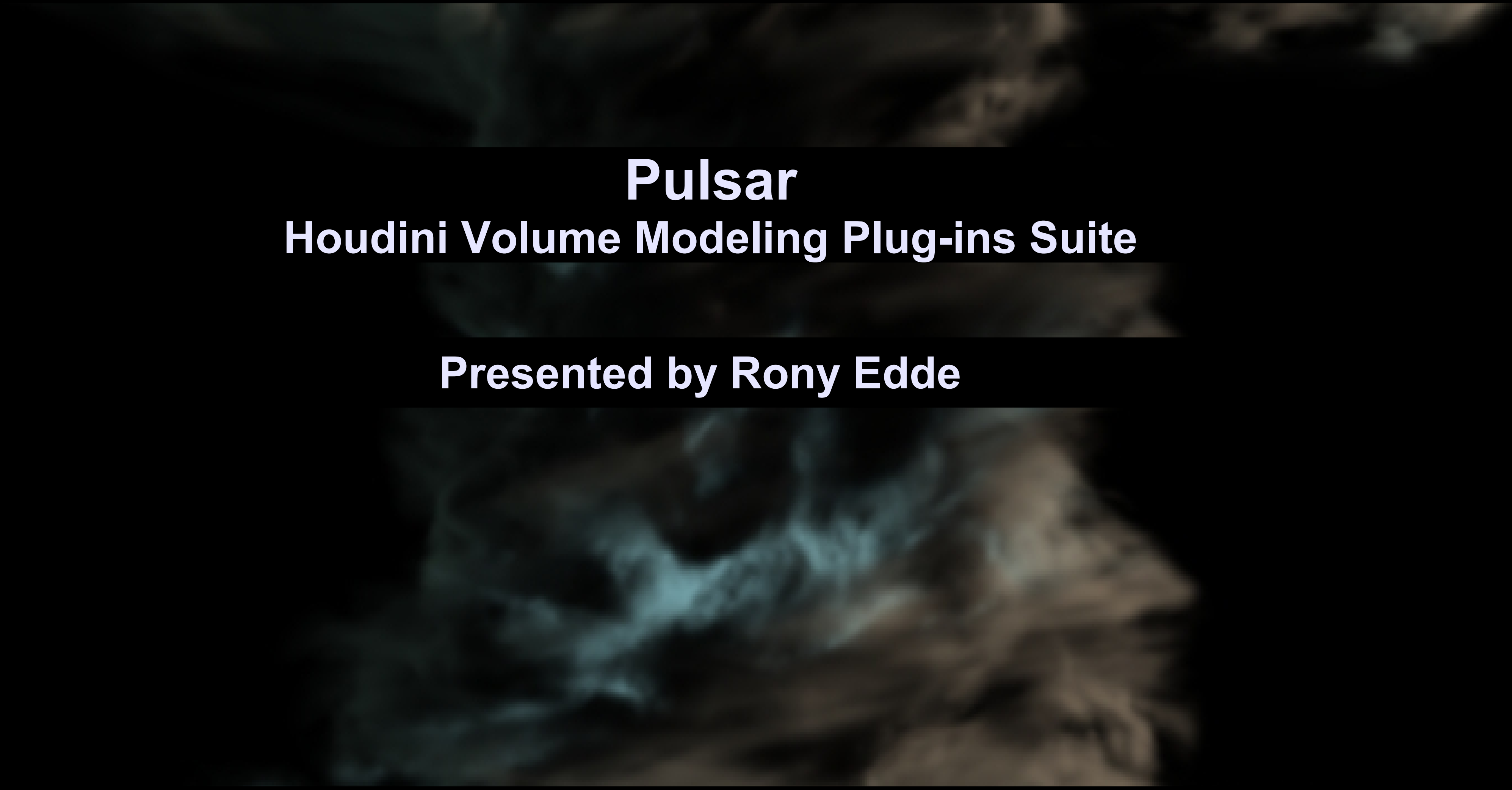
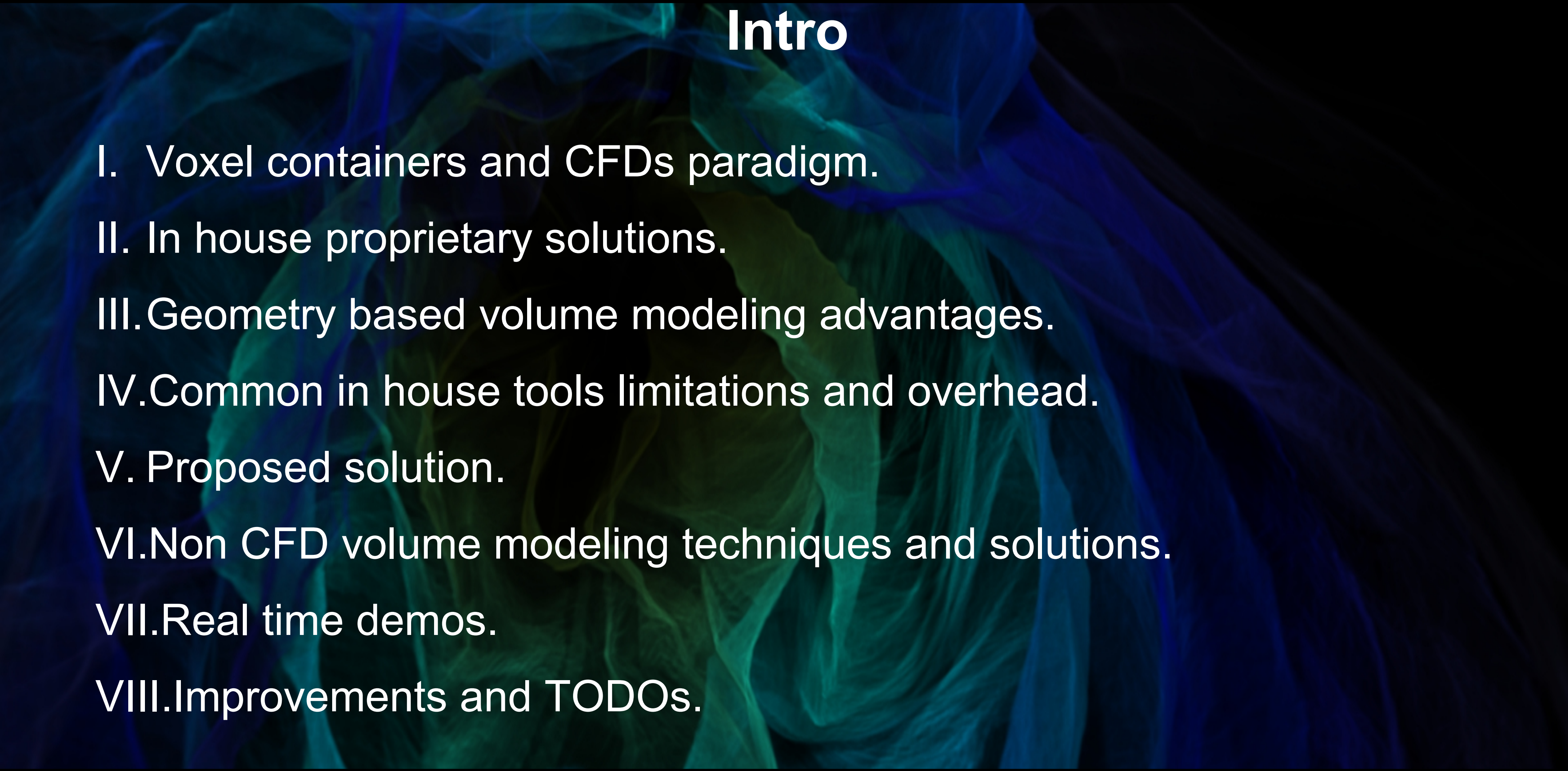


Pulsar

Houdini Volume Modeling Plug-ins Suite

Presented by Rony Edde

Intro



I. Voxel containers and CFDs paradigm.

II. In house proprietary solutions.

III. Geometry based volume modeling advantages.

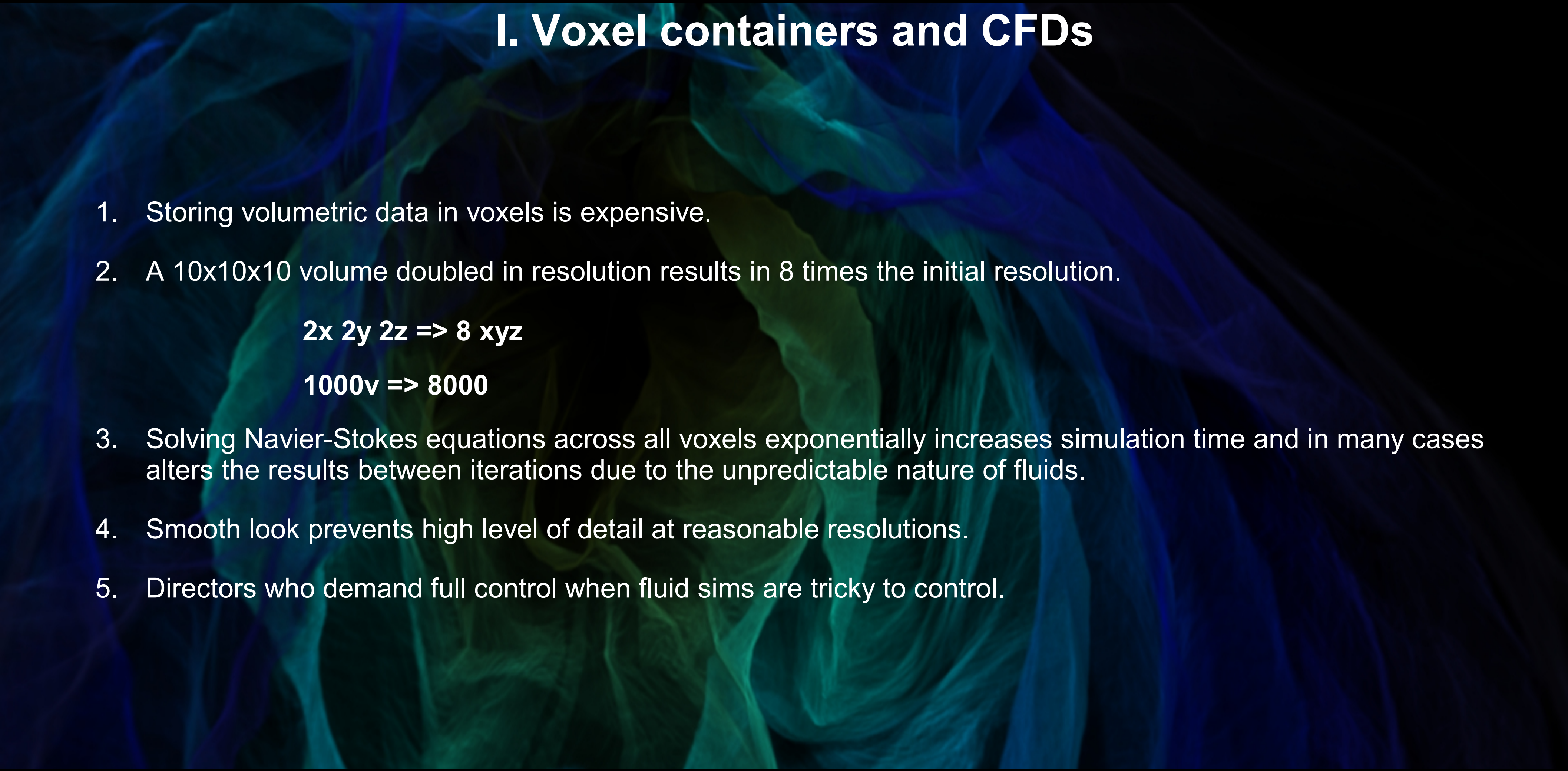
IV. Common in house tools limitations and overhead.

