Lab 4: Scalar Fields and Surface Extraction

This lab introduces students to scalar field visualization through interactive volume rendering and surface extraction using K3D. Scalar fields are 3D grids where each voxel stores a scalar value, such as temperature, pressure, or density. Students will explore how to convert such data into meaningful visual representations using volume rendering and the Marching Cubes algorithm — a standard technique for extracting surfaces from volumetric data.

Students will also learn how to attach per-vertex attributes (like color or classification) to the generated surfaces and how to experiment with thresholding, non-uniform data, and animation of scalar fields.

Learning objectives

By the end of this lab, students will be able to:

- Visualize scalar fields using 3D volume rendering techniques.
- Apply the Marching Cubes algorithm to extract isosurfaces.
- Attach metadata (e.g., color, labels) to surfaces and analyze attributes.
- Compare uniform vs. non-uniform scalar data behavior.

Theoretical Background: Scalar Fields and Surface Extraction

A scalar field is a three-dimensional function that assigns a single scalar value to every point in a volumetric space. These values often represent physical quantities such as temperature, pressure, or density, and are stored in a regular 3D grid called a voxel grid. Visualizing scalar fields is essential for interpreting spatially varying data in scientific and engineering applications.

One common method of scalar field visualization is **volume rendering**, which displays the entire data volume with color and transparency to reveal internal structures. Another key technique is **surface extraction**, which involves identifying and rendering surfaces that represent locations of constant scalar value—called **isosurfaces**. The most widely used algorithm for this purpose is the **Marching Cubes algorithm**, which analyzes each voxel cell and constructs a triangulated surface where the scalar field crosses a specified threshold. This extracted surface can then be further analyzed, rendered, or colored based on additional attributes such as height, curvature, or classification labels.

Together, these techniques provide a powerful means of exploring complex volumetric data and turning abstract scalar values into interpretable 3D visual forms.

Volume Rendering

Volume rendering is a sophisticated technique in computer graphics and scientific visualization that enables the depiction of three-dimensional scalar fields by projecting them onto two-dimensional displays. This method is pivotal in fields such as medical imaging, computational fluid dynamics, and geosciences, where understanding the internal structures of volumetric data is essential.

Fundamental Concepts:

- Voxel Representation: The foundational element of volume rendering is the voxel (volumetric pixel), representing data points on a 3D grid. Each voxel contains scalar values corresponding to properties like density, temperature, or opacity.
- Transfer Functions: These functions map voxel data to optical properties such as color and transparency, facilitating the differentiation of structures within the volume. By adjusting transfer functions, specific features can be highlighted or suppressed, enhancing the interpretability of complex data.Autodesk

Rendering Techniques:

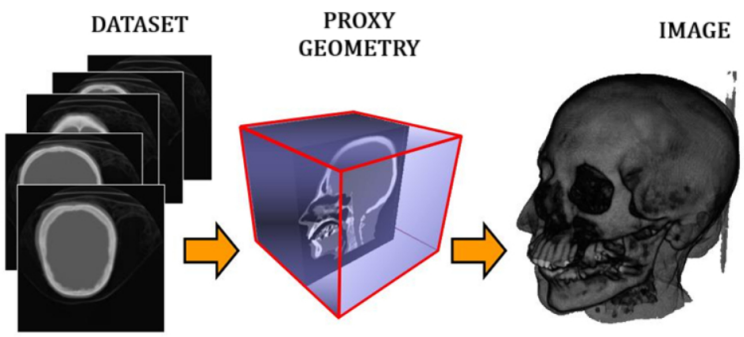
- Ray Casting: A prevalent approach where rays are projected from the viewpoint into the volume. As each ray traverses the volume, it samples data at discrete intervals, accumulating color and opacity values based on the transfer function. This accumulation determines the final pixel color displayed on the screen.
- Splatting: This technique projects voxels onto the image plane as "splats" or footprints, effectively distributing their energy contribution to the surrounding pixels. Splatting can offer performance advantages, especially in interactive applications.

