EBGeometry

Aug 27, 2024

CONTENTS

This is the user documentation for EBGeometry, a small C++ package for efficiently representing implicit functions and signed distance fields for complex geometries. Although EBGeometry is a self-contained package, it is was originally written for usage with embedded boundary (EB) and immersed boundary (IB) codes. EBGeometry provides the geometry representation through an implicit or signed distance function, but does not provide the discrete geometry generation, i.e. the generation of cut-cells for a given geometry.

The basic features of EBGeometry are as follows:

- Representation of water-tight surface grids as signed distance fields.
- Many analytic distance functions and transformations.
- Bounding volume hierarchies (BVHs) for use as acceleration structures for polygon or full object lookup. The BVHs can be represented in full or compact (i.e., linearized) forms.
- Support for both conventional and accelerated (using BVHs) constructive solid geometry (CSG).
- Examples of how to couple EBGeometry to AMReX and Chombo.

Important: This is the user documentation for EBGeometry. The source code is found at https://github.com/rmrsk/EBGeometry and a separate Doxygen-generated API of EBGeometry is available at https://rmrsk.github.io/EBGeometry/doxygen/html/index.html.

CHAPTER

ONE

INTRODUCTION

1.1 Requirements

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

`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.

1.2 Quickstart

To obtain `EBGeometry`, clone the code from github:

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

To compile the `EBGeometry` example codes, navigate to the `EBGeometry/Examples` folder. Folders that are named `EBGeometry_<something>` are pure `EBGeometry` examples and can be compiled without any third-party dependencies.

To run the `EBGeometry` examples, navigate to one of the folders in `EBGeometry/Examples/EBGeometry_*` and execute

```
g++ -O3 main.cpp && ./a.out
```

All `EBGeometry` examples can be run using this command. README files present in each folder provide more information regarding the functionality and usage of each example code.

1.3 Third-party examples

Exapmle folders that begin with e.g. `AMReX_` or `Chombo3_` are application code examples and require the user to install additional third-party software.

CHAPTER

TWO

CONCEPTS

2.1 Geometry representations

2.1.1 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. It has the additional property

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

The normal vector is always

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

`EBGeometry` uses the following convention for the sign:

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

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

2.1.2 Implicit functions

Like distance functions, implicit functions also determine whether or not a point $\mathbf{x}$ is inside or outside an object. Signed distance functions are also *implicit functions*, but not vice versa. For example, the signed distance function for a sphere with center $\mathbf{x}_0$ and radius R can be written

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

An example of an implicit function for the same sphere is

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

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. Signed distance functions are more useful objects, but many operations (e.g. CSG unions) do not preserve the signed distance property (but still provide *bounds* for the signed distance).

2.2 DCEL

2.2.1 Principle

`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, half-edges circulate the inside of the facet, with pointer access to the next half-edge. A half-edge also stores a reference to its starting vertex, as well as a reference to its pair-edge. From the DCEL structure we can obtain all edges or vertices belonging to a single facet by iterating through the half-edges, and also jump to a neighboring facet by fetching the pair edge.

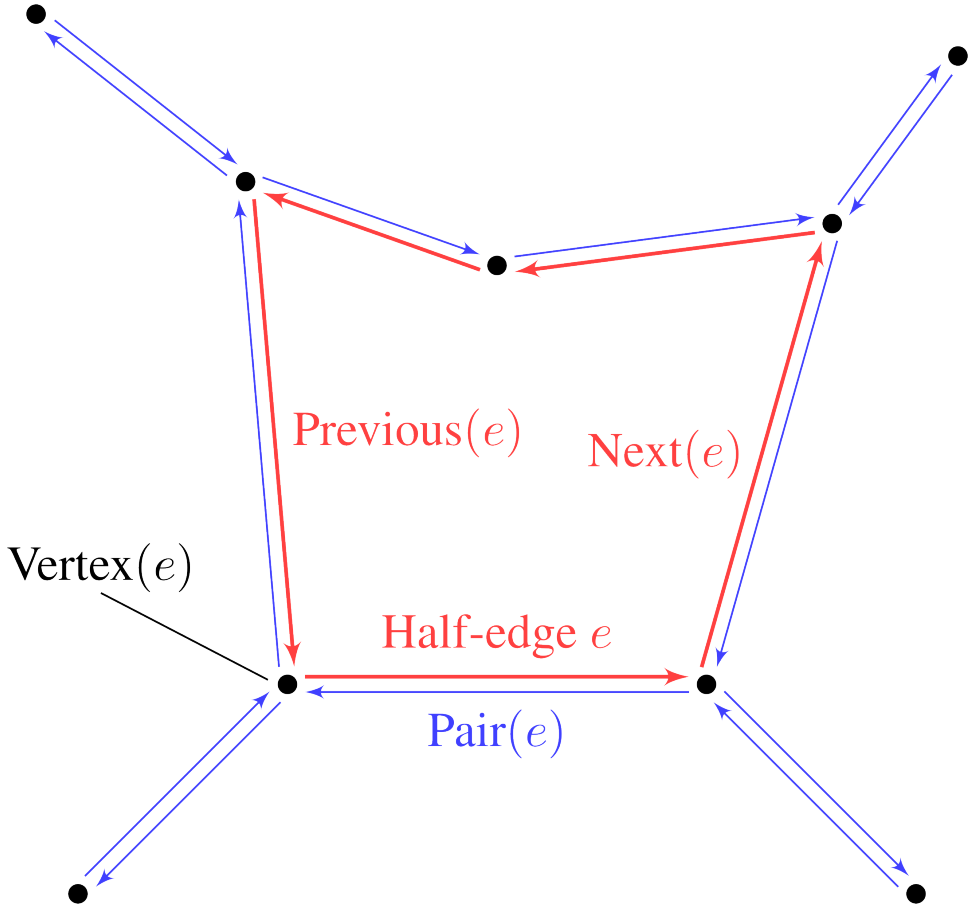

Fig. 2.1: DCEL mesh structure. Each half-edge stores references to next half-edge, the pair edge, and the starting vertex. Vertices store a coordinate as well as a reference to one of the outgoing half-edges.

In `EBGeometry` the half-edge data structure is implemented in its own namespace. 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, neither the signed distance or implicit function exist.

2.2.2 Signed distance

The signed distance to a surface mesh is equivalent to the signed distance to the closest polygon face in the mesh. When computing the signed distance from a point $\mathbf{x}$ to a polygon face (e.g., a triangle), the closest feature on the polygon can be one of the vertices, edges, or the interior of the polygon face, see Fig. 2.2.

