EBGeometry

Jul 11, 2026

CONTENTS

EBGeometry is a header-only C++17 library for constructive solid geometry (CSG) with implicit functions. It can turn surface geometries into fast, queryable signed distance functions (SDFs), and manipulate them with CSG operations.

Main features:

- Turn surface meshes into SDFs, via a half-edge (DCEL) mesh representation or raw triangles.
- Fast SDF evaluation using bounding volume hierarchies (BVHs).
- Supports both pointer-based tree BVHs, and flattened SIMD-accelerated packed BVHs.
- A library of analytic signed distance functions and implicit functions (spheres, boxes, and more).
- Composable with transforms (translation, rotation, scaling, rounding, blending).
- BVH-accelerated constructive solid geometry (CSG): unions, intersections, differences, and smooth blends, of both meshes and analytic shapes.
- Readers for triangulated surface meshes in STL, PLY, OBJ, and VTK format.
- Drop-in precision-templated for flexible usage.
- No external dependencies – drop `EBGeometry.hpp` into any C++17 project and include it.

Important: This is the user documentation for EBGeometry.

- If you are looking for the source code, it is hosted on GitHub.
- The developer documentation is a separate Doxygen-generated API of EBGeometry.

CHAPTER

ONE

GETTING STARTED

1.1 Obtaining EBGeometry

Clone the repository from GitHub:

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

The core library is header-only and completely self-contained once cloned. However, the ready-to-run examples in `Examples/` read surface meshes from the common-3d-test-models collection, which is bundled as a git submodule (`common-3d-test-models/`) at the repository root. If you intend to run the bundled examples, clone with the submodule in one step instead:

```
git clone --recurse-submodules https://github.com/rmrsk/EBGeometry.git
```

If you already cloned without `--recurse-submodules`, fetch the submodule afterwards:

```
git submodule update --init --recursive
```

The meshes are then available as `.obj` files under `common-3d-test-models/data/`. Some mesh-based examples take a mesh path on the command line, resolved relative to the run directory (each example is run from its own source folder), for example:

```
./a.out ../../common-3d-test-models/data/armadillo.obj
```

Running an example with no argument falls back to a default mesh from the submodule, so the submodule must be checked out for the examples to run.

Note: The submodule is only needed for the bundled examples. The core library and your own applications do not depend on it.

1.2 Quickstart

EBGeometry is header-only: there is nothing to separately compile or link. The single entry-point header is `EBGeometry.hpp`, located at the repository root. Including it pulls in the entire library:

```
#include "EBGeometry.hpp"
```

Because the library is header-only, the fastest way to try it is to compile one of the bundled examples directly:

```
cd Examples/Shapes
g++ -O3 -std=c++17 -I../.. main.cpp && ./a.out
```

Every folder under `Examples/<something>` is a pure EBGeometry example: it has no third-party dependencies and can be compiled with the one-liner above (see *Overview*). Folders under `Integrations/<Platform>/<something>` couple EBGeometry to a third-party application code (AMReX, Chombo) and additionally require that platform to be installed – see *Integrations*.

See *Obtaining EBGeometry* for how to clone the repository (including the submodule needed to run the bundled examples with their default mesh), and *Building* for how to point your own build system at EBGeometry.

1.3 Building

How you point your own build system at EBGeometry depends on your toolchain. Pick the section below that matches your workflow: *CMake*, *GNU make*, or *Direct compilation*. All three methods expose the same configuration knobs — described in *Configuration options*.

On this page

1.3.1 CMake

EBGeometry can be consumed in two ways from CMake: by cloning the repository (or installing it) and using `add_subdirectory`, or by letting CMake fetch it automatically at configure time with `FetchContent`.

FetchContent (recommended for quick integration)

Add the following to your `CMakeLists.txt`:

```
cmake_minimum_required(VERSION 3.16)
project(MyProject CXX)

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

include(FetchContent)
FetchContent_Declare(
```

(continues on next page)

(continued from previous page)

```
  EBGeometry
  GIT_REPOSITORY https://github.com/rmrsk/EBGeometry.git
  GIT_TAG        main   # pin to a release tag in production
)
FetchContent_MakeAvailable(EBGeometry)

add_executable(my_program main.cpp)

# Link against the interface target - this adds the include path automatically
target_link_libraries(my_program PRIVATE EBGeometry::EBGeometry)
```

After cloning, CMake configures EBGeometry as an `INTERFACE` library target named `EBGeometry::EBGeometry`. Linking against it propagates the include directory to your target automatically.

Local installation

If you have a local clone, point CMake at it with `-DCMAKE_PREFIX_PATH` or by adding a call to `add_subdirectory`:

```
# Option A - add_subdirectory
add_subdirectory(/path/to/EBGeometry EBGeometry_build)
target_link_libraries(my_program PRIVATE EBGeometry::EBGeometry)

# Option B - manually set the include path
target_include_directories(my_program PRIVATE /path/to/EBGeometry)
```

Enabling SIMD in CMake

Pass architecture flags via `target_compile_options`:

```
# AVX + FMA (recommended for modern x86-64)
target_compile_options(my_program PRIVATE -mavx -mfma -msse4.1)

# AVX-512F, if your target machines are guaranteed to support it
target_compile_options(my_program PRIVATE -mavx512f -mavx2 -mavx -mfma -msse4.1)

# Or for maximum portability, auto-detect via march=native:
# target_compile_options(my_program PRIVATE -march=native)
```

To expose the SIMD level as a CMake option:

```
set(EBGEOMETRY_SIMD "avx" CACHE STRING "SIMD level: avx512 | avx | sse41 | none")
set_property(CACHE EBGEOMETRY_SIMD PROPERTY STRINGS avx512 avx sse41 none)

if(EBGEOMETRY_SIMD STREQUAL "avx512")
  target_compile_options(my_program PRIVATE -mavx512f -mavx2 -mavx -mfma -msse4.1)
elseif(EBGEOMETRY_SIMD STREQUAL "avx")
  target_compile_options(my_program PRIVATE -mavx -mfma -msse4.1)
elseif(EBGEOMETRY_SIMD STREQUAL "sse41")
  target_compile_options(my_program PRIVATE -msse4.1)
endif()
```

This is exactly what EBGeometry's own top-level `CMakeLists.txt` does for the `EBGeometry::EBGeometry` interface target; passing `-DEBGEOMETRY_SIMD=...` when configuring the top-level project (or a project that pulls it in via `add_subdirectory`) controls it directly. Configure and build:

```
cmake -B build -DEBGEOMETRY_SIMD=avx512
cmake --build build -j$(nproc)
```

See *Configuration options* for what SIMD acceleration means for EBGeometry.

Enabling assertions in CMake

```
option(EBGEOMETRY_ENABLE_ASSERTIONS "Enable EBGeometry runtime assertions" OFF)

if(EBGEOMETRY_ENABLE_ASSERTIONS)
  target_compile_definitions(my_program PRIVATE EBGEOMETRY_ENABLE_ASSERTIONS)
endif()
```

Then at configure time:

```
cmake -B build -DEBGEOMETRY_ENABLE_ASSERTIONS=ON -DCMAKE_BUILD_TYPE=Debug
cmake --build build
```

See *Configuration options* for assertion semantics and the recommended build-type/assertion matrix.

Tip: If you are building EBGeometry itself (rather than consuming it from another project) — for example to run its unit test suite or the bundled examples — the repository ships ready-made CMake presets that set these options for you. See *Running the unit tests locally*.

1.3.2 GNU make

A minimal `Makefile`

A minimal `Makefile` that compiles a single `main.cpp` against EBGeometry:

```
# Path to the root of the EBGeometry source tree
EBGEOMETRY_DIR := /path/to/EBGeometry

CXX      := g++
CXXFLAGS := -std=c++17 -O3 -mavx -mfma -msse4.1
CXXFLAGS += -I$(EBGEOMETRY_DIR)

# Uncomment to enable runtime assertions:
# CXXFLAGS += -DEBGEOMETRY_ENABLE_ASSERTIONS

TARGET   := my_program
SRCS     := main.cpp

$(TARGET): $(SRCS)
     $(CXX) $(CXXFLAGS) $^ -o $@

.PHONY: clean
clean:
     rm -f $(TARGET)
```

Because EBGeometry is header-only, there are no object files or static libraries to build; the `$(TARGET)` rule is the only build step required. Every example under `Examples/<something>` ships a `GNUmakefile` following this same pattern — see *Overview*.

Conditional SIMD selection

To choose the SIMD level at make-time rather than hard-coding it:

```
EBGEOMETRY_DIR := /path/to/EBGeometry

CXX      := g++
CXXFLAGS := -std=c++17 -O3 -I$(EBGEOMETRY_DIR)

# SIMD=avx (default), sse41, or none
SIMD ?= avx
ifeq ($(SIMD),avx)
  CXXFLAGS += -mavx -mfma -msse4.1
```

(continues on next page)

(continued from previous page)

```
else ifeq ($(SIMD),sse41)
  CXXFLAGS += -msse4.1
endif

# Assertions: make ASSERTIONS=1 to enable
ifeq ($(ASSERTIONS),1)
  CXXFLAGS += -DEBGEOMETRY_ENABLE_ASSERTIONS
endif

TARGET := my_program
$(TARGET): main.cpp
	$(CXX) $(CXXFLAGS) $< -o $@
```

Invoke with, for example:

```
make SIMD=avx                  # AVX + FMA (default)
make SIMD=sse41 ASSERTIONS=1   # SSE4.1, assertions on
make SIMD=none                 # scalar fallback
```

See *Configuration options* for what SIMD acceleration means for EBGeometry, and for the semantics of `EBGEOMETRY_ENABLE_ASSERTIONS`.

1.3.3 Direct compilation

EBGeometry is header-only, so the simplest way to use it is to point your compiler's include path directly at the repository and compile. There is no library to build or link against.

Minimal build

The only mandatory flag is an include path:

```
g++ -std=c++17 -O3 -I/path/to/EBGeometry main.cpp -o my_program
```

Replace `g++` with `clang++`, `icpx`, or another C++17-capable compiler as needed.

Note: `-O3` is strongly recommended. The innermost loops (BVH traversal, SIMD triangle evaluation, and SDF queries) contain tight `if constexpr` branches and inlined intrinsics that only collapse into efficient machine code with full optimisation enabled.

Enabling SIMD acceleration

EBGeometry detects the available SIMD instruction set at compile time using the standard pre-defined macros `__AVX512F__`, `__AVX__`, `__SSE4_1__`, and `__FMA__`. Pass the corresponding flags to expose the widest register set supported by your CPU:

Target ISA	GCC / Clang flag(s)
AVX-512F (recent server/HEDT CPUs)	`-mavx512f -mfma`
AVX + FMA (recommended on x86-64 since ~2013)	`-mavx -mfma`
SSE 4.1 (older or constrained targets)	`-msse4.1`
No SIMD (portable fallback)	*(none)*

A typical production build targeting a modern x86-64 workstation:

```
g++ -std=c++17 -O3 -mavx -mfma -msse4.1 \
    -I/path/to/EBGeometry \
    main.cpp -o my_program
```

The `-msse4.1` flag is subsumed by `-mavx` on GCC but is harmless to include. On Apple Silicon (M-series) no flag is needed — the library automatically uses the scalar path, which remains correct though not SIMD-vectorised.

Tip: Use `-march=native` to let the compiler choose every ISA extension supported by the build machine. This gives the fastest binary but produces a non-portable executable:

```
g++ -std=c++17 -O3 -march=native \
    -I/path/to/EBGeometry \
    main.cpp -o my_program
```

See *Configuration options* for what SIMD acceleration means for EBGeometry.

Enabling runtime assertions

EBGeometry ships an assertion macro `EBGEOMETRY_EXPECT(cond)` (defined in `Source/EBGeometry_Macros.hpp`, included automatically through the library), disabled by default. To activate it:

```
g++ -std=c++17 -O0 -g \
    -DEBGEOMETRY_ENABLE_ASSERTIONS \
    -I/path/to/EBGeometry \
    main.cpp -o my_program_debug
```

See *Configuration options* for assertion semantics, the diagnostic message format, and the recommended build-type/assertion matrix.

1.4 SIMD acceleration

EBGeometry can vectorise the innermost signed-distance computation for some primitives using compiler intrinsics rather than relying on the compiler to auto-vectorise scalar code. SIMD support is also included for BVH traversal itself, in `BVH::PackedBVH`. Everything else in the library is scalar code. However, the performance-critical parts of the BVH traversal are vectorized, and adding leaf-level vectorization support for new types of primitives is quite possible – it mostly requires storing the primitives in a structure-of-arrays (SoA) layout.

1.4.1 How it is enabled

EBGeometry detects the available SIMD instruction set at **compile time**, using the standard pre-defined compiler macros `__AVX512F__`, `__AVX__`, `__SSE4_1__`, and `__FMA__` — there is no runtime dispatch. Whichever of these macros your compiler flags define, that is the code path compiled in.

Target ISA	Compiler macro(s) required
AVX-512F (recent server/HEDT CPUs)	`__AVX512F__`
AVX + FMA (recommended on x86-64 since ~2013)	`__AVX__` and `__FMA__`
SSE 4.1 (older or constrained targets)	`__SSE4_1__`
No SIMD (portable fallback)	*(none of the above)*

See *Building* for the exact compiler/CMake/Makefile flags that define these macros for each of the three build methods.

1.4.2 Under the hood

The specific defaults this selects are the BVH branching factor `K`, the SIMD width `W` for the SoA layout of the leaf primitives, and data alignment. How to override these factors explicitly is an implementation detail, documented alongside the SIMD-supported classes in *SIMD-accelerated classes*.

1.5 Configuration options

This page documents the configuration knobs shared by all three build methods (*CMake*, *GNU make*, *Direct compilation*): floating-point precision, the target SIMD instruction set, and optional runtime assertions.

On this page

1.5.1 Floating-point precision

Every EBGeometry class and function is templated on a floating-point type `T`, almost always `float` or `double`. Precision is a **compile-time** choice, not a runtime one – there is no notion of a "current" precision inside the library itself. Pick `T` explicitly when instantiating a class or calling a function template:

```
auto sdfDouble = EBGeometry::Parser::readIntoTriangleBVH<double>("bunny.ply");
auto sdfFloat  = EBGeometry::Parser::readIntoTriangleBVH<float>("bunny.ply");
```

Consuming code (including every example under `Examples/`) typically reads precision from a preprocessor define so it can be overridden from the build system without editing source:

```
#ifndef EBGEOMETRY_PRECISION
#define EBGEOMETRY_PRECISION double
#endif

using T = EBGEOMETRY_PRECISION;
```

See *Building* for how to set `EBGEOMETRY_PRECISION` from each build method.

1.5.2 SIMD acceleration

See *SIMD acceleration* for a conceptual overview of SIMD acceleration in EBGeometry, *SIMD-accelerated classes* for exactly which classes it applies to and what it means for each, and *Building* for the compiler/CMake/Makefile flags that enable it for each build method.

1.5.3 Compile-time assertions (`static_assert`)

Alongside `EBGEOMETRY_EXPECT`, EBGeometry uses ordinary C++ `static_assert` throughout to enforce template-parameter invariants that are known at compile time – these are always active and cannot be disabled, unlike `EBGEOMETRY_EXPECT`. A violation fails the build with a compiler error rather than misbehaving, crashing, or aborting at runtime. Examples of what is guarded this way:

- The floating-point type `T` really is `float`/`double` (not, say, `int`), on essentially every class template.
- BVH branching factors and SoA widths are in range, e.g. `K >= 2`.
- Type constraints between template parameters.
- The SIMD data-alignment invariants described in *SIMD-accelerated classes*.

See the end of this page for an example combining `static_assert` with `EBGEOMETRY_EXPECT` in a custom class.

1.5.4 Runtime assertions (`EBGEOMETRY_EXPECT`)

`EBGEOMETRY_EXPECT(cond)` is the standard way to encode preconditions inside EBGeometry code. When compiling with assertions enabled, EBGeometry will emit an error when the condition is violated and print the file and corresponding line where the violation occurred.

When `EBGEOMETRY_ENABLE_ASSERTIONS` is **not** defined (the default):

```
#define EBGEOMETRY_EXPECT(cond) ((void)(cond))
```

The condition is evaluated (preventing unused-variable warnings) but the branch is absent from the generated code — a modern optimising compiler eliminates it entirely at `-O2` or higher.

When `EBGEOMETRY_ENABLE_ASSERTIONS` **is** defined:

```
EBGEOMETRY_EXPECT(a_normal.length() > T(0));
// On failure:
// EBGeometry assertion failed: (a_normal.length() > T(0))
//   file: EBGeometry_AnalyticDistanceFunctions.hpp
//   line: 98
//   function: PlaneSDF
```

The program prints the failing expression, file, line, and enclosing function name to `stderr`, then calls `std::abort()`. This produces a core dump (on POSIX systems) that can be loaded directly into a debugger.

Recommended workflow

Build type	Recommended flags
Development / testing	`-O0 -g -DEBGEOMETRY_ENABLE_ASSERTIONS`
CI / integration tests	`-O2 -g -DEBGEOMETRY_ENABLE_ASSERTIONS`
Production / benchmark	`-O3 -mavx -mfma` or `-O3 -march=native` *(no assertions)*

Warning: Assertions add measurable overhead inside hot BVH traversal and SDF evaluation loops. Enable them during development and testing; disable them (the default) for production builds.