Mathematical Foundation:

The process of volume rendering is governed by the Volume Rendering Equation (VRE), which models the interaction of light with the volumetric medium. The VRE accounts for emission, absorption, and scattering of light as it passes through the volume, providing a comprehensive framework for simulating realistic visualizations.

Applications:

- Medical Imaging: Volume rendering is extensively used to visualize complex anatomical structures from imaging modalities like MRI and CT scans, aiding in diagnosis and surgical planning.
- Scientific Research: In fields such as astrophysics and meteorology, volume rendering facilitates the exploration of phenomena like nebulae formations or storm systems by providing intuitive visual representations of simulation data.



By leveraging volume rendering techniques, professionals can gain deeper insights into three-dimensional data, leading to more informed analyses and decisions across various scientific and medical domains

Therefore, volume rendering is a technique used to visualize 3D scalar fields without explicitly extracting geometric surfaces. Unlike traditional surface rendering, which relies on creating mesh representations of data (e.g., via the Marching Cubes algorithm), volume rendering works by projecting and accumulating data values directly along the viewer's line of sight. Each voxel in the volume contributes color and opacity based on its scalar value and its position within the volume.

This technique simulates the behavior of light as it passes through semi-transparent materials, making it ideal for visualizing complex internal structures within a dataset, such as organs in medical imaging or temperature gradients in fluid simulations. Transfer functions are typically used to map scalar values to color and transparency, allowing users to highlight specific value ranges and reveal hidden structures.

Volume rendering is particularly powerful because it can represent subtle, continuous variations in data and visualize nested features within a scalar field, all without the need for surface extraction or segmentation.

Surface Extraction

Surface extraction is the process of identifying and generating geometric representations of boundaries within volumetric scalar fields. These boundaries, known as **isosurfaces**, represent points in 3D space where the scalar field equals a constant value, such as a specific temperature or pressure threshold. Surface extraction provides a way to simplify and interpret volumetric data by transforming it into a tangible, viewable surface.

One of the most commonly used algorithms for surface extraction is the **Marching Cubes algorithm**, which systematically examines small cubes of adjacent voxels in the 3D grid. It then determines where the isosurface intersects each cube by comparing scalar values at the cube's corners to the desired threshold. Based on predefined patterns, it generates triangles that approximate the surface within each cube. The final result is a smooth mesh that visually represents a continuous surface embedded in the volume.

Surface extraction is widely used in medical imaging (e.g., CT or MRI scans), fluid simulations, and scientific visualization to make complex data comprehensible and suitable for further analysis or 3D rendering.

Marching Cubes Algorithm

The Marching Cubes algorithm is a fundamental technique used in computer graphics and scientific visualization to extract isosurfaces from volumetric scalar fields. Developed by Lorensen and Cline in 1987, the algorithm processes a 3D scalar field by iterating over each cube-shaped group of eight neighboring voxels (a "voxel cube") in the grid. For each cube, the scalar values at the corners are compared to a user-defined threshold (isosurface level) to determine where the surface intersects the cube.

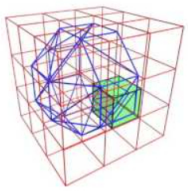
There are 256 possible configurations of inside/outside states for the cube's corners, but due to symmetries, these can be reduced to 15 unique cases. A lookup table defines how to triangulate each case, allowing the algorithm to generate a set of triangles that approximate the isosurface within each cube. The result is a smooth, continuous surface mesh that follows the contours of the data.

Marching Cubes is widely used in medical imaging (e.g., reconstructing anatomical surfaces from MRI or CT data), simulation data analysis, and 3D modeling. It is favored for its efficiency, simplicity, and ability to produce high-quality surfaces from complex volumetric data.