V. Proposed solution.

VI. Non CFD volume modeling techniques and solutions.

VII. Real time demos.

VIII. Improvements and TODOs.

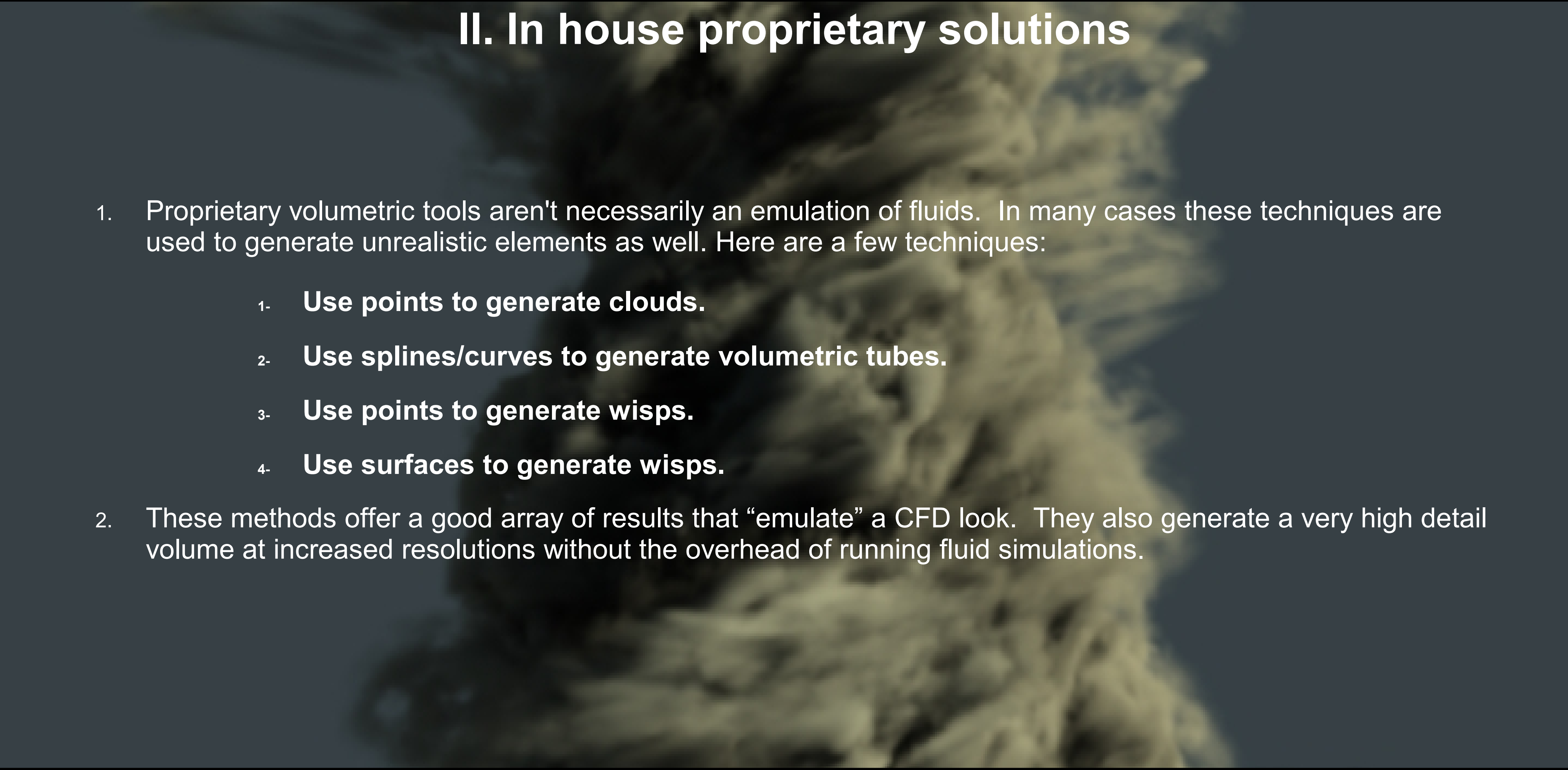


I. Voxel containers and CFDs

1. Storing volumetric data in voxels is expensive.
2. A 10x10x10 volume doubled in resolution results in 8 times the initial resolution.

   2x 2y 2z => 8 xyz

   1000v => 8000
3. Solving Navier-Stokes equations across all voxels exponentially increases simulation time and in many cases alters the results between iterations due to the unpredictable nature of fluids.
4. Smooth look prevents high level of detail at reasonable resolutions.
5. Directors who demand full control when fluid sims are tricky to control.



II. In house proprietary solutions

1. Proprietary volumetric tools aren't necessarily an emulation of fluids. In many cases these techniques are used to generate unrealistic elements as well. Here are a few techniques:

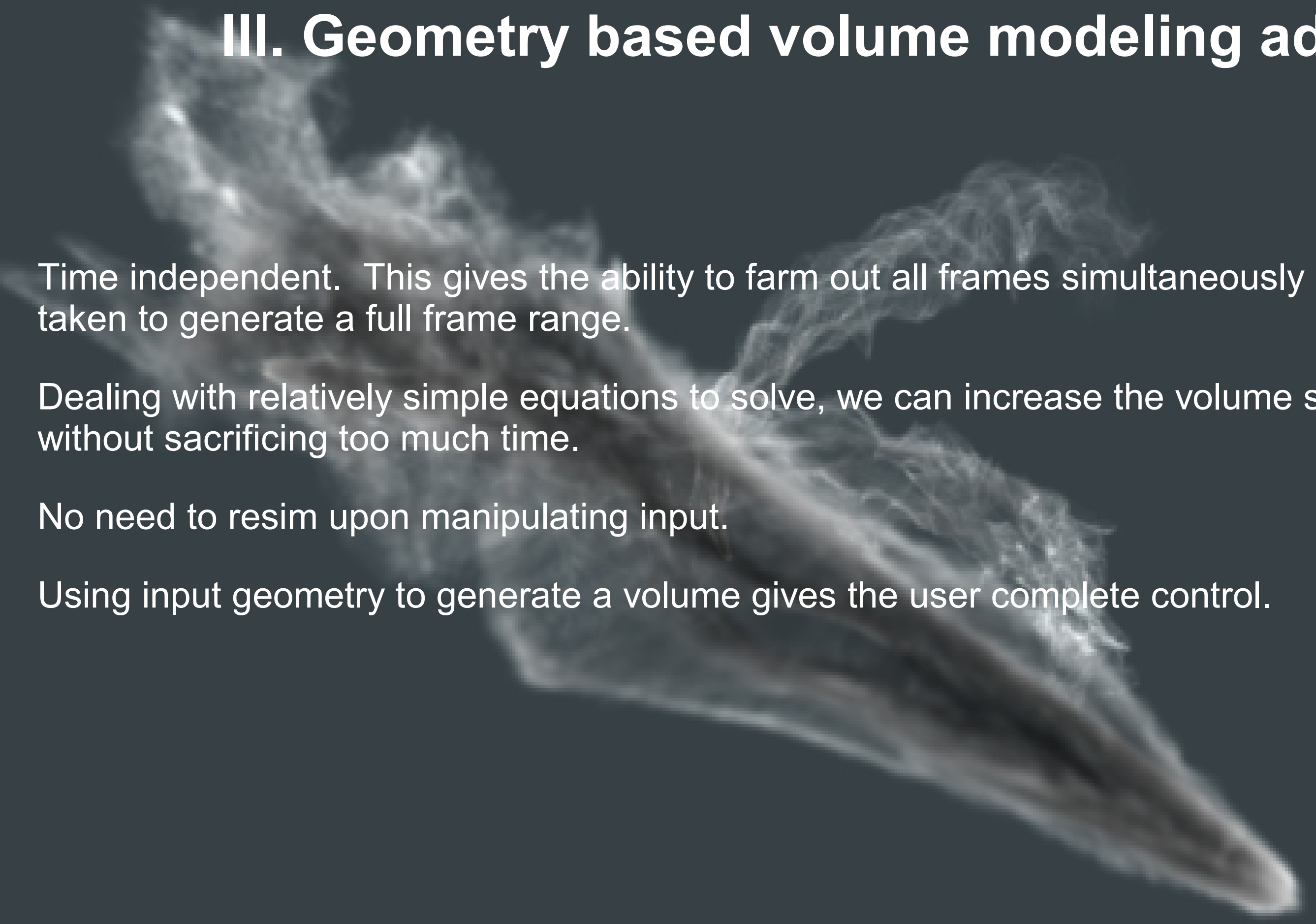
    1- **Use points to generate clouds.**

    2- **Use splines/curves to generate volumetric tubes.**

    3- **Use points to generate wisps.**

    4- **Use surfaces to generate wisps.**

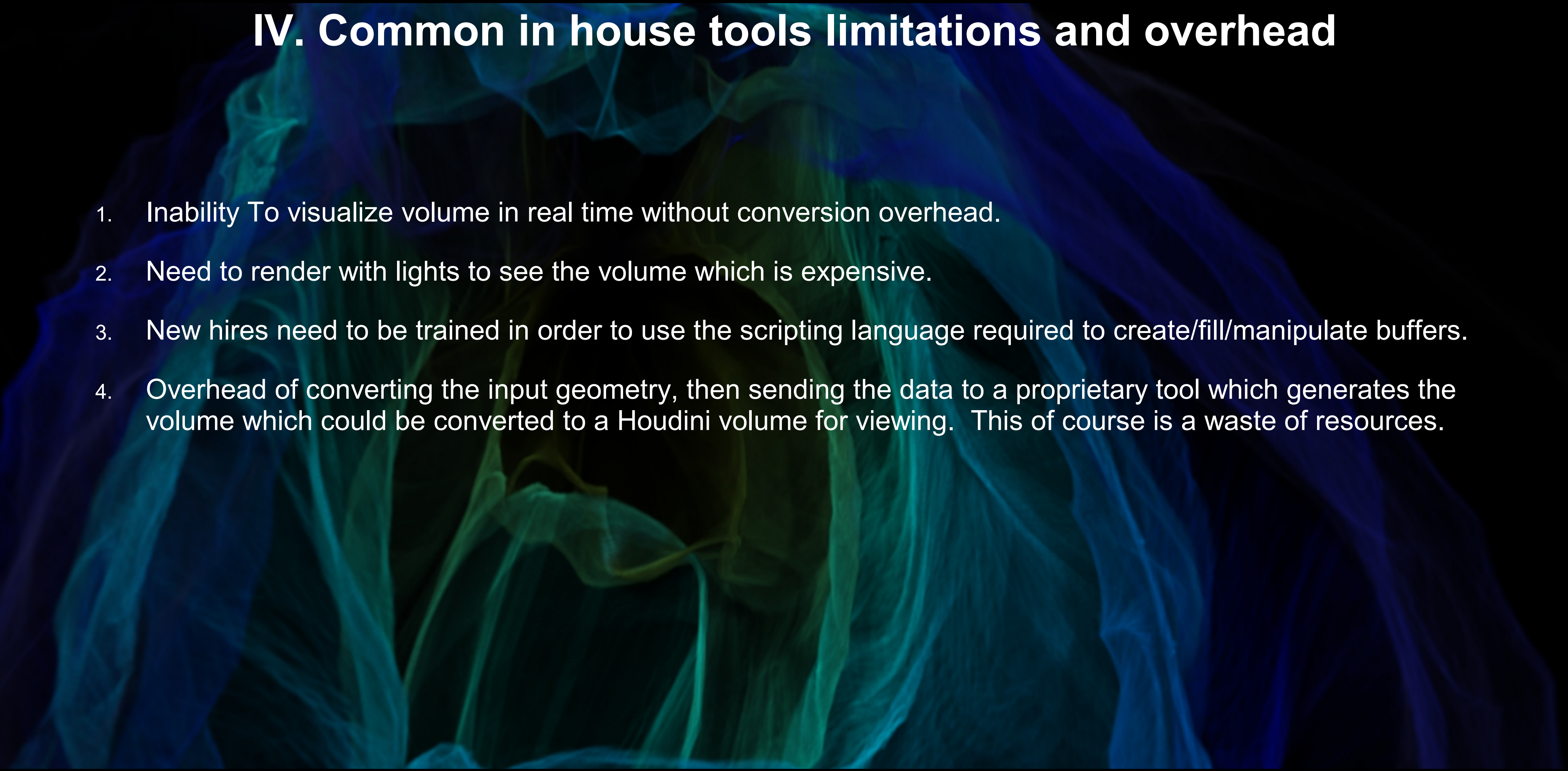
2. These methods offer a good array of results that “emulate” a CFD look. They also generate a very high detail volume at increased resolutions without the overhead of running fluid simulations.