Writing your own assertions

If you extend EBGeometry with new functionality classes, use `static_assert` to guard preconditions that are known at compile time (from the template parameters alone), and `EBGEOMETRY_EXPECT` to guard preconditions that can only be checked at runtime (from actual argument values). `EBGEOMETRY_EXPECT` is available in any translation unit that (directly or transitively) includes `EBGeometry_Macros.hpp`, which is pulled in automatically through `EBGeometry.hpp`. An example combining both is given below:

```
#include "EBGeometry.hpp"
#include <type_traits>

template <class T>
class MySDF : public EBGeometry::SignedDistanceFunction<T>
{
public:
  static_assert(std::is_floating_point_v<T>, "MySDF requires a floating-point type T");

  MySDF(T a_radius)
  {
    EBGEOMETRY_EXPECT(a_radius > T(0));

    m_radius = a_radius;
  }
};
```

CHAPTER

TWO

CONCEPTS

2.1 Geometry representations

2.1.1 Signed distance fields

The signed distance function is defined as a function $S : \mathbb{R}^3 \to \mathbb{R}$, 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.

Conceptually, every signed distance field in EBGeometry is just such a function S, mapping a query point $\mathbf{x}$ to a distance

$$d = S(\mathbf{x}).$$

Every kind of signed distance field described on the following pages – analytic shapes, surface meshes, and constructive solid geometry (CSG) combinations of either – implements exactly this mapping, which is why they can be freely combined and swapped for one another.

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 Half-edge meshes (DCEL)

2.2.1 Principle

EBGeometry uses a doubly-connected edge list (DCEL) structure – also known as a half-edge mesh – 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 knowledge of 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.

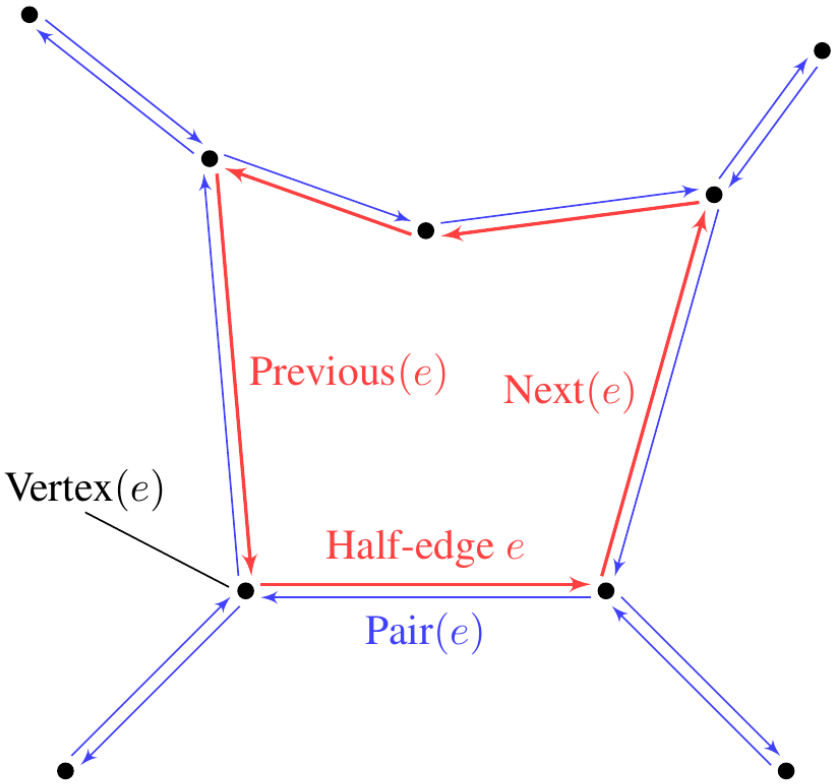

Fig. 2.1: DCEL mesh structure. Each half-edge stores references to next half-edge, the pair edge, and the starting vertex. Vertices store a coordinate as well as a reference to one of the outgoing half-edges.

While a DCEL mesh usually only requires storage of the edge structure, one can turn a DCEL mesh into a signed distance field by also storing normal vectors on each of the primitives, as indicated in the table below.

Concept	Stores
Vertex	Position, pseudonormal, one outgoing half-edge.
Half-edge	Owning face, next edge, pair edge, starting vertex, pseudonormal.
Facet	One boundary half-edge, face normal, a 2D embedding of the polygon, centroid.

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 nor the implicit function exists. If the mesh contains such deficiencies, or is self-intersecting, only the *unsigned* distance is well-defined.

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.

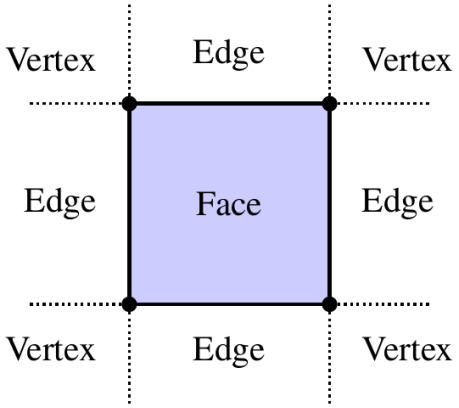

Fig. 2.2: Possible closest-feature cases after projecting a point $\mathbf{x}$ to the plane of a polygon face.

Three cases can be distinguished, and each has a direct counterpart in the DCEL structure: a vertex, a half-edge, and a facet can each be asked for both the signed distance $d = S(\mathbf{x})$ and the *squared unsigned* distance d^2 to a query point, where the latter avoids a square root when only the closest feature (not the exact distance to it) needs to be determined – this is the quantity used for comparisons when descending through a bounding volume hierarchy, see *Bounding volume hierarchies*.

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. The 2D embedding itself is computed once, at mesh-construction time, and reused for every subsequent query.

   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.

The mesh-level signed distance is then the minimum, in the unsigned-squared sense, over every facet's contribution, taking a final square root only once the closest facet has been identified. Scanning every facet this way works but has $\mathcal{O}(N)$ complexity for N facets; *Bounding volume hierarchies* describes how the query cost can be reduced dramatically by embedding the facets in a bounding volume hierarchy.

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.

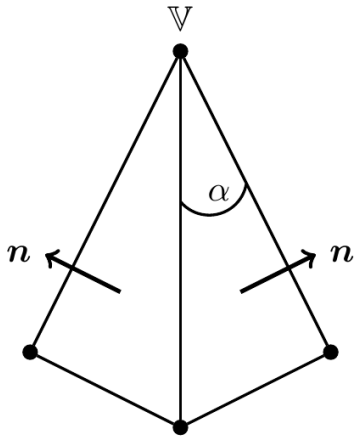

Fig. 2.3: Edge and vertex pseudonormals.

Both pseudonormals are computed once, at mesh construction time, and then simply stored and reused by every subsequent distance query.

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; EBGeometry ships an axis-aligned box and a bounding sphere, among others. 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.

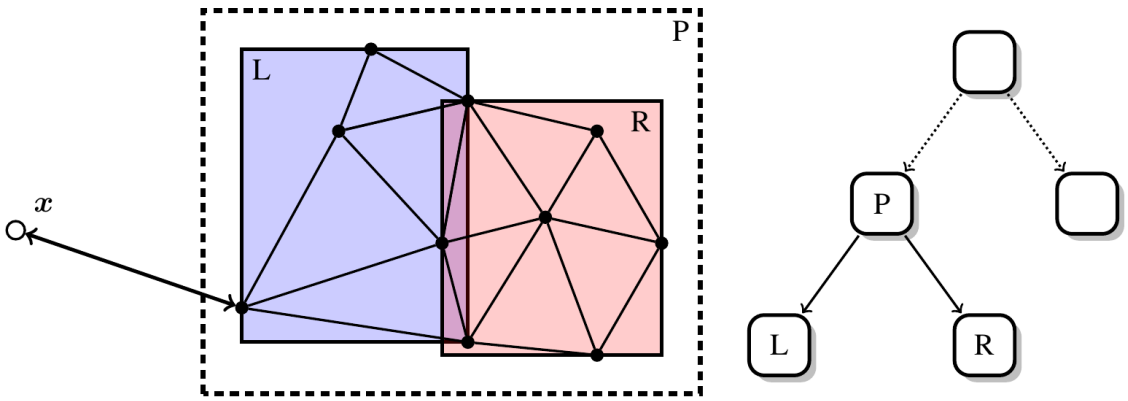

Fig. 2.4: Example of BVH partitioning for enclosing triangles. The regular node P contains two leaf nodes L and R which contain the primitives (triangles).

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 often done recursively, from top to bottom (so-called top-down construction). In this case, the BVH construction of a k-ary tree can be represented 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 the input list of objects/primitives, which is *partitioned* into k new lists of primitives. Note that the lists $\vec{O}_i$ do not contain duplicates; there is a unique set of primitives associated with 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 implements top-down construction using a user-supplied partitioning rule (the $\mathrm{Partition}$ above) and a termination criterion. Several ready-made partitioning strategies are provided, splitting on bounding-volume centroids (the default) or on primitive centroids; a more expensive strategy based on the Surface Area Heuristic typically yields the best query performance, at a higher construction cost.

Bottom-up construction is also possible, in which case one constructs the leaf nodes first, and then merges the nodes upward until one reaches a root node. In EBGeometry, bottom-up construction is done by means of space-filling curves – Morton codes, Hilbert curves, or nested indices. The Hilbert curve has better spatial locality than Morton (its consecutive codes are always spatially adjacent), so it tends to produce tighter leaf groupings.

Important: BVHs do not need to be stored with pointer referencing between nodes; it is quite possible to pack nodes tightly in memory on a linear array with direct indexing. This is useful when applying SIMD instructions for querying multiple bounding volumes simultaneously.

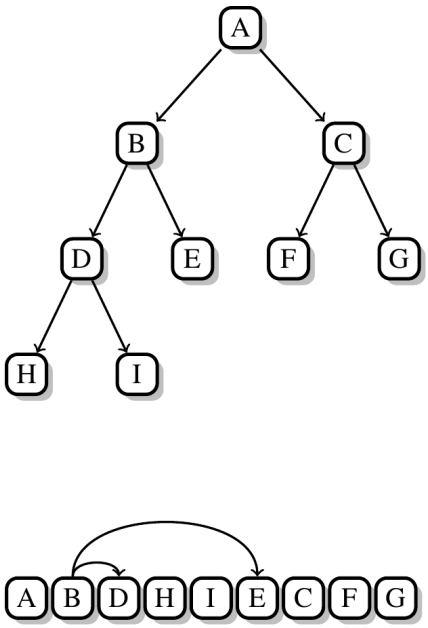

Fig. 2.5: A pointer-based tree (top) and an equivalent, flattened representation of the same tree (bottom). The tree is traversed top-to-bottom along its branches and laid out in that same order in a linear array; each interior node then stores index *offsets* to its children within the array, rather than pointers.

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. 2.6, 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 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 – it is often beneficial to use an SoA layout for the triangles so this calculation can be vectorized.
- 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, and there is never any need to compute the distance between $\mathbf{x}$ and the triangles in this subtree.

Other types of tree traversal, that do not compute a signed distance, are also possible. EBGeometry supports a fairly flexible approach to the tree traversal and update algorithms, so that the same hierarchical traversal-and-pruning idea can also be used for other operations – for instance, finding every facet within a specified distance of a point uses exactly the same principle.

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.

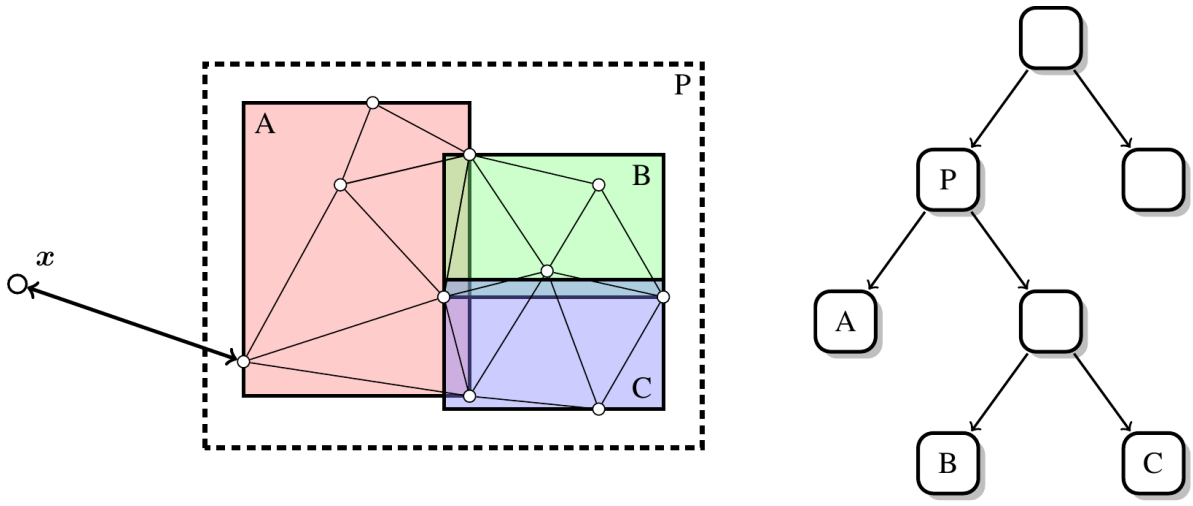

Fig. 2.6: Example of BVH tree pruning.

2.4 Point clouds

A *point cloud* is simply a set of points in space, with no connectivity, orientation, or inside/outside notion. Unlike a surface mesh, there is therefore no *signed* distance to a point cloud – only unsigned distances between points. The operations of interest are proximity queries:

- **Closest point.** Given an arbitrary query point, find the cloud point nearest to it.
- **k nearest.** Find the k closest cloud points to a query, in ascending order of distance.
- **k-nearest-neighbor graph.** For *every* point in the cloud, find its k nearest *other* points – the classic all-nearest-neighbors problem.

Answering one query by scanning all N points is $\mathcal{O}(N)$, so the all-nearest-neighbors graph is $\mathcal{O}(N^2)$ – infeasible for large clouds. As with mesh distance queries, the cost is reduced by a spatial acceleration structure that lets a query rule out the vast majority of points without ever measuring the distance to them. EBGeometry offers two such structures, built on two different ideas.

2.4.1 Hierarchical partitioning

The first idea is to build a tree over the points, recursively subdividing them into spatially tight groups – exactly the bounding volume hierarchy described in *Bounding volume hierarchies*, with the points (or small groups of them) as the primitives. A query then descends the tree, at each node visiting the nearer child first and *pruning* any subtree whose bounding volume is already farther than the best match found so far. Because the subdivision follows the points themselves, the tree adapts to the cloud's density – it stays balanced whether the points are uniform, lie on a surface, or clump into clusters – and a query touches only $\mathcal{O}(\log N)$ nodes on average.

2.4.2 Uniform grid

The second idea dispenses with the tree entirely and lays a single regular grid of fixed-size cells over the cloud, bucketing each point into the cell that contains it. A query starts in the cell containing the query point and searches outward one shell of cells at a time (Chebyshev radius 0, 1, 2, …). It can stop as soon as the best distance found so far is closer than the nearest edge of the next unvisited shell – no closer point can lie beyond that shell, so the search terminates *exactly*, never missing a neighbor. If the cells are sized to hold about one point each, a query resolves in the first shell or two.

The grid trades adaptivity for simplicity. A single global cell size fits a **near-uniform** cloud best – there, both building the grid (a counting sort) and querying it are cheaper than a tree. For a **strongly clustered or multi-scale** cloud a single cell size is a poor compromise (too coarse where the cloud is dense, too fine where it is sparse), and the density-adaptive hierarchy is the better choice.

2.4.3 Self queries

The all-nearest-neighbors graph is a special case worth calling out: every query point is *itself* a member of the cloud. A point therefore already knows which cell or leaf it lives in, so its search can be *seeded* from that group – giving a tight pruning bound immediately – and must exclude the point itself (otherwise it would trivially find itself at distance zero). This makes a batch of self queries strictly cheaper than the same number of independent external queries.

2.5 Octree

Octrees are tree-structures where each interior node has exactly eight children, each covering one octant of the parent node's spatial region. Such trees are usually used for spatial partitioning.

Unlike a bounding volume hierarchy (see *Bounding volume hierarchies*), an octree partitions *space* itself rather than a set of primitives: a node's eight children always tile the parent's region exactly, with no gaps and no overlap, regardless of what (if anything) is inside them. A BVH, by contrast, partitions whatever primitives it is given, and its children's bounding volumes may legitimately overlap.

A node is subdivided into its eight children only if some user-supplied *refinement criterion* says the node needs to be resolved further; otherwise it remains a leaf. This makes an octree's resolution adaptive: it can be made arbitrarily fine in regions that matter and left coarse everywhere else, unlike a uniform grid over the same domain. Fig. 2.7 shows this adaptivity in two dimensions (a quadtree, splitting each node into four children rather than eight): the root is refined into four children, and one of those children is refined a second time into its own four grandchildren, while the rest of the tree stops after one level.

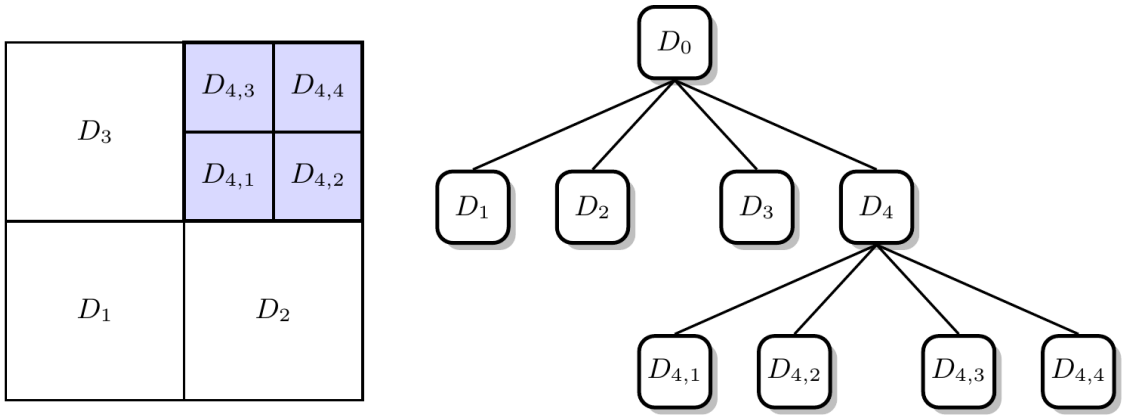

Fig. 2.7: A 2D illustration of octree-style spatial partitioning (a quadtree): the domain D is split into four sub-domains, one of which (D_4) is refined once more into four further sub-domains. The tree on the right is the exact counterpart of the partitioning on the left: each node in the tree is one region in the partitioning, and a node's children are exactly the sub-regions it was split into.