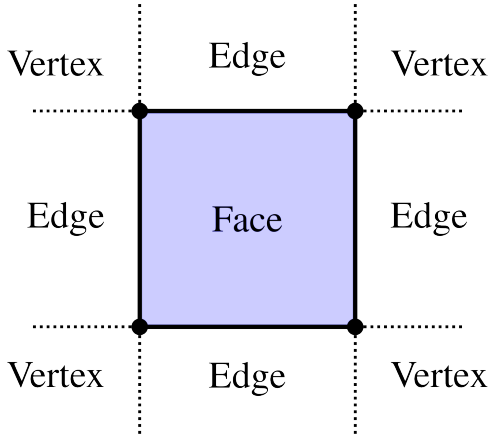

Fig. 2.2: Possible closest-feature cases after projecting a point $\mathbf{x}$ to the plane of a polygon face.

Three cases can be distinguished:

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 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.

   Tip: `EBGeometry` uses the crossing number algorithm by default.

   If the point projects to the inside of the face, the signed distance is just $\mathbf{n}_f \cdot (\mathbf{x} - \mathbf{x}_f)$ 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.

2.2.3 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 is given by

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

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

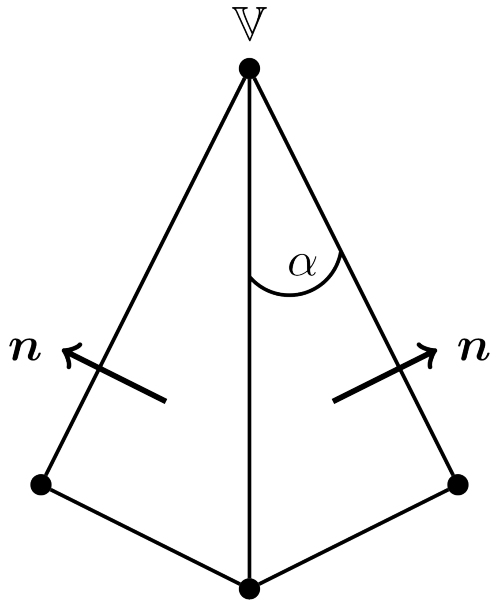

Fig. 2.3: Edge and vertex pseudonormals.

2.3 Bounding volume hierarchies

Bounding volume hierarchies (BVHs) are tree structures where the regular nodes are bounding volumes that enclose all geometric primitives (e.g. polygon faces or implicit functions) 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. There are two types of nodes in a BVH:

- **Regular/interior nodes.** These do not contain any of the primitives/objects, but store references to subtrees (aka child nodes).
- **Leaf nodes.** These lie at the bottom of the BVH tree and each of them contains 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 its subtree. Its 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 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.

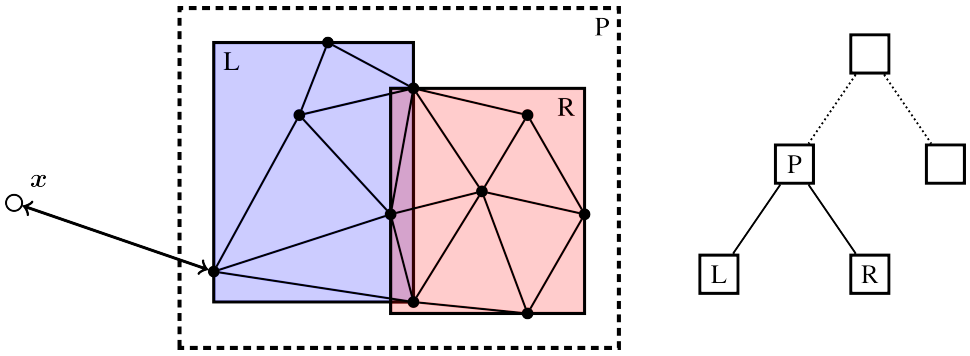

Fig. 2.4: Example of BVH partitioning for enclosing triangles. The regular node P contains two leaf nodes L and R which contain the primitives (triangles).

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 bounding volumes that enclose them.

Important: `EBGeometry` is not limited to binary trees, but supports k -ary trees where each regular node has k child nodes.

2.3.1 Construction

BVH construction is fairly flexible. 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/primitive.

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.

Bottom-up construction is also possible, in which case one constructs the leaf nodes first, and then merge the nodes upward until one reaches a root node. In `EBGeometry`, bottom-up construction is done by means of space-filling curves (e.g., Morton codes).

2.3.2 Tree traversal

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 compute the signed distance from $\mathbf{x}$ to the primitives in A. This requires us to iterate over all the triangles in A.
- When investigating the other child node of 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.

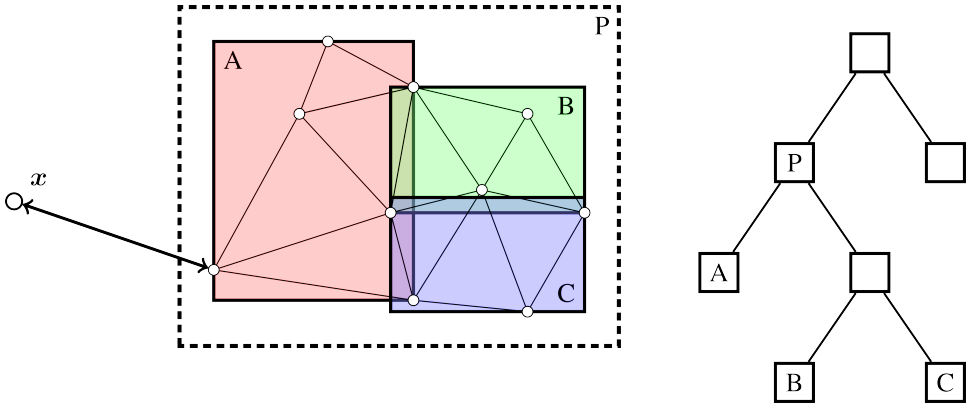

Fig. 2.5: Example of BVH tree pruning.

Warning: BVH traversal has $\log N$ complexity on average. However in the worst case the traversal algorithm may have linear complexity if 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 since all triangles and their bounding volumes are approximately at the same distance from the center.

Other types of tree traversal (that do not compute the signed distance) are also possible. EBGeometry supports a fairly flexible approach to the tree traversal and update algorithms such that the user is permitted to use the hierarhical traversal algorithm also for other types of operations (e.g., for finding all facets within a specified distance from a point).

2.4 Octree

Octrees are tree-structures where each interior node has exactly eight children. Such trees are usually used for spatial partitioning. Unlike BVH trees, the eight child nodes have no spatial overlap.

Octree construction can be done in (at least) two ways:

1. In depth-first order where entire sub-trees are built first.
2. In breadth-first order where tree levels are added one at a time.

EBGeometry supports both of these methods. Octree traversal is generally speaking quite similar to the traversal algorithms used for BVH trees.

2.5 Constructive solid geometry

2.5.1 Basic transformations

Implicit functions, and by extension also signed distance fields, can be manipulated using basic transformations (like rotations). EBGeometry supports many of these:

- Rotations.
- Translations.
- Surface offsets.
- Shell extraction.
- Mollification (e.g., smoothing)
- ... and others.