III. Geometry based volume modeling advantages

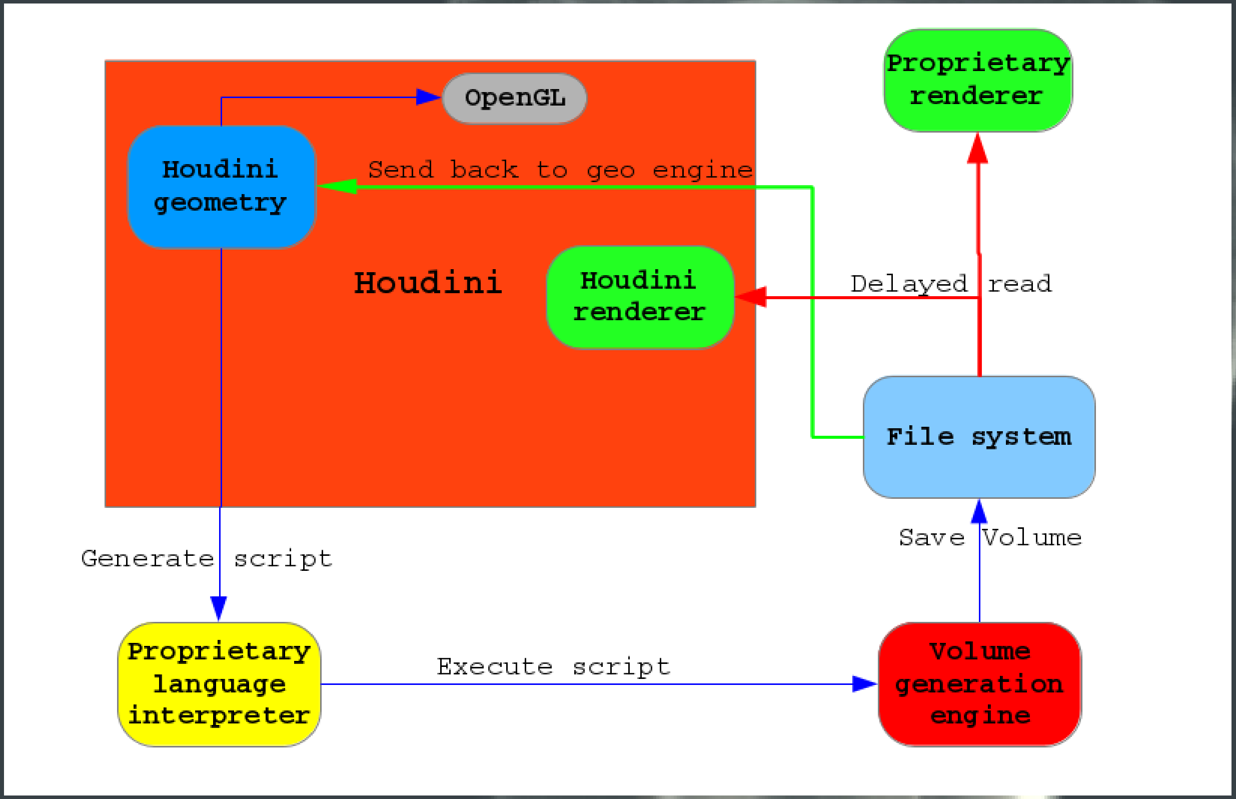
1. Time independent. This gives the ability to farm out all frames simultaneously effectively reducing the time taken to generate a full frame range.
2. Dealing with relatively simple equations to solve, we can increase the volume size or regenerate a volume without sacrificing too much time.
3. No need to resim upon manipulating input.
4. Using input geometry to generate a volume gives the user complete control.

IV. Common in house tools limitations and overhead

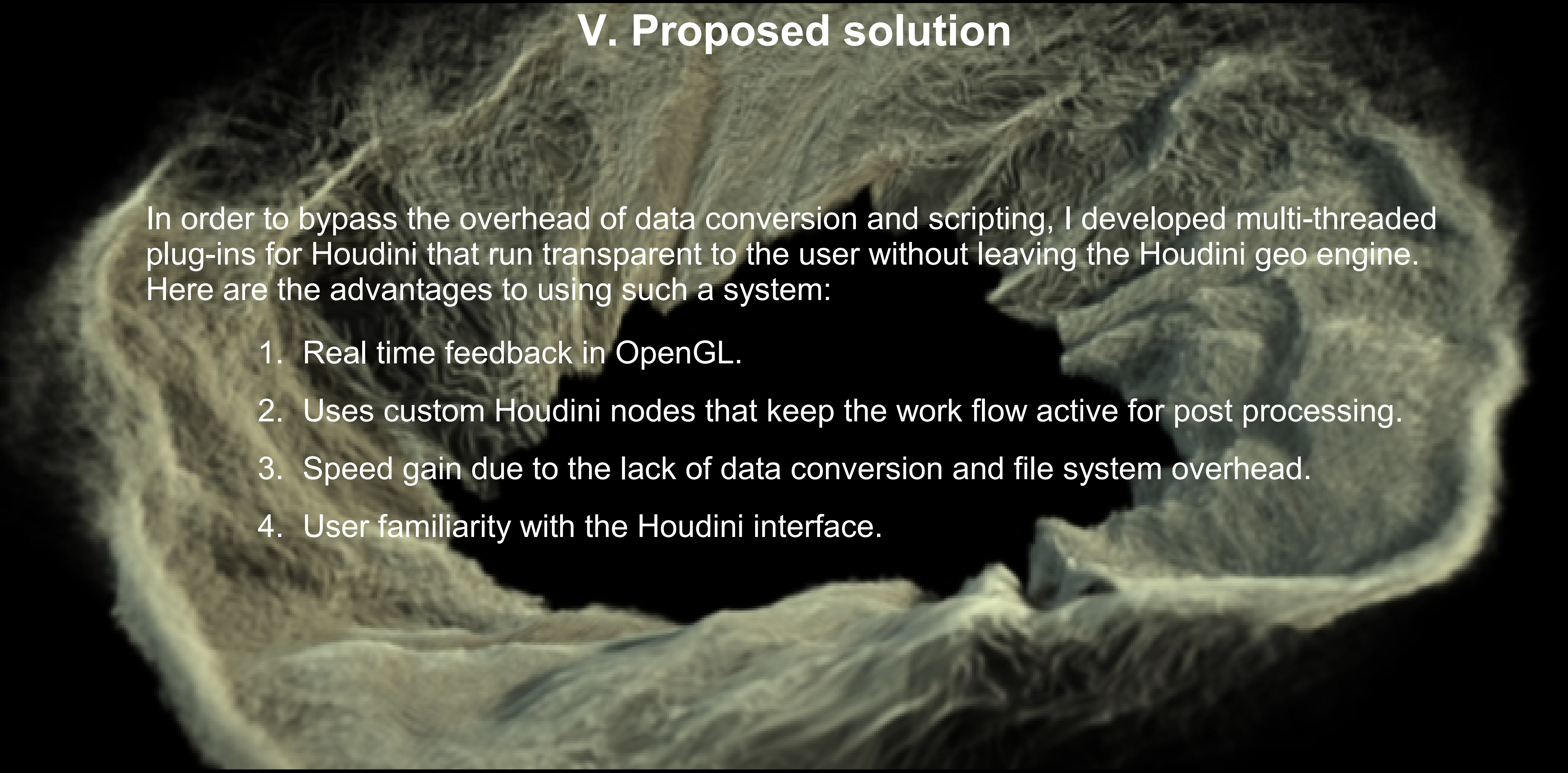


1. Inability To visualize volume in real time without conversion overhead.
2. Need to render with lights to see the volume which is expensive.
3. New hires need to be trained in order to use the scripting language required to create/fill/manipulate buffers.
4. Overhead of converting the input geometry, then sending the data to a proprietary tool which generates the volume which could be converted to a Houdini volume for viewing. This of course is a waste of resources.

IV. Common in house tools limitations and overhead



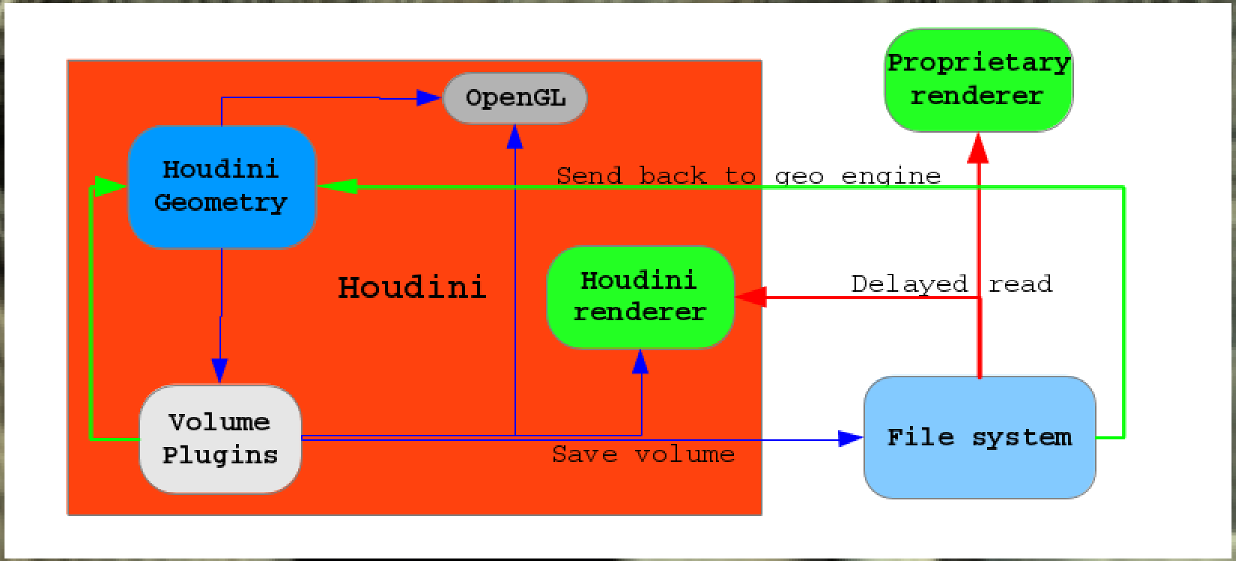
V. Proposed solution



In order to bypass the overhead of data conversion and scripting, I developed multi-threaded plug-ins for Houdini that run transparent to the user without leaving the Houdini geo engine. Here are the advantages to using such a system:

1. Real time feedback in OpenGL.
2. Uses custom Houdini nodes that keep the work flow active for post processing.
3. Speed gain due to the lack of data conversion and file system overhead.
4. User familiarity with the Houdini interface.

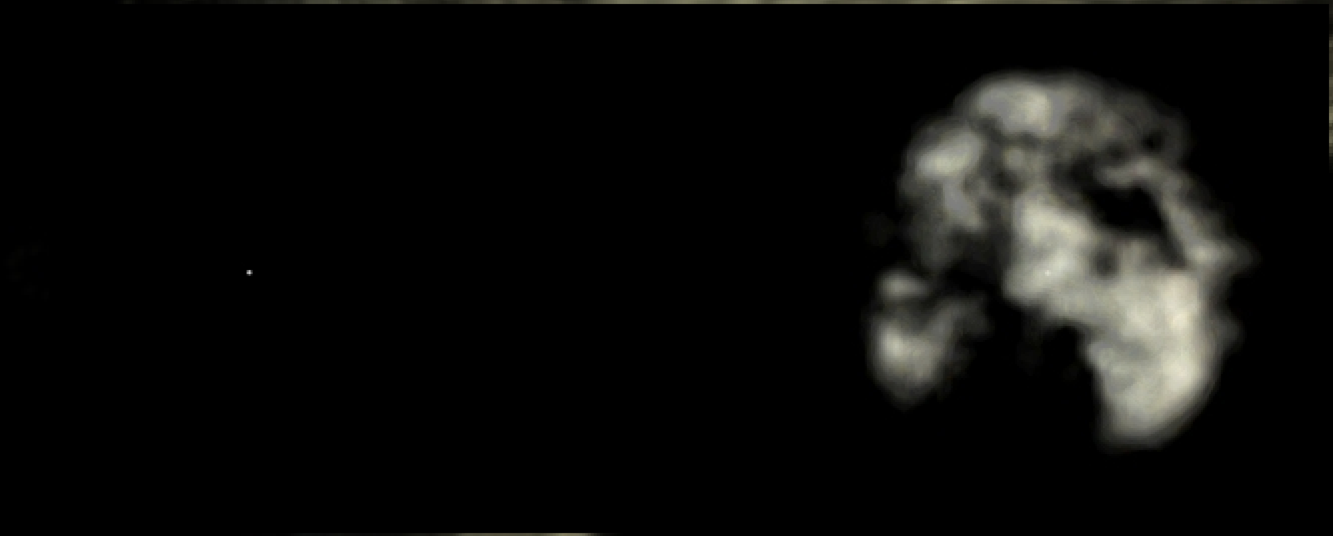
V. Proposed solution



VI. Non CFD volume modeling techniques

1. Fill with clouds plug-in.

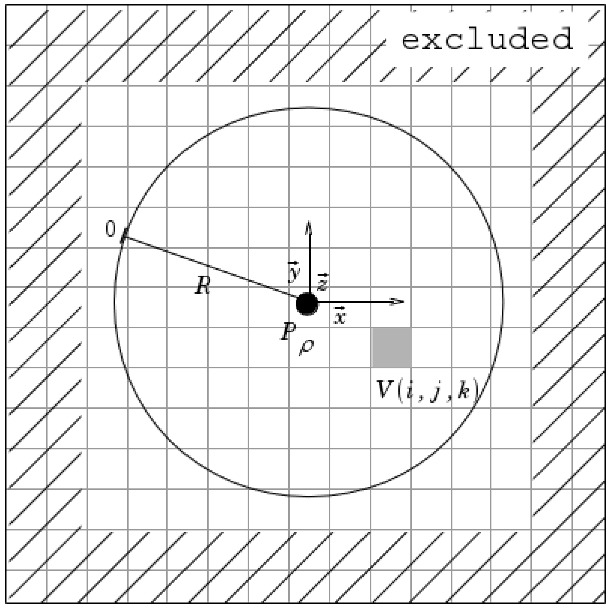
This plug-in generates cloud volumes from input points using a custom Perlin noise implementation.