Check out this video to see how Marching Cube algorithm works:

[redacted]

Implementation



[redacted]

---

[8]:
```
!pip install k3d
```

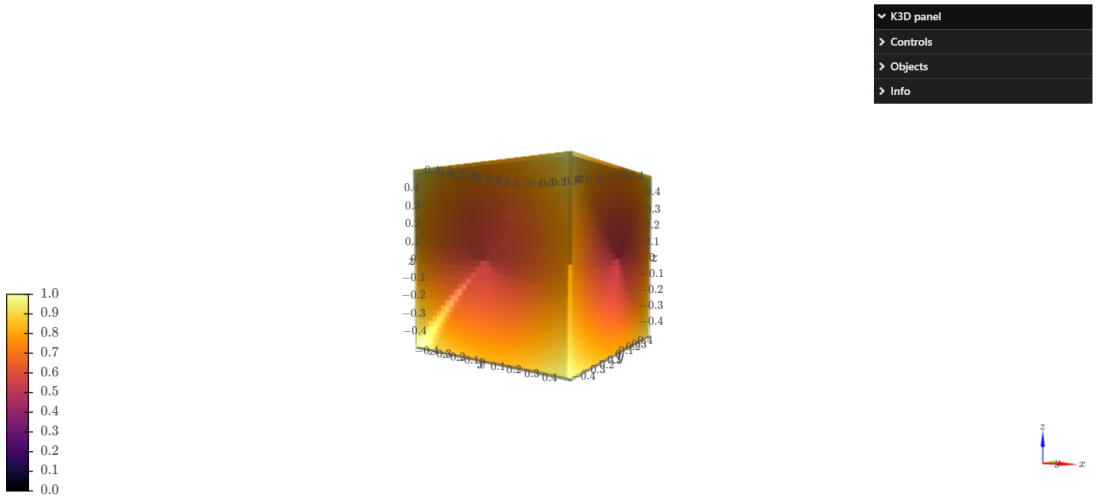
```
Requirement already satisfied: k3d in c:\users\arija\anaconda3\lib\site-packages (2.16.1)
Requirement already satisfied: ipywidgets<9.0.0,>=7.0.0 in c:\users\arija\anaconda3\lib\site-packages (from k3d) (8.1.5)
Requirement already satisfied: msgpack in c:\users\arija\anaconda3\lib\site-packages (from k3d) (1.0.3)
Requirement already satisfied: numpy in c:\users\arija\anaconda3\lib\site-packages (from k3d) (1.26.4)
Requirement already satisfied: traitlets in c:\users\arija\anaconda3\lib\site-packages (from k3d) (5.14.3)
Requirement already satisfied: traittypes in c:\users\arija\anaconda3\lib\site-packages (from k3d) (0.2.1)
Requirement already satisfied: comm>=0.1.3 in c:\users\arija\anaconda3\lib\site-packages (from ipywidgets<9.0.0,>=7.0.0->k3d) (0.2.1)
Requirement already satisfied: ipython>=6.1.0 in c:\users\arija\anaconda3\lib\site-packages (from ipywidgets<9.0.0,>=7.0.0->k3d) (8.27.0)
Requirement already satisfied: widgetsnbextension~=4.0.12 in c:\users\arija\anaconda3\lib\site-packages (from ipywidgets<9.0.0,>=7.0.0->k3d) (4.0.13)
Requirement already satisfied: jupyterlab-widgets~=3.0.12 in c:\users\arija\anaconda3\lib\site-packages (from ipywidgets<9.0.0,>=7.0.0->k3d) (3.0.13)
Requirement already satisfied: decorator in c:\users\arija\anaconda3\lib\site-packages (from ipython>=6.1.0->ipywidgets<9.0.0,>=7.0.0->k3d) (5.1.1)
Requirement already satisfied: jedi>=0.16 in c:\users\arija\anaconda3\lib\site-packages (from ipython>=6.1.0->ipywidgets<9.0.0,>=7.0.0->k3d) (0.19.1)