Warning: Some of these operations preserve the signed distance property, and others do not.

2.5.2 Combining objects

EBGeometry supports standard operations in which implicit functions can be combined:

- Union.
- Intersection.
- Difference.

Some of these CSG operations also have smooth equivalents, i.e. for smoothing the transition between combined objects. Fast CSG operations are also supported by EBGeometry, e.g. the BVH-accelerated CSG union where one uses the BVH when searching for the relevant geometric primitive(s). This functionality is motivated by the fact that a CSG union is normally implemented as $\min\left(I_1, I_2, I_3, \ldots, I_N\right)$, which has $\mathcal{O}\left(N\right)$ complexity when there are N objects. BVH trees can reduce this to $\mathcal{O}\left(\log N\right)$ complexity.

CHAPTER

THREE

IMPLEMENTATION

3.1 Overview

Here, we consider the basic `EBGeometry` API. `EBGeometry` is a header-only library, implemented under its 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` implements its own 2D and 3D 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*. `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 Geometry representation

3.3.1 Implicit functions

EBGeometry implements implement functions and signed distance functions through virtual classes

```
template <class T>
class ImplicitFunction
{
public:
  /*!
    @brief Disallowed, use the full constructor
  */
  ImplicitFunction() = default;

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

  /*!
    @brief Value function. Points are outside the object if value > 0.0 and inside
    if value < 0.0
    @param[in] a_point 3D point.
  */
  virtual T
  value(const Vec3T<T>& a_point) const noexcept = 0;

  /*!
    @brief Alternative signature for the value function.
    @param[in] a_point 3D point.
  */
  T
  operator()(const Vec3T<T>& a_point) const noexcept;

  /*!
    @brief Compute an approximation to the bounding volume for the implicit surface, using octrees.
    @details This routine will try to compute a bonding using octree subdivision of the implicit function. This routine will
    characterize a cubic region in space as being 'inside', 'outside', or intersected by computing the value function at the
    center of the cube. If the value function is larger than the extents of the cube, we assume that there is no intersection
    inside the cube. The success of this algorithm therefore relies on the implicit function also being a signed distance
    function, or at the very least not being horrendously far from being an SDF. If octree subdivision fails, this will return
    the maximally representable bounding volume.
    @param[in] a_initialLowCorner Initial low corner.
    @param[in] a_initialHighCorner Initial high corner.
    @param[in] a_maxTreeDepth Maximum permitted octree depth.
    @param[in] a_safety Safety factor when determining intersection. a_safety=1 sets safety factor to cube width, a_safety=2␣
↪sets twice the cube width, etc.
    @note The bounding volume type BV MUST have a constructor BV(std::vector<Vec3T<T>>).
  */
  template <class BV>
  inline BV
  approximateBoundingVolumeOctree(const Vec3T<T>&    a_initialLowCorner,
                                  const Vec3T<T>&    a_initialHighCorner,
                                  const unsigned int a_maxTreeDepth,
                                  const T&           a_safety = 0.0) const noexcept;
};
```

Signed distance fields inherit from implicit functions as follows:

```
template <class T>
class SignedDistanceFunction : public ImplicitFunction<T>
{
public:
  /*!
    @brief Disallowed, use the full constructor
  */
  SignedDistanceFunction() = default;

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

  /*!
```

(continues on next page)

(continued from previous page)

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

  /*!
    @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;
};
```

Note that `T` is a floating point precision, which is supported because DCEL meshes can take up quite a bit of computer memory. These declarations are found in

- `Source/EBGeometry_ImplicitFunction.hpp` for implicit functions.
- `Source/EBGeometry_SignedDistanceFunction.hpp` for signed distance field.

Various useful implementations of implicit functions and distance fields are found in:

- `Source/EBGeometry_AnalyticDistanceFields.hpp` for various pre-defined analytic distance fields.
- `Source/EBGeometry_MeshDistanceFields.hpp` for various distance field representations for DCEL meshes.

3.3.2 Transformations

Various transformations for implicit functions are defined in `Source/EBGeometry_Transform.hpp`, such as rotations, translations, etc. These are also available through functions that automatically cast the resulting implicit function to `ImplicitFunction<T>`:

```
/*!
  @brief Convenience function for taking the complement of an implicit function
  @param[in] a_implicitFunction Input implicit function
*/
template <class T>
std::shared_ptr<ImplicitFunction<T>>
Complement(const std::shared_ptr<ImplicitFunction<T>>& a_implicitFunction) noexcept;

/*!
  @brief Convenience function for translating an implicit function
  @param[in] a_implicitFunction Input implicit function to be translated
  @param[in] a_shift Distance to shift
*/
template <class T>
std::shared_ptr<ImplicitFunction<T>>
Translate(const std::shared_ptr<ImplicitFunction<T>>& a_implicitFunction, const Vec3T<T>& a_shift) noexcept;

/*!
  @brief Convenience function for rotating an implicit function.
  @param[in] a_implicitFunction Input implicit function to be rotated.
  @param[in] a_angle Angle to be rotated by (in degrees)
  @param[in] a_axis Axis to rotate about
*/
template <class T>
std::shared_ptr<ImplicitFunction<T>>
Rotate(const std::shared_ptr<ImplicitFunction<T>>& a_implicitFunction, const T a_angle, const size_t a_axis) noexcept;

/*!
  @brief Convenience function for scaling an implicit function.
```

(continues on next page)

(continued from previous page)

```
  @param[in] a_implicitFunction Input implicit function to be scaled.
  @param[in] a_scale Scaling factor
*/
template <class T>
std::shared_ptr<ImplicitFunction<T>>
Scale(const std::shared_ptr<ImplicitFunction<T>>& a_implicitFunction, const T a_scale) noexcept;

/*!
  @brief Convenience function for offsetting an implicit function
  @param[in] a_implicitFunction Input implicit function to be offset
  @param[in] a_offset Offset distance
*/
template <class T>
std::shared_ptr<ImplicitFunction<T>>
Offset(const std::shared_ptr<ImplicitFunction<T>>& a_implicitFunction, const T a_offset) noexcept;

/*!
  @brief Convenience function for creating a shell out of an implicit function
  @param[in] a_implicitFunction Input implicit function to be shelled.
  @param[in] a_delta Shell thickness
*/
template <class T>
std::shared_ptr<ImplicitFunction<T>>
Annular(const std::shared_ptr<ImplicitFunction<T>>& a_implicitFunction, const T a_delta) noexcept;

/*!
  @brief Convenience function for blurring an implicit function
  @param[in] a_implicitFunction Input implicit function to be blurred
  @param[in] a_blurDistance Smoothing distance
*/
template <class T>
std::shared_ptr<ImplicitFunction<T>>
Blur(const std::shared_ptr<ImplicitFunction<T>>& a_implicitFunction, const T a_blurDistance) noexcept;