Octree traversal is, generally speaking, quite similar to the traversal algorithms used for BVH trees, see *Bounding volume hierarchies*: descending from a node visits some or all of its (up to eight) children, in whatever order and subject to whatever pruning rule the traversal is customized with, until the leaves relevant to the query at hand have been reached.

2.6 Constructive solid geometry

2.6.1 Basic transformations

Implicit functions, and by extension also signed distance fields, can be manipulated using basic transformations (like rotations). Every such transformation takes an implicit function I and produces a new implicit function I', defined by evaluating I at a suitably transformed query point.

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

EBGeometry supports many such transformations, for example:

Translation

Translating by a vector $\mathbf{t}$ shifts the query point back by $\mathbf{t}$ before evaluating the original function:

$$I'(\mathbf{x}) = I\left(\mathbf{x} - \mathbf{t}\right).$$

Rotation

Rotating by an angle θ about one of the coordinate axes $a \in \{x, y, z\}$ applies the inverse rotation to the query point:

$$I'(\mathbf{x}) = I\left(R_a(-\theta)\,\mathbf{x}\right),$$

where $R_a(\theta)$ is the standard rotation matrix by angle θ about axis a. Rotating the query point by $-\theta$ is what makes the *shape* itself appear rotated by $+\theta$.

Scaling

Uniform scaling by a non-zero factor s shrinks the query point by s before evaluating the original function, and scales the result back up by s:

$$I'(\mathbf{x}) = s\, I\left(\mathbf{x}/s\right).$$

Rescaling the value by s alongside the query point is what preserves the signed distance property (see *Geometry representations*) for a scaled *signed distance* function – omitting it would still shrink or grow the shape correctly, but the result would no longer report true distances.

Complement

The complement simply negates the function value, swapping the roles of "inside" and "outside":

$$I'(\mathbf{x}) = -I(\mathbf{x}).$$

Reflection

Reflecting across one of the three coordinate planes (the yz-, xz-, or xy-plane) flips the sign of the one coordinate normal to that plane (x, y, or z respectively) before evaluating the original function. Writing $\mathbf{r}$ for the vector that is $+1$ in the two unaffected coordinates and -1 in the flipped one, and $\odot$ for the component-wise product:

$$I'(\mathbf{x}) = I\left(\mathbf{r} \odot \mathbf{x}\right).$$

EBGeometry supports several further transformations beyond these five, including shell extraction (hollowing out a solid into a shell of a given thickness) and mollification (smoothing a sharp surface by locally averaging the function value), among others.

2.6.2 Combining objects

EBGeometry supports the standard operations for combining implicit functions: union, intersection, and difference. Smooth equivalents of each are also available, which smooth the transition between the combined objects (controlled by a blending length) instead of leaving a sharp crease where the objects meet.

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, using the same BVH traversal-and-pruning machinery described in *Bounding volume hierarchies* for mesh signed distances.

2.7 Triangle meshes

EBGeometry also supports representing a mesh directly as a collection of self-contained triangles, without storing the half-edge topology described in *Half-edge meshes (DCEL)*. Triangles in EBGeometry are actually *pseudo-triangles* that store all the vertex positions, as well as the face, vertex, and edge *pseudo-normals*, which are used when computing the signed distance to a triangle mesh.

Triangle meshes in EBGeometry have no internal pointer chasing; they are stored in raw form, so that groups of them can be packed together tightly in memory as a single block, covering several triangles at once. Doing so enables evaluating the signed distance to several triangles simultaneously, using vectorized (SIMD) instructions rather than one triangle at a time – see *SIMD acceleration*.

CHAPTER

THREE

IMPLEMENTATION

3.1 Overview

EBGeometry is a header-only C++17 library with zero external dependencies, implemented entirely under the `EBGeometry` namespace (with sub-namespaces `EBGeometry::DCEL`, `EBGeometry::BVH`, `EBGeometry::Octree`, `EBGeometry::SFC`, and `EBGeometry::Parser` for the major components). A handful of design choices recur throughout the implementation:

- **Precision templating.** Every class and function is templated on a floating-point type `T` (almost always `float` or `double`). Precision is a compile-time choice: the library has no notion of a "current" precision at runtime, and nothing dispatches on it dynamically.
- **A common interface for composability.** Every signed distance field and implicit function – analytic shapes, surface meshes, CSG combinations of either – derives from the same small polymorphic interface, `ImplicitFunction<T>` (with `SignedDistanceFunction<T>` as a refinement of it). Because every concrete type honors exactly the same interface, the transform and CSG combinators can wrap or combine *any* of them interchangeably through ordinary virtual dispatch, without needing to know which concrete type they are actually holding. See *Geometries*.
- **The same acceleration structure for two different problems.** Finding the closest facet in a surface mesh, and finding the closest object in a CSG union of many objects, are both, structurally, "find the closest of N things" queries. EBGeometry solves both with the same bounding volume hierarchy machinery, reducing an $\mathcal{O}(N)$ linear scan to an $\mathcal{O}(\log N)$ tree traversal in either case. See *BVH*.
- **Value types where topology permits it.** A DCEL mesh needs pointer-based topology (a half-edge, its pair, and its owning face are genuinely different objects referencing each other), so it uses `weak_ptr` back-references to avoid reference cycles. Where no topology is needed, however, EBGeometry prefers small, self-contained value types instead – a triangle with no half-edge structure, or a bounding-volume-hierarchy node stored by index offset rather than by pointer – since these can be packed tightly in memory and, where the data layout allows it, evaluated with SIMD instructions. See *DCEL* and *BVH*.
- **SIMD as an opt-in, compile-time-detected layer.** SIMD acceleration is implemented with hand-written compiler intrinsics in exactly two places, selected at compile time from the standard compiler-predefined ISA macros – never a runtime dispatch. See *SIMD-accelerated classes*.
- **Two complementary assertion mechanisms.** `static_assert` guards template-parameter invariants decidable at compile time; the opt-in `EBGEOMETRY_EXPECT` macro guards runtime preconditions on actual argument values. See *Configuration options*.

The remaining pages in this section cover each component in more detail:

- *Vector types* – the `Vec2T`/`Vec3T` vector types used throughout the library.
- *Geometries* – the `ImplicitFunction`/`SignedDistanceFunction` interface, the analytic shape library, transforms, and CSG combinators.

- *DCEL* – the half-edge (DCEL) surface mesh representation.
- *BVH* – bounding volume hierarchy construction, traversal, and the packed mesh/CSG signed distance function classes built on top of it.
- *Octree* – the octree implementation.
- *Reading data* – reading surface meshes from STL/PLY/OBJ/VTK files.
- *SIMD-accelerated classes* – the two SIMD-accelerated classes, and exactly what is vectorised in each.

3.2 Vector types

EBGeometry implements its own 2D and 3D vector types, `Vec2T` and `Vec3T`. Like every other class in the library, both are templated on a floating-point precision `T` (`float` or `double`), so a single header defines both precisions with no code duplication and no runtime precision switch.

`Vec2T` is a two-dimensional Cartesian vector. Most of EBGeometry is written as three-dimensional code, but `Vec2T` is needed for DCEL functionality when determining whether a point projects onto the interior or exterior of a planar polygon, see *Half-edge meshes (DCEL)*. Its main features are:

- Public `x`/`y` components, settable either directly or through a two-argument constructor.
- Named static factories for common vectors: `zeros()`, `ones()`, `min()`, `max()`, and `infinity()`.
- The usual arithmetic operators (+, -, unary -, scalar *//, and their in-place +=/-=/*=//= counterparts), plus a scalar-left multiplication and division as free functions.
- `dot`, `length`, and `length2` member functions, mirrored by free-function equivalents (`dot`, `length`) and free-function component-wise `min`/`max`.

`Vec3T` is a three-dimensional Cartesian vector, likewise templated on precision `T`. Unlike `Vec2T`, its components are stored internally and reached only through `operator[]` (mutable and const overloads), not as public named fields. Its main features are:

- Element access through `operator[]` (indices 0, 1, 2), plus a three-argument constructor.
- Named static factories: `zeros()`, `ones()`, `unit(dir)` (a basis vector along a given axis), `min()`, `max()`, and `infinity()`.
- The usual arithmetic operators, including both scalar and component-wise *//, and their in-place counterparts, plus a scalar-left multiplication/division as free functions.
- `cross` and `dot` products, mirrored by free-function equivalents, plus free-function component-wise `min`, `max`, and `clamp`.
- Full comparison support: ==, !=, componentwise </>/<=/>=, and a lexicographic `lessLX` for use in ordered containers.
- `minDir`/`maxDir` to find the index of the smallest/largest component (optionally by magnitude), and `length`/`length2`.
- A stream insertion operator (`operator`<<) for printing.

For the full API – every constructor, operator, and free function – see the Doxygen reference for Vec2T and Vec3T.

3.3 Geometries

This page covers the concrete classes behind two Concepts pages: *Geometry representations* (signed distance fields and implicit functions in general) and *Constructive solid geometry* (transformations and CSG combinators) – read those first for the conceptual picture, and use this page for the class/function-level detail and Doxygen links.

3.3.1 ImplicitFunction

`ImplicitFunction<T>` (`Source/EBGeometry_ImplicitFunction.hpp`) is the root of EBGeometry's polymorphic geometry hierarchy. Every concrete piece of geometry the library can represent – analytic shapes, DCEL/triangle-mesh distance fields, and CSG combinations of either – ultimately derives from it. It declares a single pure virtual member function,

$$\texttt{value}(\mathbf{x}) : \mathbb{R}^3 \to \mathbb{R},$$

returning a value that is negative for points inside the object and positive for points outside it, plus a call operator (`operator()`) that simply forwards to `value()`. Because every concrete geometry type honors exactly this same contract, code elsewhere in the library (and in user code) can hold a `shared_ptr<ImplicitFunction<T>>` and call `value()` on it through ordinary virtual dispatch without ever needing to know which concrete class it actually points to – this is what lets the transform and CSG machinery below wrap or combine *any* implicit function interchangeably.

`ImplicitFunction<T>` also provides one concrete (non-virtual) member function, `approximateBoundingVolumeOctree`, for shapes that have no closed-form bounding volume. It refines an octree over a caller-supplied initial box, marking a cell as intersecting the surface once the implicit function's value at the cell center falls within a safety-scaled margin of the cell's half-width, then builds a bounding volume of the caller-chosen type `BV` from the corners of the intersected leaf cells. This is the same octree-refinement idea described conceptually in *Octree*; see that page for how the subdivision itself works. The result is only meaningful when `value()` is reasonably close to a true signed distance, since the safety margin is interpreted in units of distance.

For the full API (including the exact parameters and defaults of `approximateBoundingVolumeOctree`), see the Doxygen page for ImplicitFunction.

3.3.2 SignedDistanceFunction

`SignedDistanceFunction<T>` (`Source/EBGeometry_SignedDistanceFunction.hpp`) inherits from `ImplicitFunction<T>` and refines its contract, without adding to the public `value()` interface itself: it implements `value()` (marked `final`) to delegate to a new pure virtual member function, `signedDistance(point)`, which subclasses must implement instead. The distinction is one of guarantee rather than signature – an arbitrary `ImplicitFunction<T>` only promises that the sign of its output indicates inside/outside, whereas a `SignedDistanceFunction<T>` additionally promises that the *magnitude* of its output is the true Euclidean distance to the surface (the Eikonal property, $|\nabla S| = 1$; see *Geometry representations* for why this property matters). Every analytic shape and mesh distance field shipped with EBGeometry is a `SignedDistanceFunction<T>`.

Because the true distance is available, `SignedDistanceFunction<T>` also provides a concrete `normal(point, delta)` member function that estimates the outward unit normal from central finite differences of `signedDistance` with step size `delta` – something that cannot be done reliably from an arbitrary implicit function's value alone.

For the full API, see the Doxygen page for SignedDistanceFunction.

Tip: Various ready-to-use implementations of both interfaces are declared in `Source/EBGeometry_AnalyticDistanceFunctions.hpp` (spheres, boxes, planes, cylinders, tori, and other closed-form primitives) and in `Source/EBGeometry_MeshDistanceFunctions.hpp` (the DCEL/triangle-mesh-backed classes `FlatMeshSDF`, `MeshSDF`, and `TriMeshSDF` – see *Mesh SDF classes*).

3.3.3 Transformations

EBGeometry implements every transformation as a small wrapper class that stores a `shared_ptr<ImplicitFunction<T>>` to the wrapped function together with the transformation's own parameters, and implements `value()` by applying the (typically inverse) transformation to the query point before evaluating the wrapped function. Since every such wrapper is itself an `ImplicitFunction<T>`, transformations compose freely – wrapping a wrapper is just as valid as wrapping a leaf shape. Each wrapper class has a same-named free function that constructs it and returns it already cast to `shared_ptr<ImplicitFunction<T>>`, ready to be passed into further transformations or CSG combinators without the caller ever naming the concrete wrapper type. Both are declared in `Source/EBGeometry_Transform.hpp`.

The five simplest transformations – translation, rotation, scaling, the complement, and reflection – are described mathematically in *Constructive solid geometry*; the table below just maps each to its wrapper class, free function, and Doxygen entry:

Transformation	Wrapper class	Free function	Doxygen
Translation	`TranslateIF`	`Translate`	class / function
Rotation	`RotateIF`	`Rotate`	class / function
Scaling	`ScaleIF`	`Scale`	class / function
Complement	`ComplementIF`	`Complement`	class / function
Reflection	`ReflectIF`	`Reflect`	class / function

EBGeometry implements several further transformations that have no closed-form counterpart in *Constructive solid geometry* and are only described here:

- **Offset** (`OffsetIF`/`Offset`) subtracts a constant from the wrapped function's value, growing the object if the offset is positive and shrinking it if negative. For a true signed distance function this simply moves the isosurface outward or inward by the offset distance.
- **Annular** (`AnnularIF`/`Annular`) turns a solid into a hollow shell of total thickness 2δ by evaluating $|I(\mathbf{x})| - \delta$: the two new zero-isosurfaces sit where $I(\mathbf{x}) = \pm\delta$, i.e. the original surface offset inward and outward by δ each, hollowing out everything in between.
- **Blur** (`BlurIF`/`Blur`) smooths sharp features by passing the wrapped function through a 3D box filter sampled at the face centers, edge centers, and corners of a cube of half-width equal to the blur distance, with per-sample weights controlled by a blend factor `alpha` (`1` = no blur, `0` = maximal blur).
- **Mollify** (`MollifyIF`/`Mollify`) is a more general smoothing operation: it convolves the wrapped function with a caller-supplied mollifier implicit function (typically a small sphere SDF) sampled on a uniform grid of offsets, normalizing the sample weights to sum to one.
- **Elongate** (`ElongateIF`/`Elongate`) stretches a shape along one or more axes without changing its cross-section: the query point is clamped component-wise to $[-\mathbf{e}, \mathbf{e}]$ for a per-axis elongation vector $\mathbf{e}$, and the clamped offset is subtracted from the point before evaluating the wrapped function.

Transformation	Wrapper class	Free function	Doxygen
Offset	`OffsetIF`	`Offset`	class / function
Annular (shell)	`AnnularIF`	`Annular`	class / function
Blur	`BlurIF`	`Blur`	class / function
Mollification	`MollifyIF`	`Mollify`	class / function
Elongation	`ElongateIF`	`Elongate`	class / function

Warning: Not every transformation preserves the signed distance property. Rotation, translation, and the complement always do; scaling does provided the value is rescaled alongside the query point (which `ScaleIF` does);

> offset, annular, blur, mollification, and elongation generally do not produce an exact signed distance even when the input is one.

Every wrapper class and free function above is declared in `Source/EBGeometry_Transform.hpp`; see the Doxygen page for the file itself, EBGeometry_Transform.hpp, for the complete, single-page API listing.

3.3.4 CSG

The Boolean combinators for combining multiple implicit functions into one – union, intersection, difference, and their smooth-blended variants – are described conceptually in *Constructive solid geometry*. EBGeometry implements each with the same wrapper-class pattern as the transformations above: a class stores the combined implicit functions and implements `value()` as the appropriate combination over them, with a matching free function returning it as a `shared_ptr<ImplicitFunction<T>>`. All of the following are declared in `Source/EBGeometry_CSG.hpp`:

Combinator	Wrapper class	Free function	Doxygen
Union	`UnionIF`	`Union`	class / function
Smooth union	`SmoothUnionIF`	`SmoothUnion`	class / function
Intersection	`IntersectionIF`	`Intersection`	class / function
Smooth intersection	`SmoothIntersectionIF`	`SmoothIntersection`	class / function
Difference	`DifferenceIF`	`Difference`	class / function
Smooth difference	`SmoothDifferenceIF`	`SmoothDifference`	class / function

`Union`, `SmoothUnion`, `Intersection`, and `SmoothIntersection` each have two overloads in the Doxygen listing above – one taking a `std::vector` of any number of implicit functions, and one taking exactly two – both constructing the same underlying wrapper class.