Requirement already satisfied: matplotlib-inline in c:\users\arija\anaconda3\lib\site-packages (from ipython>=6.1.0->ipywidgets<9.0.0,>=7.0.0->k3d) (0.1.6)
Requirement already satisfied: prompt-toolkit<3.1.0,>=3.0.41 in c:\users\arija\anaconda3\lib\site-packages (from ipython>=6.1.0->ipywidgets<9.0.0,>=7.0.0->
k3d) (3.0.43)
Requirement already satisfied: pygments>=2.4.0 in c:\users\arija\anaconda3\lib\site-packages (from ipython>=6.1.0->ipywidgets<9.0.0,>=7.0.0->k3d) (2.15.1)
Requirement already satisfied: stack-data in c:\users\arija\anaconda3\lib\site-packages (from ipython>=6.1.0->ipywidgets<9.0.0,>=7.0.0->k3d) (0.2.0)
Requirement already satisfied: colorama in c:\users\arija\anaconda3\lib\site-packages (from ipython>=6.1.0->ipywidgets<9.0.0,>=7.0.0->k3d) (0.4.6)
Requirement already satisfied: parso<0.9.0,>=0.8.3 in c:\users\arija\anaconda3\lib\site-packages (from jedi>=0.16->ipython>=6.1.0->ipywidgets<9.0.0,>=7.0.0
->k3d) (0.8.3)
Requirement already satisfied: wcwidth in c:\users\arija\anaconda3\lib\site-packages (from prompt-toolkit<3.1.0,>=3.0.41->ipython>=6.1.0->ipywidgets<9.0.0,
>=7.0.0->k3d) (0.2.5)
Requirement already satisfied: executing in c:\users\arija\anaconda3\lib\site-packages (from stack-data->ipython>=6.1.0->ipywidgets<9.0.0,>=7.0.0->k3d) (0.
8.3)
Requirement already satisfied: asttokens in c:\users\arija\anaconda3\lib\site-packages (from stack-data->ipython>=6.1.0->ipywidgets<9.0.0,>=7.0.0->k3d) (2.
0.5)
Requirement already satisfied: pure-eval in c:\users\arija\anaconda3\lib\site-packages (from stack-data->ipython>=6.1.0->ipywidgets<9.0.0,>=7.0.0->k3d) (0.
2.2)
Requirement already satisfied: six in c:\users\arija\anaconda3\lib\site-packages (from asttokens->stack-data->ipython>=6.1.0->ipywidgets<9.0.0,>=7.0.0->k3
d) (1.16.0)
```