/*!
  @brief Convenience function for mollification with an input sphere.
  @param[in] a_implicitFunction Input implicit function to be mollifier
  @param[in] a_dist Mollification distance.
  @param[in] a_mollifierSamples Number of samples for the mollifier
*/
template <class T>
std::shared_ptr<ImplicitFunction<T>>
Mollify(const std::shared_ptr<ImplicitFunction<T>>& a_implicitFunction,
        const T                                     a_dist,
        const size_t                                a_mollifierSamples = 2) noexcept;

/*!
  @brief Convenience function for elongating (stretching) an implicit function
  @param[in] a_implicitFunction Implicit function to be elongated
  @param[in] a_elongation Elongation
*/
template <class T>
std::shared_ptr<ImplicitFunction<T>>
Elongate(const std::shared_ptr<ImplicitFunction<T>>& a_implicitFunction, const Vec3T<T>& a_elongation) noexcept;

/*!
  @brief Convenience function for reflecting an implicit function
  @param[in] a_implicitFunction Implicit function to be reflected
  @param[in] a_reflectPlane Plane to reflect across  (0=yz-plane, 1=xz-plane, 2=xy-plane).
*/
template <class T>
std::shared_ptr<ImplicitFunction<T>>
Reflect(const std::shared_ptr<ImplicitFunction<T>>& a_implicitFunction, const size_t& a_reflectPlane) noexcept;
```

3.3.3 CSG operations

CSG operations for implicit functions are defined in `Source/EBGeometry_CSG.hpp`. These also include accelerated variants that take advantage of BVH partitioning of the objects.

```
/*!
  @brief Convenience function for taking the union of a bunch of a implicit functions
  @param[in] a_implicitFunctions Implicit functions
  @note P must derive from ImplicitFunction<T>
*/
template <class T, class P = ImplicitFunction<T>>
std::shared_ptr<ImplicitFunction<T>>
Union(const std::vector<std::shared_ptr<P>>& a_implicitFunctions) noexcept;

/*!
  @brief Convenience function for taking the union of two implicit functions
  @param[in] a_implicitFunctionA First implicit function.
  @param[in] a_implicitFunctionB Second implicit function.
  @note P1 and P2 must derive from ImplicitFunction<T>
*/
template <class T, class P1, class P2>
std::shared_ptr<ImplicitFunction<T>>
Union(const std::shared_ptr<P1>& a_implicitFunctionA, const std::shared_ptr<P2>& a_implicitFunctionB) noexcept;

/*!
  @brief Convenience function for taking the union of a bunch of a implicit functions
  @param[in] a_implicitFunctions Implicit functions
  @param[in] a_smooth Smoothing distance.
  @note P must derive from ImplicitFunction<T>
*/
template <class T, class P = ImplicitFunction<T>>
std::shared_ptr<ImplicitFunction<T>>
SmoothUnion(const std::vector<std::shared_ptr<P>>& a_implicitFunctions, const T a_smooth) noexcept;

/*!
  @brief Convenience function for taking the union of two implicit functions
  @param[in] a_implicitFunctionA First implicit function.
  @param[in] a_implicitFunctionB Second implicit function.
  @param[in] a_smooth Smoothing distance.
  @note P1 and P2 must derive from ImplicitFunction<T>
*/
template <class T, class P1, class P2>
std::shared_ptr<ImplicitFunction<T>>
SmoothUnion(const std::shared_ptr<P1>& a_implicitFunctionA,
            const std::shared_ptr<P2>& a_implicitFunctionB,
            const T                    a_smooth) noexcept;

/*!
  @brief Convenience function for taking the BVH-accelerated union of a bunch of a implicit functions
  @param[in] a_implicitFunctions Implicit functions
  @param[in] a_boundingVolumes Bounding volumes for implicit functions.
  @note P must derive from ImplicitFunction<T>
*/
template <class T, class P, class BV, size_t K>
std::shared_ptr<ImplicitFunction<T>>
FastUnion(const std::vector<std::shared_ptr<P>>& a_implicitFunctions,
          const std::vector<BV>&                 a_boundingVolumes) noexcept;

/*!
  @brief Convenience function for taking the BVH-accelerated union of a bunch of a implicit functions
  @param[in] a_implicitFunctions Implicit functions
  @param[in] a_boundingVolumes Bounding volumes for the implicit functions.
  @param[in] a_smoothLen Smoothing length
  @note P must derive from ImplicitFunction<T>
*/
template <class T, class P, class BV, size_t K>
std::shared_ptr<ImplicitFunction<T>>
FastSmoothUnion(const std::vector<std::shared_ptr<P>>& a_implicitFunctions,
                const std::vector<BV>&                 a_boundingVolumes,
                const T                                a_smoothLen) noexcept;

/*!
  @brief Convenience function for taking the intersection of a bunch of a implicit functions
  @param[in] a_implicitFunctions Implicit functions
  @note P must derive from ImplicitFunction<T>
*/
template <class T, class P>
std::shared_ptr<ImplicitFunction<T>>
```

(continues on next page)

(continued from previous page)

```
Intersection(const std::vector<std::shared_ptr<P>>& a_implicitFunctions) noexcept;

/*!
  @brief Convenience function for taking the intersection of two implicit functions
  @param[in] a_implicitFunctionA First implicit function.
  @param[in] a_implicitFunctionB Second implicit function.
  @note P1 and P2 must derive from ImplicitFunction<T>
*/
template <class T, class P1, class P2>
std::shared_ptr<ImplicitFunction<T>>
Intersection(const std::shared_ptr<std::shared_ptr<P1>>& a_implicitFunctionA,
             const std::shared_ptr<std::shared_ptr<P2>>& a_implicitFunctionB) noexcept;

/*!
  @brief Convenience function for taking the smooth intersection of a bunch of a implicit functions
  @param[in] a_implicitFunctions Implicit functions
  @param[in] a_smooth Smoothing distance.
  @note P must derive from ImplicitFunction<T>
*/
template <class T, class P>
std::shared_ptr<ImplicitFunction<T>>
SmoothIntersection(const std::vector<std::shared_ptr<P>>& a_implicitFunctions, const T a_smooth) noexcept;

/*!
  @brief Convenience function for taking the smooth intersection of two implicit functions
  @param[in] a_implicitFunctionA First implicit function.
  @param[in] a_implicitFunctionB Second implicit function.
  @param[in] a_smooth Smoothing distance.
  @note P1 and P2 must derive from ImplicitFunction<T>
*/
template <class T, class P1, class P2>
std::shared_ptr<ImplicitFunction<T>>
SmoothIntersection(const std::shared_ptr<P1>& a_implicitFunctionA,
                   const std::shared_ptr<P2>& a_implicitFunctionB,
                   const T                    a_smooth) noexcept;

