# EBGeometry

*Release 1.0*

**Robert Marskar**

**Sep 14, 2022**

# INTRODUCTION

This is the user documentation for EBGeometry, a small C++ package for computing signed distance fields from surface tesselations and analytic shapes. Although EBGeometry is a self-contained package, it is was originally written for usage with embedded boundary (EB) and immersed boundary (IB) codes.

EBGeometry does provide the *discrete geometry generation*, i.e. the generation of cut-cells from a geometry. It only takes care of the *geometry representation*, i.e. the creation of complex geometries as numerically efficient signed distance fields.

**Important:** The EBGeometry source code is found here. A separate Doxygen-generated API of EBGeometry is available here.

CHAPTER

# ONE

# INTRODUCTION

EBGeometry is a comparatively compact code for computing signed distance functions to watertight and orientable surface grids. Originally, it was written to be used with embedded-boundary (EB) codes like Chombo or AMReX.

## 1.1 Requirements

- A C++ compiler which supports C++14.

## 1.2 Quickstart

To obtained EBGeometry, clone the code from github:

```
git clone git@github.com:rmrsk/EBGeometry.git
```

EBGeometry is a header-only library and is comparatively simple to set up and use. To use it, make `EBGeometry.hpp` (stored at the top level) visible to your code and include it.

To compile the examples, navigate to the examples folder. Examples that begin by *EBGeometry_* are pure `EBGeometry` examples. Other examples that begin with e.g. `AMReX_` or `Chombo3_` are application code examples. These examples require the user to install additional third-party software.

To run the EBGeometry examples, navigate to e.g. `Examples/EBGeometry_DCEL` and compile and run the application code as follows:

```
g++ -O3 -std=c++14 main.cpp
./a.out porsche.ply
```

This will read `porsche.ply` (stored under `Examples/PLY`) and create a signed distance function from it.

## 1.3 Features

The basic features of EBGeometry are as follows:

- Representation of water-tight surface grids as signed distance fields. EBGeometry uses a doubly-connected edge list (DCEL) representation of the mesh.
- Various analytic distance functions.
- Bounding volume hierarchies (BVHs) for use as acceleration structures. The BVHs can be represented in full or compact (i.e., linearized) forms.

- Use of metaprogramming, which exists e.g. in order to permit higher-order trees and flexibility in BVH partitioning.
- Examples of how to couple EBGeometry to AMReX and Chombo.

CHAPTER

TWO

# CONCEPTS

## 2.1 Signed distance fields

The signed distance function is defined as a function $S : \mathbb{R}^3 \rightarrow \mathbb{R}$, and returns the *signed distance* to the object. The signed distance function has the additional property:

$$|\nabla S(\mathbf{x})| = 1 \quad \text{everywhere.} \tag{2.1}$$

Note that the normal vector is always

$$\mathbf{n} = \nabla S\left(\mathbf{x}\right).$$

In EBGeometry we use the following convention:

$$S(\mathbf{x}) = \begin{cases} > 0, & \text{for points outside the object,} \\ < 0, & \text{for points inside the object,} \end{cases}$$

which means that the normal vector $\mathbf{n}$ points away from the object.

Signed distance functions are also *implicit functions* (but the reverse statement is not true). For example, the signed distance function for a sphere with center $\mathbf{x}_0$ and radius $R$ can be written

$$S_{\text{sph}}\left(\mathbf{x}\right) = \left|\mathbf{x} - \mathbf{x}_0\right| - R.$$

An example of an implicit function for the same sphere is

$$I_{\text{sph}}\left(\mathbf{x}\right) = \left|\mathbf{x} - \mathbf{x}_0\right|^2 - R^2.$$

An important difference between these is the Eikonal property in Eq. 2.1, ensuring that the signed distance function always returns the exact distance to the object.

## 2.2 Transformations

Signed distance functions retain the Eikonal property for the following set of transformations:

- Rotations.
- Translations.

## 2.3 Unions

Unions of signed distance fields are also signed distance fields *provided that the objects do not intersect or touch*. For overlapping objects the signed distance function is not well-defined (since the interior and exterior are not well-defined).

For non-overlapping objects represented as signed distance fields $\left(S_1\left(\mathbf{x}\right), S_2\left(\mathbf{x}\right), \ldots\right)$, the composite signed distance field is

$$S\left(\mathbf{x}\right) = S_k\left(\mathbf{x}\right),$$

where $k$ is index of the closest object (which is found by evaluating $\left|S_i\left(\mathbf{x}\right)\right|$.

## 2.4 DCEL mesh structure

### 2.4.1 Basic concept

EBGeometry uses a doubly-connected edge list (DCEL) structure for storing surface meshes. The DCEL structures consist of the following objects:

- Planar polygons (facets).
- Half-edges.
- Vertices.

As shown in Fig. 2.1, the half-edge is the most important data structure. Half-edges circulate the inside of the facet, with pointer-access to the previous and next half-edge. A half-edge also stores a reference to it's starting vertex, as well as a reference to it's pair-edge. From the DCEL structure we can easily obtain all edges or vertices belonging to a single facet, and also "jump" to a neighboring facet by fetching the pair edge.

In EBGeometry the half-edge data structure is implemented in it's own namespace `EBGeometry::Dcel`. This is a comparatively standard implementation of the DCEL structure, supplemented by functions that permit signed distance computations to various features on such a mesh.

**Important:** A signed distance field requires a *watertight and orientable* surface mesh. If the surface mesh consists of holes or flipped facets, the signed distance function does not exist.

### 2.4.2 Signed distance

When computing the signed distance function, the closest point on the surface mesh can be one of the vertices, (half-) edges, or faces, see Fig. 2.2.

It is therefore necessary to distinguish between three cases:

1. **Facet/Polygon face**.

   When computing the distance from a point $\mathbf{x}$ to the polygon face we first determine if the projection of $\mathbf{x}$ to the face's plane lies inside or outside the face. This is more involved than one might think, and it is done by first computing the two-dimensional projection of the polygon face, ignoring one of the coordinates. Next, we determine, using 2D algorithms, if the projected point lies inside the embedded 2D representation of the polygon face. Various algorithms for this are available, such as computing the winding number, the crossing number, or the subtended angle between the projected point and the 2D polygon.

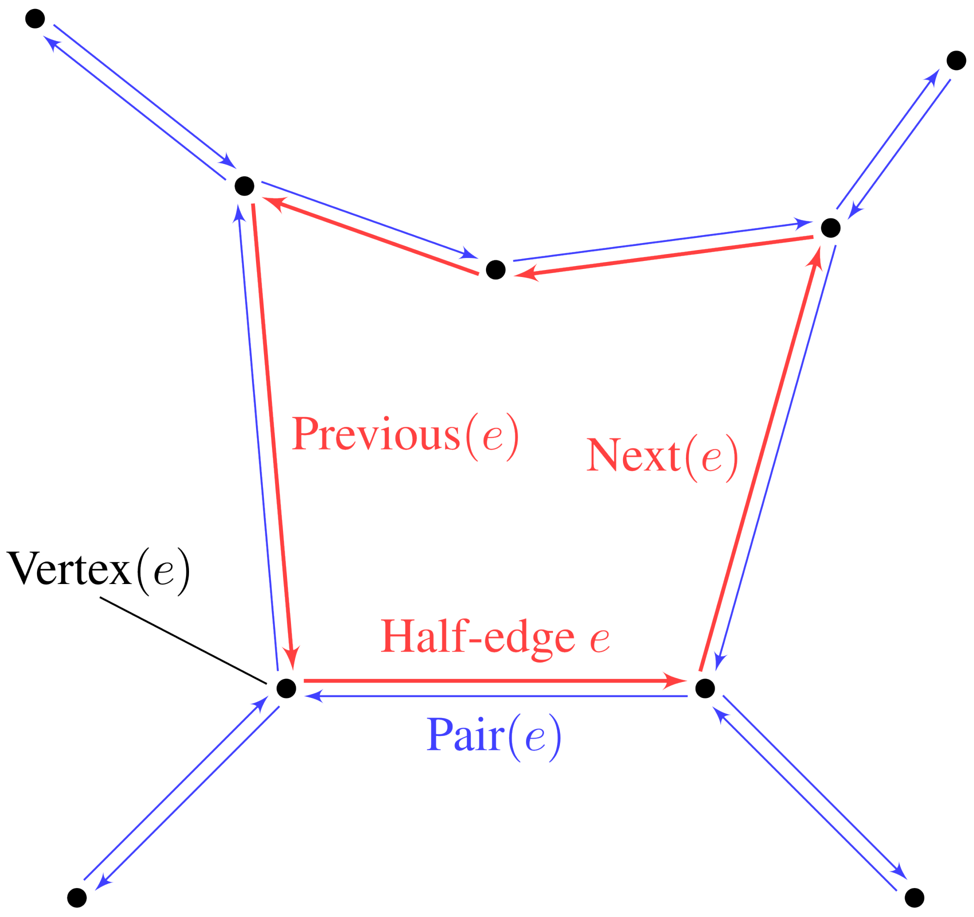

Fig. 2.1: DCEL mesh structure. Each half-edge stores references to previous/next half-edges, the pair edge, and the starting vertex. Vertices store a coordinate as well as a reference to one of the outgoing half-edges.

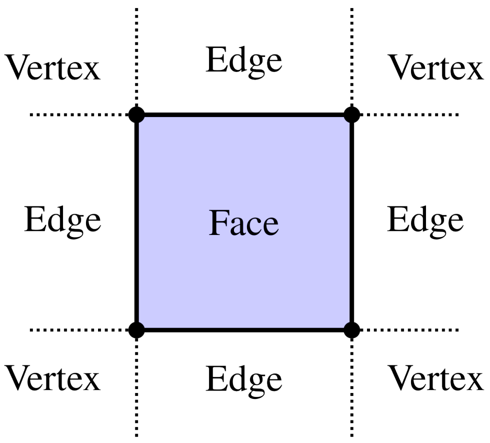

Fig. 2.2: Possible closest-feature cases after projecting a point $\mathbf{x}$ to the plane of a polygon face.

