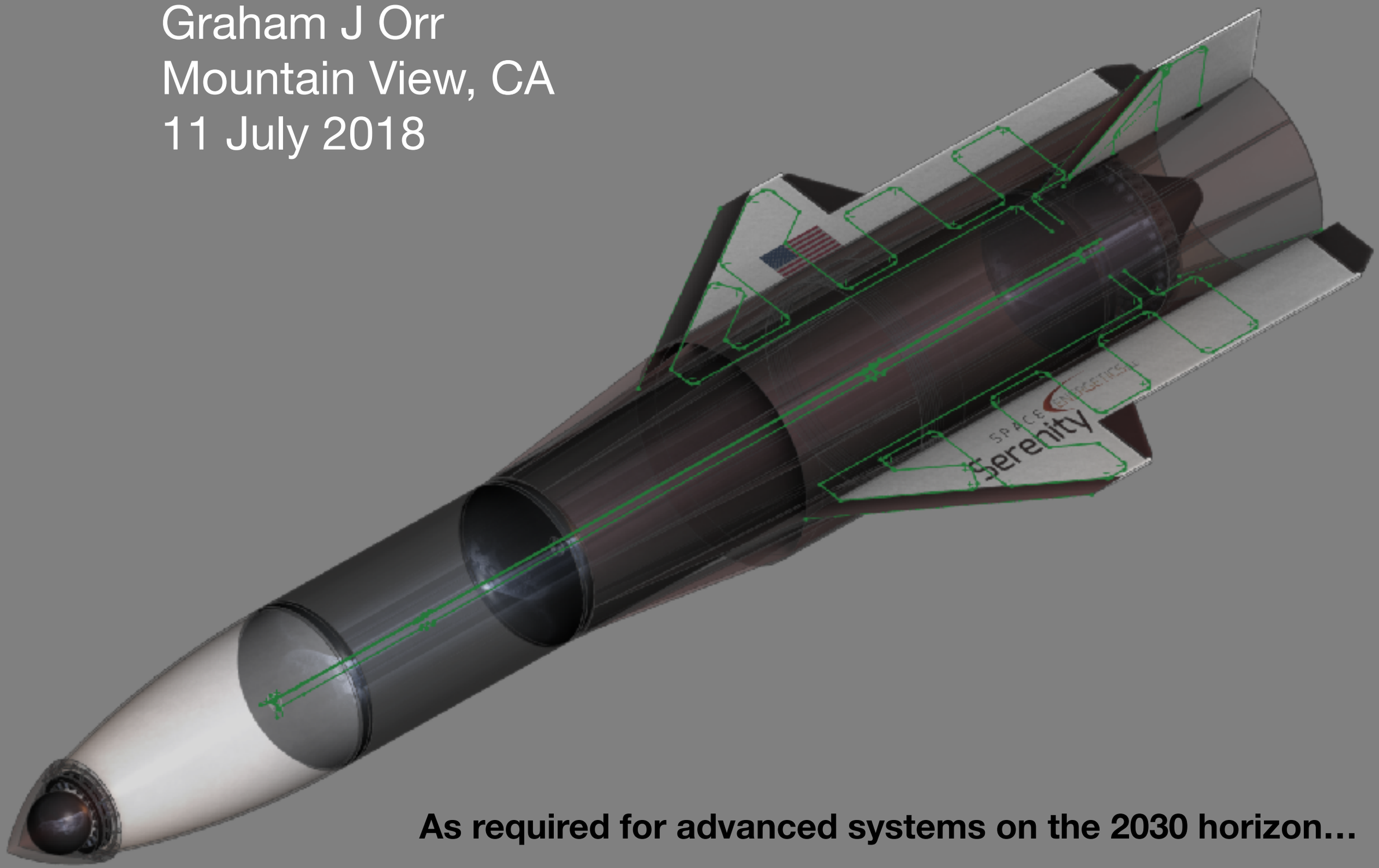
Xplicit Computing

ICCFD10-145

Unified Geometries for Dynamic HPC Modeling

Graham J Orr
Mountain View, CA
11 July 2018



As required for advanced systems on the 2030 horizon…

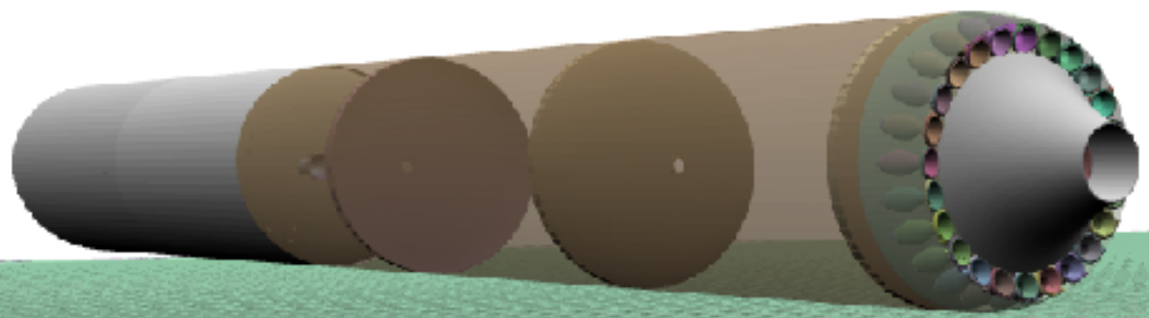
Xplicit Computing

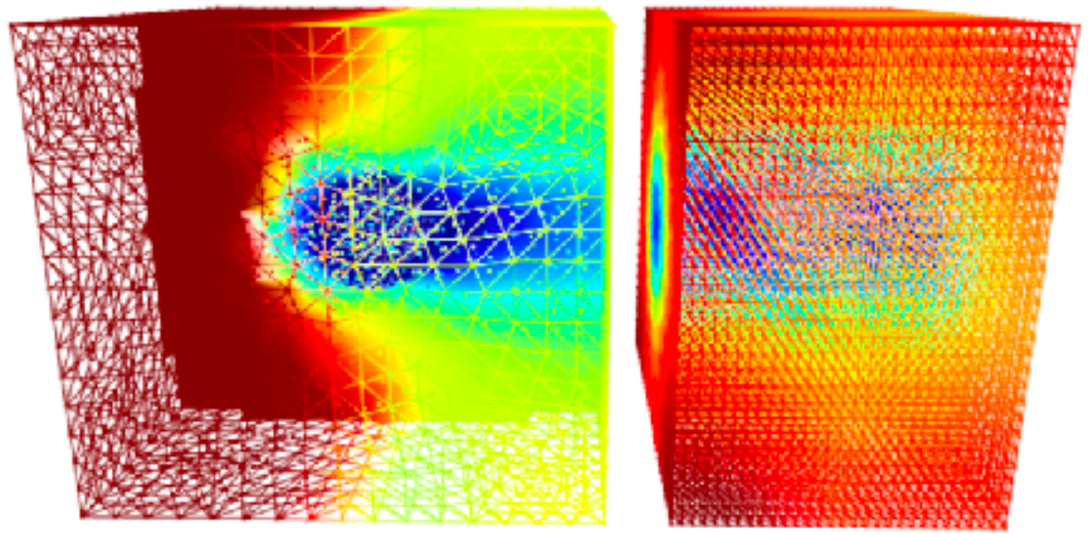
Application Examples

multi-domain multi-physics

ICCFD10-145

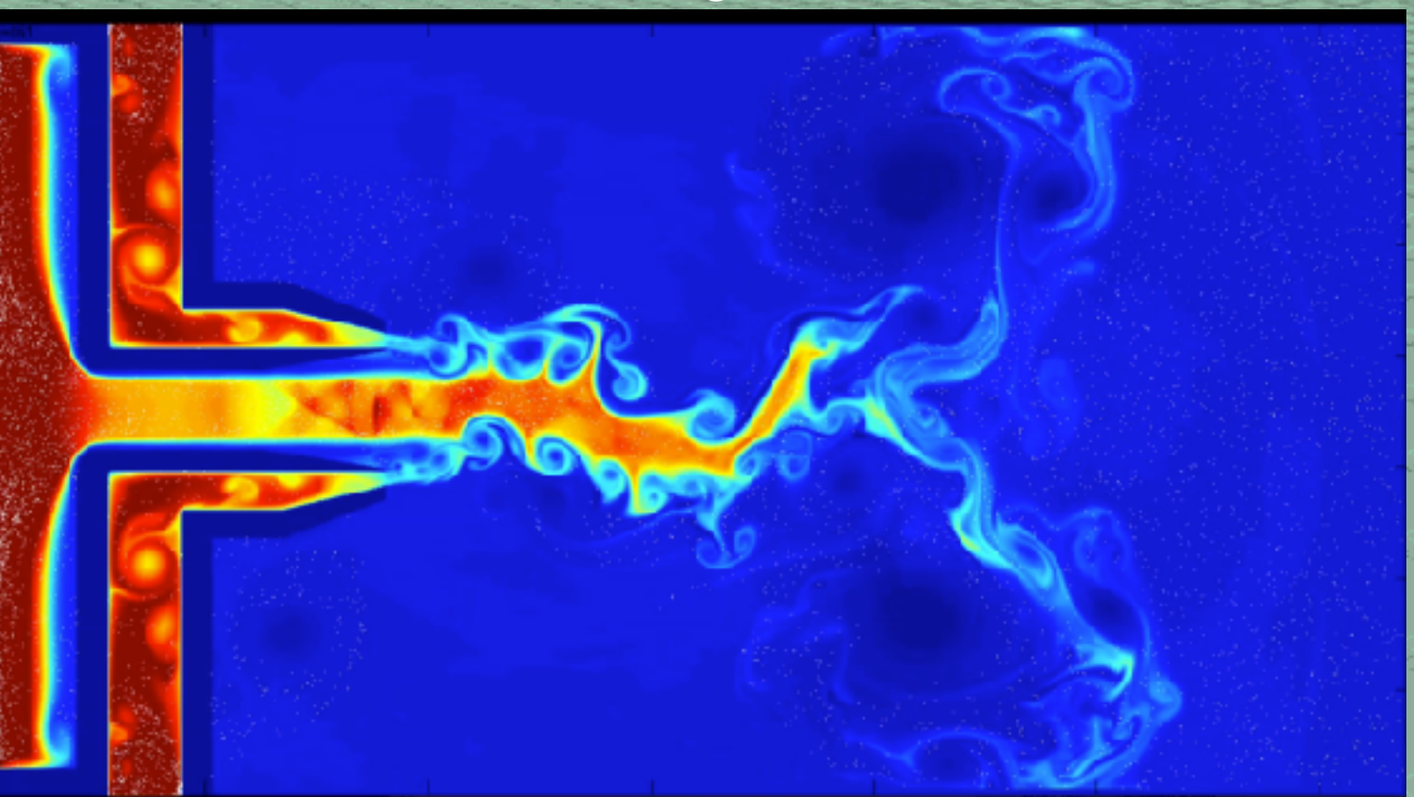
FDM RBF/SPH
FEM PIC
FVM DSMC
LBM RKPM
... ...

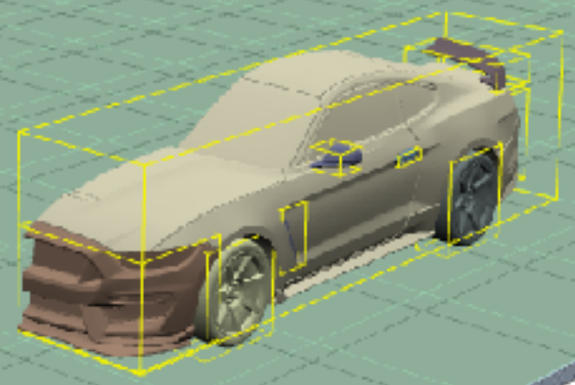




coupling disparate analyses

large sims on modest hardware

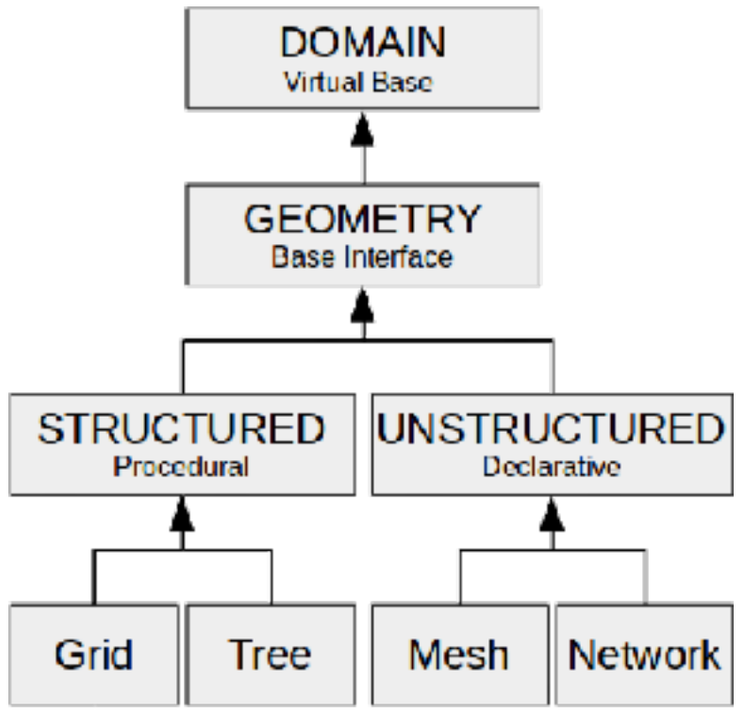




varying length scales

Geometry Virtual Base

Efficiency?



Flexibility?

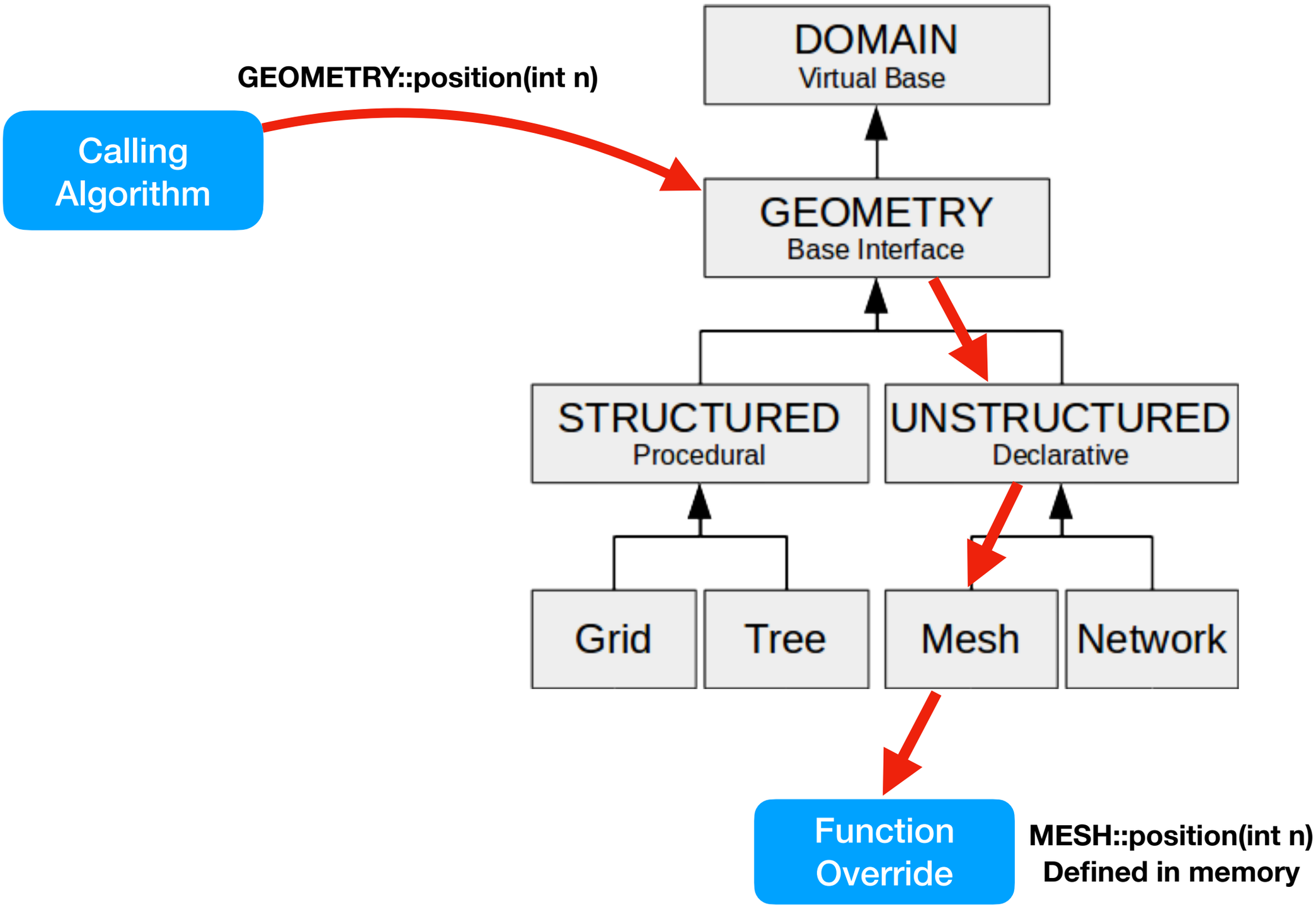
GEOMETRY is an abstract base class that derives from DOMAIN serving as a common programming interface for all geometry types. An algorithm can call through a generic GEOMETRY pointer (address) and its functions will redirect to variations available in the *vtable*. Derived types can also call static base versions of function overrides. Generic algorithms operate on any geometry type through a unified base, greatly-improving code reuse and algorithm modularity. At its core, a name, heterogeneous DATA record and revision:

```
std::string name;                          // name of the geometry
DATA data;                                 // scalar attributes, such as Position or SDF vectors
REVISION revision;                         // major and minor change counter
STRUCTURED* background{nullptr};           // ptr to background geometry (optional, see § 3.3)
```

