on the resolution of the viewing device.

Marching cubes

The marching cubes algorithm was developed in the 1980's for constructing triangular meshes to represent iso-surfaces of scalar functions. It involves constructing a discrete rectangular grid to fill the computational domain and iterating over all "cubes" formed by grid vertices. For each "cube," the values of f are measured at each of the vertices. The triangles required to represent the part of the iso-surface passing through the "cube" are then determined by choosing one of 2^8 possible configurations. Since the configuration chosen for each "cube" depends only on the vertices of that cube, the marching cubes algorithm lends itself to parallelization.

We found that a parallelized marching cubes algorithm was more performant in visualizing iso-surfaces in real-time relative to GPU ray marching. It is also possible to adjust the resolution of individual iso-surfaces by simply changing the grid-size for each marching cubes iteration. Since the output of marching cubes is a triangular mesh, we can use Unity's complex, inbuilt lighting system rather than our own implementation of the Phong model.

Coherent structures

As mentioned previously, an important application of iso-surface visualization is showing coherent structures in turbulent flow. Coherent structures have net vorticity and tend to persist in their shape. The formal definition and identification of coherent structures is itself an area of research. One widely accepted definition is the q-criterion. The value of Q can be computed from strain-rate tensor (S) and the rotaion-rate tensor (Ω), which are the symmetric and anti-symmetric parts of the velocity gradient tensor ($\nabla \mathbf{u}$).

$$
\begin{aligned}
Q &= \frac{1}{2}\left(||\Omega||^2 - ||S||^2\right) \\
\Omega &= \frac{1}{2}\left(\nabla \mathbf{u} - \nabla \mathbf{u}^T\right) \quad \text{and} \quad S = \frac{1}{2}\left(\nabla \mathbf{u} + \nabla \mathbf{u}^T\right) \\
||M|| &= \sqrt{\mathrm{tr}(MM^T)}
\end{aligned}
$$

Q is also equal to the second principal invariant of $\nabla \mathbf{u}$. For ease of representation, we have used the following formula for the second principal invariant (I_2) of a 3×3 matrix M to compute Q from $\nabla \mathbf{u}$:

$$I_2 = M_{11}M_{22} + M_{22}M_{33} + M_{11}M_{33} - M_{12}M_{21} - M_{23}M_{32} - M_{13}M_{31}$$

The q-criterion defines coherent structures as regions where $Q > 0$. Thus, visualizing the 0-surface of Q over the computational domain would enable us to "see" the evolution and decay of coherent structures.

While implementing this visualization, we observed that the velocity flow field is quite noisy, which does not allow us to observe the hypothesized long, tube-like coherent structures due to high-wavenumber eddies. We applied a spatial low-pass filter to the field using the Gaussian convolution kernel in physical space:

$$G = \left(\frac{6}{\pi\Delta^2}\right)^{\frac{1}{2}} \exp\left(-\frac{6\mathbf{r}^2}{\Delta^2}\right)$$

To apply the continuous Gaussian kernel to our discrete grid, we approximated it as a 3-tensor. The dimensions of the 3-tensor are chosen depending on the filter width Δ. Coherent structures were observed after using filtered fields.

Visualization in Unity Engine

Using the GPU-optimized marching cubes algorithm, iso-surfaces of Q were converted to a triangle mesh topology and were visualized in the Unity Engine. Performance was improved by adjusting the triangle-count allocated to each mesh depending on the iso-value. Iso-surfaces were visualized as changing in shape over time by running the marching cubes algorithm at the beginning of each frame. Animated time-dependent iso-surfaces were visualized by this method with real-time performance.