[9]:
```
import k3d
import numpy as np
```

Example 1: Volume Rendering of a Scalar Field (Stable)

This example demonstrates how to visualize the internal structure of a 3D scalar field using volume rendering — a technique that does not require surface extraction or meshing. Instead, each voxel in the 3D grid contributes color and opacity to the final image, depending on its scalar value. This allows users to see inside the volume and observe how values are distributed throughout the space. In this case, a spherical scalar gradient is used to simulate a volumetric density field. The result is a semi-transparent rendering where high-density regions near the center appear more prominently, and lower-density regions fade into the background. Volume rendering is particularly effective for representing medical scans, fluid dynamics, or any field with smoothly varying internal properties.

[11]:
```
import k3d
import numpy as np

# Create a scalar field (a spherical gradient)
size = 32
x, y, z = np.indices((size, size, size))
center = size // 2
field = np.sqrt((x - center)**2 + (y - center)**2 + (z - center)**2)

# Normalize to 0-1
field = (field - field.min()) / field.ptp()

plot = k3d.plot()
plot += k3d.volume(field.astype(np.float32))
plot.display()
```

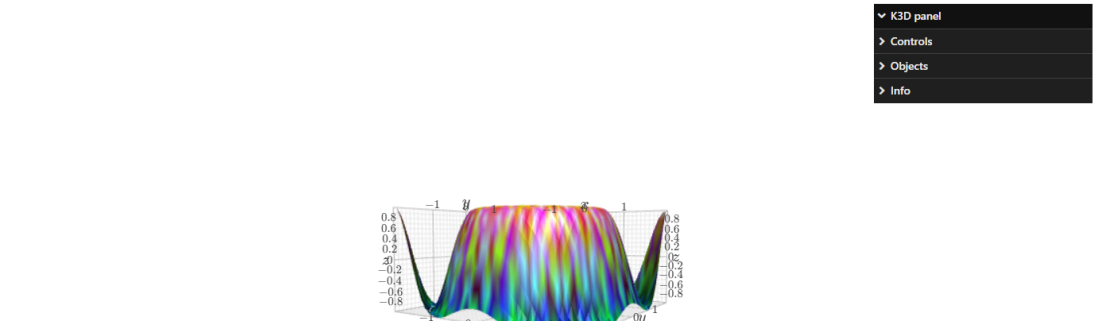


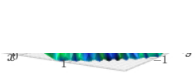
Example 2: Extract and Color a Wavy Surface (Mesh-Based, No Marching Cubes)

This example shows how to visualize a continuous 3D surface by directly generating a mesh from a mathematical function, without relying on volumetric data or the Marching Cubes algorithm. A 2D grid of points is used to define the X and Y positions, while the Z-values are calculated using a wave-like function such as `z = sin(x² + y²)`. These points are then connected into triangles to form a mesh. To enhance the visualization, each vertex is colored based on its height (Z-value), creating a smooth color gradient that reflects the shape of the surface. This technique is efficient, flexible, and ideal for visualizing mathematical surfaces, terrain models, or scalar fields sampled on a regular grid.

Now, we will generate a wavy surface from a 2D grid and assign colors based on height (Z).

[13]:
```
import k3d
import numpy as np

# Create a wavy 2D scalar field
x, y = np.meshgrid(np.linspace(-2, 2, 40), np.linspace(-2, 2, 40))
z = np.sin(x**2 + y**2)

vertices = np.stack((x.ravel(), y.ravel(), z.ravel()), axis=1)

# Generate triangle indices
faces = []
res = x.shape[0]
for i in range(res - 1):
    for j in range(res - 1):
        idx = i * res + j
        faces.append([idx, idx + 1, idx + res])
        faces.append([idx + 1, idx + res + 1, idx + res])
faces = np.array(faces, dtype=np.uint32)

# Normalize Z-values and map them to vertex colors
z_flat = z.ravel()
z_norm = (z_flat - z_flat.min()) / z_flat.ptp()
colors = (z_norm * 0xFFFFFF).astype(np.uint32)

plot = k3d.plot()
plot += k3d.mesh(vertices=vertices.astype(np.float32), indices=faces, colors=colors)
plot.display()
```



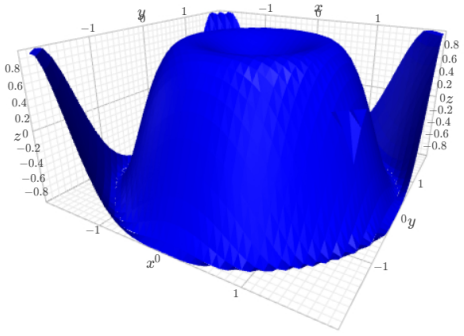


Example 3: Animate Scalar Field Over Time

Animating a scalar field over time allows for dynamic visualization of how data evolves in a 3D space. In this example, a time-dependent mathematical function, such as a sine wave, is used to update the scalar values across a mesh. At each time step, the Z-values of the surface change, simulating motion like rippling water or a breathing membrane. By updating both the geometry and the color of the mesh in real-time, students can observe how the scalar field's structure changes and flows. This approach is especially useful in simulations where values fluctuate over time, such as wave propagation, heat diffusion, or vibration analysis, offering an engaging and intuitive way to explore time-varying data.