The "smooth" combinators blend the transition between objects instead of leaving a sharp crease, using a caller-replaceable blending functor rather than a plain `min`/`max`. Two are provided as `std::function` template variables in `Source/EBGeometry_CSG.hpp`: `SmoothMin<T>` (a cheap polynomial smooth-minimum, the default for `SmoothUnion`) and `SmoothMax<T>` (its symmetric counterpart, the default for both `SmoothIntersection` and `SmoothDifference` – difference is implemented internally as the intersection of `A` with the complement of `B`, which is why it defaults to the same operator as intersection rather than to `SmoothMin<T>`), plus a more expensive exponential alternative, `ExpMin<T>`. Any of the three – or a user-supplied functor of the same signature, `T(const T&, const T&, const T&)` – can be passed as the smoothing operator to the smooth combinators' constructors/free functions.

Because a plain CSG union is evaluated as $\min(I_1, \ldots, I_N)$, querying it costs $\mathcal{O}(N)$ per point for N objects. `BVHUnionIF`/`BVHUnion` and `BVHSmoothUnionIF`/`BVHSmoothUnion` accelerate this by placing the objects' bounding volumes in a `PackedBVH` and reducing a closest-object query to an $\mathcal{O}(\log N)$ tree traversal instead of a linear scan – see *BVH* for how the BVH itself is built and traversed; this section only concerns how CSG uses it. Unlike the plain combinators, the BVH variants take the objects' bounding volumes explicitly (a `std::vector<BV>`, one per object) since these are needed up front to build the tree.

Combinator	Wrapper class	Free function	Doxygen
BVH-accelerated union	`BVHUnionIF`	`BVHUnion`	class / function
BVH-accelerated smooth union	`BVHSmoothUnionIF`	`BVHSmoothUnion`	class / function

Finally, `FiniteRepetitionIF`/`FiniteRepetition` tiles a single base implicit function periodically over a finite number of repetitions per axis, by mapping the query point into the nearest tile before evaluating the wrapped function – effectively a cheap way to instance the same shape many times without constructing a separate `ImplicitFunction<T>` (or a CSG union) per copy. See its Doxygen entries for FiniteRepetitionIF and FiniteRepetition.

For the complete, single-page API listing of every class and free function on this page, see the Doxygen page for EBGeometry_CSG.hpp.

3.4 DCEL

See *Half-edge meshes (DCEL)* for the conceptual picture of half-edge meshes (what a DCEL is, and why signed distance queries need one) before reading the concrete API below.

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 one 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 classes templated on both a floating-point precision `T` and a user-defined meta-data type `Meta` attached to each instance:

- `VertexT<T, Meta>` stores the vertex position, its (outward) normal vector, and the outgoing half-edge from the vertex, plus pointers to every polygon face sharing that vertex. It also has member functions for computing the vertex pseudonormal, see *Normal vectors*. For the full API, see the Doxygen reference for VertexT.
- `EdgeT<T, Meta>` represents a half-edge: it stores a reference to its owning face, and pointers to the next edge, its pair edge, and its starting vertex. For the full API, see the Doxygen reference for EdgeT.
- `FaceT<T, Meta>` represents a polygon face. Besides its half-edge, it also stores the face normal vector, a 2D embedding of the polygon, and its centroid position: the normal and 2D embedding exist because the signed distance computation needs them, and the centroid exists because BVH partitioners use it when partitioning the surface mesh. For the full API, see the Doxygen reference for FaceT.
- `MeshT<T, Meta>` stores an entire DCEL mesh – all of its vertices, half-edges, and faces – and provides brute-force ($\mathcal{O}(N)$) distance queries, `signedDistance()` and `unsignedDistance2()`, that scan every face directly. It is not itself a `SignedDistanceFunction<T>`: for anything beyond small meshes, one instead wraps a `MeshT<T, Meta>` in one of the BVH-accelerated classes described in *Mesh SDF classes*, which hold a `shared_ptr<MeshT<T, Meta>>` internally and only fall back to it for topology (not for the accelerated distance queries themselves). A mesh is typically never constructed by hand – it is built by a file parser reading vertices and faces from disk, see *Reading data*. For the full API, see the Doxygen reference for MeshT.

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

3.4.2 BVH integration

A `MeshT<T, Meta>` is never queried directly for anything beyond tiny meshes – see *Bounding volume hierarchies* for why an $\mathcal{O}(N)$ scan over faces doesn't scale, and *BVH* for how `TreeBVH`/`PackedBVH` are actually built and traversed. This section covers only the DCEL-specific half of that integration: how a mesh's faces become BVH primitives in the first place.

Embedding a mesh in a BVH is a matter of pairing each `FaceT<T, Meta>` with a bounding volume and handing the resulting list to a `TreeBVH`. Concretely, `MeshDistanceFunctionsDetail::buildDCELTreeBVH<T, Meta, BV, K>` (in `Source/EBGeometry_MeshDistanceFunctionsImplem.hpp`, the shared helper behind both `MeshSDF` and `TriMeshSDF`'s construction) does this by:

1. Building each face's bounding volume `BV` directly from its vertex coordinates (`FaceT::getAllVertexCoordinates()`) – this is why `FaceT` stores its vertices' positions accessibly, rather than requiring a caller to walk its half-edges to collect them.
2. Constructing a `TreeBVH<T, FaceT<T, Meta>, BV, K>` from the resulting `(face, bounding volume)` pairs.
3. Partitioning that tree according to the requested `BVH::Build` strategy (`TopDown`, `Morton`, `Nested`, or `SAH` – see *Construction*), where the `BVCentroidPartitioner`/`BinnedSAHPartitioner` used by the default and SAH strategies consult `FaceT::getCentroid()` (see above) when deciding how to split a set of faces.

`MeshSDF` then packs this `TreeBVH` into a `PackedBVH` of faces directly (`pack()`), while `TriMeshSDF` additionally converts each face into a triangle and groups triangles into SIMD-width `TriangleSoAT` blocks while packing (`packWith()`) – see *Mesh SDF classes* for how the two differ, and *Reading data* for the file-reading entry points that produce a `MeshSDF`/`TriMeshSDF` from a mesh file directly, without driving any of the steps above by hand.

3.5 BVH

See *Bounding volume hierarchies* for the conceptual picture of bounding volume hierarchies (node types, partitioning, tree pruning during traversal) before reading the concrete API below.

The BVH functionality is encapsulated in the namespace `EBGeometry::BVH` (`Source/EBGeometry_BVH.hpp` / `EBGeometry_BVHImplem.hpp`). For the full API, see the doxygen API. There are two representations of a BVH:

- `BVH::TreeBVH<T, P, BV, K>`, a pointer-based tree used while *building* and *partitioning* the hierarchy. See the doxygen page for TreeBVH.
- `BVH::PackedBVH<T, P, K, StoragePolicy>`, where the nodes are stored in depth-first order in a flat array and contain index offsets to children and primitives rather than pointers. This is the representation used for fast queries, see *PackedBVH*. See the doxygen page for PackedBVH.

The template parameters shared by both are:

- `T` Floating-point precision.
- `P` Primitive type. Neither representation imposes an interface requirement of its own: `TreeBVH` construction/partitioning only ever calls `getCentroid()` on it, and `PackedBVH` holds primitives opaquely, handing them back only to whatever leaf-visit callback a caller supplies to `traverse()` or `pruneTraverse()` (see below). Whether `P` needs a `signedDistance(Vec3T<T>)` member (or anything else) is entirely up to that callback – see `MeshSDF`/`TriMeshSDF::signedDistance()` in *Mesh SDF classes* for the signed-distance case; a callback could equally perform, say, a nearest-neighbor search over a point cloud whose primitive carries no `signedDistance()` at all.
- `BV` Bounding volume type (`TreeBVH` only — `PackedBVH` always uses `BoundingVolumes::AABBT<T>` internally).
- `K` BVH degree. `K=2` will yield a binary tree, `K=3` yields a tertiary tree and so on.

TreeBVH is the BVH builder node, i.e. it is the class through which we recursively build the BVH, see *Construction*. TreeBVH has no constraint that P be wrapped in a shared_ptr itself, but the *containers* that hold primitives during construction (PrimitiveList<P>, PrimAndBV<P, BV>) always wrap each primitive in a std::shared_ptr<const P>, so that primitives can be safely shared between the tree and any higher-level object that still refers to them by pointer (e.g. a DCEL::FaceT).

3.5.1 Bounding volumes

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

- **Bounding sphere**, templated as EBGeometry::BoundingVolumes::SphereT<T>. Various constructors are available. See the doxygen page for SphereT.
- **Axis-aligned bounding box**, which is templated as EBGeometry::BoundingVolumes::AABBT<T>. See the doxygen page for AABBT.

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. Note that PackedBVH hard-codes AABBT<T> as its bounding volume (see above), so a custom bounding volume can only be used with TreeBVH while building, and must still be convertible to an AABB before the tree is packed.

3.5.2 Construction

Building a TreeBVH always starts from a list of (primitive, bounding volume) pairs; at that point the tree is a single, unpartitioned leaf holding every primitive. Partitioning it into an actual hierarchy is then a separate step, done by calling one of the partitioning member functions described below – this recursively subdivides the tree in place.

Tip: The default construction methods perform 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 member function topDownSortAndPartition(), which takes two optional arguments: a *partitioner* and a *leaf predicate*.

The partitioner is a functor that splits a list of (primitive, BV) pairs into K new lists whenever a leaf is subdivided. Four ready-made partitioners are provided: BVCentroidPartitioner (splits on bounding-volume centroids along the longest axis – the default), PrimitiveCentroidPartitioner (the same idea, but splits on primitive centroids instead), BinnedSAHPartitioner (a Surface-Area-Heuristic partitioner, used automatically when building via BVH::Build::SAH – see below – and typically producing the best-performing trees at a higher construction cost), and MidpointPartitioner (splits on the midpoint of the bounding-volume centroids' extent along the longest axis, with a single std::partition pass – no sorting and no per-plane cost evaluation, making it the fastest of the four to build, at the cost of not adapting to the primitive distribution the way the other three do). BinnedSAHPartitioner and MidpointPartitioner both produce K groups by recursively splitting into two (std::floor(K/2) and std::ceil(K/2)) halves, exact for power-of-two K. BinnedSAHPartitioner takes an optional final template argument LongestAxisOnly (default false); setting it true bins candidate planes on only the longest centroid-bounding-box axis instead of all three, cutting roughly a third of the binning work (measured ~20% faster SAH builds on point clouds) for a tree-quality cost that is negligible on near-uniform inputs. The leaf predicate takes a TreeBVH node and decides whether it should become a leaf (i.e. not be split any further); a default is provided, but callers are free to supply their own of either kind.

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 template `bottomUpSortAndPartition<S>()` (no arguments). The template argument is the space-filling curve that the user wants to apply, from namespace `EBGeometry::SFC` (`Source/EBGeometry_SFC.hpp`, see the doxygen API). Currently, we support Morton codes, Hilbert curves, and nested indices. For Morton curves, one would e.g. call `bottomUpSortAndPartition<SFC::Morton>`; the Hilbert curve (better spatial locality than Morton, since consecutive codes are always spatially adjacent) is `bottomUpSortAndPartition<SFC::Hilbert>`; 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 it tends to produce worse trees such that traversal becomes slower. For the full API, see the Doxygen reference for TreeBVH.

Direct construction (no TreeBVH)

Both construction methods above build a `TreeBVH` first, then require a separate `pack()`/ `packWith()` call to obtain a `PackedBVH`. For workloads with many small, cheaply-copyable primitives (points, particles) built and rebuilt often, the per-node `shared_ptr<TreeBVH>` allocation this implies can dominate build time. `PackedBVH` has a constructor that skips `TreeBVH` entirely:

```
BVH::PackedBVH<T, P, K> packed(std::move(primsAndBVs), targetLeafSize);
```

It takes primitives **by value** (`std::vector<std::pair<P, BV>>`, a sink parameter the caller can `std::move` in) rather than requiring a `shared_ptr`-wrapped list, regardless of this `PackedBVH`'s `StoragePolicy` (see *PackedBVH*'s "Storage policy" section) — combined with `BVH::ValueStorage<P>`, this is genuinely pointer-free from input to final storage.

Internally, this constructor:

1. Sorts primitives along a space-filling curve (`SFC::Morton` by default; pass e.g. `SFC::Hilbert{}` or `SFC::Nested{}` as an optional trailing argument to select another curve — a constructor template's own parameters can't be explicitly named the way a regular function template's can, so this is a stateless tag value purely to let the curve type be deduced).
2. Cuts leaves via a single linear left-to-right scan at a caller-chosen **target leaf size**, rather than deriving a leaf count purely from primitive count and `K` the way `bottomUpSortAndPartition()` does — giving direct control over leaf occupancy.
3. Merges the resulting leaves upward in groups of `K`, padding the leaf count up to the next power of `K` (by reusing the last real leaf's node in place of any missing child, rather than inventing an empty placeholder) whenever it isn't already one, so every interior node still has exactly `K` children — no change to `Node`'s shape or to `traverse()`/`pruneTraverse()`.

Since this still produces an ordinary `PackedBVH`, every existing traversal/query facility (`traverse()`, `pruneTraverse()`, the SIMD dispatch) works with it identically, unchanged.

`PackedBVH` also has a second, overloaded direct constructor covering top-down (and SAH) construction rather than the SFC-based one above:

```
BVH::PackedBVH<T, P, K> packed(std::move(primsAndBVs));                                    // top-down
BVH::PackedBVH<T, P, K> packed(std::move(primsAndBVs), BVH::BinnedSAHPartitioner<T, P, AABBT<T>, K>, stopCrit); // SAH
```

It reuses `TreeBVH`'s own `Partitioner`/`LeafPredicate` machinery unchanged (any of `BVCentroidPartitioner`, `BinnedSAHPartitioner`, `PrimitiveCentroidPartitioner`, or a caller-supplied one), so it accepts the same arguments `topDownSortAndPartition()` does — but writes nodes directly into the flat node array in depth-first pre-order as the recursion unwinds, rather than building a persistent, `shared_ptr`-linked `TreeBVH` first. Since

top-down recursion visits the root before its children, this needs no relayout pass (unlike the SFC-build constructor above, where a bottom-up merge naturally produces the root last). Each split still shared_ptr-wraps primitives once, up front (to reuse the existing `Partitioner`/`LeafPredicate` signatures) and constructs one lightweight, stack-local `TreeBVH` per split purely to evaluate the stop criterion and read off its primitive list — proportionate to what `topDownSortAndPartition()` already does at every node, and immediately discarded rather than kept alive as part of a persistent tree. What this constructor avoids is exactly the *persistent* `shared_ptr<TreeBVH>` node allocation kept alive for the tree's lifetime, which the `Examples/BuildBVH` benchmark measures as the traditional path's dominant build-time cost.

A third direct constructor builds via **ClusterSAH**, a fast approximation of a full SAH tree:

```
BVH::PackedBVH<T, P, K> packed(std::move(primsAndBVs), BVH::ClusterSpec{maxClusterSize});
```

It first groups the primitives into small, spatially-tight *clusters* (buckets of at most `maxClusterSize` primitives, formed by a cheap density-adaptive midpoint subdivision that stops early), then runs binned SAH top-down over those clusters — so SAH partitions roughly `N / maxClusterSize` boxes instead of all `N` primitives. The result is near-SAH tree quality at a fraction of the single-threaded SAH build cost, and it stays robust across uniform, surface, and clustered primitive distributions (a fixed Cartesian grid, by contrast, overcrowds on non-uniform data). `BVH::ClusterSpec::maxClusterSize` trades build time (larger → fewer, cheaper SAH units) against query quality (larger → coarser leaves); `Examples/BuildBVH` benchmarks its build time against the other strategies.

Tip: Higher-level entry points such as `Parser::readIntoPackedBVH` don't require you to call `topDownSortAndPartition/bottomUpSortAndPartition` directly — they take a single `BVH::Build` enum value (`TopDown`, `Morton`, `Nested`, or `SAH`) and dispatch to the corresponding construction method internally. See *Reading data*.

3.5.3 PackedBVH

In addition to the standard BVH node `TreeBVH<T, P, BV, K>`, EBGeometry provides a `PackedBVH` where nodes are stored in depth-first order in a flat array. The `PackedBVH` can be automatically constructed from a `TreeBVH` but not vice versa.

The rationale for the `PackedBVH` is its tighter memory footprint and depth-first ordering, which allows more efficient traversal, particularly when primitives are sorted in the same order. `PackedBVH<T, P, K, StoragePolicy>` is templated on the same `T`, `P`, `K` as `TreeBVH` (its bounding volume is always `AABBT<T>`), plus a fourth `StoragePolicy` parameter described below, and internally stores two things: a flat, depth-first array of nodes, and a global primitive array holding every primitive in leaf order.

Each entry of the node array plays the same role a `TreeBVH` node plays, but stores offsets into the flat arrays rather than pointers to children: a bounding volume for the node's subtree, a primitive offset and count identifying its range in the global primitive array (used only if the node is a leaf), and the depth-first indices of its `K` children. A node is a leaf exactly when its primitive count is non-zero; the root node is always at index 0 of the node array. See the doxygen page for PackedBVH::Node for the exact member list.

Constructing a `PackedBVH` is simply a matter of flattening an already-partitioned `TreeBVH`, via one of two `TreeBVH` member functions:

- `pack()` performs a straight flatten: the resulting `PackedBVH<T, P, K>` stores exactly the same primitive type `P` that the source tree held. The original tree is left untouched and can simply be discarded (or allowed to go out of scope) once it is no longer needed.
- `packWith<Q, Converter>()` additionally *converts* the primitive type while flattening: the source tree holds primitives of type `P`, and a user-supplied `Converter` is called once per leaf to produce the `Q` values that the resulting `PackedBVH<T, Q, K>` will store. This is how `TriMeshSDF` turns a tree of individual `Triangle<T,`

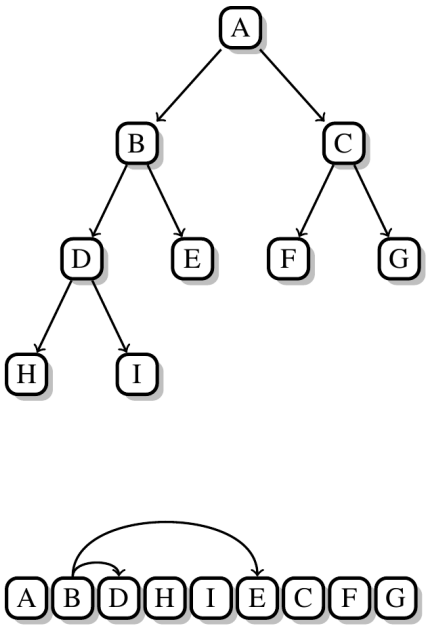

Fig. 3.1: PackedBVH representation. The original BVH is traversed from top-to-bottom along the branches and laid out in linear memory. Each interior node stores index offsets to its children and primitives.