**Note:** EBGeometry uses the crossing number algorithm by default.

If the point projects to the inside of the face, the signed distance is just $d = \mathbf{n}_f \cdot (\mathbf{x} - \mathbf{x}_f)$ where $\mathbf{n}_f$ is the face normal and $\mathbf{x}_f$ is a point on the face plane (e.g., a vertex). If the point projects to *outside* the polygon face, the closest feature is either an edge or a vertex.

2. **Edge**.

   When computing the signed distance to an edge, the edge is parametrized as $\mathbf{e}(t) = \mathbf{x}_0 + (\mathbf{x}_1 - \mathbf{x}_0)\,t$, where $\mathbf{x}_0$ and $\mathbf{x}_1$ are the starting and ending vertex coordinates. The point $\mathbf{x}$ is projected to this line, and if the projection yields $t' \in [0, 1]$ then the edge is the closest point. In that case the signed distance is the projected distance and the sign is given by the sign of $\mathbf{n}_e \cdot (\mathbf{x} - \mathbf{x}_0)$ where $\mathbf{n}_e$ is the pseudonormal vector of the edge. Otherwise, the closest point is one of the vertices.

3. **Vertex**.

   If the closest point is a vertex then the signed distance is simply $\mathbf{n}_v \cdot (\mathbf{x} - \mathbf{x}_v)$ where $\mathbf{n}_v$ is the vertex pseudonormal and $\mathbf{x}_v$ is the vertex position.

### 2.4.3 Normal vectors

The normal vectors for edges $\mathbf{n}_e$ and vertices $\mathbf{n}_v$ are, unlike the facet normal, not uniquely defined. For both edges and vertices we use the pseudonormals from [1]:

$$\mathbf{n}_e = \frac{1}{2}\left(\mathbf{n}_f + \mathbf{n}_{f'}\right).$$

where $f$ and $f'$ are the two faces connecting the edge. The vertex pseudonormal are given by

$$\mathbf{n}_v = \frac{\sum_i \alpha_i \mathbf{n}_{f_i}}{\left|\sum_i \alpha_i\right|},$$

where the sum runs over all faces which share $v$ as a vertex, and where $\alpha_i$ is the subtended angle of the face $f_i$, see Fig. 2.3.

## 2.5 Bounding volume hierarchies

### 2.5.1 Basic concept

Bounding Volume Hierarchies (BVHs) are comparatively simple data structures that can accelerate closest-point searches by orders of magnitude. BVHs are tree structures where the regular nodes are bounding volumes that enclose all geometric primitives (e.g. polygon faces) further down in the hierarchy. This means that every node in a BVH is associated with a *bounding volume*. The bounding volume can, in principle, be any type of volume. Moreover, there are two types of nodes in a BVH:

- **Regular nodes.** These do not contain any of the primitives/objects. They are also called interior nodes, and store references to their child nodes.
- **Leaf nodes.** These lie at the bottom of the BVH tree and each of them contain a subset of the geometric primitives.

Fig. 2.4 shows a concept of BVH partitioning of a set of triangles. Here, $P$ is a regular node whose bounding volume encloses all geometric primitives in it's subtree. It's bounding volume, an axis-aligned bounding box or AABB for short, is illustrated by a dashed rectangle. The interior node $P$ stores references to the leaf nodes $L$ and $R$. As shown in Fig. 2.4, $L$ contains 5 triangles enclosed by another AABB. The other child node $R$ contains 6 triangles that are also

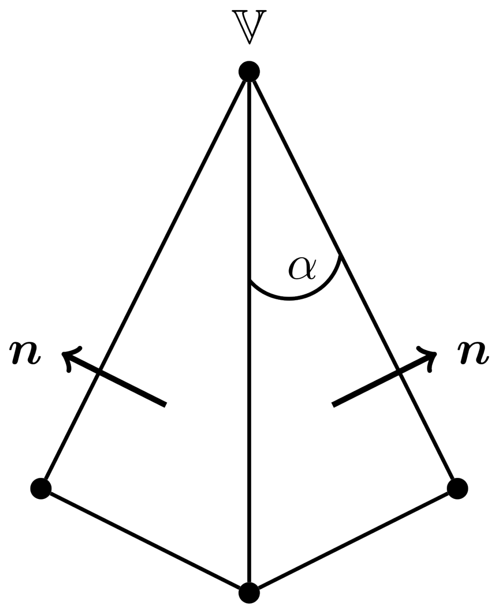

Fig. 2.3: Edge and vertex pseudonormals.

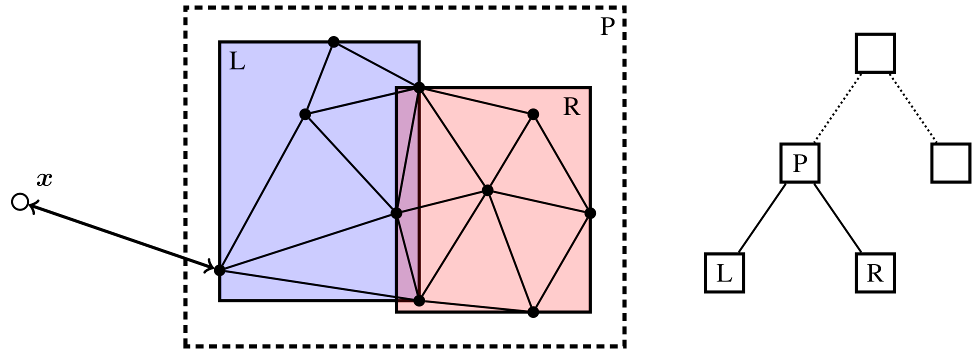

Fig. 2.4: Example of BVH partitioning for enclosing triangles. The regular node $P$ contains two leaf nodes $L$ and $R$ which contain the primitives (triangles).

enclosed by an AABB. Note that the bounding volume for $P$ encloses the bounding volumes of $L$ and $R$ and that the bounding volumes for $L$ and $R$ contain a small overlap.

There is no fundamental limitation to what type of primitives/objects can be enclosed in BVHs, which makes BVHs useful beyond triangulated data sets. For example, analytic signed distance functions can also be embedded in BVHs, provided that we can construct a bounding volume that encloses the object.

**Note:** EBGeometry limited to binary trees, but supports $k$ -ary trees where each regular node has $k$ children nodes.

### 2.5.2 Construction

BVHs have extremely flexible rules regarding their construction. For example, the child nodes $L$ and $R$ in Fig. 2.4 could be partitioned in any number of ways, with the only requirement being that each child node gets at least one triangle.

Although the rules for BVH construction are highly flexible, performant BVHs are completely reliant on having balanced trees with the following heuristic properties:

- **Tight bounding volumes** that enclose the primitives as tightly as possible.
- **Minimal overlap** between the bounding volumes.
- **Balanced**, in the sense that the tree depth does not vary greatly through the tree, and there is approximately the same number of primitives in each leaf node.

Construction of a BVH is usually done recursively, from top to bottom (so-called top-down construction). Alternative construction methods also exist, but are not used in EBGeometry. In this case one can represent the BVH construction of a $k$ -ary tree is done through a single function:

$$\text{Partition}\left(\vec{O}\right) : \vec{O} \rightarrow \left(\vec{O}_1, \vec{O}_2, \ldots, \vec{O}_k\right), \tag{2.2}$$

where $\vec{O}$ is an input a list of objects/primitives, which is *partitioned* into $k$ new list of primitives. Note that the lists $\vec{O}_i$ do not contain duplicates, there is a unique set of primitives associated in each new leaf node. Top-down construction can thus be illustrated as a recursive procedure:

```
topDownConstruction(Objects):
   partitionedObjects = Partition(Objects)

   forall p in partitionedObjects:
      child = insertChildNode(newObjects)

      if(enoughPrimitives(child)):
         child.topDownConstruction(child.objects)
```

In practice, the above procedure is supplemented by more sophisticated criteria for terminating the recursion, as well as routines for creating the bounding volumes around the newly inserted nodes. EBGeometry provides these by letting the top-down construction calls take polymorphic lambdas as arguments for partitioning, termination, and bounding volume construction.

### 2.5.3 Signed distance function

When computing the signed distance function to objects embedded in a BVH, one takes advantage of the hierarchical embedding of the primitives. Consider the case in Fig. 2.5, where the goal of the BVH traversal is to minimize the number of branches and nodes that are visited. For the traversal algorithm we consider the following steps:

- When descending from node $P$ we determine that we first investigate the left subtree (node $A$) since its bounding volume is closer than the bounding volumes for the other subtree. The other subtree will is investigated after we have recursed to the bottom of the $A$ subtree.
- Since $A$ is a leaf node, we find the signed distance from $\mathbf{x}$ to the primitives in $A$. This requires us to iterate over all the triangles in $A$.
- When moving back to $P$, we find that the distance to the primitives in $A$ is shorter than the distance from $\mathbf{x}$ to the bounding volume that encloses nodes $B$ and $C$. This immediately permits us to prune the entire subtree containing $B$ and $C$.

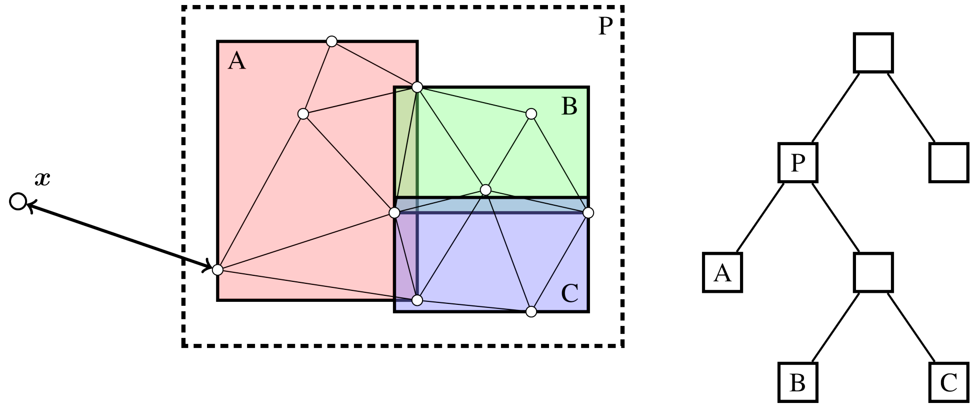

Fig. 2.5: Example of BVH tree pruning.