/*!
  @brief Convenience function for taking the CSG difference.
  @param[in] a_implicitFunctionA Implicit function.
  @param[in] a_implicitFunctionB Implicit function to subtract.
  @note P1 and P2 must derive from ImplicitFunction<T>
*/
template <class T, class P1 = ImplicitFunction<T>, class P2 = ImplicitFunction<T>>
std::shared_ptr<ImplicitFunction<T>>
Difference(const std::shared_ptr<P1>& a_implicitFunctionA, const std::shared_ptr<P2>& a_implicitFunctionB) noexcept;

/*!
  @brief Convenience function for taking the smooth CSG difference.
  @param[in] a_implicitFunctionA Implicit function.
  @param[in] a_implicitFunctionB Implicit function to subtract.
  @param[in] a_smoothLen Smoothing length.
  @note P1 and P2 must derive from ImplicitFunction<T>. This uses the default smoothMax function.
*/
template <class T, class P1 = ImplicitFunction<T>, class P2 = ImplicitFunction<T>>
std::shared_ptr<ImplicitFunction<T>>
SmoothDifference(const std::shared_ptr<P1>& a_implicitFunctionA,
                 const std::shared_ptr<P2>& a_implicitFunctionB,
                 const T                    a_smoothLen) noexcept;
```

3.3.4 Bounding volumes

For simple shapes, bounding volumes can be directly constructed given an implicit function. E.g. one can easily find the bounding volume for a sphere with a known center and radius, or for a DCEL mesh. However, more complicated implicit functions require us to compute the bounding volume, or at the very least an approximation to it. `ImplicitFunction<T>` has a member function that uses spatial subdivision (based on octrees) for doing this:

```
// Compute an approximate bounding volume BV.
template <class BV>
inline BV
approximateBoundingVolumeOctree(const Vec3T<T>&    a_initialLowCorner,
                                const Vec3T<T>&    a_initialHighCorner,
                                const unsigned int a_maxTreeDepth,
                                const T&           a_safety = 0.0) const noexcept;
```

This function initializes a cubic region in space and uses octree refinement near the implicit surface. At the end of the octree recusion the vertices of the octree leaves are collected and a bounding volume of type `BV` that encloses them is computed. The success of this method relies on the implicit function being a signed distance function (or at least an approximation to it).

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.
- **BVH functionality** for putting DCEL grids into bounding volume hierarchies.

Important: The DCEL functionality is *not* restricted to triangles, but supports N-sided polygons, including *meta-data* attached to the vertices, edges, and facets. The latter is particularly useful in case on wants to associate e.g. boundary conditions to specific triangles.

3.4.1 Main types

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 Meta>
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 Meta>
class EdgeT
```

  The half-edges store a reference to their face, as well as pointers to the next edge, pair edge, and starting vertex.

  The full API is given in the doxygen documentation here.

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

```
template <class T, class Meta>
class FaceT
```

  Faces also store

  - 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 Meta>
class MeshT : public SignedDistanceFunction<T>
```

The mesh stores all the vertices, half-edges, and faces, and if it is watertight and orientable it is also a signed distance function. Typically, the mesh is not created by the user but automatically created when reading the mesh from an input file.

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;
```

which can be used to compute the distance to the various features on the mesh.

Meta-data can be attached to the DCEL primitives by selecting an appropriate type for `Meta` above.

3.4.2 BVH integration

DCEL grids can easily be embedded in BVHs by enclosing bounding volumes around the polygons (e.g., triangles). Partitioning and bounding volume constructors are provided in `Source/EBGeometry_MeshDistanceFunctions.hpp`.

3.5 BVH

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.

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

The full BVH is encapsulated by a class

```
template <class T, class P, class BV, size_t 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*.

3.5.1 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**, which is templated as EBGeometry::BoundingVolumes::AABBT<T>.

For full API details, see the doxygen API. Other types of bounding volumes can in principle be added, with the only requirement being that they conform to the same interface as the AABB and BoundingSphere volumes.

3.5.2 Construction

Constructing a BVH is done by:

1. Creating a root node and providing it with the geometric primitives and their bounding volumes.
2. Partitioning the BVH by providing a partitioning function.

The first step is usually a matter of simply constructing the root node using the full constructor, which takes a list of primitives and their associated bounding volumes. The second step is to recursively build the BVH. We currently support top-down and bottom-up construction (using space-filling curves).

Tip: The default construction methods performs the hierarchical subdivision by only considering the *bounding volumes*. Consequently, the build process is identical regardless of what type of primitives (e.g., triangles or analytic spheres) are contained in the BVH.

Top-down construction

Top-down construction is done through the function topDownSortAndPartition(), see the doxygen API for the BVH implementation.

The optional input arguments to topDownSortAndPartition are polymorphic functions of type indicated above, and have the following responsibilities:

- PartitionerT is the partitioner function when splitting a leaf node into K new leaves. The function takes a list of primitives which it partitions into K new lists of primitives.
- StopFunctionT simply takes a NodeT as input argument and determines if the node should be partitioned further.

Default arguments for these are provided, bubt users are free to partition their BVHs in their own way should they choose.

Bottom-up construction

The bottom-up construction uses a space-filling curve (e.g., a Morton curve) for first building the leaf nodes. This construction is done such that each leaf node contains approximately the number of primitives, and all leaf nodes exist on the same level. To use bottom-up construction, one may use the member function

```
/*!
  @brief Function for doing bottom-up construction using a specified space-filling curve.
  @details The template parameter is the space-filling curve type. This function will partition the BVH
  by first sorting the bounding volume centroids along the space-filling curve. The tree is then constructed
  by placing at least K primitives in each leaf, and the leaves are then merged upwards until we reach the
```

(continues on next page)

(continued from previous page)