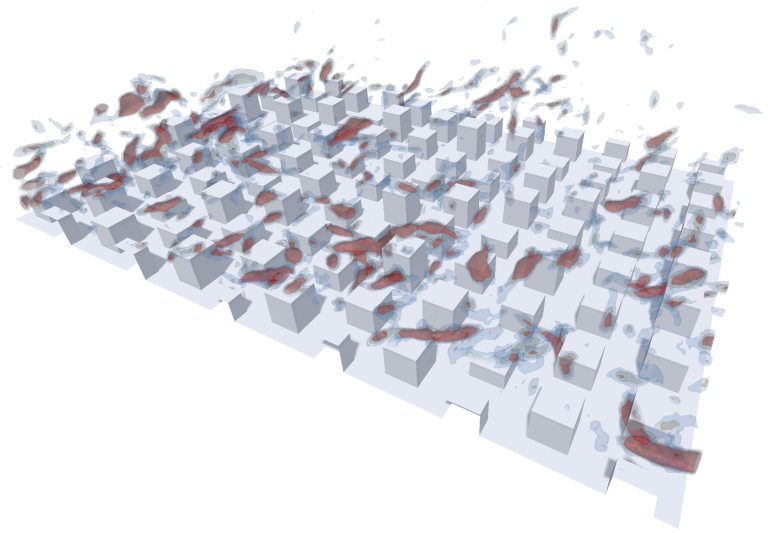

Figure 2: *Coherent structures for flow over model environment*

3.3 Streamlines

Streamlines are basic flow visualization patterns. They are defined as curves that, at any instant, are everywhere tangent to the flow velocity. To visualize a set of streamlines,

a surface is chosen (typically selected by the user in VR), a set of points uniformly distributed over the surface are chosen, streamlines through each chosen point are solved for, in parallel, using a Runge-Kutta method (RK-4) on separate GPU threads.

Visualization in Unity Engine

The streamline equation was solved numerically on the GPU using a compute shader. Streamlines were visualized as a series of tiny connected linear segments using a line mesh topology in the Unity Engine. Line mesh topologies save rendering time by skipping triangle rendering steps. However, they are not supported on all systems and hardware. On systems that do not support alternate mesh topologies, a geometry shader and "billboarding" technique are used.

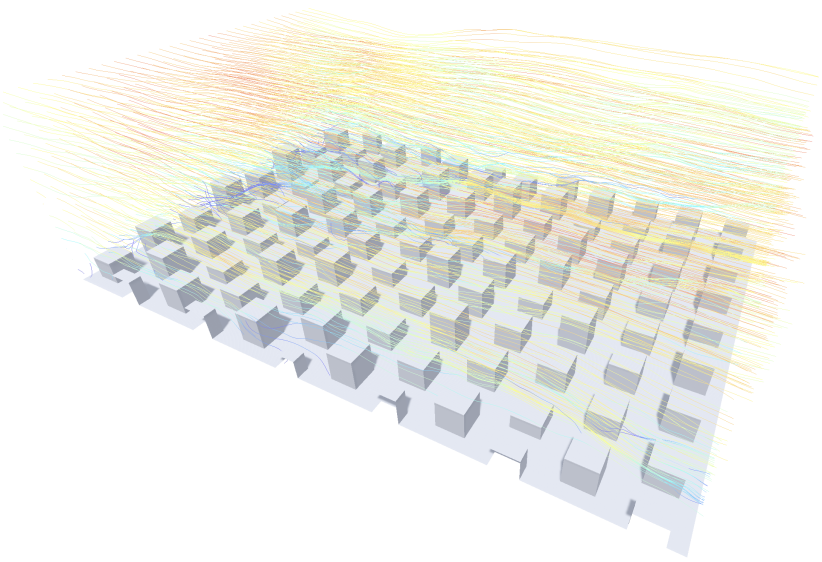

Figure 3: *Streamlines for flow over model environment*

3.4 Lagrangian particle tracking

Observing the trajectories of particles suspended in a turbulent fluid can reveal flow field characteristics. In addition to streamlines of the flow, streaklines and pathlines are common visualization techniques in the field of fluid mechanics. Pathlines are traced by the trajectories of individual particles in the fluid, whereas streaklines refer to curves formed by all particles that have previously passed through a specific point. For example, dye injected into liquids or smoke emitted in air reveal streaklines of the flow field. To visualize pathlines and streamlines, we must be able to compute the trajectories and motion of suspended particles. We will idealize suspended particles as passive inertial spheres. The motion of passive particles is governed by two interactions:

(i) Particle-particle interaction (P-P) — arises from random collisions with other suspended particles.