[15]:
```
import time

plot = k3d.plot()

# Static X, Y mesh
x, y = np.meshgrid(np.linspace(-2, 2, 40), np.linspace(-2, 2, 40))
res = x.shape[0]
faces = []
for i in range(res - 1):
    for j in range(res - 1):
        idx = i * res + j
        faces.append([idx, idx + 1, idx + res])
        faces.append([idx + 1, idx + res + 1, idx + res])
faces = np.array(faces, dtype=np.uint32)

# Initial Z
z = np.sin(x**2 + y**2)
vertices = np.stack((x.ravel(), y.ravel(), z.ravel()), axis=1)

mesh = k3d.mesh(vertices.astype(np.float32), indices=faces)
plot += mesh
plot.display()

# Animate wave
for t in range(60):
    z = np.sin(x**2 + y**2 - t * 0.2)
    vertices = np.stack((x.ravel(), y.ravel(), z.ravel()), axis=1)
    mesh.vertices = vertices.astype(np.float32)
    time.sleep(0.05)
```

K3D panel
Controls
Objects
Info

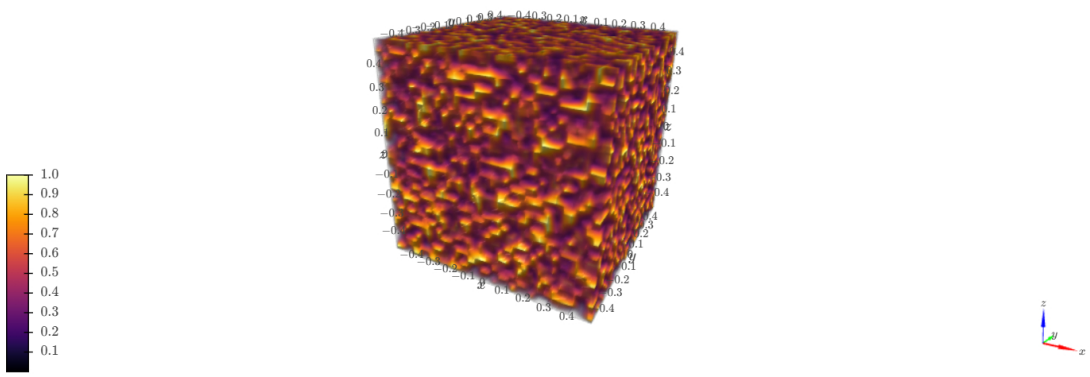


Example 4: Volume Rendering with Noise-Based Scalar Field

This example demonstrates how to visualize a 3D scalar field generated from random noise using volume rendering. Instead of a smooth, structured function, the scalar values in this case are assigned randomly throughout the voxel grid, simulating a cloud-like or turbulent distribution. When rendered with semi-transparency, this creates a complex, organic volume where denser regions appear more opaque and lighter regions fade away. This approach is commonly used in visual effects, procedural content generation, and scientific simulations involving chaotic systems. It allows students to explore how irregular or unstructured data can still produce meaningful visual patterns through volume rendering techniques.

[17]:
```
import k3d
import numpy as np

# Random noise scalar field
size = 32
field = np.random.rand(size, size, size).astype(np.float32)

plot = k3d.plot()
plot += k3d.volume(field)
plot.display()
```