```
    root node.
    @note S must have an encode and decode function which returns an SFC index. See the SFC namespace for
    examples for Morton and Nested indices.
  */
  template <typename S>
  inline void
  bottomUpSortAndPartition() noexcept;
```

The template argument is the space-filling curve that the user wants to apply. Currently, we support Morton codes and nested indices. For Morton curves, one would e.g. call `bottomUpSortAndPartition<SFC::Morton>` while for nested indices (which are not recommended) the signature is likewise `bottomUpSortAndPartition<SFC::Nested>`.

Build times for SFC-based bottom-up construction are generally speaking faster than top-down construction, but tends to produce worse trees such that traversal becomes slower.

3.5.3 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.

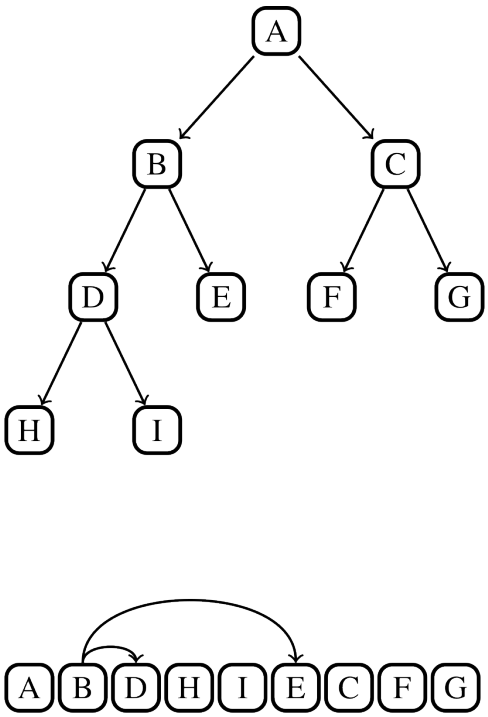

Fig. 3.1: Compact BVH representation. The original BVH is traversed from top-to-bottom along the branches and laid out in linear memory. Each interior node gets a reference (index offset) to their children nodes.

The rationale for reorganizing the BVH in compact form is it's tighter memory footprint and depth-first ordering which occasionally allows a more efficient traversal downwards in the BVH tree, particularly if the geometric primitives are sorted in the same order. 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:

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:

  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` or ensure correct scoping.

3.5.4 Tree traversal

Both `NodeT` (full BVH) and `LinearBVH` (flattened BVH) include routines for traversing the BVH with user-specified criteria. For both BVH representations, tree traversal is done using a routine

```
template <class Meta>
inline void
traverse(const BVH::Updater<P>&                    a_updater,
         const BVH::Visiter<LinearNode, Meta>&     a_visiter,
         const BVH::Sorter<LinearNode, Meta, K>&   a_sorter,
         const BVH::MetaUpdater<LinearNode, Meta>& a_metaUpdater) const noexcept;
```

The BVH trees use a stack-based traversal pattern based on visit-sort rules supplied by the user.

Node visit

Here, `a_visiter` is a lambda function for determining if the node/subtree should be investigated or pruned from the traversal. This function has a signature

```
template <class NodeType, class Meta>
using Visiter = std::function<bool(const NodeType& a_node, const Meta a_meta)>;
```

where `NodeType` is the type of node (which is different for full/flat BVHs), and the `Meta` template parameter is discussed below. If this function returns true, the node will be visisted and if the function returns false then the node will be pruned from the tree traversal. Typically, the `Meta` parameter will contain the necessary information that determines whether or not to visit the subtree.

Traversal pattern

If a subtree is visited in the traversal, there is a question of which of the child nodes to visit first. The `a_sorter` argument determines the order by letting the user sort the nodes based on order of importance. Note that a correct visitation pattern can yield large performance benefits. The user is given the option to sort the child nodes based on what he/she thinks is a good order, which is done by supplying a lambda which sorts the children. This function has the signature:

```
template <class NodeType, class Meta, size_t K>
using Sorter = std::function<void(std::array<std::pair<std::shared_ptr<const NodeType>, Meta>, K>& a_children)>;
```

Sorting the child nodes is completely optional. The user can leave this function empty if it does not matter which subtrees are visited first.

Update rule

If a leaf node is visited in the traversal, distance or other types of queries to the geometric primitive(s) in the nodes are usually made. These are done by a user-supplied update-rule:

```
template <class P>
using Updater = std::function<void(const PrimitiveListT<P>& a_primitives)>;
```

Typically, the `Updater` will modify parameters that appear in a local scope outside of the tree traversal (e.g. updating the minimum distance to a DCEL mesh).

Meta-data

During the traversal, it might be necessary to compute meta-data that is helpful during the traversal, and this meta-data is attached to each node that is queried. This meta-data is usually, but not necessarily, equal to the distance to the nodes' bounding volumes. The signature for meta-data construction is

```
template <class NodeType, class Meta>
using MetaUpdater = std::function<Meta(const NodeType& a_node)>;
```

The biggest difference between `Updater` and `MetaUpdater` is that `Updater` is *only* called on leaf nodes whereas `MetaUpdater` is also called for internal nodes. One typical example for DCEL meshes is that `Updater` computes the distance from an input point to the triangles in a leaf node, whereas `MetaUpdater` computes the distance from the input point to the bounding volumes of a child nodes. This information is then used in `Sorter` in order to determine a preferred child visit pattern when descending along subtrees.

Traversal algorithm

The code-block below shows the implementation of the BVH traversal. The implementation uses a non-recursive queue-based formulation when descending along subtrees. Observe that each entry in the stack contains both the node itself *and* any meta-data we want to attach to the node. If the traversal decides to visit a node, it immediately computes the specified meta-data of the node, and the user can then sort the children based on that data.

Listing 3.1: Tree traversal algorithm for the BVH tree.

```
template <class T, class P, class BV, size_t K>
template <class Meta>
inline void
NodeT<T, P, BV, K>::traverse(const BVH::Updater<P>&              a_updater,
                             const BVH::Visiter<Node, Meta>&     a_visiter,
                             const BVH::Sorter<Node, Meta, K>&   a_sorter,
                             const BVH::MetaUpdater<Node, Meta>& a_metaUpdater) const noexcept
{
  std::array<std::pair<std::shared_ptr<const Node>, Meta>, K> children;
  std::stack<std::pair<std::shared_ptr<const Node>, Meta>>    q;

  q.emplace(this->shared_from_this(), a_metaUpdater(*this));

  while (!(q.empty())) {
    const auto& node = q.top().first;
    const auto& meta = q.top().second;

    q.pop();

    if (a_visiter(*node, meta)) {
      if (node->isLeaf()) {
        a_updater(node->getPrimitives());
      }
      else {
        for (size_t k = 0; k < K; k++) {
          children[k].first  = node->getChildren()[k];
          children[k].second = a_metaUpdater(*children[k].first);
        }

        // User-based visit pattern.
        a_sorter(children);

        for (const auto& child : children) {
          q.push(child);
        }
      }
    }
  }
```

Traversal examples

Below, we consider two examples for BVH traversal. The examples show how we compute the signed distance from a DCEL mesh, and how to perform a *smooth* CSG union where the search for the two closest objects is done by BVH traversal.

Signed distance

The DCEL mesh distance fields use a traversal pattern based on

- Only visit bounding volumes that are closer than the minimum distance computed (so far).
- When visiting a subtree, investigate the closest bounding volume first.
- When visiting a leaf node, check if the primitives are closer than the minimum distance computed so far.

These rules are given below.

Listing 3.2: Tree traversal criterion for computing the signed distance to a DCEL mesh using the BVH accelerator. See `Source/EBGeometry_MeshDistanceFunctionsImplem.hpp` for details.