**Warning:** Note that all BVH traversal algorithms become inefficient when the primitives are all at approximately the same distance from the query point. For example, it is necessary to traverse almost the entire tree when one tries to compute the signed distance at the origin of a tessellated sphere.

CHAPTER

THREE

# IMPLEMENTATION

## 3.1 Overview

Here, we consider the basic EBGeometry API. EBGeometry is a header-only library, implemented under it's own namespace `EBGeometry`. Various major components, like BVHs and DCEL, are implemented under namespaces `EBGeometry::BVH` and `EBGeometry::DCEL`. Below, we consider a brief introduction to the API and implementation details of EBGeometry.

## 3.2 Vector types

EBGeometry runs it's own vector types `Vec2T` and `Vec3T`.

`Vec2T` is a two-dimensional Cartesian vector. It is templated as

```
namespace EBGeometry {
   template<class T>
   class Vec2T {
   public:
      T x; // First component.
      T y; // Second component.
   };
}
```

Most of EBGeometry is written as three-dimensional code, but `Vec2T` is needed for DCEL functionality when determining if a point projects onto the interior or exterior of a planar polygon, see *DCEL mesh structure*. `Vec2T` has "most" common arithmetic operators like the dot product, length, multiplication operators and so on.

`Vec3T` is a three-dimensional Cartesian vector type with precision `T`. It is templated as

```
namespace EBGeometry {
   template<class T>
   class Vec3T {
   public:
      T[3] x;
   };
}
```

Like `Vec2T`, `Vec3T` has numerous routines for performing most vector-related operations like addition, subtraction, dot products and so on.

## 3.3 Bounding volume hierarchy

The BVH functionality is encapsulated in the namespace `EBGeometry::BVH`. For the full API, see the doxygen API. There are two types of BVHs supported.

- **Direct BVHs** where the nodes are stored in build order and contain references to their children, and the leaf holds primitives.
- **Compact BVHs** where the nodes are stored in depth-first order and contain index offsets to children and primitives.

The direct BVH is encapsulated by a class

```
template <class T, class P, class BV, int K>
class NodeT;
```

The above template parameters are:

- `T` Floating-point precision.
- `P` Primitive type to be partitioned.
- `BV` Bounding volume type.
- `K` BVH degree. `K=2` will yield a binary tree, `K=3` yields a tertiary tree and so on.

`NodeT` describes regular and leaf nodes in the BVH, and has member functions for setting primitives, bounding volumes, and so on. Importantly, `NodeT` is the BVH builder node, i.e. it is the class through which we recursively build the BVH, see *Construction*. The compact BVH is discussed below in *Compact form*.

### 3.3.1 Template constraints

The template parameter `T` must be a valid C++ floating-point type (e.g. `float` or `double`) and the tree degree must be $K \geq 2$. The primitive type `P` must have the following:

- A signed distance function `T P::signedDistance(const Vec3& x) const`.

The bounding volume type `BV` must obey the following:

- A default constructor `BV()`.
- A constructor `BV(const std::vector<BV>&)` which constructs a bounding volume guaranteed to enclose other bounding volumes.
- A function `T getDistance(const Vec3T<T>& x) const` which returns a positive distance if point `x` is outside the bounding volume and zero if `x` is inside the bounding volume.

### 3.3.2 Signed distance

For getting the signed distance, `NodeT` has provide the following functions:

```
inline T
signedDistance(const Vec3T<T>& a_point) const noexcept override;

inline T
signedDistance(const Vec3T<T>& a_point, const Prune a_pruning) const noexcept;
```

The first version simply calls the other version with a stack-based pruning algorithm for the tree traversal.

### 3.3.3 Template constraints

- The primitive type P must have the following function:
  - T signedDistance(const Vec3T<T>& x), which returns the signed distance to the primitive.
- The bounding volume type BV must have the following functions:
  - T getDistance(const Vec3T<T>& x) which returns the distance from the point x to the bounding volume. Note that if x lies within the bounding volume, the function should return a value of zero.
  - A constructor BV(const std::vector<BV>& a_otherBVs) that permit creation of a bounding volume that encloses other bounding volumes of the same type.
- K should be greater or equal to 2.
- Currently, we do not support variable-sized trees (i.e., mixing branching ratios).

Note that the above constraints apply only to the BVH itself. Partitioning functions (which are, in principle, supplied by the user) may impose extra constraints.

**Important:** EBGeometry's BVH implementations fulfill their own template requirements on the primitive type P. This means that objects that are themselves described by BVHs (such as triangulations) can be embedded in another BVH, permitting BVH-of-BVH type of scenes.

### 3.3.4 Bounding volumes

EBGeometry supports the following bounding volumes, which are defined in EBGeometry_BoundingVolumes.hpp`:

- **BoundingSphere**, templated as EBGeometry::BoundingVolumes::BoundingSphereT<T> and describes a bounding sphere. Various constructors are available.
- **Axis-aligned bounding box**, or AABB for short. This is templated as EBGeometry::BoundingVolumes::AABBT<T>.

For full API details, see the doxygen API.

### 3.3.5 Construction

Constructing a BVH is done by

- Creating a root node and providing it with the geometric primitives.
- Partitioning the BVH by providing a partitioning function.

The first step is usually a matter of simply constructing the root node using the following constructor:

```
template <class T, class P, class BV, int K>
NodeT(const std::vector<std::shared_ptr<P> >& a_primitives).
```

That is, the constructor takes a list of primitives to be put in the node. For example:

```
using T    = float;
using Node = EBGeometry::BVH::NodeT<T>;

std::vector<std::shared_ptr<MyPrimitives> > primitives;
```

(continues on next page)

(continued from previous page)

```
auto root = std::make_shared<Node>(primitives);
```

The second step is to recursively build the BVH, which is done through the function

```
template <class T, class P, class BV, int K>
using StopFunctionT = std::function<bool(const NodeT<T, P, BV, K>& a_node)>;

template <class P, class BV>
using BVConstructorT = std::function<BV(const std::shared_ptr<const P>& a_primitive)>;

template <class P, int K>
using PartitionerT = std::function<std::array<PrimitiveListT<P>, K>(const PrimitiveListT
↪<P>& a_primitives)>;

template <class T, class P, class BV, int K>
NodeT<T, P, BV, K>::topDownSortAndPartitionPrimitives(const BVConstructorT<P, BV>,
                                                      const PartitionerT<P, K>,
                                                      const StopFunction<T, P, BV, K>);
```

Although seemingly complicated, the input arguments are simply polymorphic functions of the type indicated above, and have the following responsibilities:

- `StopFunctionT` simply takes a `NodeT` as input argument and determines if the node should be partitioned further. A basic implementation which terminates the recursion when the leaf node has reached the minimum number of primitives is

  ```
  EBGeometry::BVH::StopFunction<T, P, BV, K> stopFunc = [](const NodeT<T, P, BV, K>&␣
  ↪a_node) -> bool {
     return a_node.getNumPrimitives() < K;
  };
  ```

  This will terminate the partitioning when the node has less than `K` primitives (in which case it *can't* be partitioned further).

- `BVConstructorT` takes a single primitive (or strictly speaking a pointer to the primitive) and returns a bounding volume that encloses it. For example, if the primitives `P` are signed distance function spheres (see *Analytic functions*), the BV constructor can be implemented with AABB bounding volumes as;

  ```
  using T      = float;
  using Vec3   = EBGeometry::Vec3T<T>;
  using AABB   = EBGeometry::BoundingVolumes::AABBT<T>;
  using Sphere = EBGeometry::SphereSDF<T>;

  EBGeometry::BVH::BVConstructor<SDF, AABB> bvConstructor = [](const std::shared_ptr
  ↪<const SDF>& a_sdf){
     const Sphere& sph = static_cast<const Sphere&> (*a_sdf);

     const Vec3& sphereCenter = sph.getCenter();
     const T&    sphereRadius = sph.getRadius();

     const Vec3  lo = sphereCenter - r*Vec3::one();
     const Vec3  hi = sphereCenter + r*Vec3::one();
  ```

(continues on next page)

(continued from previous page)

```
  return AABB(lo, hi);
};
```

- PartitionerT is the partitioner function when splitting a leaf node into K new leaves. The function takes an list of primitives which it partitions into K new list of primitives, i.e. it encapsulates Eq. 2.2. As an example, we include a partitioner that is provided for integrating BVH and DCEL functionality.

```
template <class T, class BV, size_t K>
EBGeometry::BVH::PartitionerT<EBGeometry::DCEL::FaceT<T>, BV, K> chunkPartitioner =
[](const PrimitiveList<T>& a_primitives) -> std::array<PrimitiveList<T>, K> {
  Vec3T<T> lo = Vec3T<T>::max();
  Vec3T<T> hi = -Vec3T<T>::max();
  for (const auto& p : a_primitives) {
    lo = min(lo, p->getCentroid());
    hi = max(hi, p->getCentroid());
  }

  const size_t splitDir = (hi - lo).maxDir(true);

  // Sort the primitives along the above coordinate direction.
  PrimitiveList<T> sortedPrimitives(a_primitives);

  std::sort(
    sortedPrimitives.begin(), sortedPrimitives.end(),
    [splitDir](const std::shared_ptr<const FaceT<T>>& f1, const std::shared_ptr
↪<const FaceT<T>>& f2) -> bool {
    return f1->getCentroid(splitDir) < f2->getCentroid(splitDir);
    });

  return EBGeometry::DCEL::equalCounts<T, K>(sortedPrimitives);
};
```

In the above, we are taking a list of DCEL facets in the input argument (PrimitiveList<T> expands to std::vector<std::shared_ptr<const FaceT<T> >). We then compute the centroid locations of each facet and figure out along which coordinate axis we partition the objects (called splitDir above). The input primitives are then sorted based on the facet centroid locations in the splitDir direction, and they are partitioned into K almost-equal chunks. These partitions are returned and become primitives in the new leaf nodes.

There is also an example of the same type of partitioning for the BVH-accelerated union, see UnionBVH

In general, users are free to construct their BVHs in their own way if they choose. For the most part this will include the construction of their own bounding volumes and/or partitioners.