K3D panel
Controls
Objects
Info

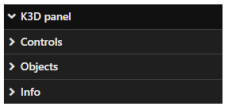


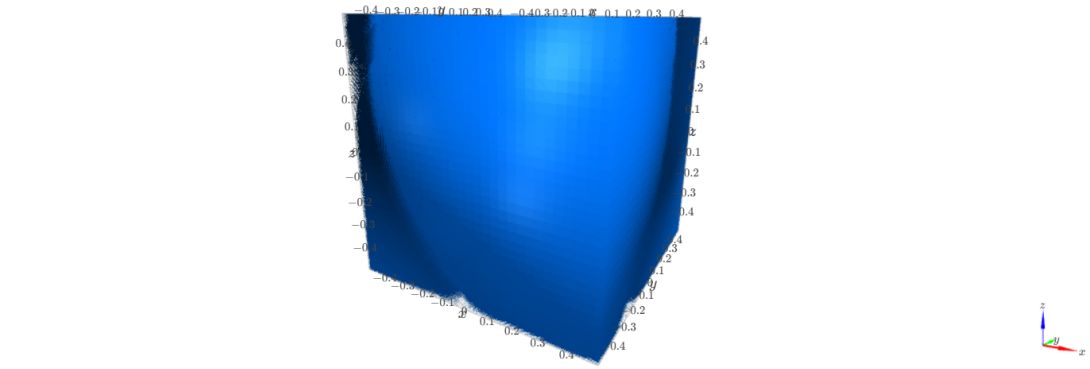
Example 5: Animated Isosurface from a Dynamic Scalar Field

This example illustrates how to create an animated isosurface by extracting it from a time-varying 3D scalar field. A mathematical function, such as a sinusoidal expression involving both space and time, is used to define the scalar values within a voxel grid. As time progresses, the scalar field evolves, and a new isosurface is extracted at each frame using the Marching Cubes algorithm. The result is a smooth, continuous surface that changes shape over time, creating a visually compelling animation. This technique is ideal for simulating dynamic phenomena such as wave propagation, fluid flow, or morphing shapes, and provides a powerful way to visualize how internal structures evolve in response to changing scalar values.

In the following example we will animate a changing scalar field and re-extract the surface using Marching Cubes at each step.

[19]:
```python
import k3d
import numpy as np
import time

# Create initial scalar field (sine wave in 3D space)
size = 32
x, y, z = np.meshgrid(
    np.linspace(-2, 2, size),
    np.linspace(-2, 2, size),
    np.linspace(-2, 2, size),
    indexing='ij'
)

# Create K3D plot
plot = k3d.plot()
iso_surface = None  # Placeholder for our animated surface
plot.display()

# Animation loop (generate changing field & re-apply marching cubes)
for t in range(50):
    scalar_field = np.sin(x**2 + y**2 + z**2 - t * 0.1)

    # Normalize for stability
    field_norm = (scalar_field - scalar_field.min()) / scalar_field.ptp()

    # Extract isosurface at level = 0.5
    new_surface = k3d.marching_cubes(
        field_norm.astype(np.float32),
        level=0.5,
        color=0x0077ff,
        wireframe=False
    )

    # Remove old surface and add new one
    if iso_surface:
        plot -= iso_surface
    iso_surface = new_surface
    plot += iso_surface

    time.sleep(0.1)
```



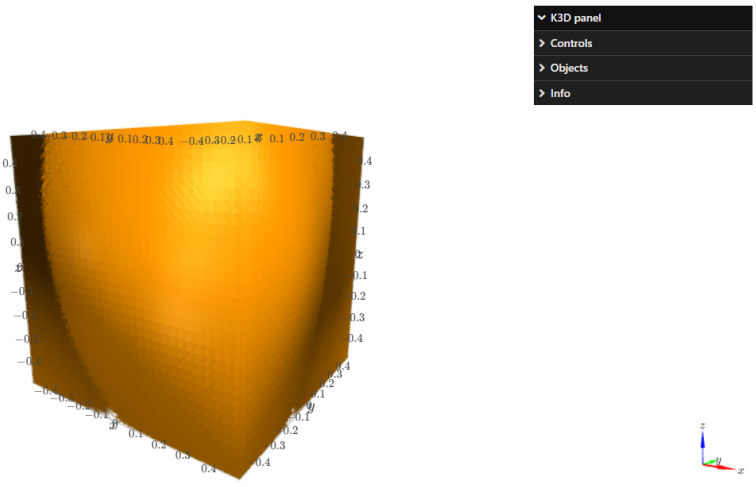


Example 6: Animated Isosurface Evolution

Animated isosurface evolution is a dynamic visualization technique that shows how surfaces derived from a scalar field change over time. By continuously updating the scalar field—using mathematical functions or simulation data—and applying the Marching Cubes algorithm at each frame, the isosurface can be made to appear as if it's morphing, pulsating, or flowing. This evolving surface helps convey complex behaviors like deformation, diffusion, or cyclic phenomena. The smooth transitions between frames provide an intuitive understanding of temporal changes in 3D data, making this technique especially useful for educational demonstrations, scientific simulations, and animated visual storytelling.