Meta> primitives into a PackedBVH whose leaves hold SIMD-width TriangleSoAT<T, W> groups instead — see *Mesh SDF classes*.

See the doxygen page for TreeBVH for the exact signatures of pack() and packWith().

Storage policy

pack()/packWith() both take an optional StoragePolicy template argument controlling *how* the resulting PackedBVH's global primitive array stores each primitive. A storage policy is a stateless struct exposing a StorageType type alias plus the static methods get(), appendTreeLeaf(), and appendAliased() that PackedBVH calls internally when packing and querying; see the BVH namespace's doxygen page for the exact member list each one must provide. Two are provided out of the box:

- BVH::SharedPtrStorage<P> (the default for both pack() and packWith()) stores each primitive as a std::shared_ptr<const P>, matching the pre-existing behavior of every caller that does not name a StoragePolicy explicitly. Primitives are shared with whatever else still holds a pointer to them (e.g. a DCEL::FaceT referenced by a TreeBVH), at the cost of one pointer indirection per access during traversal.
- BVH::ValueStorage<P> stores each primitive inline, by value: StorageType is P itself, with no indirection at all. This trades away sharing (the PackedBVH now owns an independent copy of every primitive) for a smaller, more cache-friendly primitive array — most useful when P is a small, cheaply-copyable value type (e.g. a point or particle) rather than something large or already reference-counted elsewhere.

Both policies are drop-in compatible with every existing PackedBVH consumer: swapping the policy only changes the element type of getPrimitives() and the leaf-primitive vector handed to LeafEvaluator/PackedLeafEvaluator callbacks, never the tree structure, traversal order, or query results. A caller that wants ValueStorage instead of the default simply names it explicitly, e.g. tree->pack<BVH::ValueStorage<P>>().

Copy and move semantics

`TreeBVH` and `PackedBVH` differ in whether copying is allowed, precisely because of the storage-sharing question above:

- `TreeBVH` deletes its copy constructor and copy assignment operator. It is a recursive structure of `shared_ptr`-linked children, so a naive (compiler-generated) copy would only alias the same child subtrees rather than cloning them, which `topDownSortAndPartition()`/`bottomUpSortAndPartition()` could then mutate out from under a supposedly independent "copy". Copying is disallowed outright rather than silently doing the wrong thing. Its move constructor and move assignment operator are explicitly defaulted and fully supported. To *replicate* a tree independently – e.g. to build once and then partition two copies with different strategies, or to keep a pristine copy alongside one you go on to mutate – use `deepCopy()`, which recursively clones the node hierarchy (returning a new `std::shared_ptr<TreeBVH>`) while still sharing the immutable `std::shared_ptr<const P>` primitives by handle. (Copying a `std::shared_ptr<TreeBVH>` is, of course, always fine – that is shared ownership of the *same* tree, not a replica.)
- `PackedBVH` allows both copying and moving. Its members (the flattened node array, the primitive array, and the SIMD AABB cache) are all owned value containers with no shared mutable substructure, so the compiler-generated deep copy is correct and safe under both `BVH::SharedPtrStorage` (primitives are aliased `shared_ptr`, the same sharing model as `TreeBVH`) and `BVH::ValueStorage` (primitives are copied by value – safe as long as the primitive type's own copy constructor is complete; see the note on `DCEL::FaceT` in *Mesh SDF classes* for a case where it deliberately is not).

Both classes' destructors are non-virtual: neither is intended to be subclassed or used polymorphically.

When `ValueStorage` is the wrong choice

There are two situations where `ValueStorage` should not be used and `SharedPtrStorage` (the default) must be kept:

- **Polymorphic primitives.** `ValueStorage<P>` stores `P` by value, so it requires `P` to be a concrete, copyable value type. If `P` is an abstract base (or you rely on virtual dispatch through a base pointer), value storage either fails to compile or slices the object to its static type. This is exactly the situation of the BVH-accelerated CSG unions (*Geometries*), whose primitive is `ImplicitFunction<T>` and whose leaf evaluator calls a virtual `value()` through a `std::shared_ptr<const ImplicitFunction<T>>`; those classes therefore always use `SharedPtrStorage` and do not expose the policy. Use `ValueStorage` only when `P` is a self-contained value type (a point, a particle, an SoA triangle group), never for a polymorphic hierarchy.
- **Nesting a BVH inside a BVH.** A `PackedBVH` whose primitive is itself another `PackedBVH` (or a mesh SDF that owns one) is a supported construction, and nothing stops it recursing further — `PackedBVH` of `PackedBVH` of `PackedBVH`, to any depth. At every level the *outer* `PackedBVH` should stay on `SharedPtrStorage` so it shares each inner BVH by pointer. Naming `ValueStorage` on an outer level instead copies every inner `PackedBVH` wholesale — its node, SoA, and primitive arrays — into the outer array: packing draws each primitive from the source tree as a `std::shared_ptr<const P>`, so each inner BVH is *copied* (not moved) into place, even though `PackedBVH` is otherwise fully movable (see *Copy and move semantics* above). The cost of that copy is proportional to the *entire* memory footprint reachable below the primitive, so it becomes exceedingly expensive whenever an inner BVH is large, and compounds with nesting depth: each by-value level duplicates everything beneath it, which may itself be duplicating everything beneath *it*. `SharedPtrStorage` at every outer level avoids this for no loss of correctness. The common realisation of nesting — a `BVHUnion` over several mesh SDFs, each holding its own inner `PackedBVH` — is exactly the polymorphic-primitive case above, so it already sits on `SharedPtrStorage` at the outer level and shares each mesh SDF by pointer.

3.5.4 Tree traversal

Both `TreeBVH` (full BVH) and `PackedBVH` (flattened BVH) include routines for traversing the BVH with user-specified criteria. For both BVH representations, tree traversal is done through a single `traverse()` member function taking four caller-supplied callbacks, and uses a stack-based traversal pattern driven by those callbacks. The full type signatures for the four callback roles below are documented on the BVH namespace's doxygen page (look for `PrunePredicate`, `ChildOrderer`/ `PackedChildOrderer`, `LeafEvaluator`/`PackedLeafEvaluator`, and `NodeKeyFactory`).

Node visit

The *prune-predicate* callback decides, for a given node and its associated node key (see below), whether that node's subtree should be investigated or pruned from the traversal: it is a predicate taking the node and its node key, returning `true` to visit the subtree and `false` to prune it. Typically, the node key will contain the necessary information that determines whether or not to visit the subtree.

Child ordering

If a subtree is visited in the traversal, there is a question of which of the child nodes to visit first. The *child-orderer* callback determines this order by letting the user sort the `K` children (each paired with its node key) in-place based on order of importance – for `PackedBVH` the children are identified by their node index rather than a pointer, halving the per-entry stack size relative to `TreeBVH`. Note that a correct visitation pattern can yield large performance benefits. Ordering the child nodes is completely optional; the user can leave this function empty if it does not matter which subtrees are visited first.

Leaf evaluation

If a leaf node is visited in the traversal, distance or other types of queries to the geometric primitive(s) in the node are usually made. These are done by the *leaf-evaluator* callback. For `PackedBVH` this callback receives an offset and count into the BVH's global primitive array, rather than a freshly-allocated sub-list, avoiding a heap allocation per leaf visit; for `TreeBVH` it receives the leaf's primitive list directly. Typically, the leaf-evaluator will modify parameters that appear in a local scope outside of the tree traversal (e.g. updating the minimum distance to a DCEL mesh).

Node key

During the traversal, it might be necessary to compute a per-node key that is helpful during the traversal, and this key is attached to each node that is queried. This key is usually, but not necessarily, equal to the distance to the nodes' bounding volumes. The *node-key-factory* callback produces this key for a node's children, given the node itself. The biggest difference between the leaf-evaluator and the node-key-factory is that the leaf-evaluator is *only* called on leaf nodes whereas the node-key-factory is also called for internal nodes. One typical example for DCEL meshes is that the leaf-evaluator computes the distance from an input point to the triangles in a leaf node, whereas the node-key-factory computes the distance from the input point to the bounding volumes of a child node. This information is then used by the child-orderer in order to determine a preferred child visit pattern when descending along subtrees.

Traversal algorithm

`PackedBVH::traverse()` implements this with a non-recursive, vector-backed stack rather than recursion. Each stack entry holds a node index together with that node's already-computed node key. The root is pushed first; then, until the stack is empty, the traversal pops an entry, asks the prune-predicate whether to visit it, and if so either runs the leaf-evaluator (if it is a leaf) or computes the node-key-factory for each of its `K` children, lets the child-orderer reorder them, and pushes them all onto the stack. For the full API, see the Doxygen reference for PackedBVH.

Distance-pruned traversal: `pruneTraverse()`

`PackedBVH` has no `signedDistance()` of its own, and does not privilege any one query. Alongside the generic `traverse()` described above, it exposes a second traversal, `PackedBVH::pruneTraverse()`, that implements the same "closest bounding volume first, prune anything already known to be farther than the best answer so far" strategy, but with two differences: the box-vs-point distance test is SIMD-batched across all `K` children at once (see *SIMD-accelerated classes* for exactly which instructions run for which `(K, T)`, and the list-table below for the ISA-to-`K` mapping), and – unlike `traverse()`'s four independent callbacks – the search is expressed through exactly three cooperating pieces supplied by the caller:

- **State** – whatever the search needs to remember between leaf visits. Often just "the best value found so far", but it can be richer (e.g. a running best paired with the primitive that produced it). This is the only thing that persists across the whole traversal.
- **Leaf-eval** – called once per leaf, with the leaf's offset and count into the BVH's global primitive array (never a freshly-allocated sub-list). It is the *only* place primitives are actually touched, and the only place `State` is allowed to change.
- **Pruning rule** – called on the *current* `State` to produce a squared-distance bound: a node farther than this (in squared distance) is skipped without being visited. It never touches primitives directly, only whatever `Leaf-eval` has already written into `State`.

Precisely, the traversal seeds a stack with the root node (at distance zero, so it is never pruned), then repeatedly pops the top entry and: skips it outright if its already-known squared distance to the query point exceeds the pruning rule's *current* bound; otherwise, if it is a leaf, calls leaf-eval once for the whole leaf; otherwise (an interior node), computes all `K` children's squared distances to the query point in a single SIMD batch, sorts them so the closest child is visited next, and pushes every child still within the (freshly re-evaluated) pruning bound. Because the bound is re-read from the current `State` at every node visited – never cached from the start of the traversal – a leaf visited anywhere earlier on the stack immediately tightens the pruning applied to every node visited afterwards, regardless of which subtree it came from.

Splitting the pruning rule apart from the leaf-eval like this is what lets a primitive with no notion of "signed distance" reuse the same SIMD box test: a nearest-neighbor search over a point cloud can track a plain running squared distance as its `State` (no `abs()`, no extra squaring, no square root anywhere in the hot path) with a pruning rule that returns the state unchanged, whereas `MeshSDF`/`TriMeshSDF::signedDistance()` (see *Mesh SDF classes*) track a signed distance and square its magnitude for the bound – both are ordinary instantiations of the same `pruneTraverse()`, not special cases hardcoded into `PackedBVH`. The former needs nothing more than a bare point struct with no `signedDistance()` member at all, searched for its nearest neighbor via `pruneTraverse()` against a running squared distance.

For the exact template signature and callback contracts, see the doxygen page for PackedBVH::pruneTraverse.

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.

`MeshSDF::signedDistance()` implements these rules directly as the four traversal callbacks: the leaf-evaluator scans a leaf's faces and keeps the signed distance with the smallest magnitude seen so far; the prune-predicate prunes any node whose bounding-volume distance already exceeds that magnitude; the child-orderer visits the closest child first; and the node-key-factory supplies each node's distance to its bounding volume. For the full API, see the Doxygen reference for MeshSDF.

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.

`BVHSmoothUnionIF::value()` traverses the tree while tracking the two smallest values seen so far, `a` and `b` (`a` the closest, `b` the second-closest): the leaf-evaluator updates both as leaves are scanned, the prune-predicate prunes any node whose bounding-volume distance exceeds both, the child-orderer visits the closest child first, and the node-key-factory again supplies each node's distance to its bounding volume. Once traversal completes, the two values are blended with the stored smooth-minimum operator. See *Geometries* for the CSG combinators themselves, and the Doxygen reference for BVHSmoothUnionIF / BVHUnionIF for the exact API.

3.5.5 Mesh SDF classes

EBGeometry provides three concrete classes for evaluating signed distances to surface meshes. They share the same sign convention (negative inside, positive outside) but differ in data layout, BVH type, and supported geometry:

Table 3.1: Mesh SDF classes

Class	Input	BVH type	Traversal	Notes
`FlatMeshSDF<T, Meta>`	DCEL mesh	None	O(N) scan	Debug / tiny meshes only; no build cost
`MeshSDF<T, Meta, K>`	DCEL mesh	`PackedBVH` over `DCEL::FaceT` (always `BVH::SharedPtrStorage`)	`pruneTraverse()` (SIMD when `(K, P)` matches a compiled ISA path)	Any polygon mesh; not restricted to triangles
`TriMeshSDF<T, Meta, K, W, StoragePolicy>`	DCEL mesh or triangle soup	`PackedBVH` over SoA triangle groups (`BVH::ValueStorage` by default)	`pruneTraverse()` over SoA-packed leaves	Triangle meshes only; highest throughput

FlatMeshSDF is useful for correctness checks and tiny meshes. See its doxygen page.

MeshSDF handles arbitrary polygon meshes; its signedDistance() builds the traversal criteria shown above (a leaf-eval and a pruning rule, not the full four-callback traverse() shape) and drives them through PackedBVH::pruneTraverse(), picking up SIMD node pruning whenever (K, T) matches a compiled ISA path and falling back to the generic, scalar traverse() otherwise. See its doxygen page.

TriMeshSDF is the recommended default for triangle meshes: it packs triangles into Structure-of-Arrays groups of width W (via TreeBVH::packWith(), see above) and builds the same kind of thin pruneTraverse() wrapper as MeshSDF, over TriangleSoAT<T, W> leaves instead of individual faces, so that on a matching (K, T) combination each BVH leaf evaluates W triangles with a single SIMD register operation, and even the AABB-vs-running-best comparisons during descent are done on squared distances (no square root) until the very last step. See its doxygen page, and the doxygen page for TriangleSoAT for the SoA storage itself.

What is actually vectorised in TriMeshSDF/PackedBVH is covered in *SIMD-accelerated classes* – see that page for the full detail rather than repeating it here.

Primitive storage: Facets or triangles

Both classes' underlying PackedBVH accepts the StoragePolicy axis described above, but they default – and, for MeshSDF, are restricted – differently:

- MeshSDF always uses BVH::SharedPtrStorage<DCEL::FaceT<T, Meta>> and does not expose a StoragePolicy template parameter at all. This is not merely a default: DCEL::FaceT's copy constructor deliberately does *not* copy its cached 2D polygon embedding (used by signedDistance()'s point-in-face test) or its inside/outside algorithm choice, since FaceT's half-edge back-reference is only topologically meaningful relative to a specific mesh (the same reasoning documented for VertexT/EdgeT's copy constructors). A BVH::ValueStorage-style plain copy would therefore leave every packed face with an uninitialized embedding, crashing on the first query rather than merely losing sharing – so MeshSDF simply never offers that choice.
- TriMeshSDF defaults to BVH::ValueStorage<TriangleSoAT<T, W>> instead of BVH::SharedPtrStorage, and *does* expose StoragePolicy as an overridable template parameter (its 5th, after W). Unlike DCEL::FaceT, each TriangleSoAT<T, W> group is a plain aggregate of coordinate arrays with no cached derived state or back-references, built fresh by groupTrianglesIntoSoA() during packing and shared with nothing else – so storing it inline is both safe and, avoiding one heap allocation and pointer indirection per group, the better default.

Neither default is affected by instancing the same mesh multiple times (e.g. placing several Translate/Rotate/Scale-wrapped copies of one mesh into a Union): those wrappers hold a shared_ptr to the whole MeshSDF/TriMeshSDF object (see *Geometries*), so its packed data is shared once per wrapper regardless of how that one object's own PackedBVH stores its primitives internally.

Parser::readIntoPackedBVH mirrors MeshSDF (no StoragePolicy parameter); Parser::readIntoTriangleBVH mirrors TriMeshSDF (StoragePolicy as its 5th template parameter, defaulting to BVH::ValueStorage<TriangleSoAT<T, W>>). See *Reading data*.

SIMD-optimal K and W by ISA

The helper BVH::DefaultBranchingRatio<T>() returns the SIMD-optimal branching factor for the current compilation target. EBGeometry::TriangleSoA::DefaultWidth<T>() gives the matching SoA width. Both are used as template defaults for TriMeshSDF and Parser::readIntoTriangleBVH.

Table 3.2: Default K and W by ISA and precision

ISA	Precision	`DefaultBranchingRatio<T>()`	`TriangleSoA::DefaultWidth<T>()`
AVX-512F	`float`	16	16
AVX-512F	`double`	8	8
AVX (256-bit)	`float`	8	8
AVX (256-bit)	`double`	4	4
SSE4.1 / scalar	`float` or `double`	4	4

The K=16/float and K=8/double paths use 512-bit-wide SIMD loads and require the `ChildAABBSoA` struct to be 64-byte aligned, which is guaranteed by `alignas(sizeof(T)*K)` on the struct. The K=8/float and K=4/double paths use 256-bit-wide loads instead. All other (K, T) combinations fall back to a scalar loop that goes through the generic `traverse()` described above.