Input Point **Result**

Video

VI. Non CFD volume modeling techniques



$P = input\ point$

$R = radius$

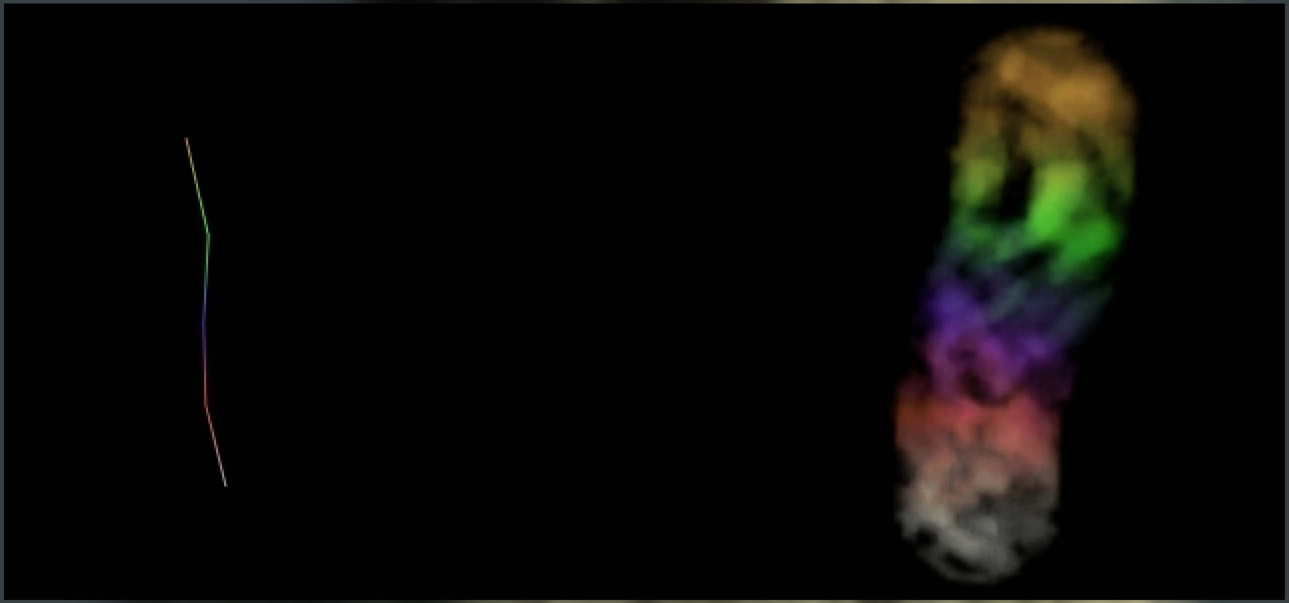
$\rho = density$

$V = voxel\ at\ index(i, j, k)$

VI. Non CFD volume modeling techniques

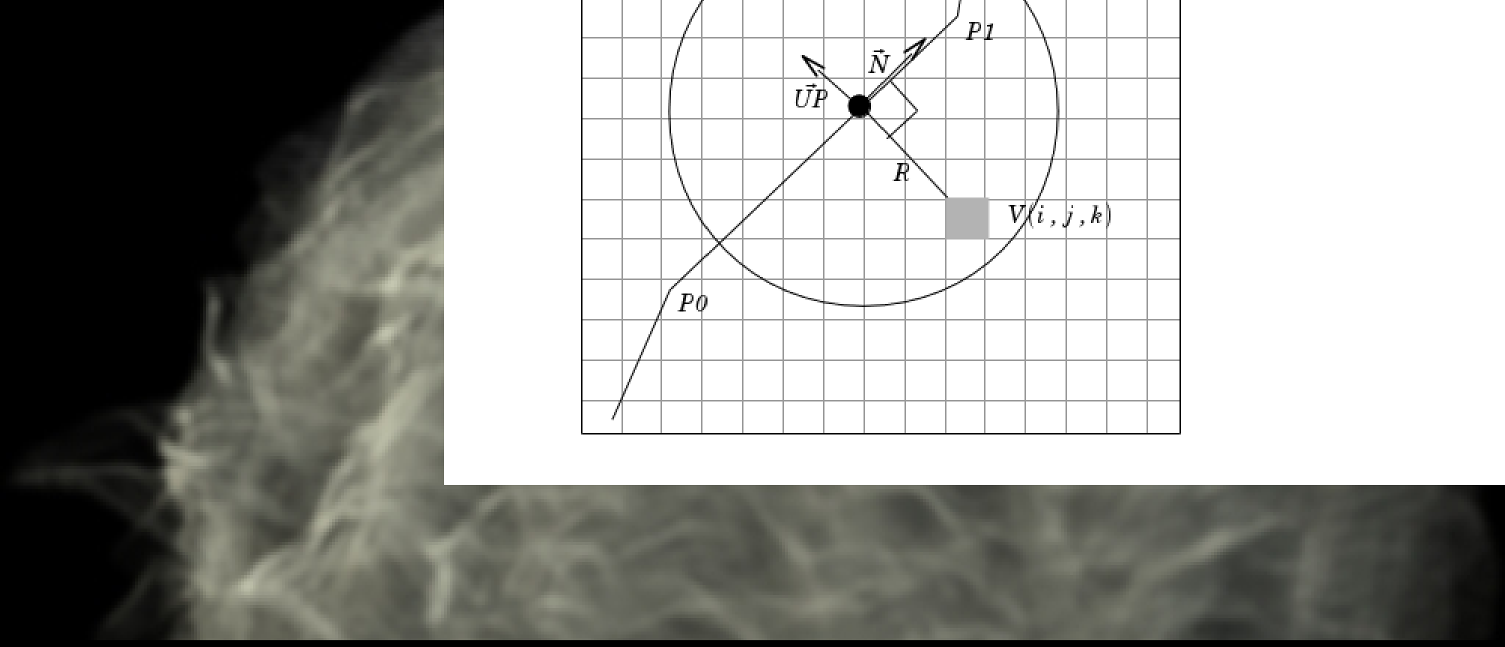
2. Fill with curves plug-in.

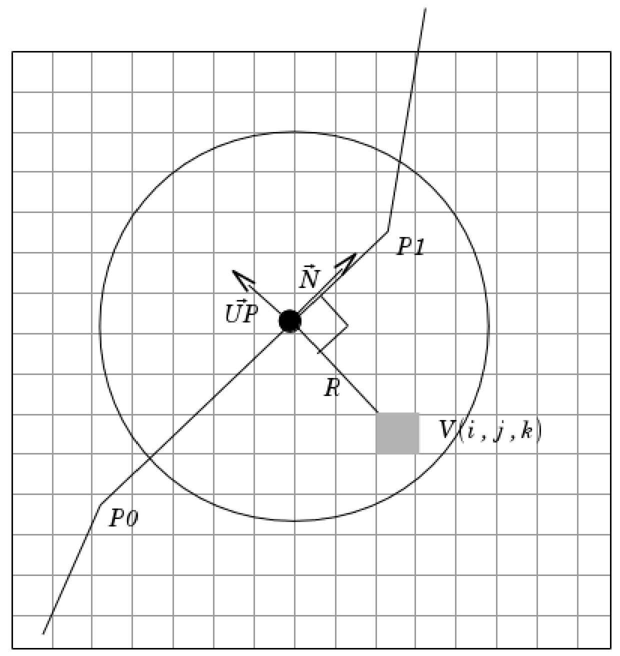
This plug-in generates a noised tube volume from input curves using a custom Perlin noise.



Video

VI. Non CFD volume modeling techniques





$P0, P1 = 2$ *closest points*

$\vec{N} =$ *normal vector*

$\vec{UP} =$ *up vector*

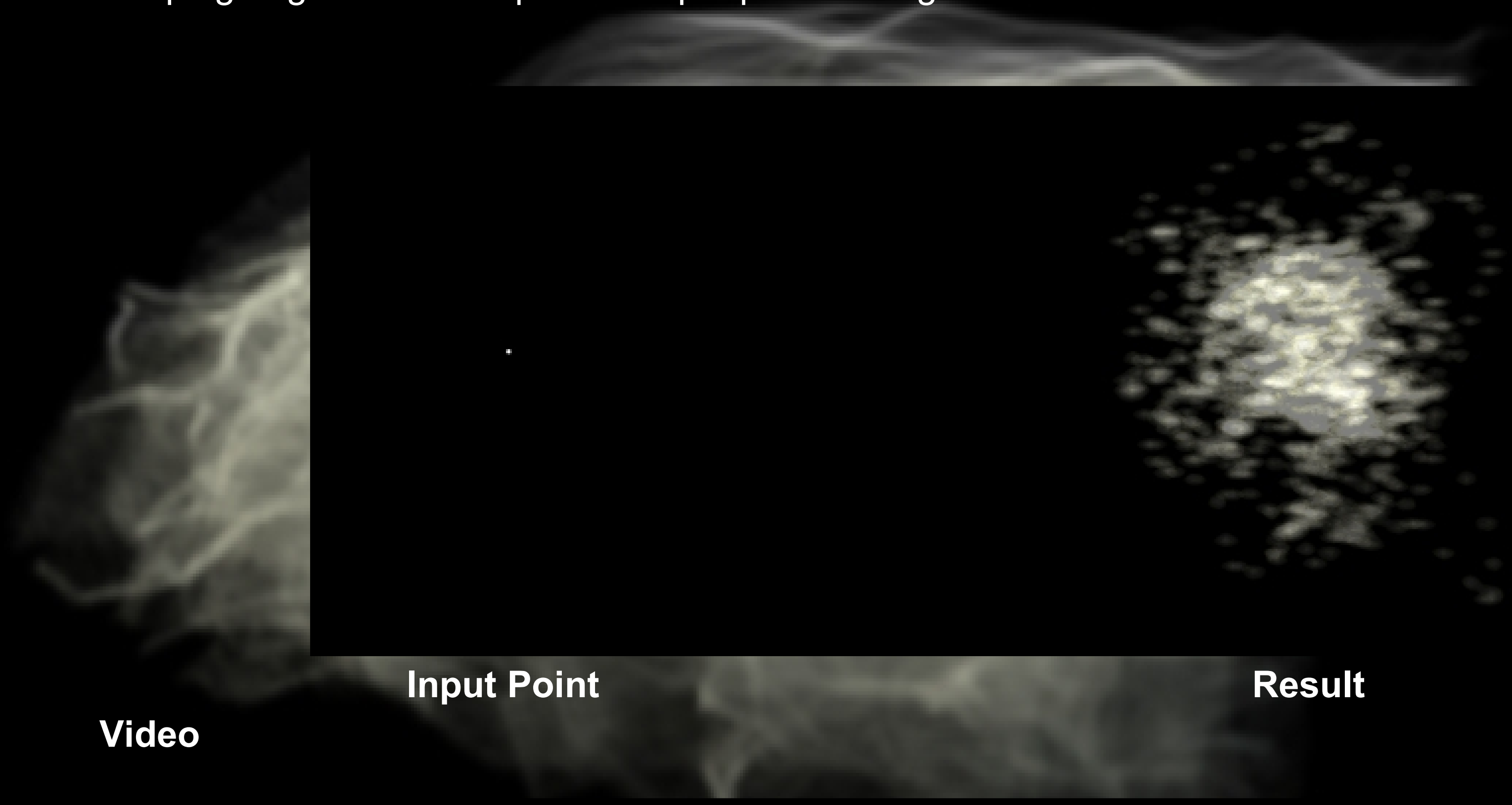
$V =$ *current voxel at index* (i, j, k)

$R =$ *radius*

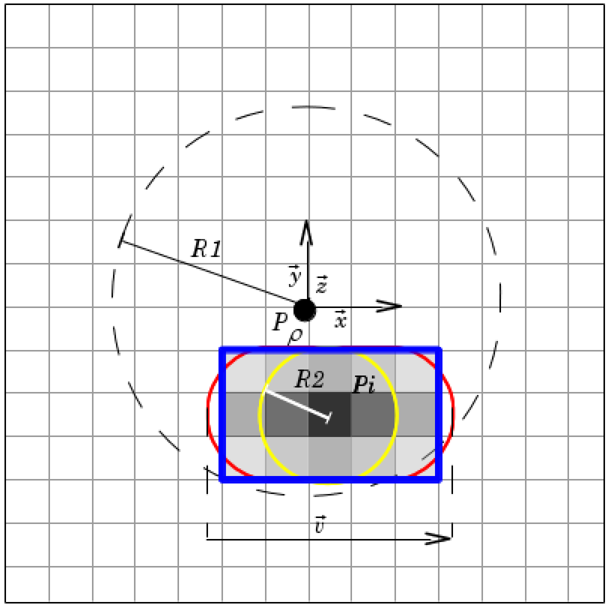
VI. Non CFD volume modeling techniques

3. Fill with wisps plug-in.

This plug-in generates wisps from input points using custom Perlin noise distribution.