### 3.3.6 Compact form

In addition to the standard BVH node `NodeT<T, P, BV, K>`, EBGeometry provides a more compact formulation of the BVH hierarchy where the nodes are stored in depth-first order. The "linearized" BVH can be automatically constructed from the standard BVH but not vice versa.

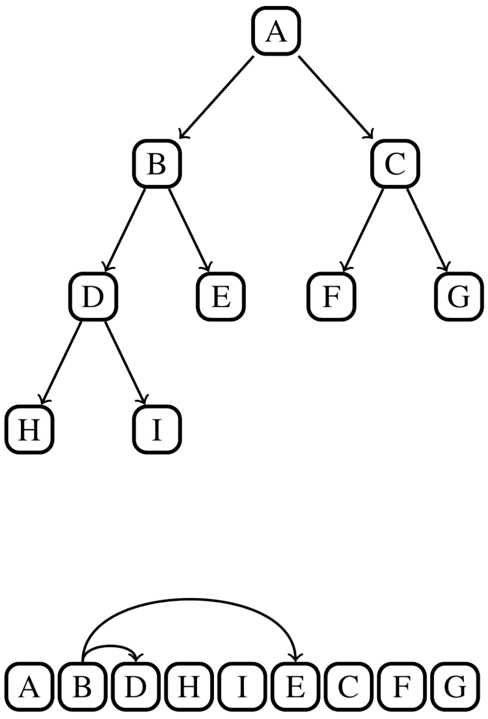

Fig. 3.1: Compact BVH representation. The original BVH is traversed from top-to-bottom along the branches and laid out in linear memory. Each interior node gets a reference (index offset) to their children nodes.

The rationale for reorganizing the BVH in compact form is it's tighter memory footprint and depth-first ordering which allows more efficient traversal downwards in the BVH tree. To encapsulate the compact BVH we provide two classes:

- `LinearNodeT` which encapsulates a node, but rather than storing the primitives and pointers to child nodes it stores offsets along the 1D arrays. Just like `NodeT` the class is templated:

  ```
  template <class T, class P, class BV, size_t K>
  class LinearNodeT
  ```

  `LinearNodeT` has a smaller memory footprint and should fit in one CPU word in floating-point precision and two CPU words in double point precision. The performance benefits of further memory alignment have not been investigated.

  Note that `LinearNodeT` only stores offsets to child nodes and primitives, which are assumed to be stored (somewhere) as

  ```
  std::vector<std::shared_ptr<LinearNodeT<T, P, BV, K> > > linearNodes;
  std::vector<std::shared_ptr<const P> > primitives;
  ```

  Thus, for a given node we can check if it is a leaf node (`m_numPrimitives > 0`) and if it is we can get the children through the `m_childOffsets` array. Primitives can likewise be obtained; they are stored in the primitives array from index `m_primitivesOffset` to `m_primitivesOffset + m_numPrimities - 1`.

- LinearBVH which stores the compact BVH *and* primitives as class members. That is, LinearBVH contains the nodes and primitives as class members.

  ```
  template <class T, class P, class BV, size_t K>
  class LinearBVH : public SignedDistanceFunction<T>
  {
  public:

  protected:

    std::vector<std::shared_ptr<const LinearNodeT<T, P, BV, K>>> m_linearNodes;
    std::vector<std::shared_ptr<const P>> m_primitives;
  };
  ```

  The root node is, of course, found at the front of the m_linearNodes vector. Note that the list of primitives m_primitives is stored in the order in which the leaf nodes appear in m_linearNodes.

Constructing the compact BVH is simply a matter of letting NodeT aggregate the nodes and primitives into arrays, and return a LinearBVH. This is done by calling the NodeT member function flattenTree():

```
template <class T, class P, class BV, size_t K>
class NodeT : public SignedDistanceFunction<T>
{
public:

  inline std::shared_ptr<LinearBVH<T, P, BV, K>>
  flattenTree() const noexcept;
};
```

which returns a pointer to a LinearBVH. For example:

```
// Assume that we have built the conventional BVH already
std::shared_ptr<EBGeometry::BVH::NodeT<T, P, BV, K> > builderBVH;

// Flatten the tree.
std::shared_ptr<LinearBVH> compactBVH = builderBVH.flattenTree();

// Release the original BVH.
builderBVH = nullptr;
```

**Warning:** When calling flattenTree, the original BVH tree is *not* destroyed. To release the memory, deallocate the original BVH tree. E.g., the set pointer to the root node to nullptr if using a smart pointer.

Note that the primitives live in LinearBVH and not LinearNodeT, and the signed distance function is therefore implemented in the LinearBVH member function:

```
template <class T, class P, class BV, size_t K>
class LinearBVH : public SignedDistanceFunction<T>
{
public:

  inline T
```

(continues on next page)

(continued from previous page)

```
  signedDistance(const Vec3& a_point) const noexcept override;
};
```

## 3.3.7 Signed distance

The signed distance can be obtained from both the full BVH storage and the compact BVH storage. Replicating the code above, we can do:

```
// Assume that we have built the conventional BVH already
std::shared_ptr<EBGeometry::BVH::NodeT<T, P, BV, K> > fullBVH;

// Flatten the tree.
std::shared_ptr<EBGeometry::BVH::LinearBVH<T, P, BV, K> > compactBVH = fullBVH.
↪flattenTree();

// These give the same result.
const T s1 = fullBVH   ->signedDistance(Vec3T<T>::zero());
const T s2 = compactBVH->signedDistance(Vec3T<T>::zero());
```

We point out that the compact BVH only supports:

- Recursive, unordered traversal through the tree.
- Recursive, ordered traversal through the tree.
- Stack-based ordered traversal through the tree.

Out of these, the ordered traversals (discussed in *Bounding volume hierarchies*) are faster.

The compact BVH only supports stack-based ordered traversal (which tends to be faster).

## 3.4 DCEL

The DCEL functionality exists under the namespace `EBGeometry::DCEL` and contains the following functionality:

- **Fundamental data types** like vertices, half-edges, polygons, and entire surface grids.
- **Signed distance functionality** for the above types.
- **File parsers for reading files** into DCEL structures.
- **BVH functionality** for putting DCEL grids into bounding volume hierarchies.

**Note:** The DCEL functionality is *not* restricted to triangles, but supports N-sided polygons.

### 3.4.1 Classes

The main DCEL functionality (vertices, edges, faces) is provided by the following classes:

- **Vertices** are implemented as a template `EBGeometry::DCEL::EdgeT`

  ```
  template <class T>
  class VertexT
  ```

  The DCEL vertex class stores the vertex position, normal vector, and the outgoing half-edge from the vertex. Note that the class has member functions for computing the vertex pseudonormal, see *Normal vectors*.

  The full API is given in the doxygen documentation here.

- **Edges** are implemented as a template `EBGeometry::DCEL::EdgeT`

  ```
  template <class T>
  class EdgeT
  ```

  The half-edges store a reference to their face, as well as pointers to the previous edge, next edge, pair edge, and starting vertex. For performance reasons, the edge also stores the length and inverse length of the edge.

  The full API is given in the doxygen documentation here.

- **Faces** are implemented as a template `EBGeometry::DCEL::FaceT`

  ```
  template <class T>
  class FaceT
  ```

  For performance reasons, a polygon face stores all it's half-edges (to avoid iteration when computing the signed distance). It also stores:

  - The normal vector.
  - A 2D embedding of the polygon face.
  - Centroid position.

  The normal vector and 2D embedding of the facet exist because the signed distance computation requires them. The centroid position exists only because BVH partitioners will use it for partitioning the surface mesh.

  The full API is given in the doxygen documentation here.

- **Mesh** is implemented as a template `EBGeometry::DCEL::MeshT`

  ```
  template <class T>
  class MeshT
  ```

  The Mesh stores all the vertices, half-edges, and faces. For example, to obtain all the facets one can call `EBGeometry::DCEL::MeshT<T>::getFaces()` which will return all the DCEL faces of the surface mesh. Typically, the mesh is not created by the user but automatically created when reading the mesh from an input file.

  The full API is given in the doxygen documentation here.

All of the above DCEL classes have member functions of the type:

```
T signedDistance(const Vec3T<T>& a_point) const noexcept;
T unsignedDistance2(const Vec3T<T>& a_point) const noexcept;
```

Thus, they fulfill the template requirements of the primitive type for the BVH implementation, see *Template constraints*. See *BVH integration* for details regarding DCEL integration with BVHs.

## 3.4.2 BVH integration

DCEL functionality can easily be embedded in BVHs. In this case it is the facets that are embedded in the BVHs, and we require that we can create bounding volumes that contain all the vertices in a facet. Moreover, partitioning functions that partition a set of polygon faces into K new sets of faces are also required.

EBGeometry provides some simplistic functions that are needed (see *Construction*) when building BVHs for DCEL geometries .

**Note:** The functions are defined in `Source/EBGeometry_DCEL_BVH.hpp`.

For the bounding volume constructor, we provide a function

```
template <class T, class BV>
EBGeometry::BVH::BVConstructorT<EBGeometry::DCEL::FaceT<T>, BV> defaultBVConstructor =
[](const std::shared_ptr<const EBGeometry::DCEL::FaceT<T>>& a_primitive) -> BV {
  return BV(a_primitive->getAllVertexCoordinates());
};
```

Note the extra template constraint on the bounding volume type BV, which must be able to construct a bounding volume from a finite point set (in this case the vertex coordinates).

For the stop function we provide a simple function

```
template <class T, class BV, size_t K>
EBGeometry::BVH::StopFunctionT<T, EBGeometry::DCEL::FaceT<T>, BV, K> defaultStopFunction␣
↪=
[](const BVH::NodeT<T, EBGeometry::DCEL::FaceT<T>, BV, K>& a_node) -> bool {
  return (a_node.getPrimitives()).size() < K;
};
```

Note that this simply terminates the leaf partitioning if there are not enough primitives (polygon faces) available, or there are fewer than a pre-defined number of primitives.