```
template <class T, class Meta, class BV, size_t K>
T
FastMeshSDF<T, Meta, BV, K>::signedDistance(const Vec3T<T>& a_point) const noexcept
{
  T minDist = std::numeric_limits<T>::infinity();

  BVH::Updater<Face> updater = [&minDist,
                                &a_point](const std::vector<std::shared_ptr<const Face>>& faces) noexcept -> void {
    for (const auto& f : faces) {
      const T curDist = f->signedDistance(a_point);

      minDist = std::abs(curDist) < std::abs(minDist) ? curDist : minDist;
    }
  };

  BVH::Visiter<Node, T> visiter = [&minDist, &a_point](const Node& a_node, const T& a_bvDist) noexcept -> bool {
    return a_bvDist <= std::abs(minDist);
  };

  BVH::Sorter<Node, T, K> sorter =
    [&a_point](std::array<std::pair<std::shared_ptr<const Node>, T>, K>& a_leaves) noexcept -> void {
    std::sort(
      a_leaves.begin(),
      a_leaves.end(),
      [&a_point](const std::pair<std::shared_ptr<const Node>, T>& n1,
                 const std::pair<std::shared_ptr<const Node>, T>& n2) -> bool { return n1.second > n2.second; });
  };

  BVH::MetaUpdater<Node, T> metaUpdater = [&a_point](const Node& a_node) noexcept -> T {
    return a_node.getDistanceToBoundingVolume(a_point);
  };

  // Traverse the tree.
  m_bvh->traverse(updater, visiter, sorter, metaUpdater);

  return minDist;
```

CSG Union

Combinations of implicit functions in EBGeometry into aggregate objects can be done by means of CSG unions. One such union is known as the *smooth union*, in which the transition between two objects is gradual rather than abrupt. Below, we show the traversal code for this union, where we traverse through the tree and obtains the distance to the *two* closest objects rather than a single one.

Listing 3.3: Tree traversal when computing the smooth CSG union. See Source/EBGeometry_CSGImplem.hpp for details.

```
template <class T, class P, class BV, size_t K>
T
FastSmoothUnionIF<T, P, BV, K>::value(const Vec3T<T>& a_point) const noexcept
{
  // For the smoothed CSG union we use a smooth min operator on the two closest
  // primitives.

  // Closest and next closest primitives.
  T a = std::numeric_limits<T>::infinity();
  T b = std::numeric_limits<T>::infinity();

  BVH::Updater<P> updater =
    [&a, &b, &a_point](const std::vector<std::shared_ptr<const P>>& a_implicitFunctions) noexcept -> void {
    for (const auto& implicitFunction : a_implicitFunctions) {
      const auto& d = implicitFunction->value(a_point);

      if (d < a) {
        b = a;
        a = d;
      }
      else if (d < b) {
        b = d;
      }
    }
  };

  BVH::Visiter<Node, T> visiter = [&a, &b](const Node& a_node, const T& a_bvDist) noexcept -> bool {
    return a_bvDist <= 0.0 || a_bvDist <= a || a_bvDist <= b;
  };

  BVH::Sorter<Node, T, K> sorter =
    [&a_point](std::array<std::pair<std::shared_ptr<const Node>, T>, K>& a_leaves) noexcept -> void {
    std::sort(
      a_leaves.begin(),
      a_leaves.end(),
      [&a_point](const std::pair<std::shared_ptr<const Node>, T>& n1,
                 const std::pair<std::shared_ptr<const Node>, T>& n2) -> bool { return n1.second > n2.second; });
  };

  BVH::MetaUpdater<Node, T> metaUpdater = [&a_point](const Node& a_node) noexcept -> T {
    return a_node.getDistanceToBoundingVolume(a_point);
  };

  this->m_bvh->traverse(updater, visiter, sorter, metaUpdater);

  return m_smoothMin(a, b, m_smoothLen);
}
```

3.6 Octree

The octree functionality is encapsulated in the namespace EBGeometry::Octree. For the full API, see the doxygen API. Currently, only full octrees are supported (i.e., using a pointer-based representation).

Octrees are encapsulated by a class

```
template <typename Meta, typename Data = void>
class Node : public std::enable_shared_from_this<Node<Meta, Data>>
```

where the template parameters are:

- `Meta` Meta-information contained in the node (e.g. upper/lower corners).
- `Data` Data contained in the node.

`Node` describes both regular and leaf nodes in the octree.

> **Warning:** `Octree::Node<Meta, Data>` should only be used as `std::shared_ptr<Octree::Node<Meta, Data>>`.

3.6.1 Construction

Constructing the octree is done by first initializing the root node and then building it in either depth-first or breadth-first ordering.

```
  template <typename Meta, typename Data = void>
  class Node : public std::enable_shared_from_this<Node<Meta, Data>>
  {
    using StopFunction = std::function<bool(const Node<Meta, Data>& a_node)>;
    using MetaConstructor = std::function<Meta(const OctantIndex& a_index, const Meta& a_parentMeta)>;
    using DataConstructor =
      std::function<std::shared_ptr<Data>(const OctantIndex& a_index, const std::shared_ptr<Data>& a_parentData)>;
    using Updater = std::function<void(const Node<Meta, Data>& a_node)>;
    using Visiter = std::function<bool(const Node<Meta, Data>& a_node)>;
    using Sorter = std::function<void(std::array<std::shared_ptr<const Node<Meta, Data>>, 8>& a_children)>;

    inline void
    buildDepthFirst(const StopFunction&    a_stopFunction,
                    const MetaConstructor& a_metaConstructor,
                    const DataConstructor& a_dataConstructor) noexcept;

    inline void
    buildBreadthFirst(const StopFunction&    a_stopFunction,
                      const MetaConstructor& a_metaConstructor,
                      const DataConstructor& a_dataConstructor) noexcept;

  };
} // namespace Octree
```

The input functions to `buildDepthFirst` and `buildBreadthFirst` are as follows:

1. `StopFunction` determines if the node should be split or not. If it returns true, the node will *not* be split.
2. `MetaConstructor` constructs meta-data in the child nodes. This can/should include the physical corners of the node, but this is not a requirement.
3. `DataConstructor` constructs data in the child node. This can e.g. be a partitioning of the parent data.

3.6.2 Tree traversal

```
  template <typename Meta, typename Data = void>
  class Node : public std::enable_shared_from_this<Node<Meta, Data>>
  {
    inline void
    traverse(
      const Updater& a_updater,
      const Visiter& a_visiter,
      const Sorter&  a_sorter = [](std::array<std::shared_ptr<const Node<Meta, Data>>, 8>& a_children) -> void {
        return;
      }) const noexcept;
```

The input functions to `traverse` are as follows:

1. `Updater` executes a user-specified update rule when visiting a leaf node.
2. `Visiter` determines if the node should be visited during the traversal or not.

3. `Sorter` permits the user to sort the nodes in the current subtrees and visit them in a specified pattern. By default, no sorting is done and the nodes are visited in lexicographical order.

3.6.3 Example

An example of how to use the Octree functionality is given in `EBGeometry_ImplicitFunctionImplem.hpp` where the octree functionality is used for spatial partitioning of an implicit function. This includes both the octree construction and traversal.

3.7 Reading data

Routines for parsing surface grid from files 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 binary and ASCII STL files and reconstructing DCEL grids from those. However, it is also possible to build DCEL grids from polygon soups read using third-party codes (see *Using third-party sources*).

3.7.1 Quickstart

If you have one or multiple STL files, you can quickly turn them into signed distance fields using