VI. Non CFD volume modeling techniques



$P = input\ point$

$Pi = child\ point\ position\ at\ index\ i$

$\rho = density$

$R1 = noise\ distribution\ radius$

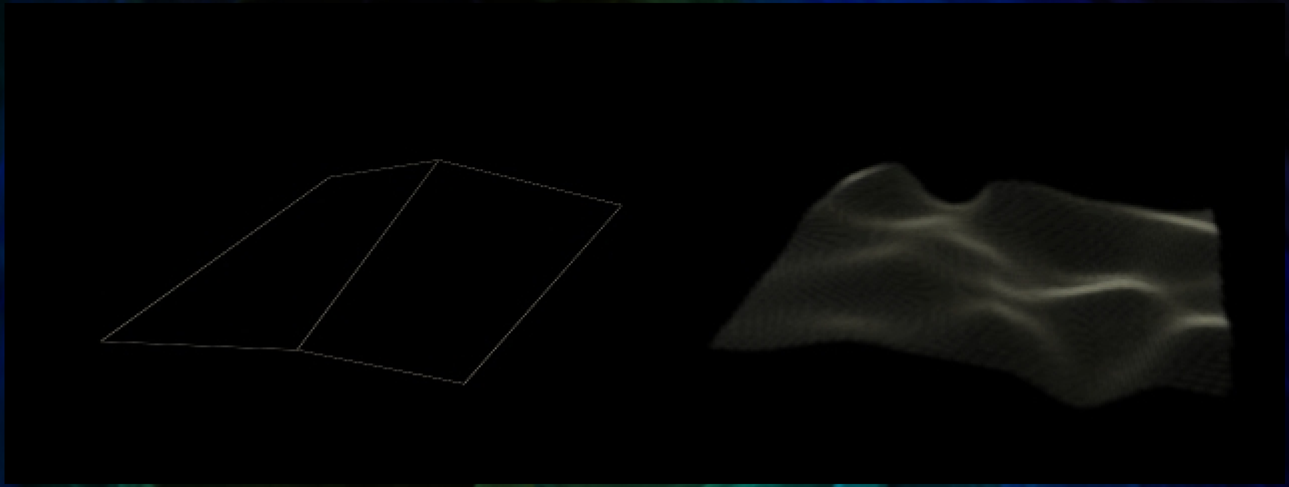
$R2 = child\ radius$

$\vec{v} = velocity\ vector$

VI. Non CFD volume modeling techniques

4. Fill with surface wisps plug-in.

This plug-in generates wisps from a mesh using custom Perlin noise distribution based on UVW coordinates.

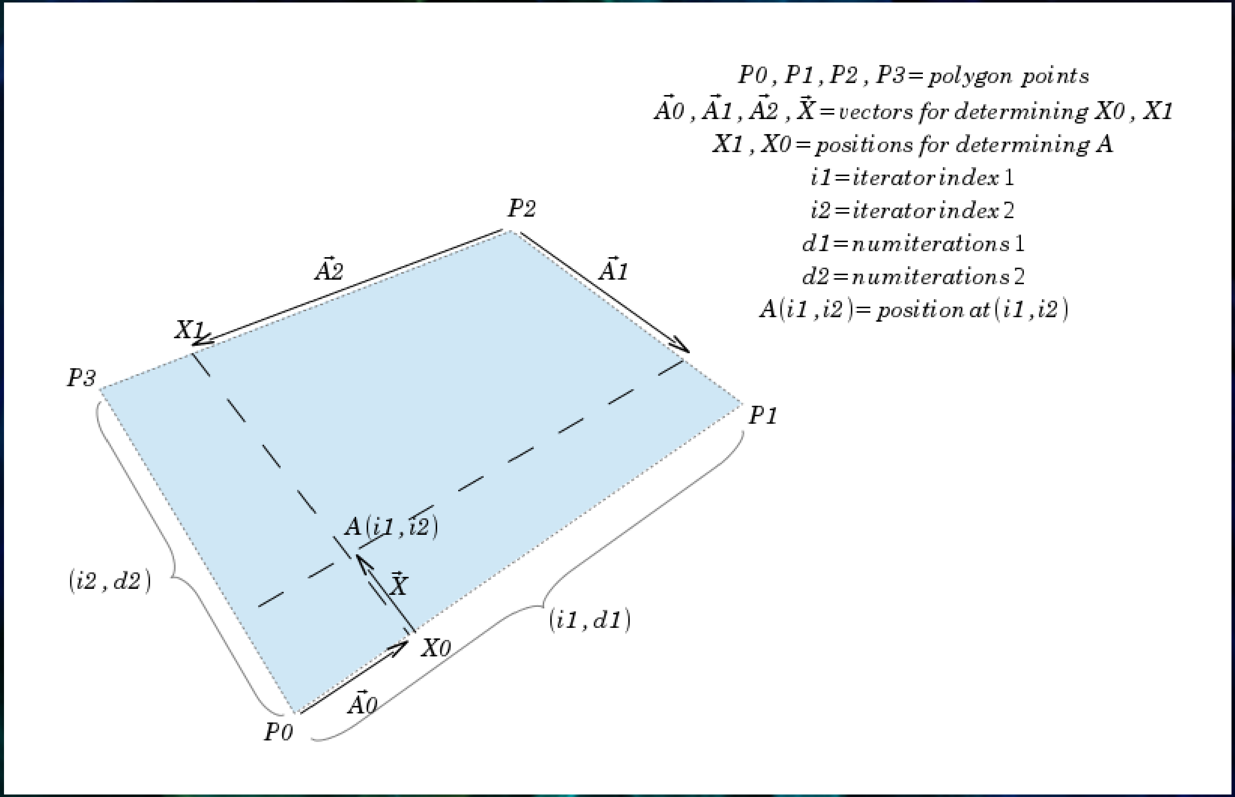


Input Mesh

Result

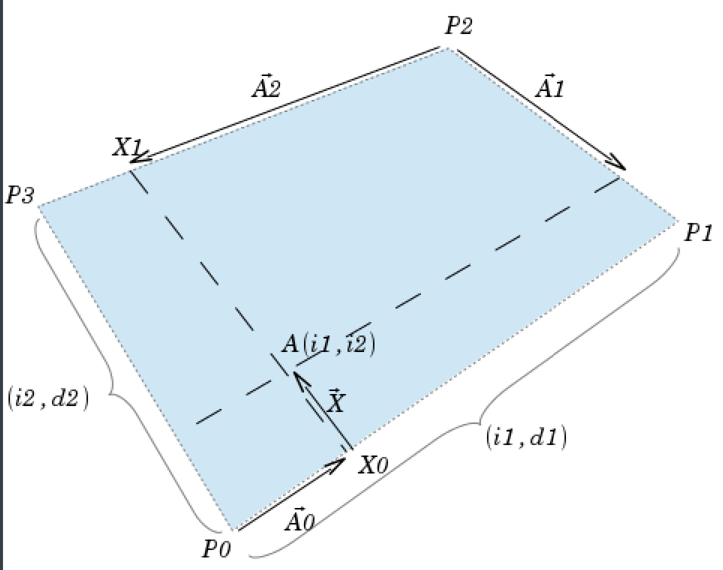
Video

VI. Non CFD volume modeling techniques



VI. Non CFD volume modeling techniques

Inefficient vector interpolation example:



$$A = P0 + \vec{A0} + \vec{X}$$

$$\vec{A0} = (\frac{P1-P0}{d1})i1 \qquad \vec{A2} = (\frac{P3-P2}{d1})(d1-i1)$$

$$X0 = P0 + \vec{A0} \qquad X1 = P2 + \vec{A2}$$

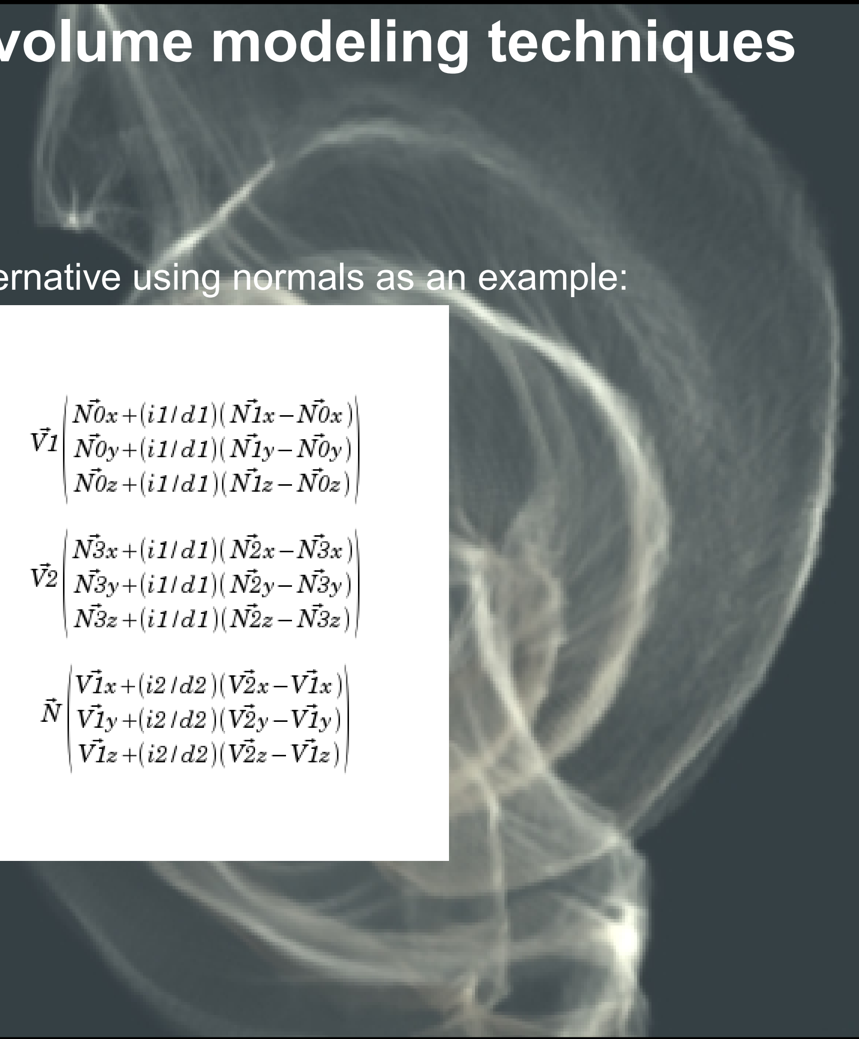
$$A = P0 + \vec{A0} + (\frac{X1-X0}{d2})i2$$

$$A = P0 + (\frac{P1-P0}{d1})i2 + (\frac{X1-X0}{d2})i2$$

$$A = P0(\frac{P1+P0}{d1})i2 + (\frac{P2+\vec{A2}-P0-\vec{A0}}{d2})i2$$

$$A = P0 + (\frac{P1-P0}{d1})i2 + (\frac{P2+(\frac{P3-P2}{d1})(d1-i1)-P0(\frac{-P1+P0}{d1})i1}{d2})i2$$

$$A = P0 + (\frac{P1-P0}{d1})i2 + (P2 + (\frac{P3-P2}{d1})(d1-i2) - P0(\frac{Po-P1}{d1})i1)(i2\,d2)$$



VI. Non CFD volume modeling techniques

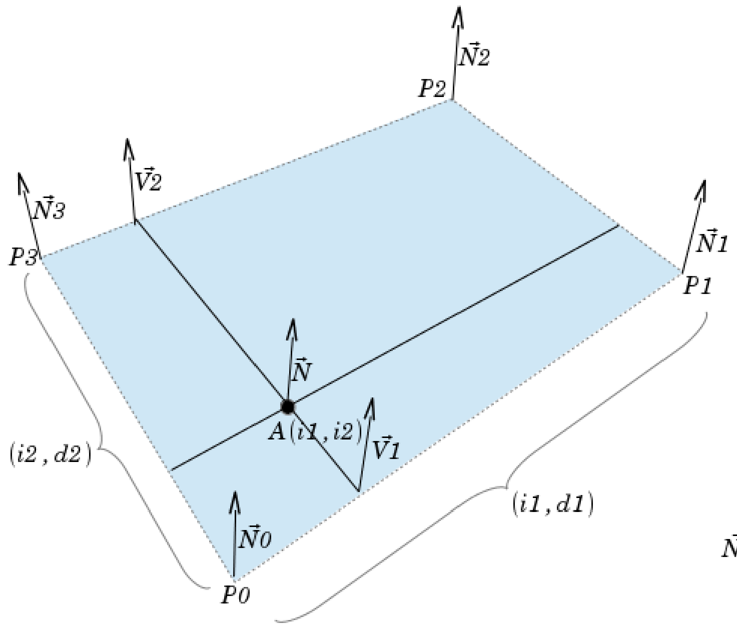
A faster alternative using normals as an example:

$$\vec{V1}\begin{pmatrix} \vec{N0}x+(i1/d1)(\vec{N1}x-\vec{N0}x) \\ \vec{N0}y+(i1/d1)(\vec{N1}y-\vec{N0}y) \\ \vec{N0}z+(i1/d1)(\vec{N1}z-\vec{N0}z) \end{pmatrix}$$

$$\vec{V2}\begin{pmatrix} \vec{N3}x+(i1/d1)(\vec{N2}x-\vec{N3}x) \\ \vec{N3}y+(i1/d1)(\vec{N2}y-\vec{N3}y) \\ \vec{N3}z+(i1/d1)(\vec{N2}z-\vec{N3}z) \end{pmatrix}$$

$$\vec{N}\begin{pmatrix} \vec{V1}x+(i2/d2)(\vec{V2}x-\vec{V1}x) \\ \vec{V1}y+(i2/d2)(\vec{V2}y-\vec{V1}y) \\ \vec{V1}z+(i2/d2)(\vec{V2}z-\vec{V1}z) \end{pmatrix}$$