Each `TriangleSoAT<T, W>` block is likewise `alignas`-aligned to its own SIMD register width (64 bytes for `<float, 16>`/`<double, 8>`, 32 bytes for `<float, 8>`, 16 bytes for `<float, 4>`/`<double, 4>`), and the library inserts `static_assert` checks that fire at compile time if the alignment invariant is violated.

Choosing W and K explicitly

`W` and the BVH branching factor `K` are explicit template parameters on `TriMeshSDF` and `Parser::readIntoTriangleBVH` – both default to `BVH::DefaultBranchingRatio<T>()` and `TriangleSoA::DefaultWidth<T>()` respectively, but either can be overridden by supplying them explicitly (e.g. requesting an 8-wide SoA packing together with a 4-ary BVH, regardless of what the current compilation target would otherwise default to). See the doxygen page for Parser::readIntoTriangleBVH for the exact signature.

Rules of thumb:

- Keep `W` equal to `EBGeometry::TriangleSoA::DefaultWidth<T>()` unless you have a specific reason to deviate. The library is tuned for this default.
- `a_maxLeafSize` (the maximum number of raw triangles per BVH leaf, before SoA packing) defaults to `2 * W`: leaves land on up to two full SoA blocks, while the SAH/TopDown partitioner is still free to split down to smaller, tighter leaves wherever the geometry calls for it. A leaf smaller than `W` simply pads its SoA block's unused lanes.
- `K = BVH::DefaultBranchingRatio<T>()` is a good default. With AVX-512F available you can try `K = 16` (float) — the child-AABB test is evaluated in a single SIMD batch, and the wider fan-out reduces tree depth.

3.6 Point clouds

EBGeometry provides two turnkey classes for nearest-neighbor and closest-point work over a point cloud: `PointCloudBVH` (`Source/EBGeometry_PointCloudBVH.hpp`) and `PointCloudHashGrid` (`Source/EBGeometry_PointCloudHashGrid.hpp`). They answer the same queries and expose the **same public interface** – the same `Hit` result type, the same `closestPoint` / `closestPoints` / `nearestNeighbor` / `nearestNeighbors` / `allNearestNeighbors` query methods, the same `O(N)` brute-force reference queries, and the same `position()` / `metadata()` accessors – so the two are drop-in interchangeable. They differ only in the spatial acceleration structure they build: a hierarchical tree, or a uniform grid. See *Point clouds* for the conceptual picture and the trade-off between the two.

Both are built directly from a raw cloud – point positions plus a parallel array of user metadata – and both return the matched point's **cloud index** (its position in the input arrays) together with the squared distance; the user metadata is reachable through `metadata()`. `closestPoint` / `closestPoints` answer an arbitrary external query point, while `nearestNeighbor` / `nearestNeighbors` (and the batch `allNearestNeighbors`) answer a point already in the cloud,

excluding it from its own result and seeding the search from the group it lives in – a strictly cheaper search an external point cannot use (see *Point clouds*).

Each accelerated query also has an `O(N)` brute-force counterpart – `closestPointBruteForce` / `closestPointsBruteForce` / `nearestNeighborBruteForce` / `nearestNeighborsBruteForce` – that answers the same question by a full linear scan. These are reference implementations for testing and debugging (verify an accelerated result against ground truth, or A/B-test a suspected bug against an unaccelerated path) and are not meant for production queries.

3.6.1 PointCloudBVH

`PointCloudBVH` specializes the *BVH* machinery: it is a subclass of `BVH::PackedBVH` with two things the general path does not offer – a much cheaper **index-based build**, and **turnkey query methods** that hide `pruneTraverse()` entirely. It is built by partitioning an index permutation in place with a longest-axis midpoint split and packing the `PointAoSoA` leaves inline (no intermediate primitive list, no `shared_ptr`, no separate packing pass), which is several times faster to build than a full Surface-Area-Heuristic tree and, for near-uniform clouds, just as tight to query. Because it is a BVH, it can also be composed as a primitive inside an outer BVH or CSG tree.

See the PointCloudBVH doxygen page for the full interface, and `Examples/ClosestPointBVH` / `Examples/NearestNeighborBVH` for worked usage.

3.6.2 PointCloudHashGrid

`PointCloudHashGrid` circumvents the tree entirely and stores the cloud in a **uniform grid**. Points are counting-sorted into a dense array of fixed-size cells (a CSR bucket array keyed by integer cell coordinates) – an `O(N)` build with no recursive partitioning and no tree nodes. A query is an **expanding-shell** search outward from the query point's cell (Chebyshev radius 0, 1, 2, …); it stops, exactly and without ever missing a neighbor, as soon as the k-th best distance found is closer than any unvisited cell can hold. With the default cell size (~1 point/cell) that is almost always one or two shells.

For a near-uniform cloud the grid both builds and queries faster than the BVH; for a strongly clustered or multi-scale cloud a single global cell size is a poor fit and `PointCloudBVH` is the better choice. The grid is also bounded-domain (dense cells sized to the bounding box, `O(N)` memory for a compact cloud) and serves only point queries; unlike `PointCloudBVH` it cannot be composed as a primitive inside an outer BVH/CSG.

See the PointCloudHashGrid doxygen page for the full interface, and `Examples/NearestNeighborHashGrid` for a worked comparison against the BVH example.

3.7 Octree

The octree functionality is encapsulated in the namespace `EBGeometry::Octree`; see *Octree* for the conceptual picture (space partitioning, adaptivity) before reading the concrete API below. For the full API, see the doxygen API. Currently, only full octrees are supported (a pointer-based representation, not a linear/pointerless one).

Octrees are encapsulated by a class template `Octree::Node<Meta, Data>`, where the template parameters are:

- `Meta` – meta-information stored in every node (e.g. the node's physical corners).
- `Data` – payload data stored at leaf nodes.

`Node` describes both regular and leaf nodes in the octree; a node with no children is a leaf.

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

See the Doxygen reference for Node for the full member list.

3.7.1 Construction

An octree is built by first default-constructing a (leaf) root node, and then calling either `buildDepthFirst` or `buildBreadthFirst` on it, which refines the tree in depth-first or breadth-first order respectively. Both take the same three user-supplied callables:

1. `SplitFunction` – a predicate `bool(const Node&)` called on each leaf node; returning `true` subdivides that leaf into eight children, `false` leaves it as-is.
2. `MetaConstructor` – constructs a child's `Meta` from its parent's `Meta` and its octant index. This is typically where the child's physical corners are computed from the parent's, but nothing requires that – `Meta` can hold whatever the refinement criterion needs to decide whether to split further.
3. `DataConstructor` – constructs a child's `Data` from its parent's `Data` and its octant index (e.g. partitioning the parent's data set among the eight children).

Refinement proceeds top-down: starting from the root, every leaf for which `SplitFunction` returns `true` is subdivided into eight children (using `MetaConstructor`/`DataConstructor` to populate them), and the process repeats on the new leaves. There is no separate "maximum depth" parameter built into `Node` itself – if a bounded depth is wanted, `SplitFunction` must encode that itself (e.g. by having `Meta` carry the current level and refusing to split past some level), exactly as *the bounding-volume estimator below* does.

3.7.2 Tree traversal

Tree traversal is done through the member function `traverse`, which visits nodes top-down using a prune-order-evaluate pattern analogous to the BVH traversal described in *BVH*. The input functions to `traverse` are as follows:

1. `PrunePredicate` – a predicate `bool(const Node&)` called on every node (interior or leaf); returning `false` prunes that entire subtree from the traversal.
2. `ChildOrderer` – reorders a node's (up to) eight children in-place before they are visited, so the traversal can, e.g., visit the closest child first. By default, no sorting is done and children are visited in lexicographical octant order.
3. `LeafEvaluator` – called on every *leaf* node that `PrunePredicate` did not prune; this is where the caller actually consumes the leaf (e.g. to accumulate a result).

3.7.3 Estimating a bounding volume for an implicit function

The octree machinery above is used directly by ImplicitFunction::approximateBoundingVolumeOctree, which estimates a bounding volume of type `BV` for an implicit function I that has no closed-form bound – for example, one built up from several nested CSG operations (see *Constructive solid geometry*), where no simple formula for a bounding box or sphere is available.

Given an initial search box and a maximum tree depth, the algorithm is:

1. Each node's meta-data records its two physical corners, its depth, and a boolean flag for whether it might contain the implicit surface.

2. A node is flagged as possibly containing the surface if

$$|I(\mathbf{x}_c)| \leq (1+\sigma)\,|\Delta\mathbf{x}|\,,$$

where $\mathbf{x}_c$ is the node's center, $\Delta\mathbf{x}$ is its half-diagonal, and $\sigma \geq 0$ is a user-supplied safety factor. This is a direct consequence of the Eikonal property (see *Geometry representations*): since I changes by at most the distance moved, evaluating it at a single point bounds how far away the surface can be from that point, so a node whose center is farther from the surface than the node itself extends cannot contain it.

3. `SplitFunction` subdivides exactly the flagged nodes, and only up to the given maximum depth – this is the "bounded refinement" mentioned above, implemented entirely inside the `SplitFunction`/`MetaConstructor` pair rather than by any feature of `Node` itself. The tree is built with `buildBreadthFirst`.
4. The tree is then traversed (`PrunePredicate` keeps only flagged nodes, `LeafEvaluator` collects each surviving leaf's eight corner points), and the final bounding volume is constructed directly from that point set: `BV` need only be constructible from a `std::vector<Vec3T<T>>`.

If the initial box doesn't intersect the surface at all (or is degenerate), the routine falls back to returning the maximally representable bounding volume instead, signalling that the initial box needs to be chosen more generously.

Tip: A deeper maximum tree depth gives a tighter bounding volume, at the cost of more evaluations of I (one per candidate node, at every level).

Warning: This is an approximation, not an exact bound: it is only as tight as the finest cell size actually reached, and an implicit function whose rate of change meaningfully exceeds unity (violating the Eikonal property) can, in principle, still have surface features that fall outside the estimate unless σ is chosen generously enough to compensate.

3.8 Reading data

Routines for parsing surface grids from files into EBGeometry's DCEL grids (see *Half-edge meshes (DCEL)* for the concept, *DCEL* for the concrete API) – or directly into BVH-accelerated signed distance functions, including the triangle-mesh representation described conceptually in *Triangle meshes* – are given in the namespace `EBGeometry::Parser`. The source code is implemented in `Source/EBGeometry_Parser.hpp`.

Important: EBGeometry is currently limited to reading STL, PLY, OBJ, and VTK (legacy or XML polydata) files, and then reconstructing DCEL grids from those. PLY and VTK files can contain associated data on the nodes and faces, but this is not automatically populated when constructing the DCEL grids. It is also possible to build DCEL grids from polygon soups read using third-party codes (see *Using third-party sources*).

3.8.1 Quickstart

Every reader function in `EBGeometry::Parser` comes in two overloads: one that takes a single file name, and one that takes a `std::vector<std::string>` of file names and returns a vector of results, one per file. If you have one or multiple mesh files, the quickest way to turn them into BVH-accelerated signed distance fields is

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

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

This will build a DCEL mesh for each input file and wrap it in a `MeshSDF`, backed by a SIMD-accelerated `PackedBVH` over the mesh's faces. See *DCEL mesh SDF with PackedBVH* for further details.

Tip: If the input files consist only of triangles, use the version

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

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

This version will convert all DCEL polygons to triangles, pack them into SIMD-width groups, and usually provides a nice code speedup over `readIntoPackedBVH`.

3.8.2 Reading raw file data

At the lowest level, `readPLY<T>`, `readSTL<T>`, `readOBJ<T>`, and `readVTK<T>` (each with a single-filename and a `std::vector<std::string>` overload) parse a file into a raw, format-specific data structure (`PLY<T>`, `STL<T>`, `OBJ<T>`, or `VTK<T>`) holding nothing more than the vertex coordinates and face index lists as they appear in the file – no DCEL topology, no signed distance functionality. Two small helpers support this layer: `getFileType` inspects a file's extension to determine which of the four formats it is (or `FileType::Unsupported`), and `getFileEncoding` inspects the file header to determine whether it is `Encoding::ASCII` or `Encoding::Binary`. Most users will not call these raw readers directly – they exist mainly as the common first step every `readInto*` function below builds on – but they are useful if you need the raw vertex/face data without paying for DCEL construction at all.

Note: If an STL file contains multiple solids (uncommon, but technically valid STL), `readSTL` only reads the first one.

For the raw readers' exact signatures, see the Doxygen entries for readPLY, readSTL, readOBJ, and readVTK.

3.8.3 Reading mesh files

Above the raw per-format readers, EBGeometry offers five ways to turn a file directly into a higher-level representation, each doing progressively more work:

1. Into a DCEL mesh, with no signed-distance functionality – `readIntoDCEL`, see *DCEL*.
2. Into a bare DCEL signed distance function (`FlatMeshSDF`, $O(N)$ scan, no BVH) – `readIntoMesh`.
3. Into a signed distance function representation of a DCEL mesh, using a `PackedBVH` over DCEL faces (any polygon) – `MeshSDF`, via `readIntoPackedBVH`.
4. Into a signed distance function representation of a triangle mesh, using a `PackedBVH` over SoA triangle groups – `TriMeshSDF`, via `readIntoTriangleBVH`.

5. Into a flat, unconnected list of `Triangle` objects (each with precomputed vertex positions, normals, and edge normals, but no half-edge topology linking them) – via `readIntoTriangles`. Internally this still parses through a DCEL mesh first and then extracts each face as an independent `Triangle`; use it when you need per-triangle data as plain values (e.g. to feed your own acceleration structure) rather than any of EBGeometry's own SDF wrappers.

See *Mesh SDF classes* for how the three SDF classes (options 2-4 above) compare.

Important: The EBGeometry parser will read input files into internal objects that represent each file type. Conversion of these objects into DCEL meshes is not required, and it is possible to create bounding volume hierarchies directly from the facets. This is useful when only an acceleration structure is needed for looking up facets or triangles, but no signed distance function is otherwise required.

DCEL representation

To read one or multiple files and turn it into DCEL meshes, use `readIntoDCEL<T, Meta>(filename)` (or the `std::vector<std::string>` overload for multiple files at once), returning a `shared_ptr<DCEL::MeshT<T, Meta>>` (or a vector thereof). Note that this will only expose the DCEL mesh, but not include any signed distance functionality.

DCEL mesh SDF

To read one or multiple files and also turn it into a bare (BVH-free) signed distance representation, use `readIntoMesh<T, Meta>(filename)`, returning a `shared_ptr<FlatMeshSDF<T, Meta>>` (or a vector thereof for the multi-file overload).

DCEL mesh SDF with PackedBVH

`readIntoPackedBVH<T, Meta, K>(filename, build)` wraps a DCEL mesh in a `PackedBVH` (depth-first flat layout) with SIMD traversal, returning a `shared_ptr<MeshSDF<T, Meta, K>>` (or a vector thereof). It supports any polygon, not just triangles; the BVH branching factor `K` defaults to 4 and the build strategy `a_build` defaults to `BVH::Build::SAH`. For maximum throughput on triangle-only meshes, prefer `readIntoTriangleBVH` below.

Triangle meshes with PackedBVH

`readIntoTriangleBVH<T, Meta, K, W, StoragePolicy>(filename, maxLeafGroups, build)` converts all DCEL polygons to triangles, packs them into SoA groups of `W`, and builds a `PackedBVH`, returning a `shared_ptr<TriMeshSDF<T, Meta, K, W, StoragePolicy>>` (or a vector thereof). SIMD intrinsics evaluate up to `W` triangles per leaf visit. `K` and `W` default to the SIMD-optimal values for `T` on the current ISA (`BVH::DefaultBranchingRatio<T>()` and `TriangleSoA::DefaultWidth<T>()`, see *Mesh SDF classes*); `maxLeafGroups` (default 4) bounds the number of full `W`-sized SoA groups per BVH leaf; `StoragePolicy` defaults to `BVH::ValueStorage<TriangleSoAT<T, W>>`, matching `TriMeshSDF`'s own default (see *Mesh SDF classes* for the rationale, and why `readIntoPackedBVH`/`MeshSDF` above has no equivalent parameter). The code will raise an error if any face is not a triangle.

Flat triangle list

`readIntoTriangles<T, Meta>(filename)` returns a flat `std::vector<shared_ptr<Triangle<T, Meta>>>` (or, for the multi-file overload, one such vector per file) – every face of the parsed mesh as an independent, self-contained `Triangle` value, with no DCEL/half-edge topology connecting them. Use this when some other part of your code wants raw triangle values (for example, to build a custom acceleration structure) rather than any of EBGeometry's own SDF wrappers.

Note: `readIntoDCEL`, `readIntoMesh`, `readIntoPackedBVH`, `readIntoTriangleBVH`, and `readIntoTriangles` are declared `inline static` in the header, so this project's current Doxygen configuration (`EXTRACT_STATIC = NO`) does not extract them into the generated API reference individually – their exact signatures are documented via the Doxygen comment on each function directly in `Source/EBGeometry_Parser.hpp`. The raw per-format readers above (`readPLY`/`readSTL`/`readOBJ`/`readVTK`, declared without `static`) do not have this limitation and are linked individually.

3.8.4 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.