(ii) Particle-fluid interaction (P-F) — arises from hydrodynamic, viscous, and gravitational forces as well as random collisions with fluid molecules.

Particle-particle interaction (P-P)

An approach to modeling this interaction is to compute collisions between individual suspended particles; however, this is computationally expensive. A Fickian diffusion model is generally used to model this interaction. Since we are focused on visualizing individual particles, we will consider them in low concentration. Thus, Fickian diffusion and P-P interaction can be neglected. Thus, the motion of particles is determined solely by P-F interactions discussed in the following section.

Particle-fluid interaction (P-F)

Given our idealization of suspended particles as rigid spheres, the equation governing P-F interaction is the Basset–Boussinesq–Oseen (BBO) equation. Given our scale of interest, some terms of the BBO equation can be neglected. Squires and Yamazaki proposed a simplified version of the BBO equation as follows:

$$\frac{d\mathbf{v}}{dt} = \alpha(\mathbf{u} - \mathbf{v} - w\hat{\mathbf{k}}) + \frac{3}{2}R\,\frac{D\mathbf{u}}{Dt}$$

$$\begin{aligned}
\frac{D\mathbf{u}}{Dt} &= \frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u}\cdot\nabla)\mathbf{u} \quad \text{(material derivative)}\\
\alpha &= \frac{6\pi r_p \mu}{m_p + \frac{1}{2}m_f} \quad \text{(inertia parameter)}\\
R &= \frac{m_f}{m_p + \frac{1}{2}m_f} \quad \text{(mass ratio)}\\
w &= \frac{(m_p - m_f)g}{6\pi r_p \mu} \quad \text{(settling velocity)}\\
\mathbf{u} &= \text{flow field velocity}\\
\mathbf{v} &= \text{particle velocity}\\
m_p &= \text{mass of suspended particle}\\
m_f &= \text{mass of fluid particle}\\
r_p &= \text{radius of suspended particle}\\
\mu &= \text{dynamic viscosity}
\end{aligned}$$

It is important to note that for the BBO equation to accurately describe the motion of particles, the particle Reynolds number must satisfy $R_e < 1$. An estimate of R_e for our

application is given as follows:

$$R_e = \frac{\max\{|\mathbf{u}-\mathbf{v}|\}\cdot 2r_p}{\mu/\rho_f} \approx \frac{10\cdot 2\cdot r_p}{(10^{-5})/1}$$
$$R_e \approx 1 \implies 2\cdot r_p \approx 1\ \mu\text{m}$$

Thus, the BBO equation is best suited for describing the motion of small, micrometer-scale particles such as smoke, pollutants, and small pollen particles in air. The BBO equation is solved using a Runge-Kutta method (RK4) for every particle. Motion for each particle was solved for, in parallel, on a separate GPU thread.

A drawback of the BBO equation is that it calculates $\mathbf{u}$ by interpolating values at the nearest grid-points. Thus, it does not model subgrid-scale velocity fluctuations for suspended particles as well as random collisions with fluid molecules. Comola et. al. propose a stochastic evolution equation for the subgrid-scale velocity component. The proposed equation was simiplified further by making necessary approximations for real-time trajectory computation. By combining the simplified subgrid-scale velocity evolution equation with the BBO equation proposed earlier, we obtain:

$$\begin{aligned}
\frac{d\mathbf{v}}{dt} &= \alpha(\Delta_u - w\hat{\mathbf{k}}) + \frac{3}{2}R\,\frac{D\mathbf{u}}{Dt} \\
\Delta_u &= \mathbf{u} + \mathbf{u}_{SGS} - \mathbf{v} \\
d\mathbf{u}_{SGS} &= -k\frac{\mathbf{u}_{SGS}}{T_p}dt + \frac{1}{2}(\nabla\sigma^2)dt + \frac{2k\sigma^2}{T_p}d\eta
\end{aligned}$$

$$\sigma^2 = \frac{2}{3}\left(\frac{\epsilon\Delta}{c_\epsilon}\right)^{\frac{2}{3}} \quad \text{(SGS velocity variance)}$$
$$\epsilon \approx \frac{u_\tau^3}{(\mathbf{r}\cdot\hat{\mathbf{k}})} \quad \text{(turbulence energy dissipation rate)}$$
$$T_p = \frac{2\sigma^2}{C_0\epsilon}\left(1+\left(\frac{\beta(\mathbf{v}\cdot\hat{\mathbf{k}})}{\sigma}\right)^2\right)^{-\frac{1}{2}} \quad \text{(autocorrelation timescale)}$$
$$d\eta \sim N(0,dt) \quad \text{(normally-distributed random step)}$$

$\mathbf{r}$ = particle position
$k \in [0,1]$ = SGS fraction of turbulent kinetic energy
Δ = filter width
u_τ = average friction velocity
$c_\epsilon \approx 0.93, \quad C_0 \approx 4, \quad \beta \approx 2$

The equations above are solved using a Runge-Kutta method (RK4) for every particle. The Box-Muller transform was used to generate random, normally-distributed $d\eta$. For

the specific flow-field considered, the approximations $u_\tau \approx 1$ and $k \approx 0.3$ were deemed appropriate and were applied.

Visualization in Unity Engine

We focused on visualizing streaklines of the flow by showing the positions of several suspended moving particles. The particle motion equation was solved numerically on the GPU using a compute shader. Particles were visualized using a point mesh topology in the Unity Engine. Similar to the line mesh topology, point mesh topologies save rendering time by skipping triangle rendering steps. Again, point mesh topologies are not supported on all systems and hardware. On systems that do not support alternate mesh topologies, a geometry shader and "billboarding" technique are used. Particle positions are updated at the beginning of every frame and the visualization for 1000 particles runs with real-time performance.

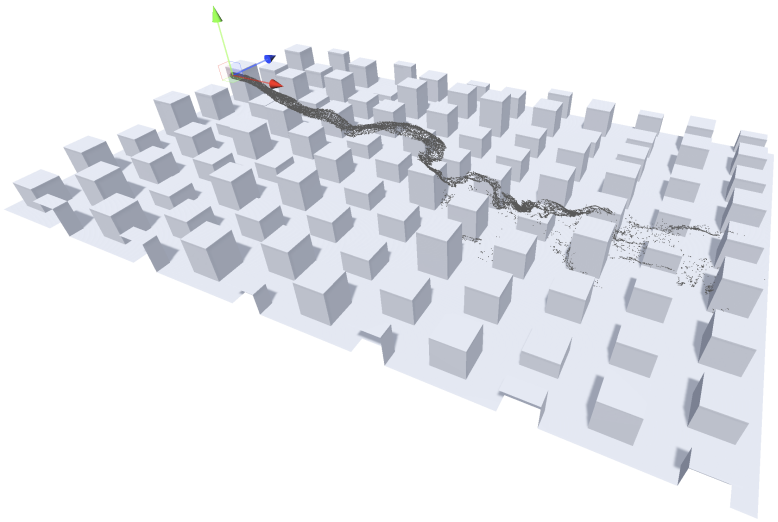

Figure 4: *Stochastic "smoke" particles released over model environment*

4 Virtual reality environment

In the previous section, we described primitives that are useful for visualizing flow field characteristics. This section details how a VR environment can be constructed from LiDAR measurements and how the aforementioned visualization primitives can be applied to simulation data in VR.