VI. Non CFD volume modeling techniques



$P0, P1, P2, P3 = polygon\ points$

$i1 = iterator\ index\ 1$

$i2 = iterator\ index\ 2$

$d1 = num\ iterations\ 1$

$d2 = num\ iterations\ 2$

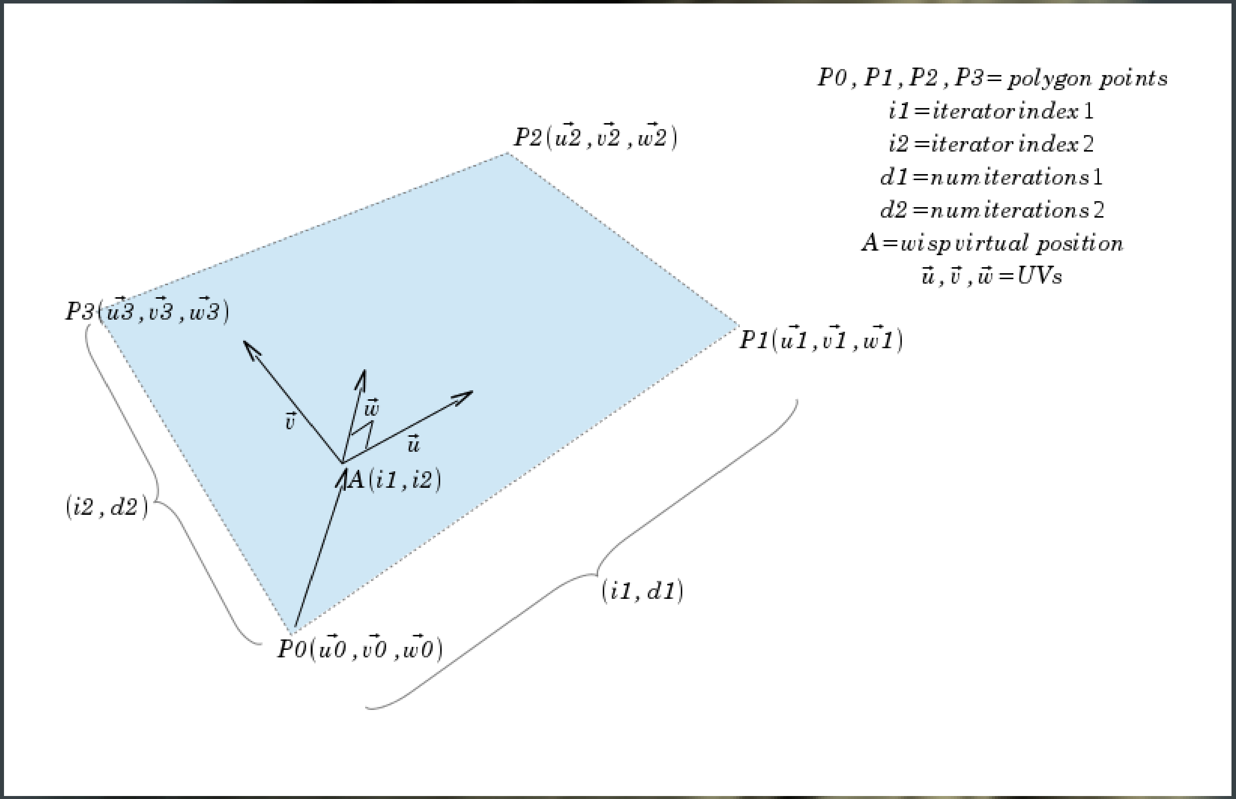
$A = wisp\ virtual\ position$

$\vec{N0}, \vec{N1}, \vec{N2}, \vec{N3} = polygon\ normals$

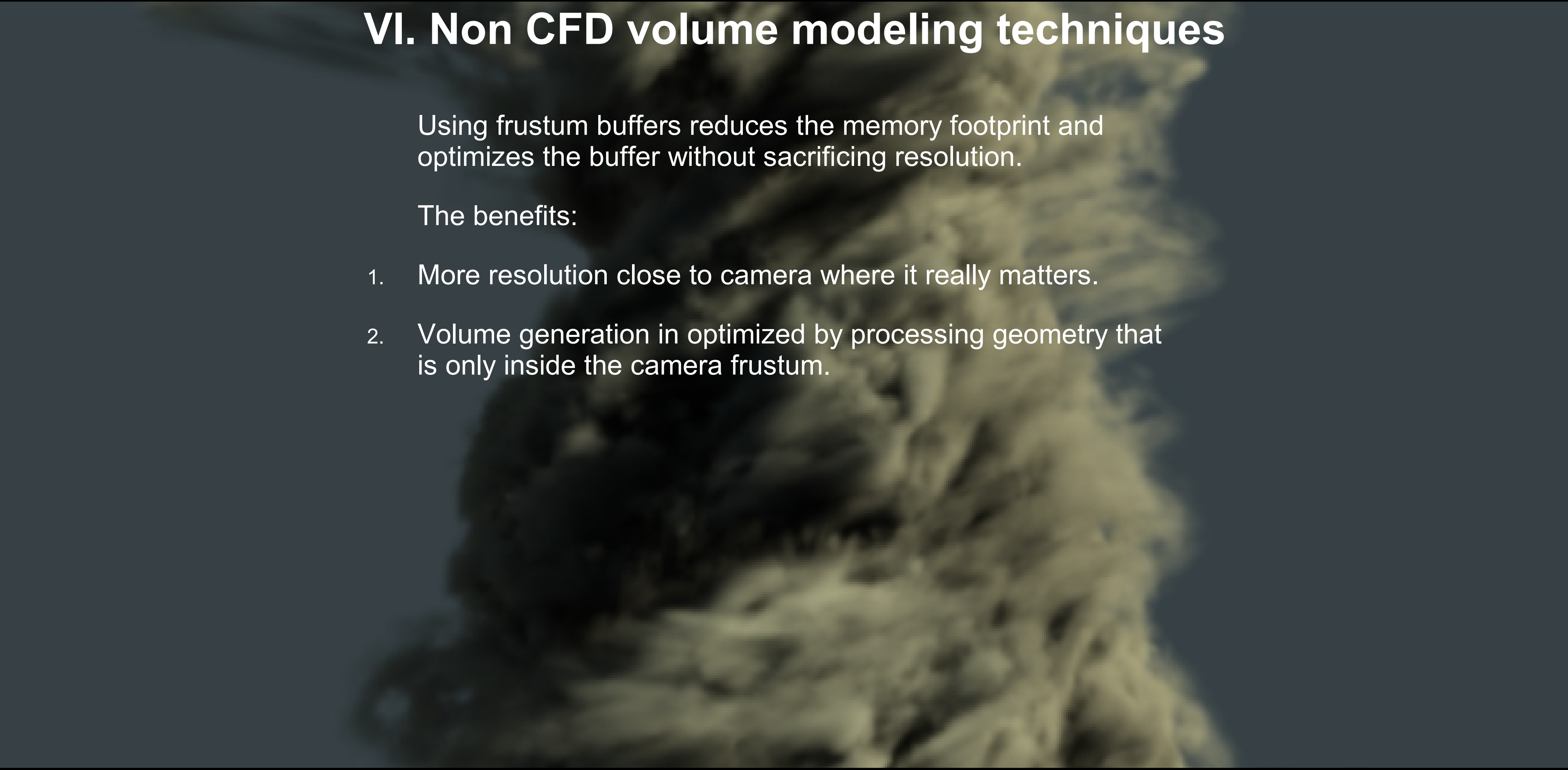
$\vec{N} = interpolated\ normal$

VI. Non CFD volume modeling techniques

UV coordinates interpolation for noise displacement:



$P0, P1, P2, P3 = polygon\ points$
$i1 = iterator\ index\ 1$
$i2 = iterator\ index\ 2$
$d1 = num\ iterations\ 1$
$d2 = num\ iterations\ 2$
$A = wisp\ virtual\ position$
$\vec{u}, \vec{v}, \vec{w} = UVs$
$P2(\vec{u2}, \vec{v2}, \vec{w2})$
$P3(\vec{u3}, \vec{v3}, \vec{w3})$
$P1(\vec{u1}, \vec{v1}, \vec{w1})$
$P0(\vec{u0}, \vec{v0}, \vec{w0})$
$\vec{v}$
$\vec{w}$
$\vec{u}$
$A(i1, i2)$
$(i2, d2)$
$(i1, d1)$



VI. Non CFD volume modeling techniques

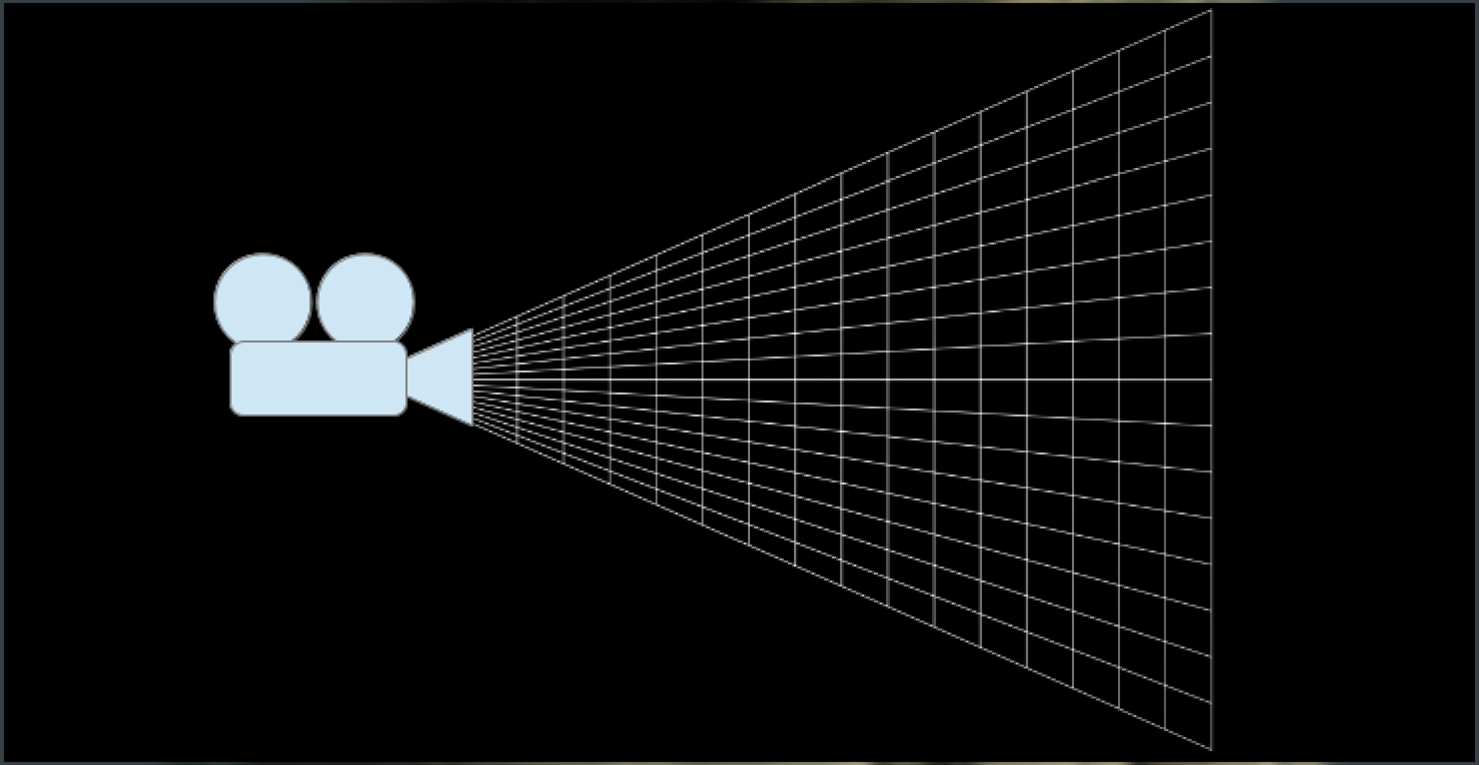
Using frustum buffers reduces the memory footprint and optimizes the buffer without sacrificing resolution.

The benefits:

1. More resolution close to camera where it really matters.
2. Volume generation in optimized by processing geometry that is only inside the camera frustum.