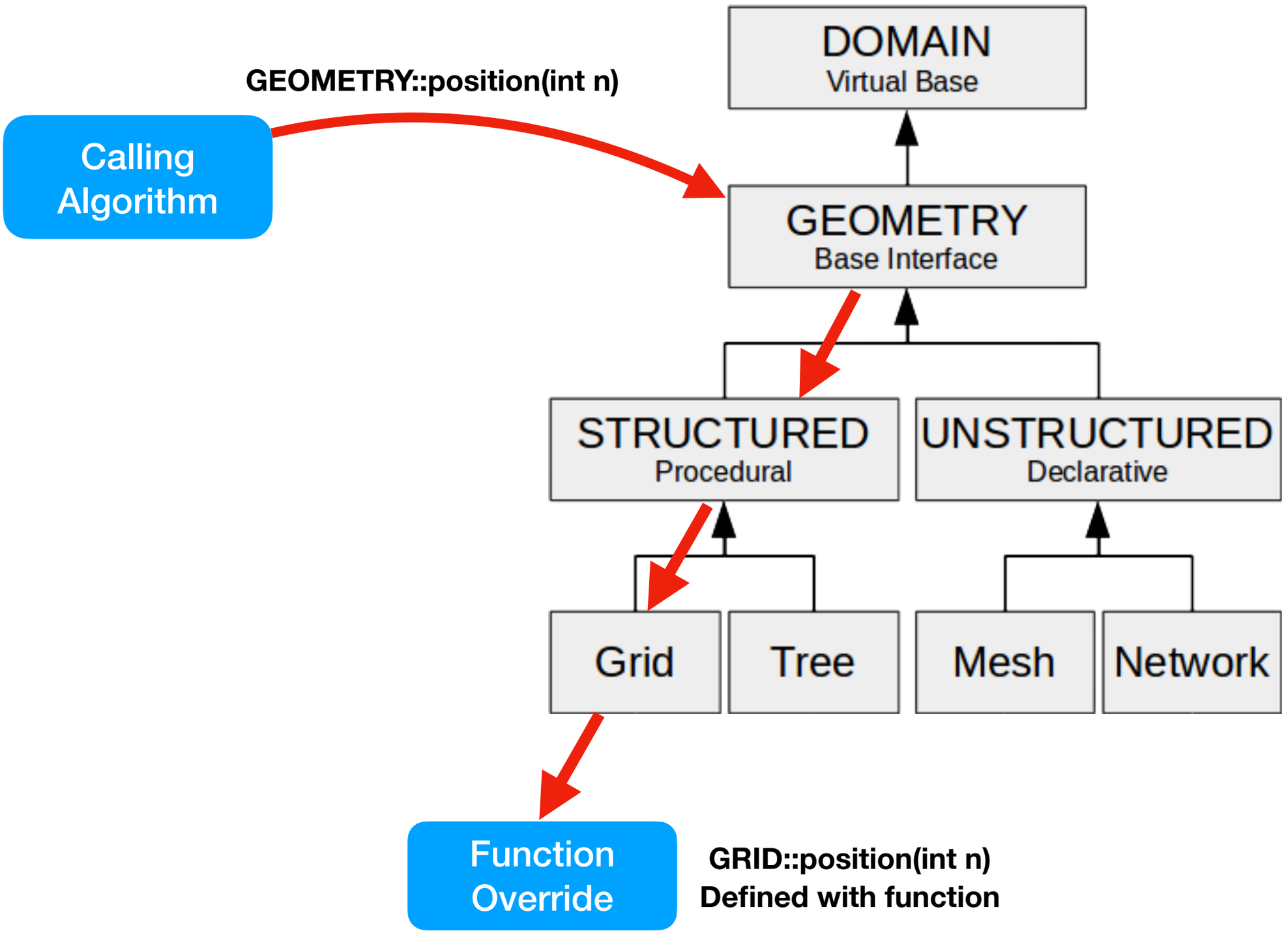
REGIONS may be defined by the user or algorithms zero through D dimensions for manifolds $\vec{x} \in \mathbb{R}^D$, including:

```
REGIONS<GROUP> groups;                     // 0d node groups of nodes
REGIONS<LOOP> loops;                       // 1d loops of edges
REGIONS<SURFACE> surfaces;                 // 2d surfaces of faces
REGIONS<VOLUME> volumes;                   // 3d volumes of cells
```

Functional Polymorphism
MESH Specialization

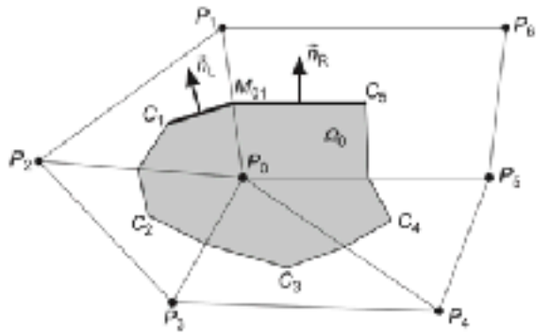


Functional Polymorphism
GRID Specialization



Xplicit Computing

Generic Spatial Algorithm Example



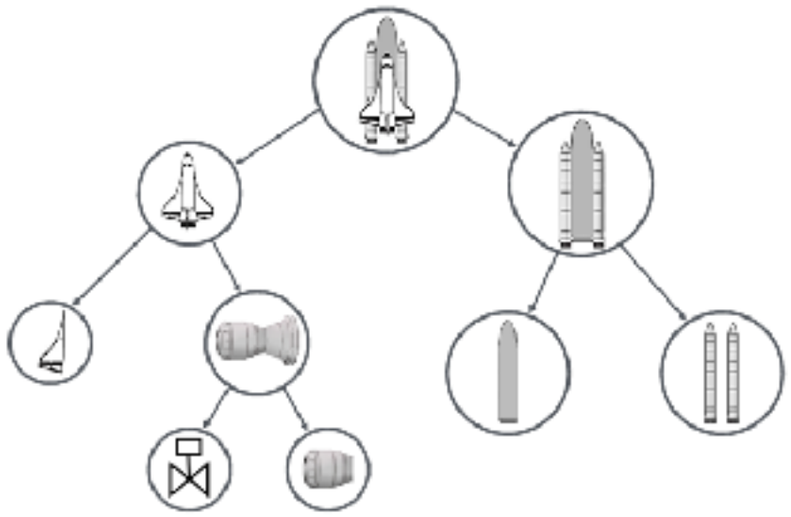
Gradients of any scalar field $\phi(\vec{x})$ can be sampled over K dual interfaces using *Green's Theorem* [1]:

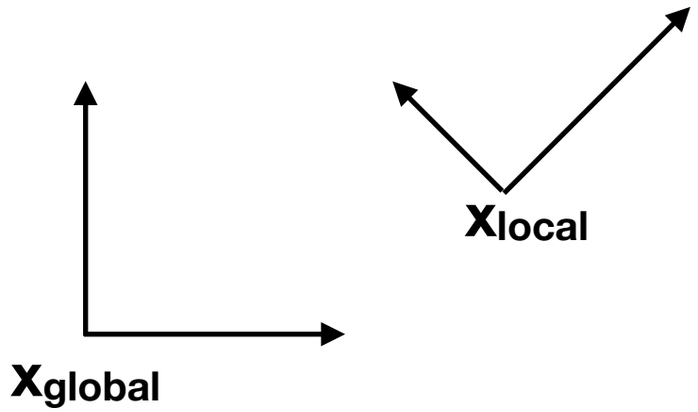
$$\nabla\phi(\vec{x})=\frac{\partial\vec{A}}{\partial V}\phi(\vec{x})=\sum_{k=0}^{K}\frac{\Delta\vec{A}_k}{\Delta V}[(1-\alpha)\phi(\vec{x})+\alpha\,\phi(\vec{x}_k)]+\epsilon(O(\vec{x})^2) \tag{1}$$

Where $\alpha=0.5$ for median sampling. Equivalent code computes gradients for any geometry type and scalar parameter:

```
void GreenGaussGradient(DATA& data, GEOMETRY& geometry, PropertyKey val_pk){
    VECTOR& values = data(val_pk);                          // get data vector for scalar property
    auto grad_pk = PropertyKey(val_pk, Gradient);           // create a new pk by appending gradient modifier
    VECTOR& gradient = data(grad_pk);                       // get data vector for gradient property
    #pragma omp parallel for
    for (int n = 0; n < geometry.nNode(); n++){             // iterate through all nodes (parallel)
      double volume = geometry.volume(n);                       // get volume at target dual node
      if (approxEqual(volume,0.))                               // skip if volume is nearly zero
            continue;
      gradient.row(n) = VECTOR::Zero(1, geometry.D);        // before first dual zero out gradient
      auto K = geometry.nNeighbor(n);                       // get number of neighbors for node n
      for (int k = 0; k < K; k++){                              // iterate through interior neighbors of n
          FACE face = geometry.getInteriorFace(n, k);               // get the interior dual node face n-k
          if(face.empty())                                          // skip fragment if face is empty
              continue;
          int& ns = face.indices[1];                                // flux source node index (opposing)
          double interfaceValue = .5 * (values(ns) + values(n));    // compute interface value as average
          for (auto d=0; d<geometry.D; d++)                         // for each spatial dimension
              gradient(n,d) += interfaceValue*face.area[d]/volume;  // sum gradient using Green's theorem
      }
    }
}
```