```
std::vector<std::string> files; // <---- List of file names.

const auto distanceFields = EBGeometry::Parser::readIntoLinearBVH<float>(files);
```

This will build DCEL meshes for each input file, and wrap the meshes in BVHs. See *DCEL mesh SDF with compact BVH* for further details.

3.7.2 Reading STL files

`EBGeometry` supports a native parser for binary and ASCII STL files, which can be read into a few different representations:

1. Into a DCEL mesh, see *DCEL*.
2. Into a signed distance function representation of a DCEL mesh, see *Geometry representation*.
3. Into a signed distance function representation of a DCEL mesh, but using a BVH accelerator in full representation.
4. Into a signed distance function representation of a DCEL mesh, but using a BVH accelerator in compact representation.

DCEL representation

To read one or multiple STL files and turn it into DCEL meshes, use

```
/*!
  @brief Read a file containing a single watertight object and return it as a DCEL mesh
  @param[in] a_filename File name
*/
template <typename T, typename Meta = DCEL::DefaultMetaData>
inline static std::shared_ptr<EBGeometry::DCEL::MeshT<T, Meta>>
readIntoDCEL(const std::string a_filename) noexcept;

/*!
  @brief Read multiple files containing single watertight objects and return them as DCEL meshes
  @param[in] a_files File names
*/
template <typename T, typename Meta = DCEL::DefaultMetaData>
inline static std::vector<std::shared_ptr<EBGeometry::DCEL::MeshT<T, Meta>>>
readIntoDCEL(const std::vector<std::string> a_files) noexcept;
```

Note that this will only expose the DCEL mesh, but not include any signed distance functionality.

DCEL mesh SDF

To read one or multiple STL files and also turn it into signed distance representations, use

```
/*!
  @brief Read a file containing a single watertight object and return it as an implicit function.
  @param[in] a_filename File name
*/
template <typename T, typename Meta = DCEL::DefaultMetaData>
inline static std::shared_ptr<MeshSDF<T, Meta>>
readIntoMesh(const std::string a_filename) noexcept;

/*!
  @brief Read multiple files containing single watertight objects and return them as an implicit functions.
  @param[in] a_files File names
*/
template <typename T, typename Meta = DCEL::DefaultMetaData>
inline static std::vector<std::shared_ptr<MeshSDF<T, Meta>>>
readIntoMesh(const std::vector<std::string> a_files) noexcept;
```

DCEL mesh SDF with full BVH

To read one or multiple STL files and turn it into signed distance representations using a full BVH representation, use

```
/*!
  @brief Read a file containing a single watertight object and return it as a DCEL mesh enclosed in a full BVH.
  @param[in] a_filename File name
*/
template <typename T,
          typename Meta = DCEL::DefaultMetaData,
          typename BV   = EBGeometry::BoundingVolumes::AABBT<T>,
          size_t K      = 4>
inline static std::shared_ptr<FastMeshSDF<T, Meta, BV, K>>
readIntoFullBVH(const std::string a_filename) noexcept;

/*!
  @brief Read multiple files containing single watertight objects and return them as DCEL meshes enclosed in BVHs.
  @param[in] a_files File names
*/
template <typename T,
          typename Meta = DCEL::DefaultMetaData,
          typename BV   = EBGeometry::BoundingVolumes::AABBT<T>,
          size_t K      = 4>
inline static std::vector<std::shared_ptr<FastMeshSDF<T, Meta, BV, K>>>
readIntoFullBVH(const std::vector<std::string> a_files) noexcept;
```

DCEL mesh SDF with compact BVH

To read one or multiple STL files and turn it into signed distance representations using a compact BVH representation, use

```
/*!
  @brief Read a file containing a single watertight object and return it as a DCEL mesh enclosed in a linearized BVH
  @param[in] a_filename File name
*/
template <typename T,
          typename Meta = DCEL::DefaultMetaData,
          typename BV   = EBGeometry::BoundingVolumes::AABBT<T>,
          size_t K      = 4>
inline static std::shared_ptr<FastCompactMeshSDF<T, Meta, BV, K>>
readIntoLinearBVH(const std::string a_filename) noexcept;

/*!
  @brief Read multiple files containing single watertight objects and return them as DCEL meshes enclosed in linearized BVHs.
  @param[in] a_files File names
*/
template <typename T,
          typename Meta = DCEL::DefaultMetaData,
          typename BV   = EBGeometry::BoundingVolumes::AABBT<T>,
          size_t K      = 4>
inline static std::vector<std::shared_ptr<FastCompactMeshSDF<T, Meta, BV, K>>>
readIntoLinearBVH(const std::vector<std::string> a_files) noexcept;
```

3.7.3 From soups to DCEL

`EBGeometry` also supports the creation of DCEL grids from polygon soups, which can then later be turned into an SDF representation. 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 compress the triangle soup (get rid of duplicate vertices) and then construct the DCEL mesh:

```
/*!
  @brief Compress triangle soup (removes duplicate vertices)
  @param[out] a_vertices   Vertices
  @param[out] a_facets     STL facets
*/
template <typename T>
inline static void
compress(std::vector<EBGeometry::Vec3T<T>>& a_vertices, std::vector<std::vector<size_t>>& a_facets) noexcept;

/*!
  @brief Turn raw vertices into DCEL vertices.
  @param[out] a_mesh Output DCEL mesh.
  @param[in]  a_verticesRaw  Raw vertices
  @param[in]  a_facets       Facets
*/
template <typename T, typename Meta>
inline static void
soupToDCEL(EBGeometry::DCEL::MeshT<T, Meta>&        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 tie the remaining polygons into a DCEL mesh. This function will also compute the vertex and edge normal vectors.

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

3.7.4 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

EXAMPLES

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.

4.1 EBGeometry

`EBGeometry`-specified examples are given in `Examples/EBGeometry_<something>`. These examples display most of the `EBGeometry` functionality:

- Generating analytic implicit or signed distance functions.
- Representation of surface grids as signed distance functions.
- Using BVH-acceleration when combining multiple analytically defined implicit functions with CSG.
- Using BVH-acceleration when combining multiple surface grids functions with CSG.

Note that these examples do not provide any output for visualization.

4.2 AMReX

The AMReX examples are given in `Examples/AMReX_<something>`. These examples are intended to expose the same features as the `EBGeometry`-specific examples.

4.3 Chombo3

The Chombo-3 examples are given in `Examples/Chombo3_<something>`. These examples are intended to expose the same features as the `EBGeometry`-specific examples.

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.