4.1 Surface representation and rendering

Performing and visualizing high-fidelity simulations of flow requires a high-fidelity representation of the urban environment surface. Such a representation can be constructed from high-resolution LiDAR measurements. Additionally, we require high-performance visualization techniques to display these environments in VR. Trees and buildings are the most important features of an urban environment's surface; hence, they will be the focus of this section.

Representing and rendering buildings

Using clustering and classification algorithms, points corresponding to trees were pruned from the point cloud scan from the LiDAR. The resulting point-cloud, which contains only buildings, was then converted to a triangle mesh using a modified marching cubes algorithm. Here are the results for a low-fidelity LiDAR scan of Columbia University's Morningside Campus:

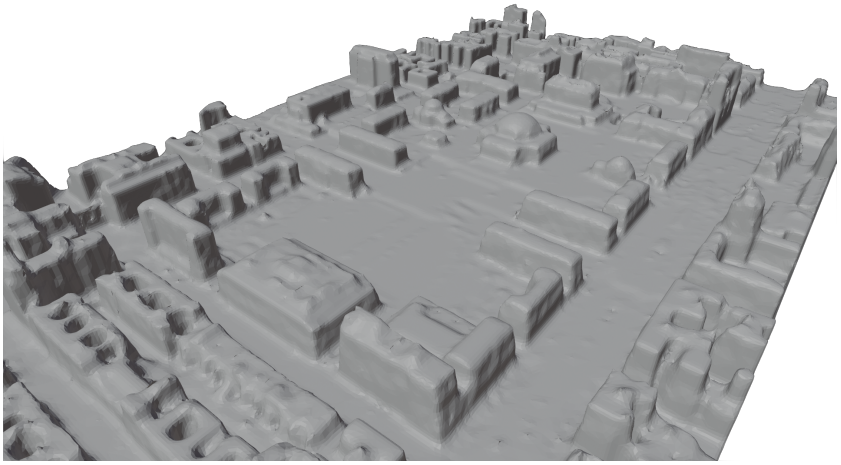

Figure 5: *Mesh reconstruction of Columbia University campus*

Representing and rendering trees

LiDAR scans of trees produce point clouds that, on their own, aren't sufficient to create a realistic representation of a tree; additional processing steps are required. In order to represent trees in LES simulations, we convert point clouds into scalar fields of leaf density based on the shapes and clusterings of points. After a discrete leaf density field is created, we need to estimate the shapes of branches to create the tree. To do so, we make use of the space colonization algorithm.

The algorithm has four main steps. (1) identify the zero iso-surface of the leaf density and randomly select several "influence" points within this surface (2) choose the root

of the tree, i.e., the starting point of our algorithm (3) add new vertices to the tree in idenitified growth directions (4) remove each "influence" point once any tree vertex is sufficiently close to it. The algorithm ends once all "influence" points have been removed.

After we have a set of vertices representing the structure of the tree and its branches, we can create a polygon mesh for the tree. This meshing operation is implemented in a separate step following the space colonization algorithm. Figure 6 (left) shows an example of a tree generated by this algorithm. In this case, the leaf density zero iso-surface is taken to be a sphere. All branches of the tree lie within this sphere and the tree itself has a realistic shape. Figure 6 (right) shows another example. In this case, two trees are generated by specifying two root points and growing two trees. Both trees are given the same set of "influence" points within a cuboid-shaped region. The trees "compete" for space and have a realistic geometry.

Figure 6: *Examples of generated trees using parallelized space-colonization*

4.2 Interaction and mixed reality

This section shows the results of applying the aforementioned visualization primitives to simulation data in interactive mixed reality environments (using the Varjo XR-3 mixed reality headset). While our approaches have currently been applied to simulations over model environments, they will soon be extended to simulations of airflow over real-world urban areas such as Columbia University's Morningside Campus, following aerial LiDAR scans of the campus by our lab (scheduled for early 2024).