Definitions





5.2 Local & Global Coordinates

There is no preferred reference frame, but interacting systems require a common global frame. Positions (and spatial vectors) within GEOMETRY are by convention described in untransformed local coordinates. One *3x3* (or *4x4)* homogeneous model matrix per domain provides the absolute transformation between local and global space $\vec{x}_{global} = M_S \vec{x}_{local}$. Global coordinates for domain *S* are a product of local position with a global model matrix, constructed from the recursive product of local transforms in hierarchy:

$$M_S = \prod_{s=root}^{S} L_s \tag{31}$$

5.1 Heterogeneous Data – Regularity

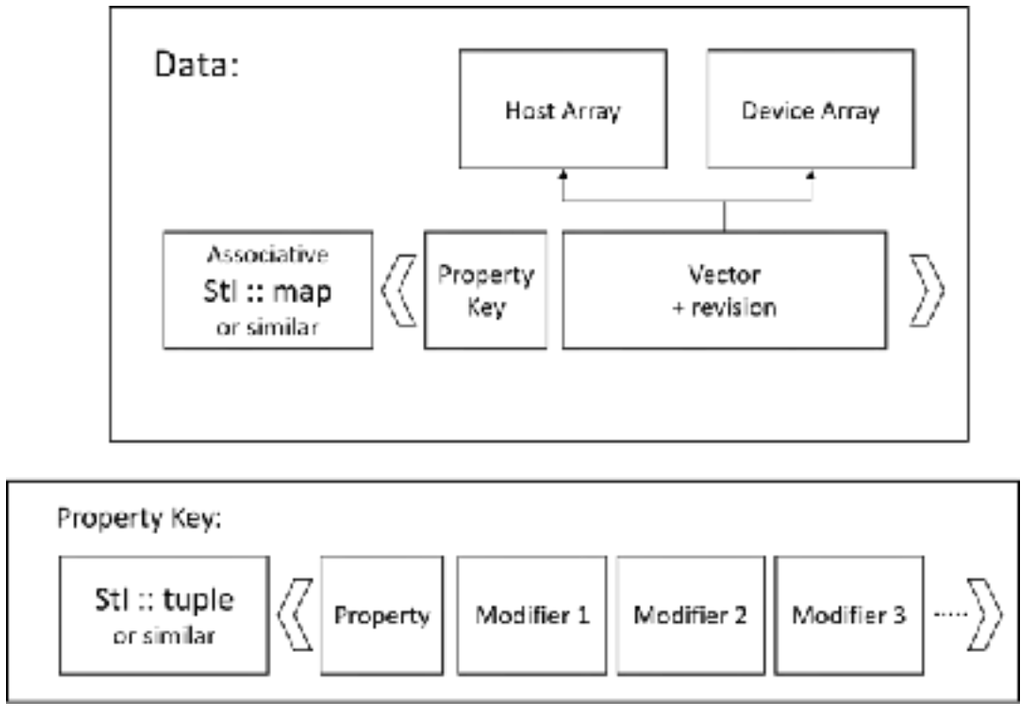
DATA defines a map-like container that manages multiple VECTOR entries by Property Key. VECTOR consists of a master host array, analogous device buffer, and revision integers for each. Property Key *pk* is a unique key composed of a property attribute followed by any number of modifiers (to facilitate lexicographical look-up in DATA). A heterogeneous (CPU-GPU) algorithm may have the following programming pattern:

Contiguous VECTOR representing element values within field $\phi(\vec{x})$ can be accessed from DATA:

```
if (!data.contains(pk))                  // check to see if data contains an entry for property key pk
      return false;                      // if not, do something else
VECTOR& values = data(pk);               // get a reference to values vector
```

Operate concurrently on host VECTOR using *OpenMP C++*:

```
double pi = Constants(PI)(0);            // get global data constant, first entry
#pragma omp parallel for                 // perform next for loop in parallel
for (int n=0; n<values.size(); n++)      // loop through all values
      values[n] = pi * c[n];             // perform some computation without write collision
```



Synchronize VECTOR to device in preparation for heterogeneous processing:

```
values.revision++;                    // increment revision to indicate host vector has changed
values.sync();                        // if device vector revision is less than host, update device vector
```

Algorithms and bound arguments consolidate code fragments into OpenCL program source. A kernel builder manages and resolves variable names based upon bindings. The parallel portion of the above loop may generate CL equivalent:

```
const int n = get_global_id(0);       // get this node (or element) index
values_arr[n] = pi_const * c_arr[n];  // perform some computation
```

Instructions are invoked from a script able to execute C++ or OpenCL versions of the algorithm. After kernel launch and processing, device values can be buffered back to CPU memory space using revision and synchronization:

```
values.device.revision++;             // increment device revision to indicate buffer has changed
values.sync();                        // if device revision is greater than host, update host vector
```

5.4 Computable Elements

ELEMENT is defined as a discrete polyhedral simplex consisting of one or more node vertexes and quadrature points, often generated through an automated construction process (e.g. mesh generator). In memory they can be represented in contiguous form, enabling concurrent functions to return a single element in constant time while processing. Access patterns remain universal for querying number of elements with *nElement*, or returning an actual element with *getElement* functions, while underlying machinery varies.

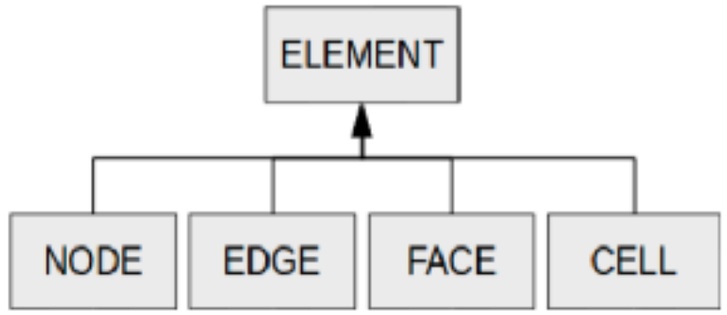


Illustration 10: Element Inheritance

ELEMENT defines a virtual simplex with base member attributes with up to E indices:

```
std::array<int, E> indices;       // indices of the nodes that constituent this element
int type;                         // simplex sub-type (Node1, Tri3, Tet4)
int id;                           // region identifier to map surfaces, volumes, edges, etc.
glm::dvec center;                 // element barycenter position
```

NODE, EDGE, FACE, CELL derive from ELEMENT and append additional member attributes:

```
double NODE::volume;              // dual-cell control volume around node
glm::dvec EDGE::length;           // distance vector between two nodes
glm::dvec FACE::area;             // area normal vector from barycenter
double CELL::volume;              // scalar volume
```

5.6 Regions of Interest

Indexing multiple elements into regions allows users and algorithms to define and select specific spaces (as a subset of the larger topology), such as required when applying boundary conditions to surfaces or immersive conditions within volumes, or simply for export. Uniform access patterns across elements and regions enables efficiency in programming, computation, and user interaction. Inherited region types specialize definition of new attributes and functions specific to dimensionality *d*.

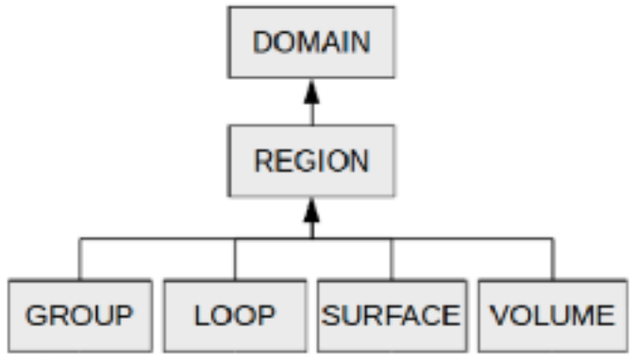


Illustration 12: Region Inheritance

REGION inherits from DOMAIN, specializing as a collection of elements of similar dimensionality (e.g. group of nodes, loop of edges, surface of faces, volume of cells):