VI. Non CFD volume modeling techniques

Frustum buffer:



VI. Non CFD volume modeling techniques

Velocity Paradigm:

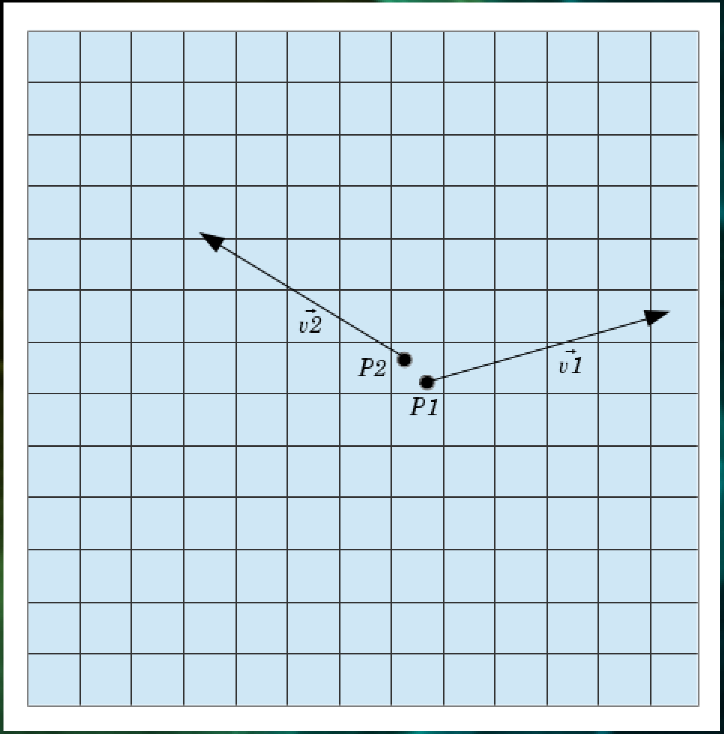
When transferring velocity vectors from points to neighboring voxels there are 2 possible options for each voxel.

1. Only one point is close enough to directly affects the voxel value.

2. More than one point is affecting the voxel and this case causes problems because we can only store one velocity component value per voxel.

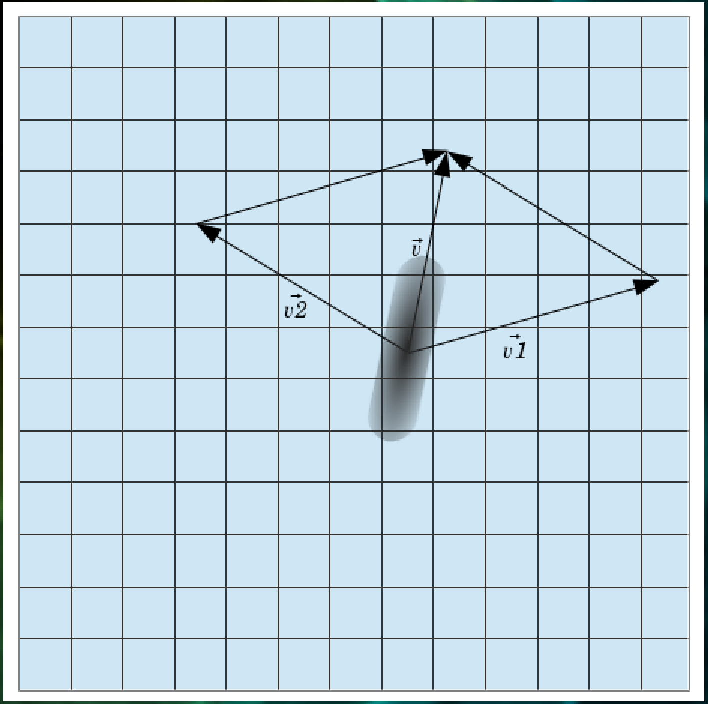
VI. Non CFD volume modeling techniques

Two points with different velocity vectors inside the same voxel.



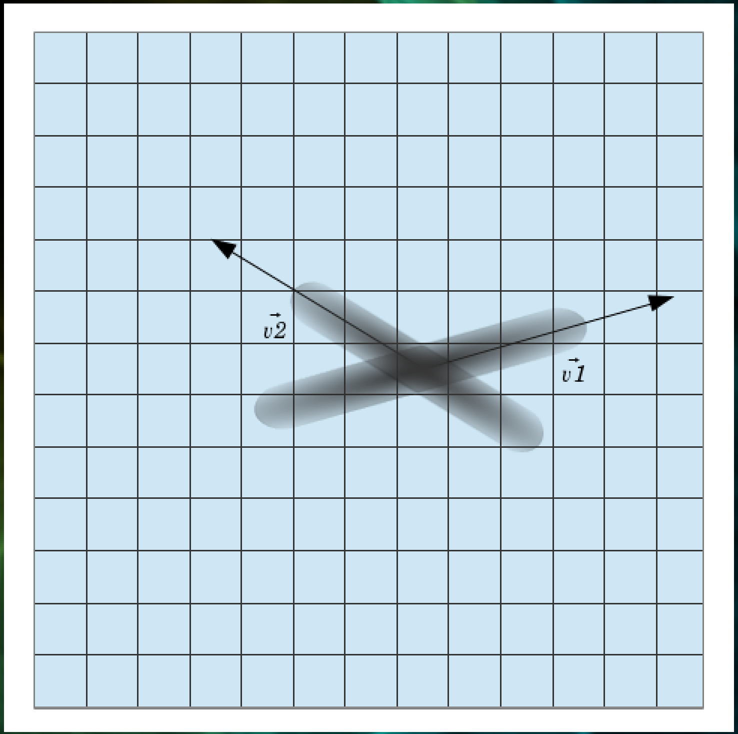
VI. Non CFD volume modeling techniques

The combined or average value is incorrect. The result is incorrect velocities.



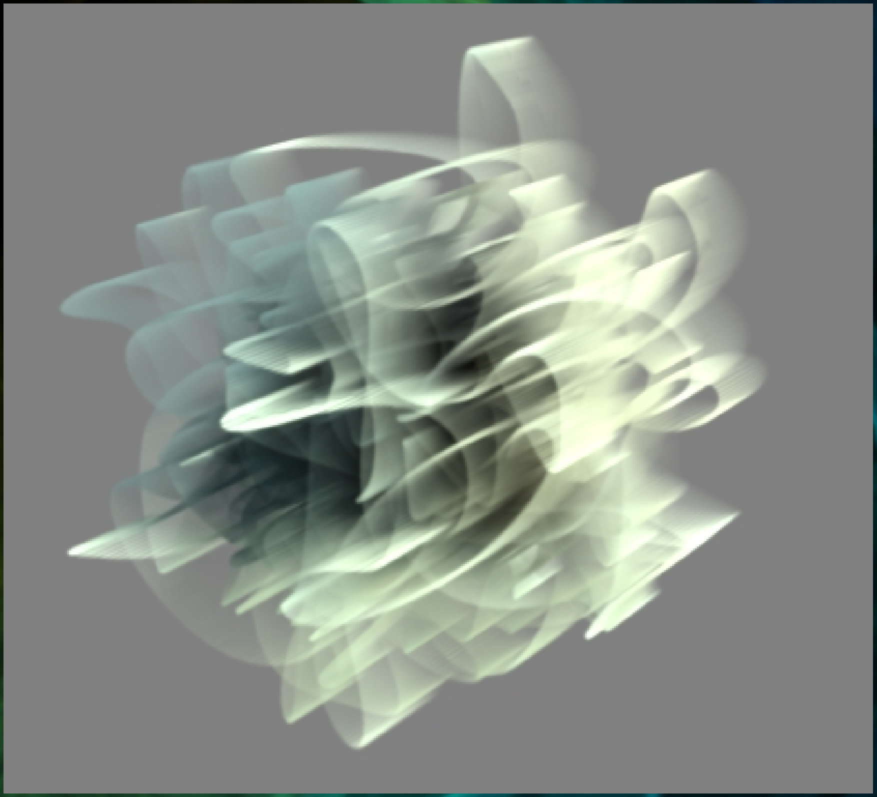
VI. Non CFD volume modeling techniques

The solution is to bake the velocity into the density and ignore the velocity field.



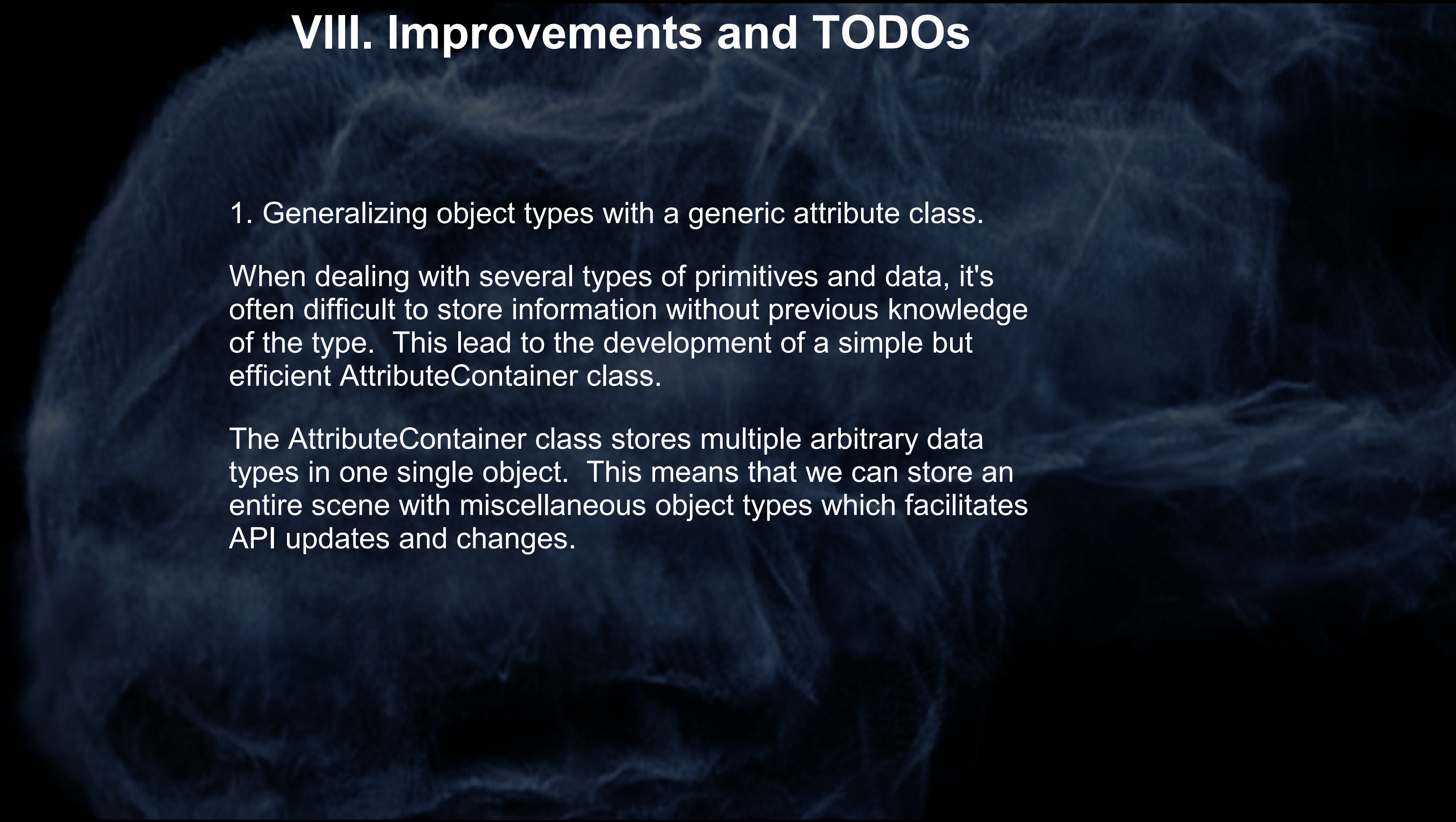
VI. Non CFD volume modeling techniques

Baked velocity example:



VII. Real time demos

VIII. Improvements and TODOs



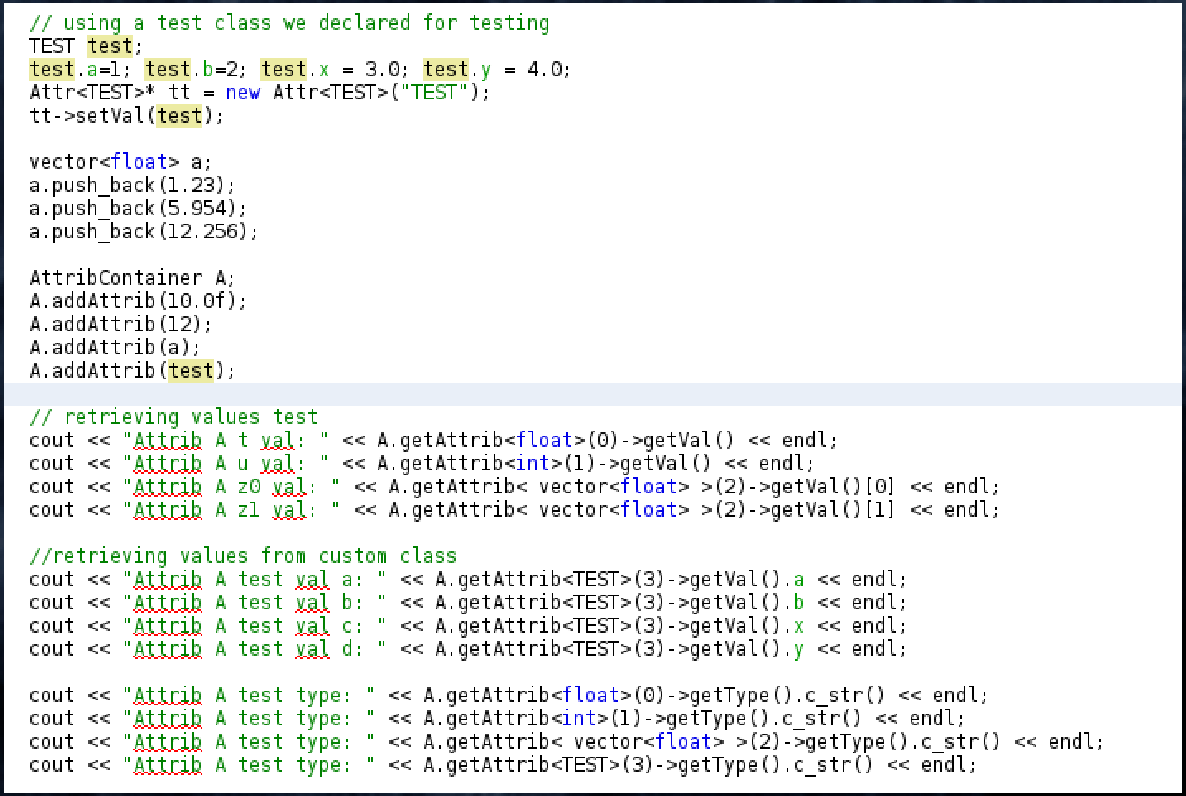
1. Generalizing object types with a generic attribute class.