Turning a soup into a DCEL mesh is a two- (optionally three-) step process, using the functions in namespace `EBGeometry::Soup`:

- `containsDegeneratePolygons(vertices, facets)` is an optional up-front check: it returns `true` if any face has fewer than three vertices, or two or more vertices that coincide after lexicographic sorting. Useful for validating a soup produced by an external tool before spending time compressing/converting it.
- `compress(vertices, facets)` discards duplicate vertices from the soup in place, updating `facets` to reference the compressed vertex list.
- `soupToDCEL(mesh, vertices, facets, id)` builds the vertices, half-edges, and faces of the (already-compressed) soup into the output DCEL mesh, reconciles pair edges (internally, via `reconcilePairEdgesDCEL`, which links each half-edge $u \to v$ to its reverse $v \to u$), and runs a mesh sanity check. This also computes the vertex and edge normal vectors.

Note: Every function in `EBGeometry::Soup` is also declared `inline static`, so – for the same reason noted above for the `readInto*` family – none of them currently appear individually in the generated Doxygen API reference; the namespace page exists but is effectively empty. Their exact signatures and contracts are documented via the Doxygen comment on each function directly in `Source/EBGeometry_Soup.hpp`.

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

3.8.5 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*.

3.9 SIMD-accelerated classes

SIMD acceleration in EBGeometry is implemented with hand-written compiler intrinsics (`__m128`/`__m256`/`__m512` and their `double` counterparts), not compiler auto-vectorisation. As of today it is confined to the places listed below; everything else in the library is scalar code. This is a statement about the current state of the implementation, not an architectural ceiling – further classes may become SIMD-accelerated in the same style in the future. This page lists the SIMD-accelerated classes as they exist today, and explains precisely what gets vectorised in each.

On this page

3.9.1 Per-triangle signed-distance evaluation: `TriangleSoAT<T, W>`

`Source/EBGeometry_TriangleSoA.hpp` / `EBGeometry_TriangleSoAImplem.hpp`

See *Triangle meshes* for the conceptual picture of why triangles are packed this way.

What it stores: the vertex positions, vertex normals, edge normals, and face normal of W triangles, laid out as a *structure of arrays* (one flat, `alignas`-aligned array per coordinate/component, rather than W separate `Triangle` objects). See *Mesh SDF classes* for the rest of the `TriMeshSDF` machinery this is packed into.

What is vectorised: the entire closest-point-on-triangle signed-distance query, `TriangleSoAT::signedDistance(point)`. This is the full algorithm – classifying the projection of the query point against the triangle's three edge regions and interior region, computing the squared distance and sign for whichever region applies, per triangle – executed for all W triangles in the block *simultaneously*, one SIMD lane per triangle. The result is the minimum signed distance over the whole block, found with vectorised compare/blend instructions rather than a loop.

What this means in practice: evaluating a `TriangleSoAT<T, W>` block costs roughly the same number of instructions as evaluating a *single* triangle scalar, but produces the answer for W of them. This is the leaf-level cost inside every `TriMeshSDF::signedDistance()` BVH leaf visit.

Choosing and tuning W: the width is chosen from ISA auto-detection at compile time (the compiler-predefined macros described in *SIMD acceleration*), via `EBGeometry::TriangleSoA::DefaultWidth<T>()`, and `TriMeshSDF`/`Parser::readIntoTriangleBVH` default to the ISA-appropriate W for whichever precision `T` is in use. See *Mesh*

SDF classes for the full ISA/precision-to-default table, how to override W explicitly, and the data-alignment requirements it relies on.

3.9.2 Per-point distance evaluation: `PointSoAT<T, W>` / `PointAoSoA<T, Meta, W>`

`Source/EBGeometry_PointSoA.hpp` / `EBGeometry_PointSoAImplem.hpp` (and the metadata-carrying wrapper `EBGeometry_PointAoSoA.hpp` / `EBGeometry_PointAoSoAImplem.hpp`)

What it stores: the positions of W points, laid out as a *structure of arrays* (one flat, `alignas`-aligned array per coordinate, rather than W separate point objects). `PointSoAT` is position-only; `PointAoSoA<T, Meta, W>` adds a physically separate `std::array<Meta, W>` of per-point metadata alongside it, never interleaved with the positions, so the distance kernel touches exactly the same bytes either way – metadata is read only afterward, via `getMetaData(lane)`.

What is vectorised: the squared distance from a query point to all W lane positions, computed in one SIMD batch – one `_mm(128\|256\|512)_load_p[sd]` per coordinate array, then vectorised subtract/multiply/add for all W lanes at once. This one kernel (`getDistances2()`, which returns the whole W-lane array) backs every distance query: `getMinimumDistance2()` / `getMaximumDistance2()` horizontally reduce the lanes to the nearest / farthest, and the `*Distance()` forms add a single `sqrt`. The full array is what an all-neighbors (k-nearest-neighbor) leaf scan wants; the reduced minimum is what a closest-point leaf scan wants.

What this means in practice: evaluating the distance to a whole W-point block costs roughly the same as one scalar point distance, but produces the answer for W of them. This is the leaf-level cost of a point-cloud `PackedBVH` search – see `Examples/ClosestPointBVH` (closest point) and `Examples/NearestNeighborBVH` (k nearest neighbors), both built on the `PointCloudBVH` class.

Choosing and tuning W: the width is chosen from ISA auto-detection at compile time via `EBGeometry::PointSoA::DefaultWidth<T>()` – the width that fills one SIMD register exactly for `T`, on the same ISA/precision table as `TriangleSoA`'s. Pass a different `W` explicitly as the final template argument to override it.

3.9.3 SIMD-accelerated bounding-box pruning: `BVH::PackedBVH<T, P, K>`

`Source/EBGeometry_BVH.hpp` / `EBGeometry_BVHImplem.hpp`

See *Bounding volume hierarchies* for the conceptual picture of bounding volume hierarchies and tree pruning.

What it stores: each interior node's K children's bounding boxes, laid out as a *structure of arrays* (`ChildAABBSoA`: flat, `alignas`-aligned low/high-corner coordinate arrays across all K children), alongside the usual index-offset node data. See *PackedBVH* for the rest of the packed representation.

What is vectorised: the point-to-bounding-box squared-distance test used to decide which children to descend into (or prune) during traversal, in `PackedBVH::pruneTraverse()`. All K children of a node are tested against the query point in a single SIMD batch – one `_mm(256\|512)_load_p[sd]` per coordinate array, then vectorised subtract/max/multiply/add to get all K squared distances at once – rather than a scalar loop over children. `PackedBVH` has no `signedDistance()` of its own; `MeshSDF`/`TriMeshSDF::signedDistance()` each build a thin wrapper around `pruneTraverse()`, supplying a signed-distance leaf-eval and pruning rule. Any caller can invoke `pruneTraverse()` directly with its own leaf-eval/pruning- rule pair to get the same SIMD node pruning over a different search (e.g. nearest-neighbor search over a primitive type with no `signedDistance()` at all). See *Distance-pruned traversal: pruneTraverse()* for the full callback contract and traversal algorithm.

What this means in practice: the cost of deciding which subtree(s) to visit next no longer scales with K the way a scalar loop would; a wider branching factor (larger K) is close to "free" for this step as long as it still fits in one SIMD batch for the target ISA, which is exactly why K is chosen per-ISA (see below).

Choosing and tuning K: the branching factor is likewise chosen from ISA auto-detection at compile time, via `BVH::DefaultBranchingRatio<T>()` – a separate function from `TriangleSoA`'s, but driven by the same underlying ISA macros, so `TriMeshSDF`/ `Parser::readIntoTriangleBVH` still end up with a matching (K, W) pair for

whichever precision `T` is in use. See *Mesh SDF classes* for the full ISA/precision-to-default table and how to override K explicitly.

CHAPTER

FOUR

EXAMPLES

4.1 Overview

The `Examples/` folder contains pure EBGeometry examples with no third-party dependencies. Each one can be built directly with a compiler, with GNU Make, or with CMake (see *Building*), and every folder ships all three (a `GNUmakefile`, a `CMakeLists.txt`, and a plain `main.cpp` compilable with a single compiler invocation).

Alongside the examples with dedicated pages below, the following BVH and point-cloud examples ship in `Examples/`:

- `BuildBVH` – compares the BVH construction strategies (top-down, SAH, and the Morton, Hilbert, and Nested space-filling-curve builds) by build time.
- `ClosestPointBVH` – closest-point search over a point cloud using the `PointCloudBVH` class (see *Point clouds*).
- `NearestNeighborBVH` – the k-nearest-neighbor graph of a point cloud via `PointCloudBVH::allNearestNeighbors()`.

Examples that couple EBGeometry to a third-party application code's embedded-boundary grid generation (AMReX, Chombo) live separately under `Integrations/` instead, and additionally require that platform to be installed — see *Integrations*.

None of the EBGeometry examples produce output for visualization; they print timing and/or correctness information to the terminal. Each example folder also ships its own `README.md` with the exact build/run commands reproduced on its page.

4.1.1 Building and running any of them

Every example needs the path to the EBGeometry root — the directory containing `EBGeometry.hpp` — which is two levels up (`../..`) when building in place. All three build methods accept the same two overrides: `PRECISION`/`EBGEOMETRY_PRECISION` (`float` or `double`, default `double`) and the location of the EBGeometry tree itself (`EBGEOMETRY_HOME`, default `../..`); all three also produce the binary (`<Example>.ex`) directly in the example's own folder, regardless of where the CMake build tree itself lives:

```
# CMake
cmake -S . -B build -DEBGEOMETRY_PRECISION=float -DEBGEOMETRY_HOME=/path/to/EBGeometry
cmake --build build
./<Example>.ex

# GNU Make
make PRECISION=float EBGEOMETRY_HOME=/path/to/EBGeometry
./<Example>.ex

# Direct compilation
g++ -std=c++17 -O3 -march=native -DEBGEOMETRY_PRECISION=float -I../.. main.cpp -o <Example>.ex
./<Example>.ex
```

Run every example from inside its own folder — several resolve a default input mesh relative to the run directory. See *Building* for the full detail on each build method.

4.2 Shapes

A very basic example of using EBGeometry for creating analytic signed distance fields (see *Geometries*). A good starting point for exploring the library's built-in shapes and transformations.

The source for this example is at `Examples/Shapes/main.cpp`. See *Building* for how to compile it with CMake, GNU Make, or a direct compiler invocation.

```
cd Examples/Shapes
./Shapes.ex
```

4.3 MeshSDF

Reads a surface mesh and evaluates its signed distance field using all three mesh-SDF representations described in *Mesh SDF classes*:

- A naive $O(N)$ scan over all facets (`FlatMeshSDF`).
- A `PackedBVH` with pointer-free, index-offset nodes storing references to the facets directly (`MeshSDF`).
- A SIMD-accelerated, SoA-packed triangle BVH (`TriMeshSDF`).

Tip: SDF query complexity depends on both the geometry and the query point. A tessellated sphere has a "blind spot" at its center where even a BVH must visit most, if not all, primitives — this example is a good way to see that effect in practice.

The source for this example is at `Examples/MeshSDF/main.cpp`. See *Building* for how to compile it with CMake, GNU Make, or a direct compiler invocation.

```
cd Examples/MeshSDF
./MeshSDF.ex                                        # defaults to armadillo.obj
./MeshSDF.ex ../../common-3d-test-models/data/cow.obj   # or pick another mesh
```

With no argument the example loads `armadillo.obj` from the `common-3d-test-models` submodule, so make sure it is checked out first (see *Obtaining EBGeometry*).

4.4 CSGUnion

Builds a CSG *union* of two different kinds of implicit function — a signed distance field read from a surface mesh, and an analytic sphere. Both derive from `ImplicitFunction<T>`, so they combine directly through `BVHUnion`, which places the objects in a bounding volume hierarchy so that closest-object queries traverse the tree instead of testing every object (see *Bounding volume hierarchies* and *Geometries*).

The source for this example is at `Examples/CSGUnion/main.cpp`. See *Building* for how to compile it with CMake, GNU Make, or a direct compiler invocation.

```
cd Examples/CSGUnion
./CSGUnion.ex                                       # defaults to cow.obj
./CSGUnion.ex ../../common-3d-test-models/data/cow.obj
```

4.5 NestedBVH

Builds a *nested* bounding volume hierarchy: an outer, BVH-accelerated CSG union whose primitives are themselves BVH-backed mesh signed distance functions. One triangle mesh is loaded once into a `TriMeshSDF` (which owns an inner `PackedBVH` over its triangle groups) and then instanced at several positions – each placement a `Translate` wrapper sharing a pointer to that same `TriMeshSDF` – and the placements are combined with `BVHUnion`, which builds the outer `PackedBVH` over them. A single distance query therefore descends two levels of BVH — the outer union hierarchy to locate the nearby placement, then the mesh's own inner hierarchy to find the nearest triangle (see *Bounding volume hierarchies* and *Geometries*).

The outer union shares each placement by pointer (the default `BVH::SharedPtrStorage`) rather than copying it, so the one inner mesh BVH is built and stored just once — the recommended way to nest BVHs. See the *Storage policy* section of *BVH* for why the outer level must not use `BVH::ValueStorage` here.

The source for this example is at `Examples/NestedBVH/main.cpp`. See *Building* for how to compile it with CMake, GNU Make, or a direct compiler invocation. Unlike the mesh-reading examples above, it defaults to the `dodecahedron.stl` fixture shipped in the repository, so it needs no submodule.

```
cd Examples/NestedBVH
./NestedBVH.ex                      # defaults to ../../Tests/data/dodecahedron.stl
./NestedBVH.ex path/to/mesh.stl     # place copies of your own triangle mesh
```

4.6 PackedSpheres

Creates a scene of N^3 analytic spheres and combines them with two kinds of union: a standard union that scans every object, and a BVH-accelerated union (see *Bounding volume hierarchies*). Demonstrates the algorithmic-complexity benefit of the BVH for closest-object queries as the object count grows.

The source for this example is at `Examples/PackedSpheres/main.cpp`. See *Building* for how to compile it with CMake, GNU Make, or a direct compiler invocation.

```
cd Examples/PackedSpheres
./PackedSpheres.ex
```

4.7 RandomCity

Creates a scene of randomly placed boxes ("buildings") and, like `PackedSpheres`, compares a standard union against a BVH-accelerated union for closest-object queries.

The source for this example is at `Examples/RandomCity/main.cpp`. See *Building* for how to compile it with CMake, GNU Make, or a direct compiler invocation.

```
cd Examples/RandomCity
./RandomCity.ex
```

4.8 OctreeBoundingVolume

Computes an approximate bounding volume for an implicit function using octree refinement (see *Octree* and `ImplicitFunction::approximateBoundingVolumeOctree`).

The source for this example is at `Examples/OctreeBoundingVolume/main.cpp`. See *Building* for how to compile it with CMake, GNU Make, or a direct compiler invocation.

```
cd Examples/OctreeBoundingVolume
./OctreeBoundingVolume.ex
```

4.9 Integrations

Important: These examples are illustrative only, meant to show how EBGeometry can be integrated into a third-party application code – they are not actively maintained or built as part of EBGeometry's continuous integration. Unlike everything under *Overview*, which is compiled and run on every pull request, changes to a third-party platform (or to EBGeometry itself) may or may not break these examples without anyone noticing. Treat them as a starting point for your own integration, not a guarantee that they work out of the box.

4.9.1 AMReX

The `Integrations/AMReX/<something>` folders couple EBGeometry to AMReX's embedded-boundary grid generation: an EBGeometry signed distance function is handed to AMReX, which uses it to cut cells at the implicit surface. AMReX must be installed separately, with the `AMREX_HOME` environment variable pointing to it. Available examples:

- `Integrations/AMReX/Shapes`
- `Integrations/AMReX/MeshSDF`
- `Integrations/AMReX/PackedSpheres`
- `Integrations/AMReX/RandomCity`
- `Integrations/AMReX/PaintEB`

See the README in each folder for exact build and run instructions.

4.9.2 Chombo

The `Integrations/Chombo/<something>` folders couple EBGeometry to Chombo's embedded-boundary grid generation, in the same spirit as the AMReX examples above. Chombo must be installed separately, with the `CHOMBO_HOME` environment variable pointing to it. Available examples:

- `Integrations/Chombo/Shapes`
- `Integrations/Chombo/MeshSDF`
- `Integrations/Chombo/PackedSpheres`
- `Integrations/Chombo/RandomCity`

See the README in each folder for exact build and run instructions.

CHAPTER

FIVE

CONTRIBUTING AND TESTING

5.1 Overview

This section describes how to build and run the EBGeometry test suite locally, what the continuous integration (CI) pipeline checks, and the conventions expected of contributions. It is split into three pages, one per topic:

- *Running the unit tests locally* — building and running the Catch2 unit test suite, CMake presets, sanitizers, and a table of what each test binary covers.
- *Continuous integration* — what the GitHub Actions CI pipeline checks on every pull request, and how to reproduce those checks locally with `pre-commit`.
- *Contribution guidelines* — code style, static and dynamic assertions, and test coverage expected of new code.

5.2 Running the unit tests locally

EBGeometry ships a Catch2 v3 unit test suite under `Tests/`. Catch2 is fetched automatically at CMake configure time via `FetchContent` — no manual installation is needed.