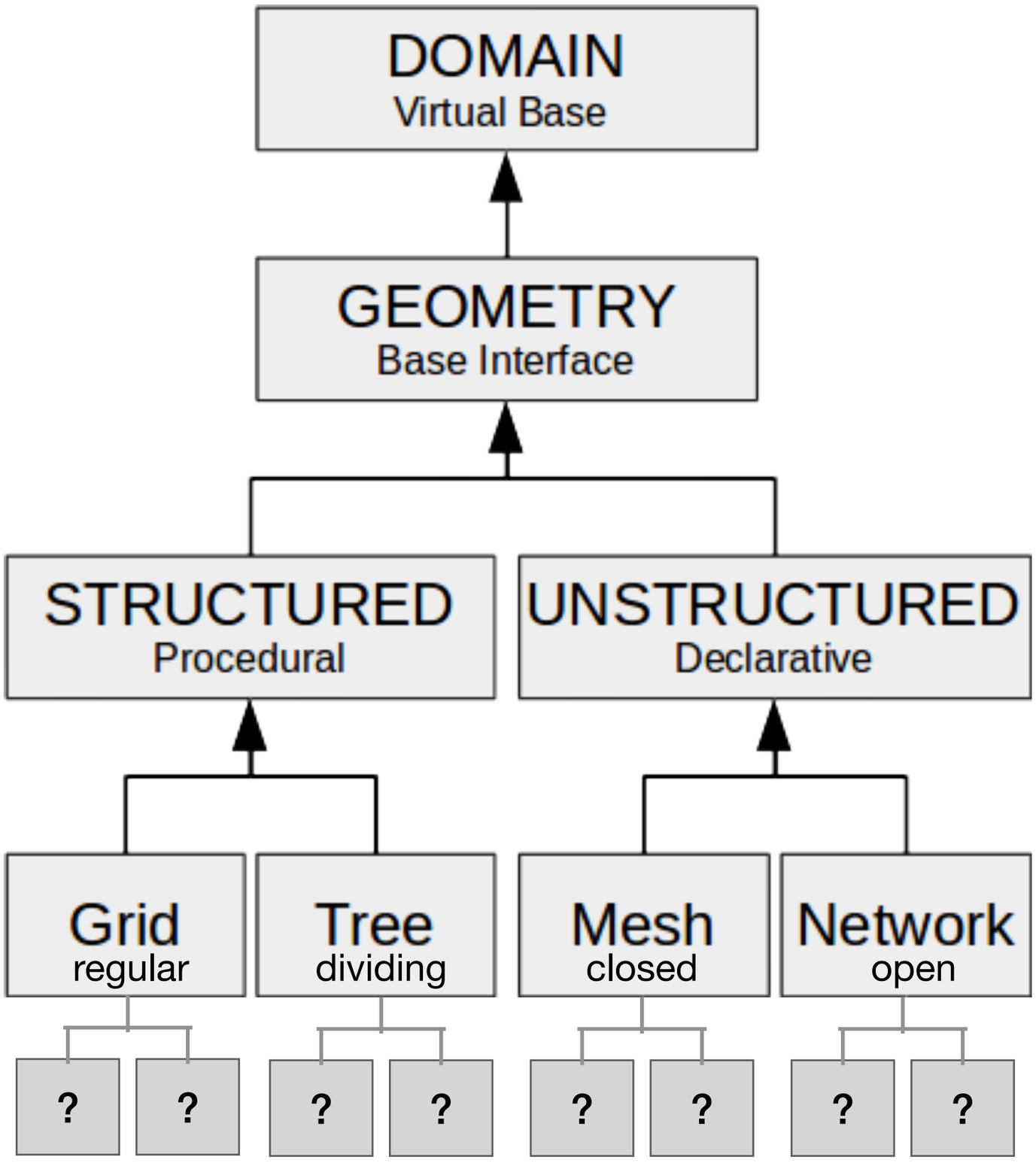
```
GEOMETRY* geometry;              // ptr to parent that owns this region
int id;                          // identifier linked to elements with same id
string name;                     // assigned literal name
std::set<int> nodes;             // unique node indices that comprise this region
int nElement, offset;            // regions don't store elements, but jump to entry via geometry
```

Region-Element-Simplex Compatibility

d	Region	Element	Simplex
0	GROUP	NODE	Node1
1	LOOP	EDGE	Line2, Line3**, Line4**
2	SURFACE	FACE	Tri3, Tri6*, Tri10**, Quad4, Quad8**, Quad9**
3	VOLUME	CELL	Tet4, Tet10*, Prism6*, Hex8, HeX20**, HeX27**

** beta, **scheduled*

Geometry Inheritance



Concrete class
-useful by itself

Abstract class
-not useful by itself

Abstract class
-not useful by itself

Concrete class
-useful

Xplicit Computing

Structured GRID

FDM
FEM
FVM
LBM

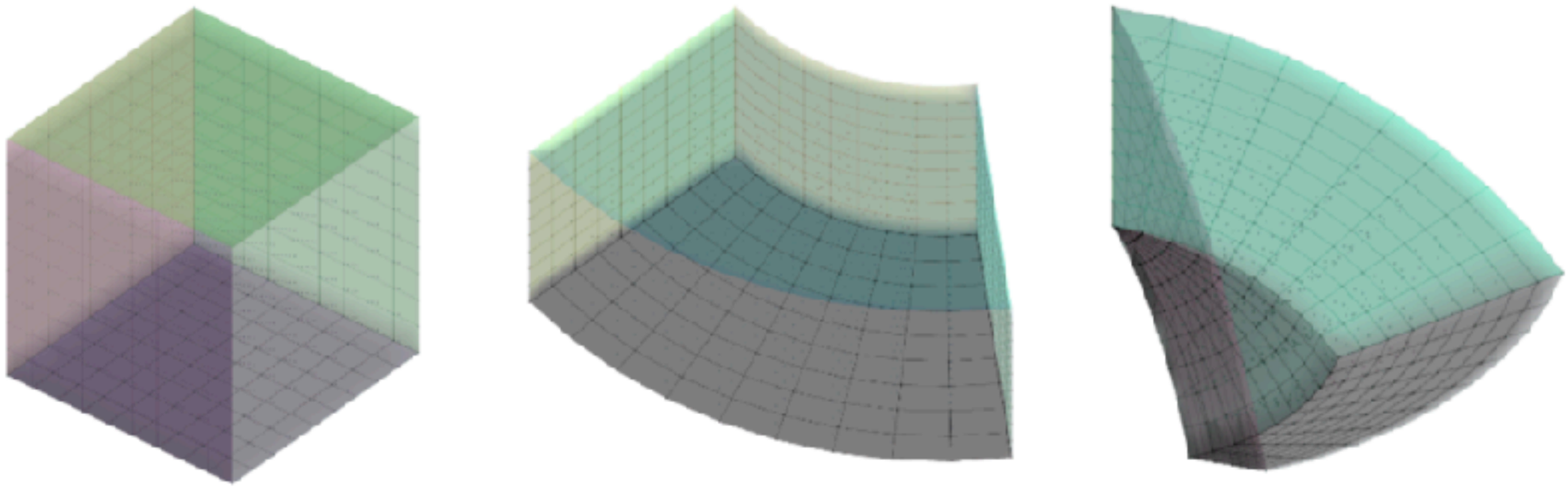
$$\phi(\vec{x}(\vec{k})) \quad \text{for} \quad \vec{k}=(i,j,k,l)\in\mathbb{Z}^D$$

Spatial vector entries are ordered left to right, map similarly to other spaces:

$$\vec{k}(i,j,k,l)\sim\vec{\xi}(\xi,\eta,\zeta,\tau)\sim\vec{x}(x,y,z,t)\sim\vec{\theta}(r,\theta,z,t)\sim\ldots$$