Using the front-facing camera on the Varjo XR-3 headset and the Ultraleap software for hand-recognition, we implemented a system of hand-tracking to enable users to interact with visualizations by gesture-control. Figure 7 (below) visualizes the user's tracked hands and a plane primitive with the local flow velocity projection. The plane primitive is scaled, rotated depending on the position, orientation of the user's hands.

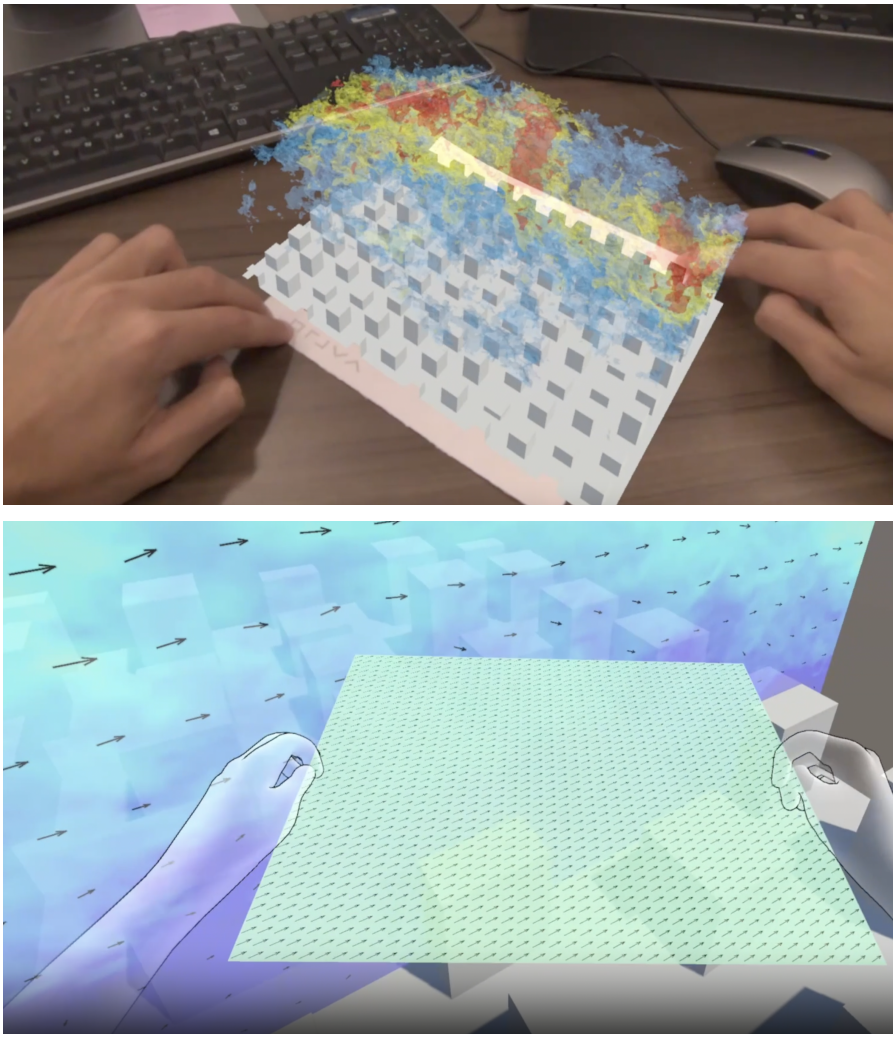

Figure 7: *Visualization of primitives in interactive AR, VR environments*

5 Conclusion

We have achieved our objective of creating an interactive mixed reality environment for visualizing fluid flow data from numerical simulations clearly. Future directions include applying our approaches to realistic urban areas, such as Columbia's Morningside Campus (following data collection in early 2024), and parameterizing uncertainties in tree-reconstruction from LiDAR scans to more accurately estimate parameters of interest, such as frontal area. Our results are currently being prepared for publication.