On this page

5.2.1 Quick start with CMake presets

The repository ships `CMakePresets.json` (requires CMake 3.22+) with pre-configured debug and release profiles. The **debug preset** is the recommended starting point: it enables assertions, disables SIMD (cleaner debugger output), and turns on both the test suite and the examples.

```
# 1. Configure (Catch2 is fetched automatically on the first run)
cmake --preset debug

# 2. Build
cmake --build --preset debug --parallel $(nproc)

# 3a. Run unit tests only  (< 1 s)
ctest --preset debug

# 3b. Run example programs (allow several minutes in Debug mode)
ctest --preset examples
```

A successful unit-test run looks like:

```
100% tests passed, 0 tests failed out of 220
Label Time Summary:
unit    =   1.43 sec*proc (220 tests)
```

Most test files are written with Catch2's `TEMPLATE_TEST_CASE` so they can run under both `float` and `double`, but locally, by default, only `double` runs (fast iteration, matching whatever the CMake preset otherwise builds) – the count above is double-only. CI additionally configures with `-DEBGEOMETRY_TEST_BOTH_PRECISIONS=ON` to run the full suite under both precisions (422 tests). To do the same locally:

```
cmake --preset debug -DEBGEOMETRY_TEST_BOTH_PRECISIONS=ON
cmake --build --preset debug --parallel $(nproc)
ctest --preset debug
```

5.2.2 Running with sanitizers (AddressSanitizer + UBSan)

```
cmake --preset debug-san
cmake --build --preset debug-san --parallel $(nproc)
ctest --preset debug-san
```

5.2.3 Preset reference

Preset	Build type	Assertions	SIMD	Tests / Examples
`debug`	`Debug`	ON	none	Unit tests + examples (labelled separately)
`debug-san`	`Debug`	ON	none	Unit tests + examples with ASan/UBSan
`release`	`Release`	OFF	avx	OFF (library only)
`release-test`	`Release`	OFF	avx	Unit tests + examples

5.2.4 Manual CMake options (without presets)

If you cannot use presets (CMake < 3.22 or an IDE that does not support them), pass the variables directly:

```
cmake -B build \
      -DCMAKE_BUILD_TYPE=Debug \
      -DEBGEOMETRY_BUILD_TESTS=ON \
      -DEBGEOMETRY_ENABLE_ASSERTIONS=ON \
      -DEBGEOMETRY_SIMD=none
cmake --build build --parallel $(nproc)
cd build && ctest -L unit --output-on-failure
```

5.2.5 CMake options

Option	Default	Effect
`EBGEOMETRY_BUILD_TESTS`	`OFF`	Fetch Catch2 and build the `Tests/` suite.
`EBGEOMETRY_BUILD_EXAMPLES`	`OFF`	Build the standalone examples under `Examples/`.
`EBGEOMETRY_ENABLE_ASSERTIONS`	`OFF`	Compile with `EBGEOMETRY_EXPECT()` guards active. Strongly recommended when running tests so that precondition violations abort with a clear message rather than silently producing wrong results.
`EBGEOMETRY_ENABLE_SANITIZERS`	`OFF`	Add `-fsanitize=address,undefined` to tests and examples.
`EBGEOMETRY_SIMD`	`avx`	`avx512` enables `-mavx512f -mavx2 -mavx -mfma -msse4.1`; `avx` enables `-mavx -mfma -msse4.1`; `sse41` enables `-msse4.1`; `none` uses the scalar fallback.

5.2.6 Selecting individual tests

Catch2 test cases are registered with CTest so you can filter by name or tag:

```
# Run only the Vec tests
ctest --output-on-failure -R "Vec3T"

# Run only the SFC tests
ctest --output-on-failure -R "SFC"

# Run a single test binary directly (all Catch2 options available).
# Each preset gets its own build directory (build/<preset-name>/...); adjust
# the path if you configured with a preset instead of the manual example above.
./Tests/TestVec --list-tests
./Tests/TestAnalyticSDF "[SphereSDF]"
```

5.2.7 Test coverage

Test binary	What is covered
`TestVec`	`Vec2T` and `Vec3T`: construction, arithmetic, dot/cross products, length, component-wise min/max, `minDir`/`maxDir`, lexicographic ordering, scalar-over-vector `operator/`.
`TestBoundingVolumes`	`AABBT` and `SphereT`: construction from corners and point clouds, volume, surface area, point distance, intersection predicate, overlapping volume.
`TestAnalyticSDF`	`SphereSDF`, `BoxSDF`, `PlaneSDF`, `CylinderSDF`, `TorusSDF`; CSG `Union()`, `Intersection()`, `Complement()`.
`TestDCEL`	DCEL topology of a hardcoded tetrahedron (face/vertex/edge counts, half-edge pairing, unit normals, `sanityCheck`); signed-distance correctness for `FlatMeshSDF`; agreement between `FlatMeshSDF` (brute-force) and `TriMeshSDF` (SIMD BVH).
`TestSFC`	`Morton`, `Nested`, and `Hilbert` space-filling curves: encode/decode roundtrip across the full valid coordinate range, monotonicity along one axis, injectivity, `ValidSpan` boundary regression, and (for Hilbert) the consecutive-code adjacency property.
`TestTriangle`	`Triangle`: face normal from vertex ordering, and signed-distance correctness for points closest to the face interior, an edge, and a vertex.
`TestSTL`	`STL`: construction, copy/move semantics; `Parser::readSTL` reading raw vertex/facet data from a test file; round-trip through `convertToDCEL` into a valid, watertight mesh.
`TestPLY`	`PLY`: construction, copy/move semantics; named vertex/face property storage and retrieval (including the out-of-range-name error path); round-trip through `convertToDCEL`.
`TestOBJ`	`OBJ`: construction, copy/move semantics; round-trip through `convertToDCEL` into a valid, watertight mesh.
`TestVTK`	`VTK`: construction, copy/move semantics; named point-data/cell-data scalar array storage and retrieval (including the out-of-range-name error path); round-trip through `convertToDCEL`.
`TestBVH`	A regular dodecahedron (20 vertices, 36 triangulated faces), read from disk in all four supported formats, used to verify: identical topology/geometry across formats; `BVH::TreeBVH`/`BVH::PackedBVH` `signedDistance` agreement with a brute-force scan across every partitioning strategy (top-down with the default and SAH partitioners, bottom-up with Morton, Nested, and Hilbert space-filling curves); `MeshSDF` and `TriMeshSDF` agreement with `FlatMeshSDF` for every `BVH::Build` strategy; and `MeshSDF::getClosestFaces()` ordering.
`TestCSG`	`SmoothMin()`/`SmoothMax()`/`ExpMin()` blending primitives; sharp and smooth `UnionIF`/`IntersectionIF`/`DifferenceIF` (and their BVH-accelerated counterparts); `FiniteRepetitionIF` tiling and boundary clamping.
`TestTransform`	`ComplementIF`, `TranslateIF`, `RotateIF`, `ScaleIF`, `OffsetIF`, `AnnularIF`, `BlurIF`, `MollifyIF`, `ElongateIF`, `ReflectIF`: each transform's free-function/class-constructor equivalence, and correctness against a hand-computable expected value.
`TestOctree`	`Octree::Node`: depth-first/breadth-first construction, traversal pruning and custom sort order; `ImplicitFunction::approximateBoundingVolumeOctree()` correctness (tightening bound with depth, degenerate/non-intersecting input fallback).
`TestPolygon2D`	`Polygon2D`: winding-number/crossing-number/subtended-angle containment algorithms, agreement between them on convex and concave (notched) polygons, and on a non-axis-aligned embedding plane.
`TestTriangleSoA`	`TriangleSoAT`: `signedDistance` agreement with the minimum over the individual packed `Triangle` instances (including padding when fewer than `W` triangles are packed), and bounding-volume construction from the packed data.
`TestSimpleTimer`	`SimpleTimer`: near-zero elapsed time on construction, measured duration against a requested sleep, restart-on-`start()` semantics, and relative ordering of two different sleep durations.

`Tests/InstantiateAll.cpp` is not itself a test binary and does not appear in the table above: it is a compile-only target (built by `cmake --build`, but never run by `ctest`) that explicitly instantiates every public class template for both `float` and `double`. Its purpose is to give `clang-tidy` and the project's warning set (in particular `-Wdouble-promotion`) something to analyse for both precisions, regardless of what the Catch2 tests above happen to exercise – `float` support would otherwise only be checked by whichever public classes a test happens to instantiate. See *Contribution guidelines* for when to add a new class to it.

Note: All of the mesh- and BVH-related tests above rely on the sign convention (negative inside, positive outside) and face-normal computation described in *Geometry representations* and *Half-edge meshes (DCEL)* – see those pages rather than this one for the convention itself.

5.3 Continuous integration

Every pull request targeting `main` triggers the CI pipeline defined in `.github/workflows/CI.yml`, on GitHub-hosted `ubuntu-latest` runners. Together, the jobs check: code formatting (`clang-format`) and static analysis (`clang-tidy`, advisory); code correctness and assurance (the Catch2 unit-test suite, under multiple compilers, SIMD levels, and both `float` and `double` precision; every bundled example, built and run via CMake, GNU Make, and direct compiler invocation, under GCC, Clang, and Intel's `icpx`; AddressSanitizer and UndefinedBehaviorSanitizer runs of the same test suite); spelling (`codespell`); license and copyright compliance (REUSE); and the project's documentation (a warnings-as-errors Doxygen build, and HTML/PDF Sphinx builds). A single aggregator job (`CI-passed`) then gates on all of the above so branch-protection rules only need to target one required check.

On this page

- *Running CI checks locally with* `pre-commit`

5.3.1 Jobs overview

Four *setup* jobs run first with no dependencies – `Formatting`, `Codespell`, `Reuse`, and `Doxygen-check`. Every other job depends on all four of them and runs only once they pass; the *Dependency graph* below shows the full picture. Each job is described in turn.

Formatting

Runs `clang-format` (version 21) over `Source/` and `Examples/` (matrix over the two directories) via jidicula/clang-format-action. The PR is blocked if any file differs from the formatted output.

Codespell

Runs the `codespell` pre-commit hook over every tracked file.

Reuse

Runs the `reuse` pre-commit hook (REUSE license/copyright header compliance) over every tracked file.

Doxygen-check

Runs the `doxygen-check` pre-commit hook: builds the Doxygen API reference from `Docs/doxygen.conf` with warnings treated as errors.

Static-analysis

Runs `clang-tidy-18` (via `run-clang-tidy-18`) over every `Tests/*.cpp` and `Examples/*.cpp` translation unit, using a compile-command database exported from the `debug` preset built with `clang++-14`. Marked `continue-on-error: true` (there is a known review backlog), so it is advisory and does not gate `CI-passed`.

Linux-GNU

Compiles and runs every example under `Examples/` directly with `g++` (matrix over `{g++-11, g++-12}` × the six example directories), using `-std=c++17 -pedantic -Wall -Wextra` plus a large set of additional diagnostic flags.

Linux-Intel

Compiles and runs a subset of examples (`MeshSDF`, `PackedSpheres`, `RandomCity`, `Shapes`) with Intel's `icpx` compiler, `-std=c++17 -Wall -Werror` plus additional diagnostic flags.

Examples-GNUMake

Builds and runs every example (matrix over the six example directories) via its own `GNUmakefile` (`make run`).

Examples-CMake

Configures and builds the top-level project with the `debug` preset (assertions on, `double` precision) and runs every example via `ctest --preset examples`. This is the path that exercises the CMake-driven build with assertions enabled, unlike `Linux-GNU/Linux-Intel` (raw compiler invocation, no assertions) or `Unit-Tests/Sanitizers` (examples disabled).

Examples-FloatPrecision

The same as `Examples-CMake`, but configured with `-DEBGEOMETRY_PRECISION=float` (a cache variable shared by every example's own `CMakeLists.txt`).

Build-documentation

Installs Doxygen, Graphviz, a LaTeX toolchain, Poppler (for the documentation figure pipeline), and Sphinx (with `sphinx_rtd_theme` and `sphinxcontrib-bibtex`); builds the Doxygen API reference; renders the documentation figures from their LaTeX/TikZ sources (`Scripts/build-doc-figures.sh`); builds the Sphinx HTML and PDF documentation; uploads the result as a workflow artifact.

Unit-Tests

Configures with the `debug` preset (examples disabled) across a matrix of compilers `{g++-12, clang++-14}` × SIMD levels `{none, avx, avx512}`, with `-DEBGEOMETRY_TEST_BOTH_PRECISIONS=ON`, and runs `ctest --preset debug`. The `avx512` configurations always compile (catching ISA-specific errors in the SIMD-accelerated code paths) but only actually run on a runner whose CPU supports AVX-512F.

Release-Test

Configures with the `release-test` preset (optimised, AVX, examples and tests both enabled) plus `-DEBGEOMETRY_TEST_BOTH_PRECISIONS=ON`, and runs `ctest --preset release-test` (the full unit-test and example suite).

Sanitizers

Configures with the `debug-san` preset (examples disabled) across a matrix of compilers `{g++-12, clang++-14}` × SIMD levels `{none, avx}`, with `-DEBGEOMETRY_TEST_BOTH_PRECISIONS=ON`, and runs `ctest --preset debug-san` under AddressSanitizer and UndefinedBehaviorSanitizer.

CI-passed

Dummy job whose only role is to aggregate every job above – except the advisory `Static-analysis` – as a single required status check. Branch-protection rules can target this job instead of each individual job.

5.3.2 Dependency graph

```
Formatting, Codespell, Reuse, Doxygen-check
 +-- Static-analysis          (advisory; not required by CI-passed)
 +-- Linux-GNU
 +-- Linux-Intel
 +-- Examples-GNUMake
 +-- Examples-CMake
 +-- Examples-FloatPrecision
 +-- Build-documentation
 +-- Unit-Tests
 +-- Release-Test
 +-- Sanitizers
      (all of the above except Static-analysis) --> CI-passed
```

`Formatting`, `Codespell`, `Reuse`, and `Doxygen-check` themselves have no dependencies and run first, in parallel; every other job depends on all four of them.

5.3.3 Running CI checks locally with `pre-commit`

A subset of the CI checks can be reproduced locally before pushing using pre-commit:

```
pip install pre-commit
pre-commit install          # installs the git hook
pre-commit run --all-files  # run all hooks on the whole tree
```

The hooks configured in `.pre-commit-config.yaml` include:

- **clang-format** — formats `Source/` and `Examples/` C/C++ files (default stage; runs on every `git commit` once `pre-commit install` has been run).
- **reuse** — REUSE license/copyright header compliance (default stage).
- **codespell** — typo detection across `Source/`, `Docs/`, and `Exec/` (default stage).
- **doxygen-check** — builds Doxygen from `Docs/doxygen.conf` (warnings as errors; default stage).
- **clang-tidy** — static analysis over the library headers, via `Scripts/clang-tidy-check.sh` (`stages: [manual]`; needs a compile database, so it (re)configures the `debug` preset first).
- **build-tests** — compiles the unit test suite with the `debug` preset (`stages: [manual]`), catching template-instantiation errors locally before they show up first in CI.
- **check-docs** — enforces the ban on `.. literalinclude::` in the Sphinx docs (`stages: [manual]`; see `Scripts/CheckDocs.py`): it fails if any `.. literalinclude::` directive exists anywhere under `Docs/Sphinx/source/`. A clean run only guarantees the banned directive is absent, not that the surrounding prose still accurately describes the code – that still needs a manual read-through.
- **build-doc-figures** — renders the documentation figures from their LaTeX/TikZ sources under `Docs/Sphinx/source/_static/` (`stages: [manual]`; requires `pdflatex` and `pdftoppm` on `PATH`, see *Overview*).
- **sphinx-build-html** — builds the Sphinx HTML docs in a managed Python virtual environment (`stages: [manual]`; run with `pre-commit run sphinx-build-html --hook-stage manual`).
- **sphinx-build-pdf** — builds the Sphinx PDF docs via `make latexpdf` (`stages: [manual]`; requires a full LaTeX toolchain — `texlive-latex-extra` and `latexmk` — on `PATH`; run with `pre-commit run sphinx-build-pdf --hook-stage manual`).

Tip: The manual-stage hooks above are not run by a plain `pre-commit run` invocation (they need system LaTeX/Doxygen/CMake tooling and take longer than the default hooks); use the explicit `--hook-stage manual` flag (with the specific hook id, or omit it to run all manual-stage hooks) when you want to verify them locally. Run `build-doc-figures` before either `sphinx-build-*` hook if you changed a figure's `.tex` source — pre-commit runs local hooks in the order they appear in the config file, so `pre-commit run --all-files --hook-stage manual` already gets the ordering right.

5.4 Contribution guidelines

On this page

5.4.1 Code style

- Format all C++ files with `clang-format` version 18 before committing. The repository's `.clang-format` file defines the style; running `clang-format -i <file>` will apply it in-place.
- Follow the naming conventions already present in the codebase (`UpperCamel` for types, `lowerCamel` with `a_` prefix for function parameters, `m_` prefix for member variables).

5.4.2 Static and dynamic assertions

EBGeometry guards its preconditions with two complementary mechanisms: a compile-time `static_assert` for anything decidable from template parameters alone, and the runtime `EBGEOMETRY_EXPECT(cond)` macro for anything that can only be checked from actual argument values. When adding new functionality, follow the same split:

- Guard all public-facing runtime preconditions (non-zero radii, valid axis indices, non-null pointers in public API, non-empty containers) with `EBGEOMETRY_EXPECT(cond)`. Do **not** guard internal invariants that the surrounding code already enforces — this adds noise without safety benefit.
- Guard template-parameter invariants that are known at compile time (a floating-point type, an in-range branching factor, …) with `static_assert` instead — a violation should fail the build rather than exercise a runtime check that can never actually be reached.

See *Configuration options*'s "Compile-time assertions (`static_assert`)" and *Runtime assertions (EBGEOMETRY_EXPECT)* ("Runtime assertions (`EBGEOMETRY_EXPECT`)") subsections for the full detail on how each mechanism behaves, including with and without `EBGEOMETRY_ENABLE_ASSERTIONS`.

5.4.3 Adding tests

New classes and functions should be accompanied by Catch2 tests under `Tests/`. Register the test binary in `Tests/CMakeLists.txt` using the `ebgeometry_add_test(TestName)` helper, which handles linking against `Catch2::Catch2WithMain`, setting the C++17 standard, and exposing `EBGEOMETRY_TEST_DATA_DIR` for any test data files placed under `Tests/data/`. See *Running the unit tests locally* for the existing test binaries and what each one covers — new tests should follow the same one-binary-per-class convention.

Adding new public classes should also be reflected in `Tests/InstantiateAll.cpp`, which explicitly instantiates every public class template so that `clang-tidy` and the project's warning set analyse them regardless of what the tests exercise.

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.