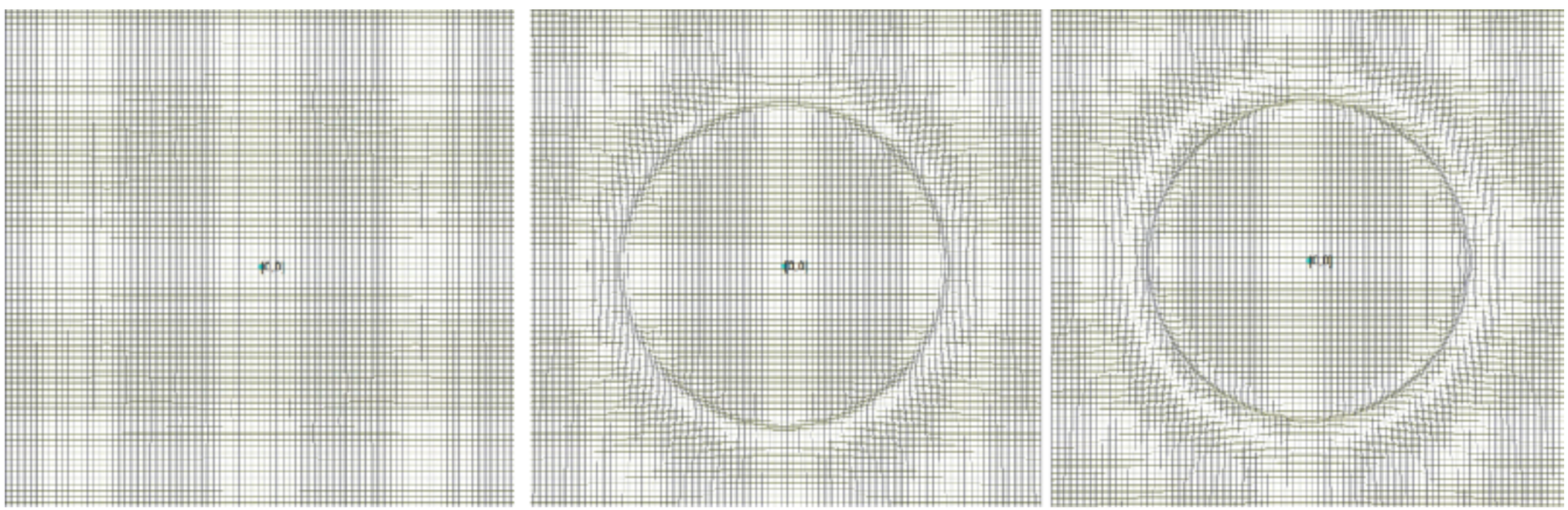
Orthogonal

$\vec{x}'=Q\vec{x}$ where $Q^{-1}=Q^{T}$



Local Tensor

$\Delta(Q\vec{x})<\Delta\vec{x}$



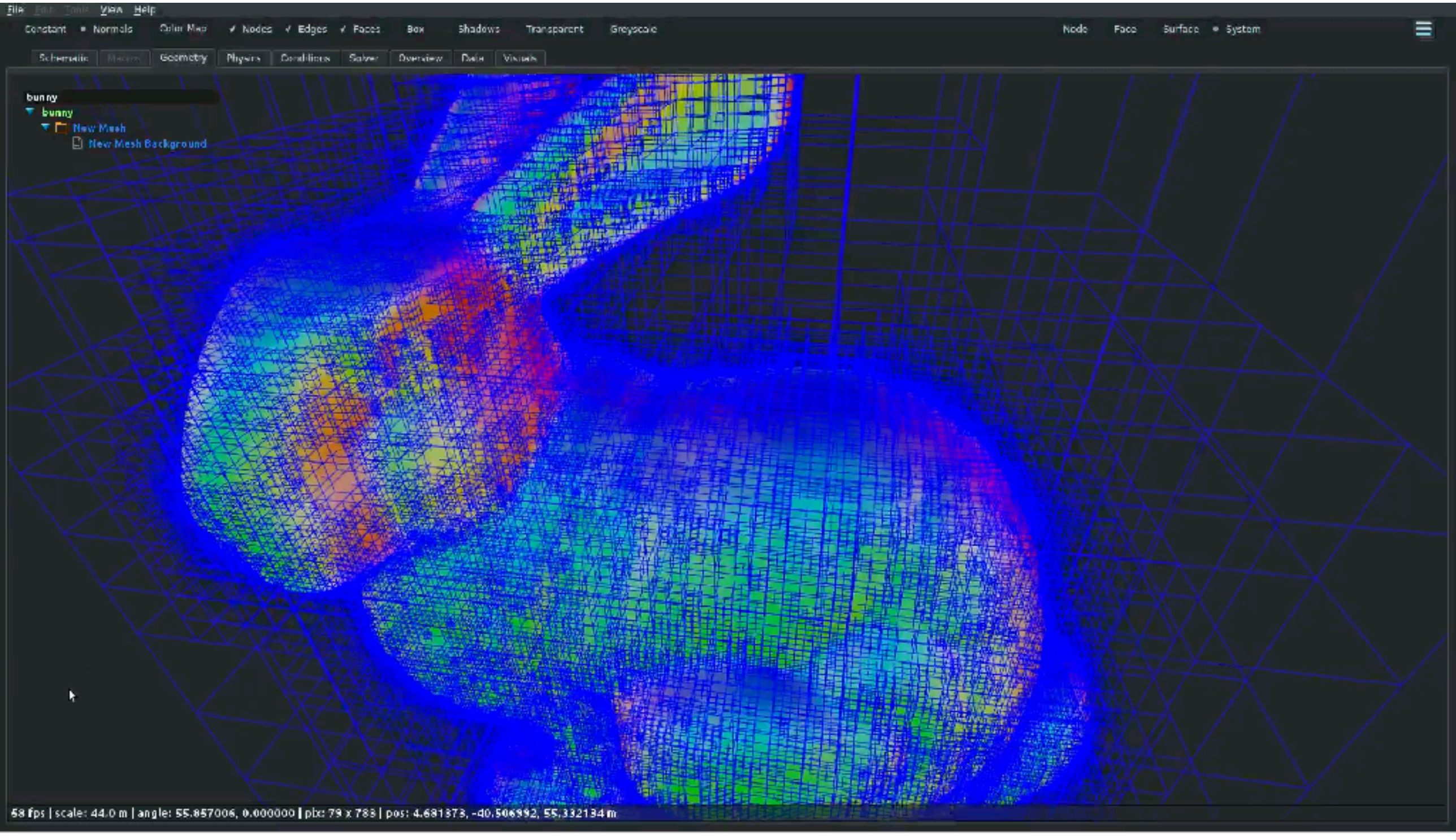
Structured TREE

Locations within $\bar{K}$ divide domain in powers of two up to max L levels deep (starting at root location $k_o=2^{L-1}$, computing all attributes from an encoded large integer for each spatial dimension. Location $\bar{k}$ has center position:

FVM

...?

$$\vec{x}(\vec{k})=\vec{x_{min}}+(\vec{x_{max}}-\vec{x_{min}})\vec{k}/2^{L} \tag{14}$$



Unstructured MESH

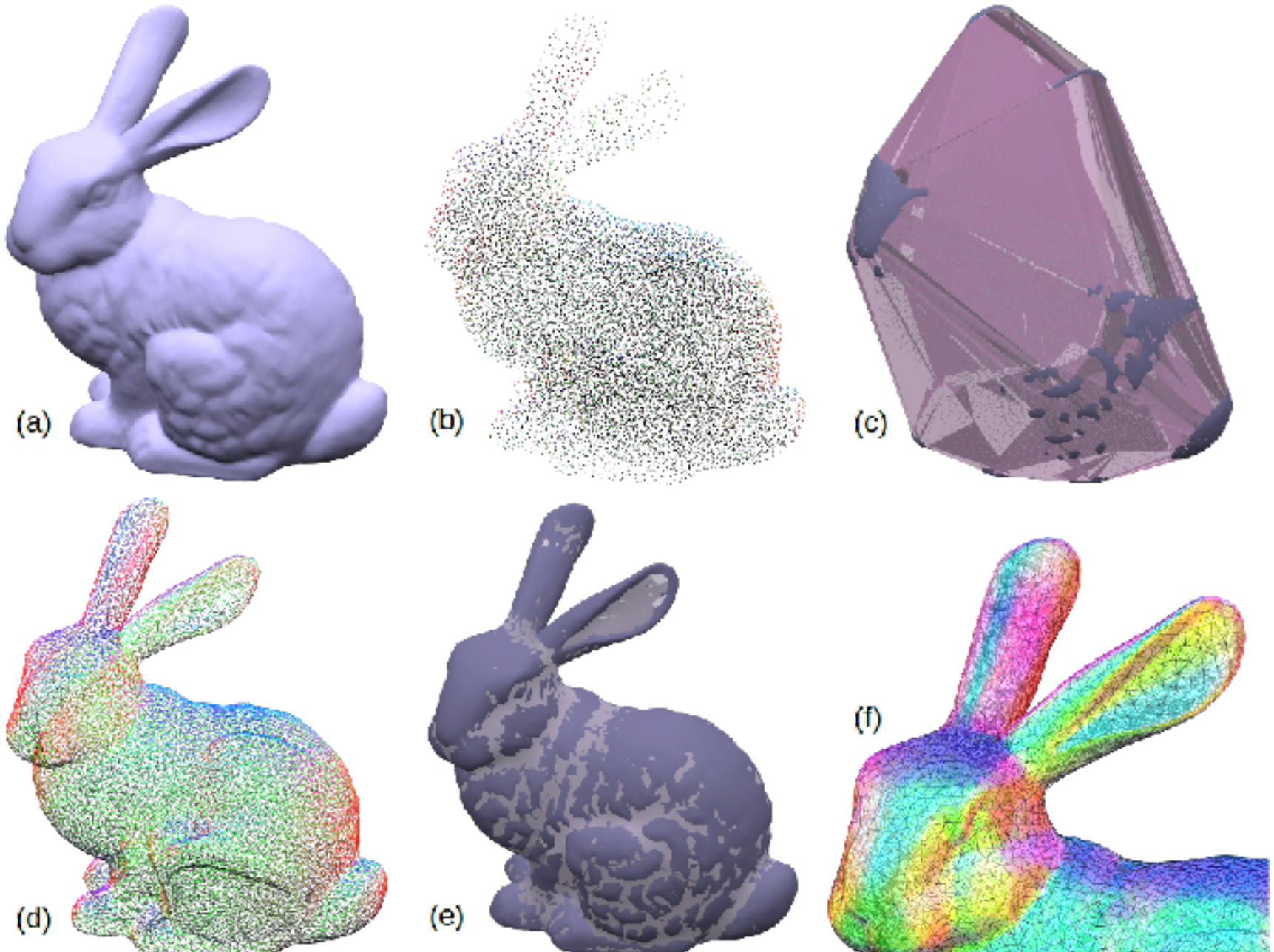
ICCFD10-145

FEM
FVM

$$\nabla \cdot \phi(\vec{x}) \stackrel{\text{def}}{=} 1 \quad \text{and} \quad \phi(A) \stackrel{\text{def}}{=} 0 \tag{16}$$