For the partitioning function we include a simple function that partitions the primitives along the longest axis:

```
template <class T, class BV, size_t K>
EBGeometry::BVH::PartitionerT<EBGeometry::DCEL::FaceT<T>, BV, K> chunkPartitioner =
[](const PrimitiveList<T>& a_primitives) -> std::array<PrimitiveList<T>, K> {
  Vec3T<T> lo = Vec3T<T>::max();
  Vec3T<T> hi = -Vec3T<T>::max();

  for (const auto& p : a_primitives) {
    lo = min(lo, p->getCentroid());
    hi = max(hi, p->getCentroid());
  }

  const size_t splitDir = (hi - lo).maxDir(true);

  // Sort the primitives along the above coordinate direction.
```

(continues on next page)

(continued from previous page)

```
  PrimitiveList<T> sortedPrimitives(a_primitives);

  std::sort(
    sortedPrimitives.begin(), sortedPrimitives.end(),
    [splitDir](const std::shared_ptr<const FaceT<T>>& f1, const std::shared_ptr<const␣
↪FaceT<T>>& f2) -> bool {
      return f1->getCentroid(splitDir) < f2->getCentroid(splitDir);
    });

  return EBGeometry::DCEL::equalCounts<T, K>(sortedPrimitives);
};
```

For a list of all DCEL partitioner, see `Source/EBGeometry_DCEL_BVH.hpp`.

### 3.4.3 Code example

Constructing a compact BVH representation of polygon mesh is therefore done as follows:

```
using T    = float;
using BV   = EBGeometry::BoundingVolumes::AABBT<T>;
using Vec3 = EBGeometry::Vec3T<T>;
using Face = EBGeometry::DCEL::FaceT<T>;

constexpr int K = 4;

// Read the mesh from file and put it in a DCEL format.
std::shared_ptr<EBGeometry::DCEL::Mesh<T> > mesh = EBGeometry::Parser::read("MyFile.stl
↪");

// Make a BVH node and build the BVH.
auto root = std::make_shared<EBGeometry::BVH::NodeT<T, Face, BV, K> >(mesh->getFaces());

// Build the BVH hierarchy
root->topDownSortAndPartitionPrimitives(EBGeometry::DCEL::defaultBVConstructor<T, BV>,
                                        EBGeometry::DCEL::spatialSplitPartitioner<T, K>,
                                        EBGeometry::DCEL::defaultStopFunction<T, BV, K>);

// Flatten the tree onto a tighter representation. Then delete the old tree.
auto compactBVH = root->flattenTree();

root = nullptr;
```

## 3.5 Signed distance function

In EBGeometry we have encapsulated the concept of a signed distance function in an abstract class

```
/* EBGeometry
 * Copyright © 2022 Robert Marskar
 * Please refer to Copyright.txt and LICENSE in the EBGeometry root directory.
 */

/*!
  @file   EBGeometry_SignedDistanceFunction.hpp
  @brief  Abstract base class for representing a signed distance function.
  @author Robert Marskar
*/

#ifndef EBGeometry_SignedDistanceFunction
#define EBGeometry_SignedDistanceFunction

#include <memory>
#include <deque>

// Our includes
#include "EBGeometry_ImplicitFunction.hpp"
#include "EBGeometry_NamespaceHeader.hpp"

/*!
  @brief Abstract representation of a signed distance function.
  @details Users can put whatever they like in here, e.g. analytic functions,
  DCEL meshes, or DCEL meshes stored in full or compact BVH trees. The
  signedDistance function must be implemented by the user. When computing it,
  the user can apply transformation operators (rotations, scaling, translations)
  by calling transformPoint on the input coordinate.
*/
template <class T>
class SignedDistanceFunction : public ImplicitFunction<T>
{
public:
  /*!
    @brief Disallowed, use the full constructor
  */
  SignedDistanceFunction() = default;

  /*!
    @brief Destructor (does nothing)
  */
  virtual ~SignedDistanceFunction() = default;

  /*!
    @brief Implementation of ImplicitFunction::value
    @param[in] a_point 3D point.
  */
  virtual T
  value(const Vec3T<T>& a_point) const noexcept override final;
```

(continues on next page)

(continued from previous page)

```
  /*!
    @brief Signed distance function.
    @param[in] a_point 3D point.
  */
  virtual T
  signedDistance(const Vec3T<T>& a_point) const noexcept = 0;

  /*!
    @brief Signed distance normal vector.
    @details Computed using finite differences with step a_delta
    @param[in] a_point 3D point
    @param[in] a_delta Finite difference step
  */
  inline virtual Vec3T<T>
  normal(const Vec3T<T>& a_point, const T& a_delta) const noexcept;
};

#include "EBGeometry_NamespaceFooter.hpp"

#include "EBGeometry_SignedDistanceFunctionImplem.hpp"

#endif
```

We point out that the BVH and DCEL classes are fundamentally also signed distance functions, and they also inherit from `SignedDistanceFunction`. The `SignedDistanceFunction` class also exists so that we have a common entry point for performing distance field manipulations like rotations and translations.

When implementing the `signedDistance` function, one can transform the input point by first calling `transformPoint`. The functions `translate` and `rotate` will translate or rotate the object. It is also possible to *scale* an object, but this is not simply a coordinate transform so it is implemented as a separate signed distance function. For example, in order to rotate a DCEL mesh (without using the BVH accelerator) we can implement the following signed distance function:

```
template <class T>
class MySignedDistanceFunction : public SignedDistanceFunction<T> {
public:
   T signedDistance(const Vec3T<T>& a_point) const noexcept override {
      return m_mesh->signedDistance(this->transformPoint(a_point));
   }

protected:
   // DCEL mesh object, must be constructed externally and
   // supplied to MyDistanceFunction (e.g. through the constructor).
   std::shared_ptr<EBGeometry::Dcel::MeshT<T> > m_mesh;
};
```

Alternatively, using a BVH structure:

```
template <class T, class P, class BV, int K>
class MySignedDistanceFunction : public SignedDistanceFunction<T> {
public:
   T signedDistance(const Vec3T<T>& a_point) const noexcept override {
```

(continues on next page)

(continued from previous page)

```
      return m_bvh->signedDistance(this->transformPoint(a_point));
   }

protected:
   // BVH object, must be constructed externally
   // and supplied to MyDistanceFunction (e.g. through the constructor).
   std::shared_ptr<EBGeometry::BVH::LinearBVH<T, P, BV, K> > m_bvh;
};
```

### 3.5.1 Normal vector

The normal vector of `EBGeometry::SignedDistanceFunction<T>` is computed using centered finite differences:

$$n_i\left(\mathbf{x}\right) = \frac{1}{2\Delta}\left[S\left(\mathbf{x}+\Delta\hat{\mathbf{i}}\right) - S\left(\mathbf{x}-\Delta\hat{\mathbf{i}}\right)\right],$$

where $i$ is a coordinate direction and $\Delta > 0$. This is done for each component, and the normalized vector is then returned.

### 3.5.2 Transformations

The following transformations are possible:

- Translation, which defines the operation $\mathbf{x}^\prime = \mathbf{x} - \mathbf{t}$ where $\mathbf{t}$ is a translation vector.
- Rotation, which defines the operation $\mathbf{x}^\prime = R\left(\mathbf{x}, \theta, a\right)$ where $\mathbf{x}$ is rotated an angle $\theta$ around the coordinate axis $a$.

Transformations are applied sequentially. The APIs are as follows:

```
void translate(const Vec3T<T>& a_translation) noexcept;  // a_translation are Cartesian␣
↪translations vector
void rotate(const T a_angle, const int a_axis) noexcept; // a_angle in degrees, and a_
↪axis being the Cartesian axis
```

E.g. the following code will first translate, then 90 degrees about the $x$-axis.

```
MySignedDistanceFunction<float> sdf;

sdf.translate({1,0,0});
sdf.rotate(90, 0);
```

Note that if the transformations are to be applied, the implementation of `signedDistance(...)` must transform the input point, as shown in the examples above.

### 3.5.3 Rounding

Distance functions can be rounded by displacing the SDF by a specified distance. For example, given a distance functions $S(\mathbf{x})$, the rounded distance functions is

$$S_r(\mathbf{x}) = S(\mathbf{x}) - r,$$

where $r$ is some rounding radius. Note that the rounding does not preserve the volume of the original SDF, so subsequent *scaling* of the object is usually necessary.

The rounded SDF is implemented in `Source/EBGeometry_AnalyticDistanceFunctions.hpp`

```
template <class T>
class RoundedSDF : public SignedDistanceFunction<T>
{
public:
  /*!
    @brief Disallowed weak construction
  */
  RoundedSDF() = delete;

  /*!
    @brief Rounded SDF. Rounds the input SDF
    @param[in] a_sdf  Input signed distance function.
    @param[in] a_curv Rounding radius.
  */
  RoundedSDF(const std::shared_ptr<SignedDistanceFunction<T>> a_sdf, const T a_curv)
  {
    m_sdf  = a_sdf;
    m_curv = a_curv;
  }

  /*!
    @brief Destructor
  */
  virtual ~RoundedSDF()
  {}

  /*!
    @brief Signed distance field.
  */
  virtual T
  signedDistance(const Vec3T<T>& a_point) const noexcept override
  {
    return m_sdf->signedDistance(a_point) - m_curv;
  }

protected:
  /*!
    @brief Original signed distance function
  */
  std::shared_ptr<const SignedDistanceFunction<T>> m_sdf;

  /*!
```

(continues on next page)

(continued from previous page)

```
    @brief Rounding radius
  */
  T m_curv;
```

To use it, simply pass an SDF into the constructor and use the new distance function.

## 3.5.4 Scaling

Scaling of distance functions are possible through the transformation

$$S_c\left(\mathbf{x}\right) = cS\left(\frac{\mathbf{x}}{c}\right),$$

where $c$ is a scaling factor. We point out that anisotropic stretching does not preserve the distance field.