When dealing with several types of primitives and data, it's often difficult to store information without previous knowledge of the type. This lead to the development of a simple but efficient AttributeContainer class.

The AttributeContainer class stores multiple arbitrary data types in one single object. This means that we can store an entire scene with miscellaneous object types which facilitates API updates and changes.

VIII. Improvements and TODOs

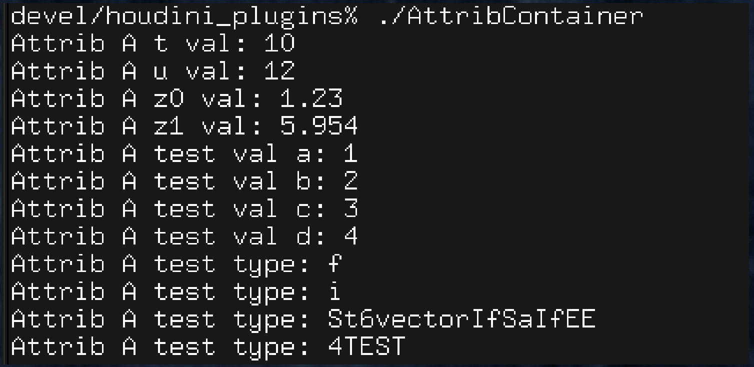
Sample code usage:



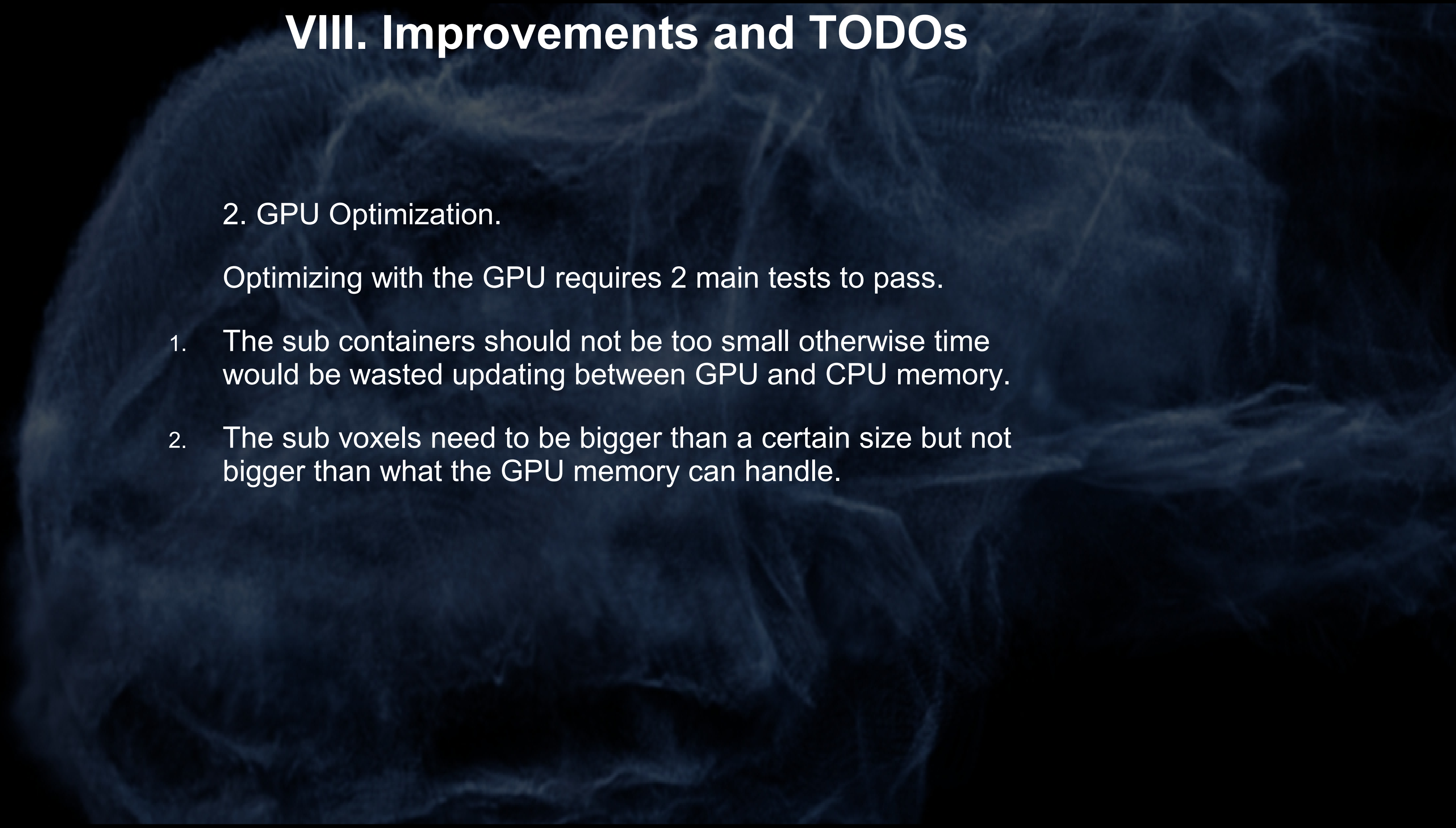
```
// using a test class we declared for testing
TEST test;
test.a=1; test.b=2; test.x = 3.0; test.y = 4.0;
Attr<TEST>* tt = new Attr<TEST>("TEST");
tt->setVal(test);

vector<float> a;
a.push_back(1.23);
a.push_back(5.954);
a.push_back(12.256);

AttribContainer A;
A.addAttrib(10.0f);
A.addAttrib(12);
A.addAttrib(a);
A.addAttrib(test);

// retrieving values test
cout << "Attrib A t val: " << A.getAttrib<float>(0)->getVal() << endl;
cout << "Attrib A u val: " << A.getAttrib<int>(1)->getVal() << endl;
cout << "Attrib A z0 val: " << A.getAttrib< vector<float> >(2)->getVal()[0] << endl;
cout << "Attrib A z1 val: " << A.getAttrib< vector<float> >(2)->getVal()[1] << endl;

//retrieving values from custom class
cout << "Attrib A test val a: " << A.getAttrib<TEST>(3)->getVal().a << endl;
cout << "Attrib A test val b: " << A.getAttrib<TEST>(3)->getVal().b << endl;
cout << "Attrib A test val c: " << A.getAttrib<TEST>(3)->getVal().x << endl;
cout << "Attrib A test val d: " << A.getAttrib<TEST>(3)->getVal().y << endl;

cout << "Attrib A test type: " << A.getAttrib<float>(0)->getType().c_str() << endl;
cout << "Attrib A test type: " << A.getAttrib<int>(1)->getType().c_str() << endl;
cout << "Attrib A test type: " << A.getAttrib< vector<float> >(2)->getType().c_str() << endl;
cout << "Attrib A test type: " << A.getAttrib<TEST>(3)->getType().c_str() << endl;
```

VIII. Improvements and TODOs

Output:



```
devel/houdini_plugins% ./AttribContainer
Attrib A t val: 10
Attrib A u val: 12
Attrib A z0 val: 1.23
Attrib A z1 val: 5.954
Attrib A test val a: 1
Attrib A test val b: 2
Attrib A test val c: 3
Attrib A test val d: 4
Attrib A test type: f
Attrib A test type: i
Attrib A test type: St6vectorIfSaIfEE
Attrib A test type: 4TEST
```

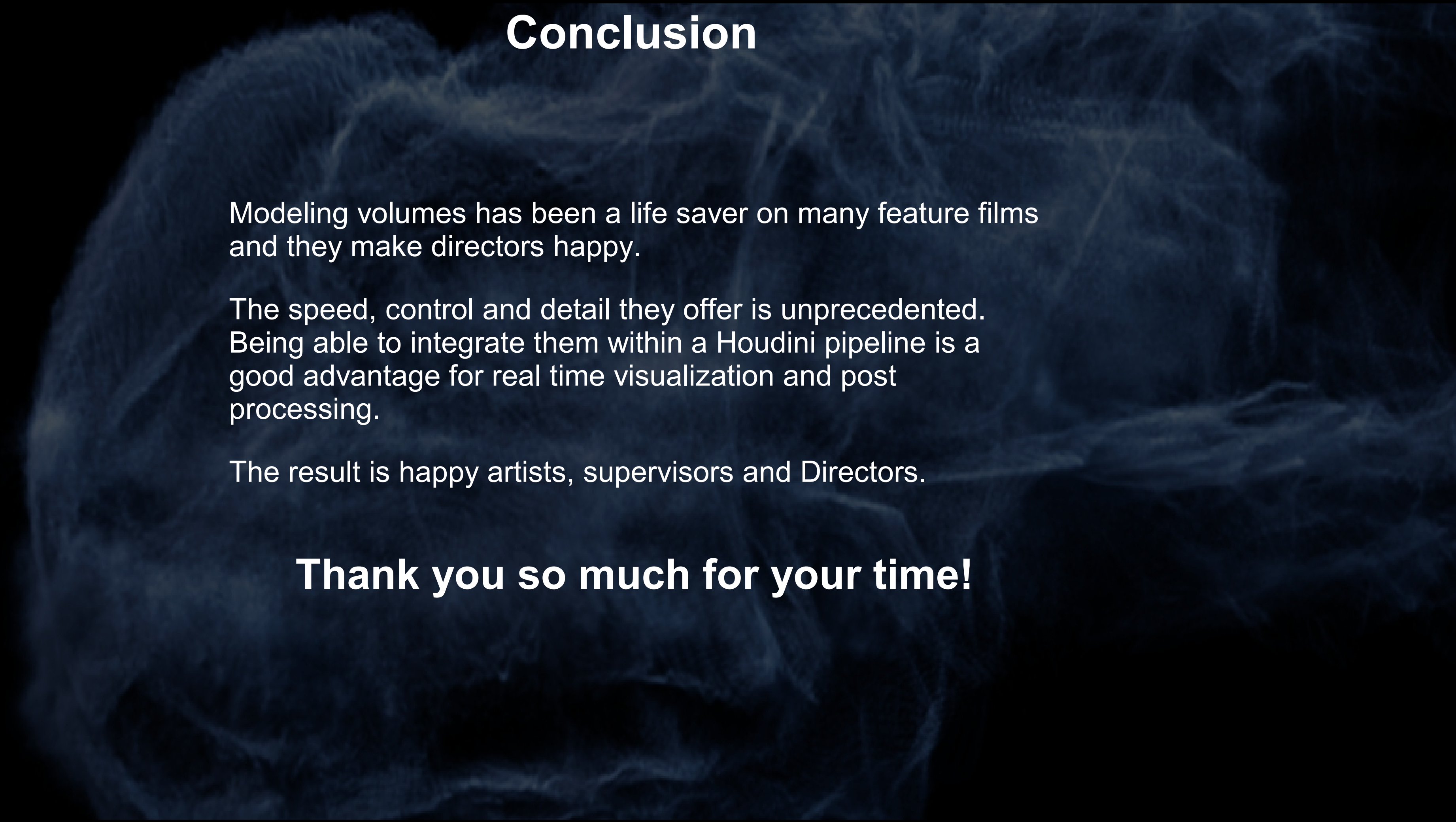


VIII. Improvements and TODOs

2. GPU Optimization.

Optimizing with the GPU requires 2 main tests to pass.

1. The sub containers should not be too small otherwise time would be wasted updating between GPU and CPU memory.
2. The sub voxels need to be bigger than a certain size but not bigger than what the GPU memory can handle.



Conclusion

Modeling volumes has been a life saver on many feature films and they make directors happy.

The speed, control and detail they offer is unprecedented. Being able to integrate them within a Houdini pipeline is a good advantage for real time visualization and post processing.

The result is happy artists, supervisors and Directors.

Thank you so much for your time!