Initial nodes are distributed across valid (positive or negative) SDF locations, followed by improvement taking an edge-wise spring-truss analogy. Displacement is constrained normal to boundaries, while nodes near SDF-defined surfaces iteratively project to progressively conform:

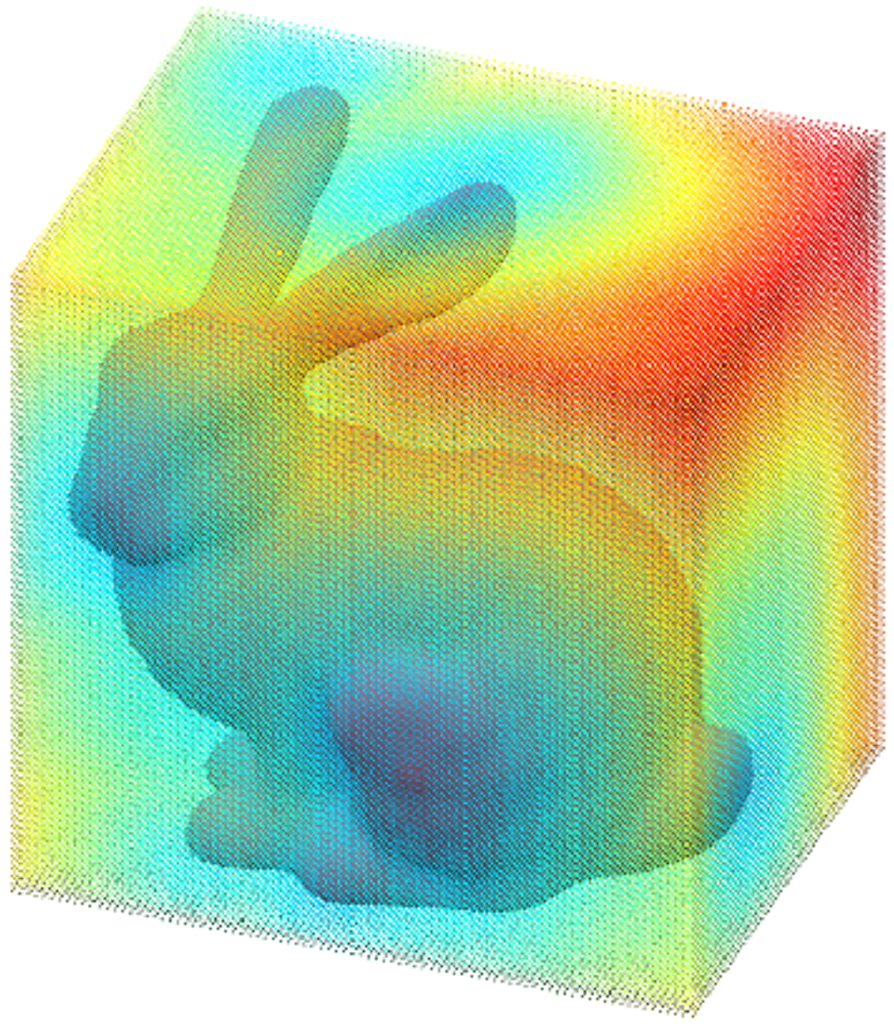
$$\Delta \vec{x} = \frac{\phi(\vec{x}) \nabla \phi}{|\nabla \phi|^2} + \epsilon(O(\vec{x})^2) \tag{17}$$



Background Geometries

Meshing
RBF
PIC
…?

- **Usually 1-way interpolation**
- **Often STRUCTURED for efficiency**
- **e.g. SDF around reference geometry**



$$\phi(\vec{x}) = \frac{1}{k_\Sigma} \sum_{n=0}^{N} k_n \phi_n^* \quad \text{, where} \quad k_\Sigma = \sum_{n=0}^{N} k_n$$

Structured Sampling

Type	Sample Count N	Weighting Coefficient k_n	Description
Grid	2^D	$\prod_{d=0}^{D} \left\lvert \frac{\Delta\vec{x} - \lvert\vec{x} - \vec{x}_n\rvert}{\Delta\vec{x}} \right\rvert_d$ (29)	linear volumetric fraction, node-centered
Tree	$l \cdot 2^D$	$\sum_{i=0}^{l} b_i \prod_{d=0}^{D} \left\lvert \frac{\Delta\vec{x}_i - \lvert\vec{x} - \vec{x}_n\rvert}{\Delta\vec{x}_i} \right\rvert_d$ (30)	recursive volumetric sum, cell-centered level weights b_i determine accuracy

Unstructured NETWORK

ICCFD10-145

FEM
RBF
PIC
…?

RBF ex: Nodes positions $\vec{x}$ are projected onto the background grid with the standard basis potential $\phi_o(\vec{x})$ resulting in net scalar potential function $\phi(\vec{x})$ as the superposition of the fields around each node. This can be described as the convolution of the standard potential with Dirac delta $\delta_n(\vec{x})$ for each node (with weights k_n, if desired) :

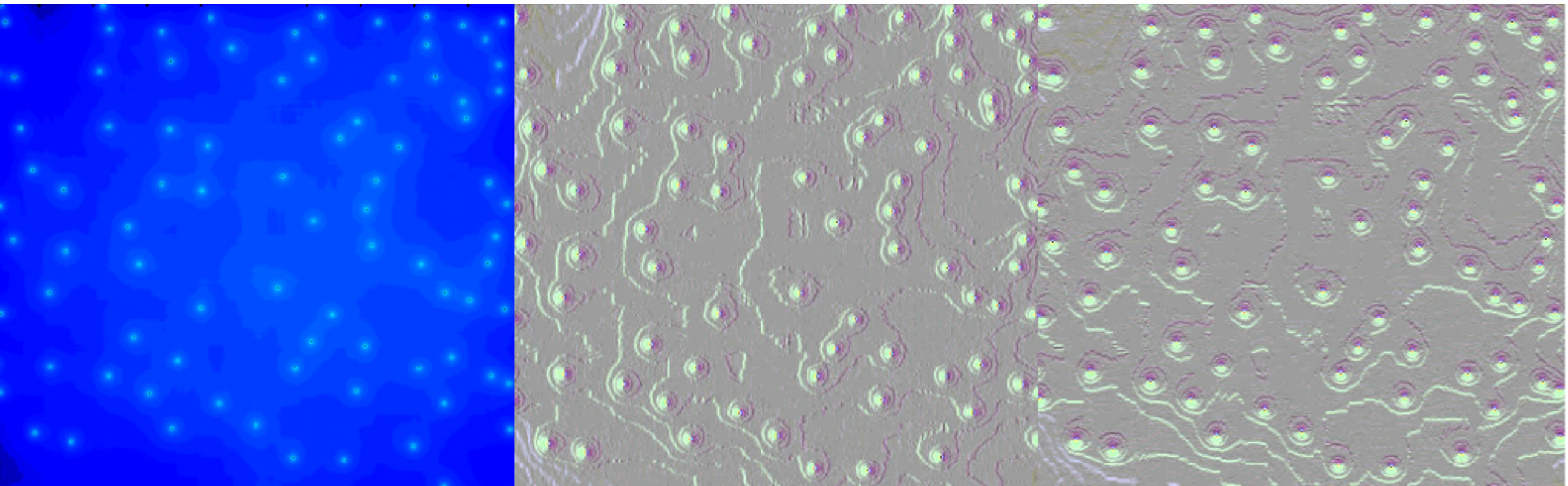
$$\phi(\vec{x}) = \sum_{n=0}^{N} k_n \phi_o(\vec{x}) * \delta_n(\vec{x}) \tag{25}$$

Net force field $\vec{f}(\vec{x})$ is determined from gradient of the potential function $\phi(\vec{x})$. Local forces are sampled at each node and pseudo-physical mass is applied to impose a timescale:

$$\vec{f}(\vec{x}) = -\nabla \phi(\vec{x}) = \frac{\partial(m\dot{\vec{x}})}{\partial t} \tag{26}$$

As an optimization, sum and gradients can be evaluated only locally around nodes. Non-linear second-order equation emerges, harmonic in $\vec{x}$:

$$m\ddot{\vec{x}} + \dot{m}\dot{\vec{x}} + \nabla \phi(\vec{x}) = \vec{0} \tag{27}$$



Virtual Function Table (vTable)