The rounded SDF is implemented in `Source/EBGeometry_AnalyticDistanceFunctions.hpp`

```
  @brief Scaled signed distance function.
*/
template <class T>
class ScaledSDF : public SignedDistanceFunction<T>
{
public:
  /*!
    @brief Disallowed weak construction
  */
  ScaledSDF() = delete;

  /*!
    @brief Scaled SDF.
    @param[in] a_sdf   Input signed distance function.
    @param[in] a_scale Scaling factor.
  */
  ScaledSDF(const std::shared_ptr<SignedDistanceFunction<T>> a_sdf, const T a_scale)
  {
    m_sdf   = a_sdf;
    m_scale = a_scale;
  }

  /*!
    @brief Destructor
  */
  virtual ~ScaledSDF()
  {}

  /*!
    @brief Signed distance field.
    @param[in] a_point Input point.
  */
  virtual T
  signedDistance(const Vec3T<T>& a_point) const noexcept override
  {
    return (m_sdf->signedDistance(a_point / m_scale)) * m_scale;
```

(continues on next page)

(continued from previous page)

```
  }

protected:
  /*!
    @brief Original signed distance function
  */
  std::shared_ptr<const SignedDistanceFunction<T>> m_sdf;

  /*!
    @brief Scaling factor.
```

### 3.5.5 Analytic functions

Above, we have shown how users can supply a DCEL or BVH structure to implement `SignedDistanceFunction`. In addition, the file `Source/EBGeometry_AnalyticSignedDistanceFunctions.hpp` defines various other analytic shapes such as:

- **Sphere**

```
template <class T>
class SphereSDF : public SignedDistanceFunction<T>
```

- **Box**

```
template <class T>
class BoxSDF : public SignedDistanceFunction<T>
```

- **Torus**

```
template <class T>
class TorusSDF : public SignedDistanceFunction<T>
```

- **Capped cylinder**

```
template <class T>
class CylinderSDF : public SignedDistanceFunction<T>
```

- **Infinite cylinder**

```
template <class T>
class InfiniteCylinderSDF : public SignedDistanceFunction<T>
```

- **Capsule/rounded cylinder**

```
template <class T>
class CapsuleSDF : public SignedDistanceFunction<T>
```

- **Infinite cone**

```
template <class T>
class InfiniteConeSDF : public SignedDistanceFunction<T>
```

- **Cone**

```
template <class T>
class ConeSDF : public SignedDistanceFunction<T>
```

## 3.6 Unions

As discussed in *Signed distance fields*, a union of signed distance fields can be created provided that the objects do not touch or overlap. EBGeometry provides two implementations:

- **Standard union** where one looks through every primitive in the union.
- **BVH-enabled union** where bounding volume hierarchies are used to find the closest object.

### 3.6.1 Standard union

The standard union is template as

```
class Union : public ImplicitFunction<T>
{
public:
  // Regular CSG union only requires implicit functions.
  static_assert(std::is_base_of<EBGeometry::ImplicitFunction<T>, P>::value);

  /*!
    @brief Disallowed, use the full constructor
  */
  Union() = delete;

  /*!
    @brief Full constructor. Computes the CSG union
    @param[in] a_primitives List of primitives
    @param[in] a_flipSign   Hook for turning inside to outside
  */
  Union(const std::vector<std::shared_ptr<P>>& a_primitives, const bool a_flipSign);

  /*!
    @brief Destructor (does nothing)
  */
  virtual ~Union() = default;

  /*!
    @brief Value function
    @param[in] a_point 3D point.
  */
  T
  value(const Vec3T<T>& a_point) const noexcept override;

protected:
  /*!
    @brief List of primitives
  */
  std::vector<std::shared_ptr<const P>> m_primitives;
```

(continues on next page)

(continued from previous page)

```
  /*!
    @brief Hook for turning inside to outside
  */
  bool m_flipSign;
};

#include "EBGeometry_NamespaceFooter.hpp"
```

Note that EBGeometry::Union inherits from EBGeometry::SignedDistanceFunction and thus provides a signedDistance(...) function. The implementation of the standard union is

```
Union<T, P>::Union(const std::vector<std::shared_ptr<P>>& a_primitives, const bool a_
↪flipSign)
{
  for (const auto& prim : a_primitives) {
    m_primitives.emplace_back(prim);
  }

  m_flipSign = a_flipSign;
}

template <class T, class P>
T
Union<T, P>::value(const Vec3T<T>& a_point) const noexcept
{
  T ret = std::numeric_limits<T>::infinity();

  for (const auto& prim : m_primitives) {
    ret = std::min(ret, prim->value(a_point));
  }

  T sign = (m_flipSign) ? -1.0 : 1.0;

  return sign * ret;
}

#include "EBGeometry_NamespaceFooter.hpp"
```

That is, it iterates through *all* the objects in order to find the signed distance.

### 3.6.2 BVH-enabled union

The BVH-enabled union is implemented by EBGeometry::UnionBVH as follows:

```
*/
template <class T, class P, class BV, size_t K>
class UnionBVH : public ImplicitFunction<T>
{
public:
  static_assert(std::is_base_of<EBGeometry::SignedDistanceFunction<T>, P>::value);
```

(continues on next page)

(continued from previous page)

```
  using BVConstructor = EBGeometry::BVH::BVConstructorT<P, BV>;

  /*!
    @brief Disallowed, use the full constructor
  */
  UnionBVH() = delete;

  /*!
    @brief Full constructor.
    @param[in] a_distanceFunctions Signed distance functions.
    @param[in] a_flipSign          Hook for turning inside to outside
    @param[in] a_bvConstructor     Bounding volume constructor.
  */
  UnionBVH(const std::vector<std::shared_ptr<P>>& a_distanceFunctions,
           const bool                             a_flipSign,
           const BVConstructor&                   a_bvConstructor);

  /*!
    @brief Destructor (does nothing)
  */
  virtual ~UnionBVH() = default;

  /*!
    @brief Value function
    @param[in] a_point 3D point.
  */
  T
  value(const Vec3T<T>& a_point) const noexcept override;

  /*!
    @brief Get the bounding volume
  */
  const BV&
  getBoundingVolume() const noexcept;

protected:
  /*!
    @brief Root node for linearized BVH tree
  */
  std::shared_ptr<EBGeometry::BVH::LinearBVH<T, P, BV, K>> m_rootNode;

  /*!
    @brief Hook for turning inside to outside
  */
  bool m_flipSign;

  /*!
    @brief Build BVH tree for the input objects. User must supply a partitioner
    and a BV constructor for the SDF objects.
    @param[in] a_bvConstructor Constructor for building a bounding volume that
    encloses an object.
```

(continues on next page)

(continued from previous page)

```
  */
  inline void
  buildTree(const std::vector<std::shared_ptr<P>>& a_distanceFunctions, const␣
↪BVConstructor& a_bvConstructor) noexcept;
};

#include "EBGeometry_NamespaceFooter.hpp"

#include "EBGeometry_UnionBVHImplem.hpp"

#endif
```

As always, the template parameter T indicates the precision, BV the bounding volume type and K the tree degree. UnionBVH takes a bounding volume constructor in addition to the list of primitives, see *Construction*.

Internally, UnionBVH defines its own partitioning function which is identical to the implementation for DCEL meshes (see *BVH integration*), with the exception that the partitioning is based on the centroids of the bounding volumes rather than the centroid of the primitives. After partitioning the primitives, the original BVH tree is flattened onto the compact representation.

The implementation of the signed distance function for the BVH-enabled union is

```
  // distance but there is still a CSG union, and the BVH is still a useful thing.
  // So, we can't use LinearNode::signedDistanceFunction because it returns the
  // closest object and not the object with the smallest value function.
  //
  // Fortunately, when I wrote the LinearNode accelerator I wrote it such that we can␣
↪determine
  // how to update the "shortest" distance using externally supplied criteria.
  // So, we just update this as f = min(f1,f2,f3) etc, and prune nodes accordingly.
  // The criteria for that are below...
```

That is, it relies on pruning from the BVH functionality for finding the signed distance to the closest object.

## 3.7 Reading data

Routines for parsing surface files from grids into EBGeometry's DCEL grids are given in the namespace EBGeometry::Parser. The source code is implemented in Source/EBGeometry_Parser.hpp.

**Warning:** EBGeometry is currently limited to reading STL files and reconstructing DCEL grids from those. However, it is a simple matter to also reconstructor DCEL grids from triangle soups read using third-party codes (see *Using third-party sources*).

### 3.7.1 Reading STL files

EBGeometry supports a native parser for binary and ASCII STL files. To read an STL file, one will use one of the following:

```
template <typename T>
inline static std::shared_ptr<Mesh>
EBGeometry::Parser::STL<T>::readSingle(const std::string a_filename) noexcept;

template <typename T>
inline static std::Vector<std::pair<std::shared_ptr<Mesh>, std::string>>
EBGeometry::Parser::STL<T>::readMulti(const std::string a_filename) noexcept;
```

The difference between these two is that `readSingle` only reads a single STL *solid* while `readMulti` will read all STL solids defined in the input file.

Alternatively, one can use `Parser::read`:

```
template <typename T>
inline static std::shared_ptr<EBGeometry::DCEL::MeshT<T>>
EBGeometry::Parser::read<T>(const std::string a_filename) noexcept;

template <typename T>
inline static std::vector<std::shared_ptr<EBGeometry::DCEL::MeshT<T>>>
EBGeometry::Parser::read<T>(const std::vector<std::string> a_files) noexcept;
```

where `a_filename` and `a_files` must be STL files.

### 3.7.2 From soups to DCEL

EBGeometry supports the creation of DCEL grids from polygon soups. A triangle soup is represented as

```
std::vector<Vec3T<T>> vertices;
std::vector<std::vector<size_t>> faces;
```

Here, `vertices` contains the $x, y, z$ coordinates of each vertex, while each entry `faces` contains a list of vertices for the face.

To turn this into a DCEL mesh, one should call

```
template <typename T>
inline static void
EBGeometry::Parser::compress(std::vector<EBGeometry::Vec3T<T>>& a_vertices,
                             std::vector<std::vector<size_t>>&  a_facets) noexcept;

template <typename T>
inline static void
EBGeometry::Parser::soupToDCEL(EBGeometry::DCEL::MeshT<T>&              a_mesh,
                               const std::vector<EBGeometry::Vec3T<T>>& a_vertices,
                               const std::vector<std::vector<size_t>>&  a_facets)␣
↪noexcept;
```

The `compress` function will discard duplicate vertices from the soup, while the `soupToDCEL` will simply turn the remaining polygon soup into a DCEL mesh.