This example animates a dynamic scalar field and visualizes it using the Marching Cubes algorithm. It's a great wrap-up exercise combining volume generation, isosurface extraction, animation, and visual storytelling.

[21]:
```python
import k3d
import numpy as np
import time

# Create a 3D grid (volumetric space)
size = 32
x, y, z = np.meshgrid(
    np.linspace(-3, 3, size),
    np.linspace(-3, 3, size),
    np.linspace(-3, 3, size),
    indexing='ij'
)

# Initialize the plot
plot = k3d.plot()
plot.display()

# Placeholder for dynamic surface object
iso_surface = None

# Animate over time
for t in range(50):
    # Generate a dynamic scalar field using trigonometric wave
    scalar_field = np.sin(x**2 + y**2 + z**2 - t * 0.2)

    # Normalize values between 0 and 1
    scalar_field = (scalar_field - scalar_field.min()) / (scalar_field.ptp() + 1e-8)

    # Extract isosurface using Marching Cubes
    surface = k3d.marching_cubes(
        scalar_field.astype(np.float32),
        level=0.5,
        color=0xff9900,
        wireframe=False
    )

    # Replace previous surface in plot
    if iso_surface:
        plot -= iso_surface
    iso_surface = surface
    plot += iso_surface

    time.sleep(0.05)  # control animation speed
```



Tasks

Task 1: Volume Rendering of a Custom Scalar Field

Create and visualize your own 3D scalar field using k3d.volume(). Steps:

- Define a 3D grid using np.indices() or np.meshgrid().
- Fill it with scalar values using a custom function (e.g., Gaussian blob, random noise, sine-based waves).
- Normalize the scalar values between 0 and 1.
- Visualize it using k3d.volume() and experiment with different scalar field resolutions.

The outcome is to understand how scalar fields look internally and how resolution affects visual detail.

Task 2: Compare Marching Cubes Results for Different Isosurface Levels

Extract and compare multiple surfaces from the same scalar field using different level values. Steps:

- Generate a scalar field (e.g., spherical or sinusoidal).
- Extract 3 surfaces using k3d.marching_cubes() with thresholds level=0.3, 0.5, and 0.7.
- Assign each surface a different color.

- Display them together in the same plot to compare.

The outcome is to visualize how the isosurface threshold affects the geometry and topology of extracted surfaces.

Task 3: Animate a Growing Sphere Using Marching Cubes

Simulate a growing or shrinking object (e.g., a pulsating sphere) using animated scalar fields and Marching Cubes. Steps:

- In a loop, generate a scalar field of a moving or growing 3D shape (e.g., increase radius over time).
- Re-extract the surface at each step using marching_cubes.
- Animate by updating the surface on each frame.
- Experiment with animation speed, resolution, and color.

The outcome is to combine surface extraction with dynamic scalar field updates.

Task 4: Surface Coloring Based on Distance from Origin

Color a mesh or isosurface based on the distance from the origin to each vertex. Steps:

- Create a 3D scalar field (e.g., spherical gradient).
- Extract the surface using marching_cubes.
- Instead of using a flat color, compute distance of each voxel/vertex from the center and use that as a scalar attribute.
- Apply a color map to visualize the attribute (e.g., blue = near center, red = far).
- If attribute mapping doesn't work, simulate the effect using per-vertex colors or extract the surface geometry manually.

Outcome is to introduce metadata or calculated attributes to enhance surface interpretation.