Spatial Common Functions

Function Name	Description	Arguments	Domain	Geometry	Str'd	Grid	Tree	Unstr'd	Network	Mesh
build	construction sequence			V		O	O		O	O
setKernelSource	define CL kernel source	incl, args, body, needed		V		O	O	O		
setKernelArgs	set CL kernel arguments	args, kernel, prefix		V		O	O	O		
clear	clear contents		V	O		O	O	O		
envelope	envelope nodes (w/wo mask)	spatial, group, reset	T							
setMinMax	set min and max	position, reset	F							
getGlobal	get global domain	glm::mat	F							
delta	difference between max and min		F							
center	center position calculation		F							
contains	contains position array	positions	T							
contains	contains position	position	V	O		O				
volume	total volume calculation		V	O		O				
intersects	intersects ray test	ray	V	O		O				
intersects	intersects another domain	domain	V	O		O				
position	get position from node index	index		V		O	O	O		
getElement*	get element from element index	index, type		V		O	O	O		
nElement*	get element count	type		V		O	O	O		
nNeighbor	get node neighbor count	index		V		O	O	O		
getAdjaceny	get element adjacency	index		V				O		
sample	sample values at position	position		V		O	O	O		
makeRegions	make regions from elements	elements		T						

Numerics Summary

4 Numerics Summary

Type	Characteristic	Supported Methods	Spatial Resolution (per dimension)Ω	Spatial Accuracy	Temporal Accuracy	IO Size (B/element)	RAM (B/element)Ξ	Speed-Up (GPU/CPU)Ψ
Mesh	closed simplex	FEM, FVM, RBF**	$1/\epsilon \approx 10^{15}$	1, 2	1, 2, +	240-360	800-2400	11-17x
Network**	open graph	FEM, PIC**, RBF**	$1/\epsilon \approx 10^{15}$	1	1	240-360 Γ	800-2400 Γ	TBD
Grid	regular indexed	FEM, FVM, LBM*, FDM**	$\sqrt[D]{2^{31}} \approx 10^{3}$	1, 2, +	1, 2, +	4	16-32	9-28x
Tree**	branch locations	FVM, FDM**	$2^{30} \approx 10^{9}$	1, 2	1, 2, +	16	16-32 Γ	TBD

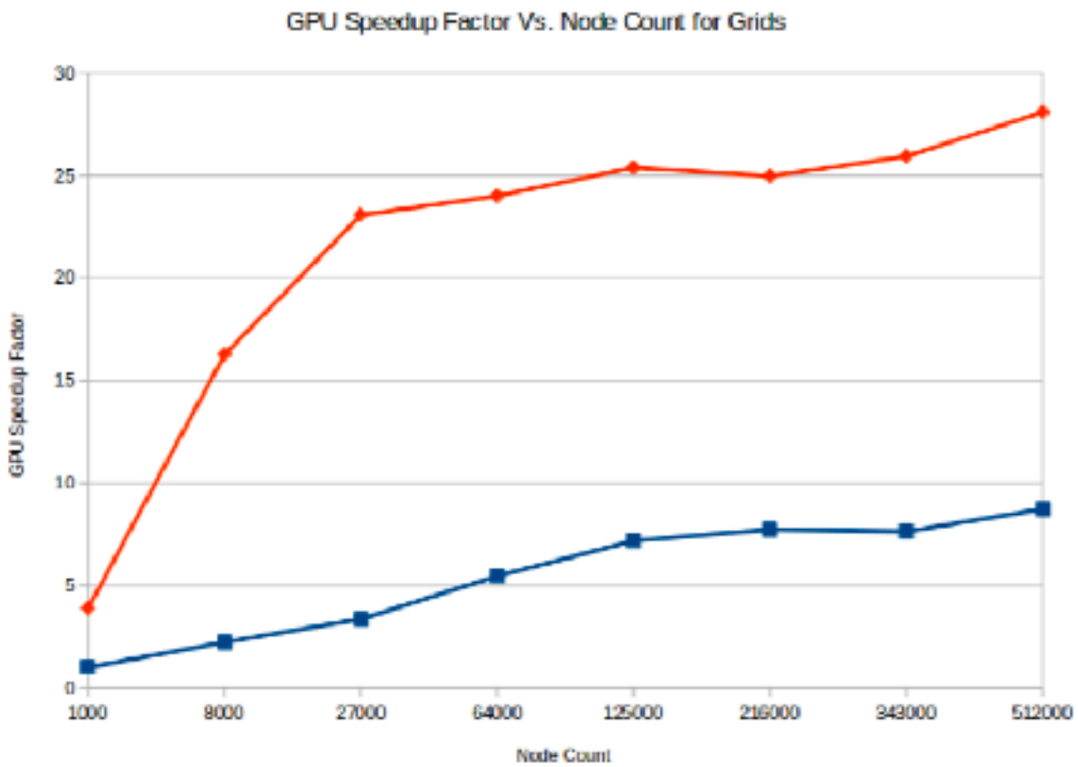
Ω assumes 64-bit-float, 32-bit integer index (~2B element max per geometry)
Ψ estimate based on single $1000 desktop with about 1M elements, see *Appendix B* [36]
Γ estimate inferred on similar branch attributes and functions
Ξ includes only geometry kernel CPU memory, not total application memory, assumes 3d

O(60) **O(50)** **O(20)**

2020 Predictions:

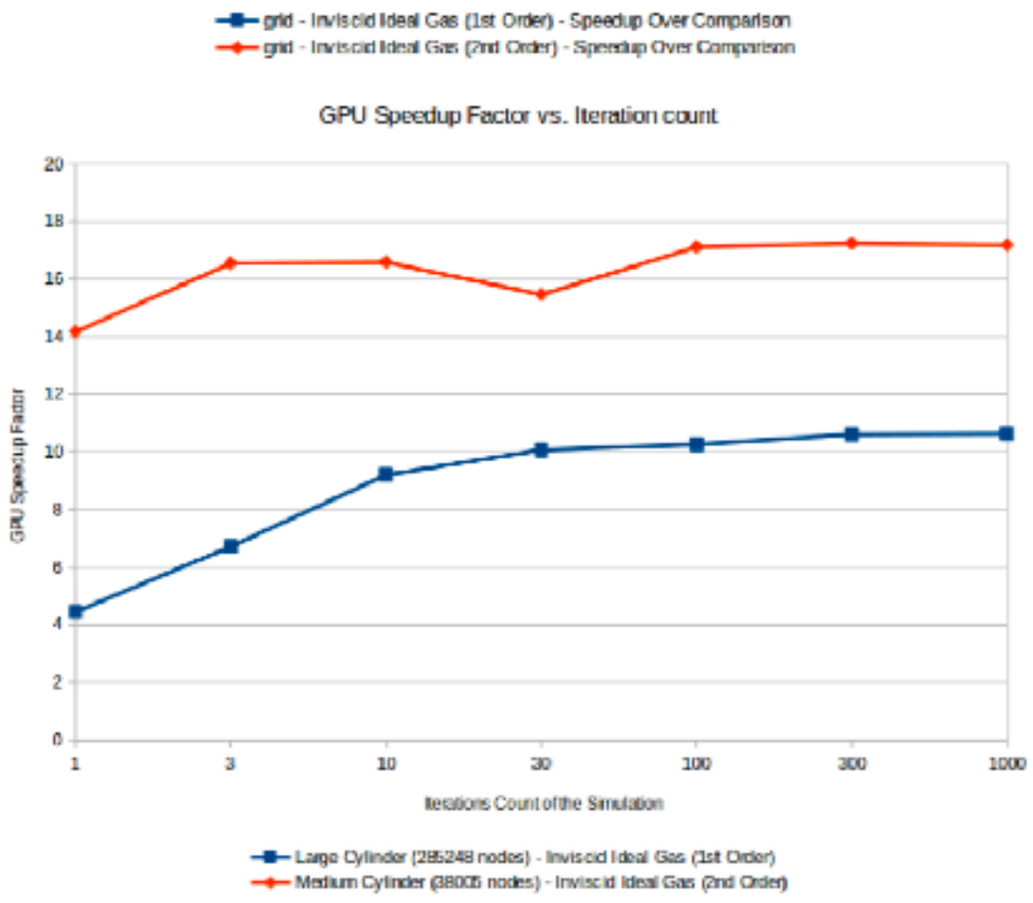
- **Trees for far-field volumes and background sampling**
- **Grids and Meshes for near-field layers**
- **Networks for dynamic multi-physics**
- **All types required, interacting**

Preliminary GPU/CPU Benchmarking



Intel i5-4670K CPU ~ $200
vs
GTX 780 GPU ~ $300

GPU about 10-30x faster
but are limited in memory



GPU w/ good FP64
about 5x cheaper over lifetime

Other devices such as TPU, FPGA can be equally disruptive co-processors

Conclusion

- **“Orthogonal” concepts leveraging computer science can potentially unify traditional and experimental infrastructure with few downsides.**
- **6K lines C++14, ~1/4 of server-side application**
- **Types have characteristic strengths and weaknesses, the union forming a complete basis for systems analysis.**
- **More than five families of numerics were implemented called by >100 algorithms. Novel methods underway.**
- **50-fold less memory and IO in structured vs unstructured, suggesting comparable speed and size increases.**
- **Geometry kernel benefits can be overshadowed by poor solver efficiency. Use higher-order (>2) unstructured strategically.**

Acknowledgements

- **Several contracted graduate researchers from Stanford, MIT, and Von Karman Institute contributed specific modules over three years.**
- **Special thanks for D Lortscher and IAP for early funding**