**Tip:** `soupToDCEL` will issue plenty of warnings if the polygon soup is not watertight and orientable.

### 3.7.3 Using third-party sources

By design, EBGeometry does not include much functionality for parsing files into polygon soups. There are many open source third-party codes for achieving this (and we have tested several of them):

1. happly or miniply for Stanford PLY files.
2. stl_reader for STL files.
3. tinyobjloader for OBJ files.

In almost every case, the above codes can be read into polygon soups, and one can then turn the soup into a DCEL mesh as described in *From soups to DCEL*.

CHAPTER

FOUR

# GUIDED EXAMPLES

## 4.1 Overview

Below, we consider a few examples that show how to use EBGeometry. All the examples are located in the examples folder. For instructions on how to compile and run the examples, refer to the README file in the example folder.

## 4.2 Basic example

This example is given in `Examples/EBGeometry_DCEL/main.cpp` and shows the following steps:

1. How to read an STL file into a DCEL mesh.
2. How to partition and flatten a BVH tree.
3. How to call the signed distance function and provide a performance comparison between SDF representations.

We will focus on the following parts of the code:

```
using T    = float;
  }

  // we convert it to a full BVH tree representation. Then we flatten that tree.
  const auto dcelSDF = EBGeometry::Parser::readIntoDCEL<T>(file);
  const auto bvhSDF  = EBGeometry::DCEL::buildFullBVH<T, BV, K>(dcelSDF);
  const auto linSDF  = bvhSDF->flattenTree();
  // Sample some random points around the object.
  constexpr size_t Nsamp = 100;

  const Vec3 lo    = bvhSDF->getBoundingVolume().getLowCorner();
  const Vec3 hi    = bvhSDF->getBoundingVolume().getHighCorner();
  std::mt19937_64 rng(static_cast<size_t>(std::chrono::system_clock::now().time_since_
↪epoch().count()));

  if (std::abs(bvhSum - dcelSum) > std::numeric_limits<T>::epsilon()) {
```

### 4.2.1 Reading the surface mesh

The first block of code parses an STL file (here called *file*) and returns a DCEL mesh description of the STL file. We point out that the parser will issue errors if the STL file is not watertight and orientable.

### 4.2.2 Constructing the BVH

The second block of code, which begins with

```
  // Sample some random points around the object.
```

creates a BVH root node and provides it with all the DCEL faces. The next block of code

```
  constexpr size_t Nsamp = 100;

  const Vec3 lo    = bvhSDF->getBoundingVolume().getLowCorner();
  const Vec3 hi    = bvhSDF->getBoundingVolume().getHighCorner();
```

partitions the BVH using pre-defined partitioning functions (see *BVH integration* for details).

Finally, the BVH tree is flattened by

```

```

### 4.2.3 Summary

Note that all the objects `directSDF`, `bvhSDF`, and `linSDF` represent precisely the same distance field. The objects differ in how they compute it:

- `directSDF` will iterate through all faces in the mesh.
- `bvhSDF` uses *full BVH tree representation*, pruning branches during the tree traversal.
- `linSDF` uses *compact BVH tree representation*, also pruning branches during the tree traversal.

All the above functions give the same result, but with different performance metrics.

## 4.3 Unions

This example is given in `Examples/EBGeometry_Union/main.cpp` and shows the following steps:

1. The creation of scene composed of an array of spheres.
2. Instantiation of a standard union for the signed distance (see *Unions*).
3. Instantiation of a BVH-enabled union for the signed distance (see *Unions*).

We focus on the following parts of the code:

```
int
  constexpr int K = 4;

  // Make a sphere array consisting of about M^3 spheres. We assume
  // that the domain is x,y,z \in [-1,1] and set the number of spheres
```

(continues on next page)

(continued from previous page)

```
// and their radii.
std::vector<std::shared_ptr<Sphere>> spheres;

constexpr T   radius = 0.02;
constexpr int M      = 40;
constexpr int Nsamp  = 1000;
constexpr T   delta  = (2.0 - 2 * M * radius) / (M + 1);

if (delta < 0.0) {
  std::cerr << "Error: 'delta < 0.0'" << std::endl;

  return 1;
}
else {
  std::cout << "delta = " << delta << std::endl;
}

for (int i = 1; i <= M; i++) {
  for (int j = 1; j <= M; j++) {
    for (int k = 1; k <= M; k++) {

      const T x = -1. + i * (delta + 2 * radius) - radius;
      const T y = -1. + j * (delta + 2 * radius) - radius;
      const T z = -1. + k * (delta + 2 * radius) - radius;

      Vec3 center(x, y, z);

      spheres.emplace_back(std::make_shared<Sphere>(center, radius, false));
    }
  }
}

// Make a standard union of these spheres. This is the union object which
// iterates through each and every object in the scene.
EBGeometry::Union<T, Sphere> slowUnion(spheres, false);
// Make a fast union. To do this we must have the SDF objects (our vector of
// spheres) as well as a way for enclosing these objects. We need to define
```

### 4.3.1 Creating the spheres

In the first block of code we are defining one million spheres that lie on a three-dimensional lattice, where each sphere has a radius of one:

```
constexpr int K = 4;

// Make a sphere array consisting of about M^3 spheres. We assume
// that the domain is x,y,z \in [-1,1] and set the number of spheres
// and their radii.
std::vector<std::shared_ptr<Sphere>> spheres;

constexpr T   radius = 0.02;
```

(continues on next page)

(continued from previous page)

```
  constexpr int M      = 40;
  constexpr int Nsamp  = 1000;
  constexpr T   delta  = (2.0 - 2 * M * radius) / (M + 1);

  if (delta < 0.0) {
    std::cerr << "Error: 'delta < 0.0'" << std::endl;

    return 1;
  }
  else {
    std::cout << "delta = " << delta << std::endl;
  }
```

## 4.3.2 Creating standard union

In the second block of code we are simply creating a standard signed distance function union:

```
      for (int k = 1; k <= M; k++) {
```

For implementation details regarding the standard union, see *Unions*.

## 4.3.3 Creating BVH-enabled union

In the third block of code we create a BVH-enabled union. To do so, we must first provide a function which can create bounding volumes around each object:

```
        const T z = -1. + k * (delta + 2 * radius) - radius;

        Vec3 center(x, y, z);

        spheres.emplace_back(std::make_shared<Sphere>(center, radius, false));
      }
    }
  }

  // Make a standard union of these spheres. This is the union object which
  // iterates through each and every object in the scene.
```

Here, we use axis-aligned boxes but we could also have used other types of bounding volumes.

## 4.3.4 Typical output

The above example shows two methods of creating unions. When running the example the typical output is something like:

```
Partitioning spheres
Computing distance with slow union
Computing distance with fast union
Distance and time using standard union = -1, which took 26.7353 ms
```

(continues on next page)

(continued from previous page)

```
Distance and time using optimize union = -1, which took 0.003527 ms
Speedup = 7580.19
```

where we note that the optimized union was about 7500 times faster than the “standard” union.

## 4.4 Integration with AMReX

**Warning:** This example requires you to install AMReX

This example is given in `Examples/AMReX_DCEL/main.cpp` and shows how to expose EBGeometry’s DCEL and BVH functionality to AMReX.

We will focus on the following parts of the code:

```
/* EBGeometry
 * Copyright © 2022 Robert Marskar
 * Please refer to Copyright.txt and LICENSE in the EBGeometry root directory.
 */

// AMReX includes
#include <AMReX.H>
#include <AMReX_EB2.H>
#include <AMReX_EB2_IF.H>
#include <AMReX_ParmParse.H>
#include <AMReX_PlotFileUtil.H>

// Our include
#include "../../EBGeometry.hpp"

using namespace amrex;

/*!
  @brief This is an AMReX-capable version of the EBGeometry BVH accelerator. It
  is templated as T, BV, K which indicate the EBGeometry precision, bounding
  volume, and tree degree.
*/
template <class T, class BV, size_t K>
class AMReXSDF
{
public:
  /*!
    @brief Full constructor.
    @param[in] a_filename File name. Must be an STL file.
  */
  AMReXSDF(const std::string a_filename)
  {
    m_rootNode = EBGeometry::Parser::readIntoLinearBVH<T, BV, K>(a_filename);
  }
```

(continues on next page)

(continued from previous page)

```
  /*!
    @brief Copy constructor.
    @param[in] a_other Other SDF.
  */
  AMReXSDF(const AMReXSDF& a_other)
  {
    this->m_rootNode = a_other.m_rootNode;
  }

  /*!
    @brief AMReX's implicit function definition.
  */
  Real operator()(AMREX_D_DECL(Real x, Real y, Real z)) const noexcept
  {
    using Vec3 = EBGeometry::Vec3T<T>;

    return m_rootNode->signedDistance(m_rootNode->transformPoint(Vec3(x, y, z)));
  };

  /*!
    @brief Also an AMReX implicit function implementation
  */
  inline Real
  operator()(const RealArray& p) const noexcept
  {
    return this->operator()(AMREX_D_DECL(p[0], p[1], p[2]));
  }

protected:
  /*!
    @brief Root node of the linearized BVH hierarchy.
  */
  std::shared_ptr<EBGeometry::BVH::LinearBVH<T, EBGeometry::DCEL::FaceT<T>, BV, K>> m_
↪rootNode;
};

int
main(int argc, char* argv[])
{
  amrex::Initialize(argc, argv);

  int n_cell        = 128;
  int max_grid_size = 32;
  int which_geom    = 0;

  std::string filename;

  // read parameters
  ParmParse pp;
  pp.query("n_cell", n_cell);
  pp.query("max_grid_size", max_grid_size);
  pp.query("which_geom", which_geom);
```

(continues on next page)

(continued from previous page)

```
Geometry geom;
{
  RealBox rb;

  if (which_geom == 0) { // Airfoil case
    rb       = RealBox({-100, -100, -75}, {400, 100, 125});
    filename = "../Resources/airfoil.stl";
  }
  else if (which_geom == 1) { // Sphere case
    rb       = RealBox({-400, -400, -400}, {400, 400, 400});
    filename = "../Resources/sphere.stl";
  }
  else if (which_geom == 2) { // Dodecahedron
    rb       = RealBox({-2., -2., -2.}, {2., 2., 2.});
    filename = "../Resources/dodecahedron.stl";
  }
  else if (which_geom == 3) { // Horse
    rb       = RealBox({-0.12, -0.12, -0.12}, {0.12, 0.12, 0.12});
    filename = "../Resources/horse.stl";
  }
  else if (which_geom == 4) { // Car
    //            rb = RealBox({-20,-20,-20}, {20,20,20}); // Doesn't work.
    rb       = RealBox({-10, -5, -5}, {10, 5, 5}); // Works.
    filename = "../Resources/porsche.stl";
  }
  else if (which_geom == 5) { // Orion
    rb       = RealBox({-10, -5, -10}, {10, 10, 10});
    filename = "../Resources/orion.stl";
  }
  else if (which_geom == 6) { // Armadillo
    rb       = RealBox({-100, -75, -100}, {100, 125, 100});
    filename = "../Resources/armadillo.stl";
  }

  Array<int, AMREX_SPACEDIM> is_periodic{false, false, false};
  Geometry::Setup(&rb, 0, is_periodic.data());
  Box domain(IntVect(0), IntVect(n_cell - 1));
  geom.define(domain);
}

// Create our signed distance function. K is the tree degree while T is the
// EBGeometry precision.
constexpr int K = 4;

using T    = float;
using Vec3 = EBGeometry::Vec3T<T>;
using BV   = EBGeometry::BoundingVolumes::AABBT<T>;

AMReXSDF<T, BV, K> sdf(filename);

auto gshop = EB2::makeShop(sdf);
```

(continues on next page)

(continued from previous page)

```
  EB2::Build(gshop, geom, 0, 0);

  // Put some data
  MultiFab mf;
  {
    BoxArray boxArray(geom.Domain());
    boxArray.maxSize(max_grid_size);
    DistributionMapping dm{boxArray};

    std::unique_ptr<EBFArrayBoxFactory> factory =
      amrex::makeEBFabFactory(geom, boxArray, dm, {2, 2, 2}, EBSupport::full);

    mf.define(boxArray, dm, 1, 0, MFInfo(), *factory);
    mf.setVal(1.0);
  }

  EB_WriteSingleLevelPlotfile("plt", mf, {"rho"}, geom, 0.0, 0);

  amrex::Finalize();
}
```

### 4.4.1 Constructing the BVH

When constructing the signed distance function we use the DCEL and BVH functionality directly in the constructor. Note that we are performing the following steps:

- Using the STL parser for creating a DCEL mesh.
- Constructing a BVH for the DCEL faces.
- Flattening the BVH tree for performance.

### 4.4.2 Exposing signed distance functions

Next, we expose the signed distance function to AMReX by implementing the functions

```
Real operator()(AMREX_D_DECL(Real x, Real y, Real z)) const noexcept
```

Note that the AMReX DECL macros expand to `(Real x, Real y)` in 2D, but here we assume that the user has compiled for 3D.

## 4.5 Integration with Chombo3

**Warning:** This example requires you to install Chombo3.

This example is given in `Examples/Chombo_DCEL/main.cpp` and shows how to expose EBGeometry's DCEL and BVH functionality to Chombo3.

We will focus on the following parts of the code:

```
// Chombo includes
#include "EBISLayout.H"
#include "DisjointBoxLayout.H"
#include "BaseIF.H"
#include "GeometryShop.H"
#include "ParmParse.H"
#include "EBIndexSpace.H"
#include "BRMeshRefine.H"
#include "EBCellFactory.H"
#include "EBLevelDataOps.H"
#include "EBAMRIO.H"

// Our includes
#include "EBGeometry.hpp"

// Binding for exposing EBGeometry's signed distance functions to Chombo
template <class T, class BV, int K>
class ChomboSDF : public BaseIF
{
public:
  ChomboSDF() = delete;

  ChomboSDF(const std::string a_filename)
  {
    m_rootNode = EBGeometry::Parser::readIntoLinearBVH<T, BV, K>(a_filename);
  }

  ChomboSDF(const ChomboSDF& a_other)
  {
    m_rootNode = a_other.m_rootNode;
  }

  Real
  value(const RealVect& a_point) const override final
  {
    using Vec3 = EBGeometry::Vec3T<T>;

#if CH_SPACEDIM == 2
    Vec3 p(a_point[0], a_point[1], 0.0);
#else
    Vec3 p(a_point[0], a_point[1], a_point[2]);
#endif

    return Real(m_rootNode->signedDistance(m_rootNode->transformPoint(p)));
  }

  BaseIF*
  newImplicitFunction() const
  {
    return (BaseIF*)(new ChomboSDF(*this));
  }

protected:
```

(continues on next page)

(continued from previous page)

```
  std::shared_ptr<EBGeometry::BVH::LinearBVH<T, EBGeometry::DCEL::FaceT<T>, BV, K>> m_
↪rootNode;
};

int
main(int argc, char* argv[])
{
  constexpr int K = 4;

  using T  = float;
  using BV = EBGeometry::BoundingVolumes::AABBT<T>;

#ifdef CH_MPI
  MPI_Init(&argc, &argv);
#endif

  // Parse input file
  char*     inFile = argv[1];
  ParmParse pp(argc - 2, argv + 2, NULL, inFile);

  int nCells    = 128;
  int whichGeom = 0;
  int gridSize  = 16;
  pp.query("which_geom", whichGeom);
  pp.query("n_cells", nCells);
  pp.query("grid_size", gridSize);

  RealVect    loCorner;
  RealVect    hiCorner;
  std::string filename;

  if (whichGeom == 0) { // Airfoil
    loCorner = -50 * RealVect::Unit;
    hiCorner = 250 * RealVect::Unit;

    filename = "../Resources/airfoil.stl";
  }
  else if (whichGeom == 1) { // Sphere
    loCorner = -400 * RealVect::Unit;
    hiCorner = 400 * RealVect::Unit;

    filename = "../Resources/sphere.stl";
  }
  else if (whichGeom == 2) { // Dodecahedron
    loCorner = -2 * RealVect::Unit;
    hiCorner = 2 * RealVect::Unit;

    filename = "../Resources/dodecahedron.stl";
  }
  else if (whichGeom == 3) { // Horse
    loCorner = -0.12 * RealVect::Unit;
    hiCorner = 0.12 * RealVect::Unit;
```

(continues on next page)

(continued from previous page)

```
    filename = "../Resources/horse.stl";
  }
  else if (whichGeom == 4) { // Porsche
    loCorner = -10 * RealVect::Unit;
    hiCorner = 10 * RealVect::Unit;

    filename = "../Resources/porsche.stl";
  }
  else if (whichGeom == 5) { // Orion
    loCorner = -10 * RealVect::Unit;
    hiCorner = 10 * RealVect::Unit;

    filename = "../Resources/orion.stl";
  }
  else if (whichGeom == 6) { // Armadillo
    loCorner = -125 * RealVect::Unit;
    hiCorner = 125 * RealVect::Unit;

    filename = "../Resources/armadillo.stl";
  }

  auto impFunc = static_cast<BaseIF*>(new ChomboSDF<T, BV, K>(filename));

  // Set up the Chombo EB geometry.
  ProblemDomain domain(IntVect::Zero, (nCells - 1) * IntVect::Unit);
  const Real    dx = (hiCorner[0] - loCorner[0]) / nCells;
  ;

  GeometryShop  workshop(*impFunc, -1, dx * RealVect::Zero);
  EBIndexSpace* ebisPtr = Chombo_EBIS::instance();
  ebisPtr->define(domain, loCorner, dx, workshop, gridSize, -1);

  // Set up the grids
  Vector<int> procs;
  Vector<Box> boxes;
  domainSplit(domain, boxes, gridSize, gridSize);
  mortonOrdering(boxes);
  LoadBalance(procs, boxes);
  DisjointBoxLayout dbl(boxes, procs);

  // Fill the EBIS layout
  EBISLayout ebisl;
  ebisPtr->fillEBISLayout(ebisl, dbl, domain, 1);

  // Allocate some data that we can output
  LevelData<EBCellFAB> data(dbl, 1, IntVect::Zero, EBCellFactory(ebisl));
  for (DataIterator dit(dbl); dit.ok(); ++dit) {
    EBCellFAB& fab = data[dit()];
    fab.setVal(0.0);

    const Box region = fab.getRegion();
```

(continues on next page)

(continued from previous page)

```
    for (BoxIterator bit(region); bit.ok(); ++bit) {
      const IntVect iv = bit();

      const RealVect pos         = loCorner + (iv + 0.5 * RealVect::Unit) * dx;
      fab.getFArrayBox()(iv, 0) = impFunc->value(pos);
    }
  }

  // Write to HDF5
  Vector<LevelData<EBCellFAB>*> amrData;
  amrData.push_back(&data);
  writeEBAMRname(&amrData, "example.hdf5");

#ifdef CH_MPI
  MPI_Finalize();
#endif
  return 0;
}
```

### 4.5.1 Constructing the BVH

When constructing the signed distance function we use the DCEL and BVH functionality directly in the constructor. Note that we are performing the following steps:

- Using the STL parser for creating a DCEL mesh.
- Constructing a BVH for the DCEL faces.
- Flattening the BVH tree for performance.

### 4.5.2 Exposing signed distance functions

Next, we expose the signed distance function to Chombo3 by implementing the functions

```
Real
value(const RealVect& a_point) const override final
{
#if CH_SPACEDIM == 2
  Vec3 p(a_point[0], a_point[1], 0.0);
#else
  Vec3 p(a_point[0], a_point[1], a_point[2]);
#endif

  return Real(m_rootNode->signedDistance(p));
}
```

Note that because Chombo3 can be compiled in either 2D or 3D, we put a Chombo preprocessor flag `CH_SPACEDIM` in order to translate the Chombo `RealVect` to EBGeometry's inherent 3D vector structure.

CHAPTER

FIVE

# REFERENCES

## 5.1 References

# BIBLIOGRAPHY

[1] J.A. Baerentzen and H. Aanaes. Signed distance computation using the angle weighted pseudonormal. *IEEE Transactions on Visualization and Computer Graphics*, 11(3):243–253, 2005. doi:10.1109/TVCG.2005.49.