FOSSEE, IIT Bombay
OpenFOAM Research Migration Project
August 06, 2026

Classy Blocks Integration for Blender: A Structured Mesh Generation GUI for OpenFOAM

B C H Benjamin
Department of Computer Science and Engineering,
Atria Institute of Technology, Bengaluru
Intern at FOSSEE, IIT Bombay,
OpenFOAM GUI Summer Semester-Long Internship 2026

Abstract

This report presents an open-source Blender add-on that replaces manual blockMeshDict authoring with a visual, Classy Blocks-backed structured meshing workflow for OpenFOAM. The tool provides parametric primitives (Box, Cylinder, Frustum, Wedge), an interactive point-and-click sketch tool with Extrude and Revolve operations for arbitrary profile geometry, and STL/terrain surface projection for conforming block faces to real-world surfaces. A single-click pipeline generates the blockMeshDict, invokes blockMesh, converts the result to VTK, and reloads it into Blender for inspection, removing the need to hand-write structured mesh dictionaries.

1 Introduction

OpenFOAM's blockMesh utility generates structured hexahedral meshes from a hand-written blockMeshDict, in which vertices, blocks, edges, grading, and boundary patches are specified as text. For anything beyond the simplest geometry, authoring this dictionary by hand is tedious and error-prone, particularly for curved profiles, revolved geometry, and surfaces that must conform to external terrain or CAD data. This project, undertaken as part of the FOSSEE Summer Internship 2026 at IIT Bombay, develops a Blender add-on that lets a user construct structured-mesh geometry visually inside Blender's 3D viewport, using the open-source Classy Blocks Python library as the backend that emits a valid blockMeshDict. The add-on targets users who are comfortable with OpenFOAM meshing concepts but want a faster, visual, less error-prone way to construct block topology, rather than abstracting away blockMesh's underlying model.

The FOSSEE (Free/Libre and Open Source Software for Education) project, under the National Mission on Education through Information and Communication Technology (NMEICT), Ministry

of Education, Government of India, promotes the adoption of open-source scientific and engineering software in academia through fellowships, textbook companion projects, and lab migration activities. OpenFOAM, the free, open-source computational fluid dynamics toolbox this project extends the tooling around, is one of the flagship FLOSS packages FOSSEE supports. This project was carried out as a Summer/Semester-Long Internship under FOSSEE's OpenFOAM initiative at IIT Bombay, with the explicit goal of lowering the barrier to constructing structured meshes for OpenFOAM users who are not already fluent in blockMeshDict's text syntax, without hiding or reimplementing blockMesh's underlying block-structured meshing model.

A deliberate design constraint distinguishes this project from a conventional CAD-to-mesh pipeline: the add-on is built as a thin, inspectable layer over two existing, independently maintained tools — Blender, used purely as an interactive 3D authoring surface, and Classy Blocks, an open-source Python library that already knows how to emit valid OpenFOAM blockMeshDict files from a structured description of blocks, edges, and grading. The add-on's own responsibility is narrow: translate what a user does in Blender's viewport into the specification dictionaries Classy Blocks expects, and translate Classy Blocks' and OpenFOAM's output back into something visible in Blender. This separation of concerns keeps the generated blockMeshDict a first-class, independently inspectable artifact at every stage, rather than an opaque intermediate file the user is not expected to read.

The remainder of this report is organized as follows. Section 2 states the specific gaps in the existing blockMesh workflow that motivated this project. Section 3 develops the computational geometry underlying the add-on's three correctness-critical operations: object transform handling, sketch profile winding, and surface projection. Section 4 describes the simulation (meshing) procedure in detail, covering geometry authoring, the module architecture, boundary condition handling, the automated pipeline, and the testing methodology used to validate it. Section 5 presents results from representative test cases, documents defects found and resolved during development, discusses the key design trade-offs made, and states the current limitations and directions for future work. Section 6 concludes.

Related approaches to structured and semi-automatic mesh generation for OpenFOAM include snappyHexMesh, OpenFOAM's own utility for generating predominantly unstructured, automatically-refined meshes around arbitrary surface geometry from a background hex mesh; cfMesh, a similarly automatic unstructured mesher; and commercial, general-purpose meshing environments such as ANSYS ICEM CFD or Pointwise, which support structured block meshing through their own proprietary interfaces and export to a range of solvers, OpenFOAM among them. This project differs from the first two in scope, since it targets structured, block-based meshing specifically, and from the third in being free, open-source, built directly on top of blockMesh and Classy Blocks rather than a proprietary meshing kernel, and integrated into Blender, an environment many users already have installed for unrelated 3D modelling purposes rather than a dedicated, purchased meshing application.

Problem Statement Constructing a structured hexahedral mesh for a non-trivial geometry in native OpenFOAM requires manually computing and listing every block's eight vertices, its edges (including any arcs or splines), inter-block connectivity, per-direction cell counts and grading ratios, and boundary patch assignments, all as plain text. Three specific gaps motivated this work. First, there was no visual, CAD-like way to define arbitrary 2D profiles and turn them into 3D structured blocks via extrusion or revolution; profile-based blocks had to be worked out algebraically by hand. Second, there was no way to conform a block face to an external surface such as a terrain DEM

or an arbitrary CAD surface without manually computing projected vertex coordinates. Third, the standard workflow of editing the dictionary, running blockMesh, and visually inspecting the result required leaving the modeling environment entirely, slowing down iteration.

1.1 Design Requirements

From the three gaps above, four design requirements were set for the add-on. First, geometry authoring should happen directly in Blender's 3D viewport, using either typed primitives or an interactive point-and-click sketch tool, so that a user never has to hand-derive vertex coordinates. Second, any block face should be projectable onto an externally supplied surface (an STL file), so that terrain-following or CAD-conforming geometry does not require manual coordinate computation. Third, the full sequence of generating the blockMeshDict, invoking blockMesh, converting the result to a visualizable format, and displaying it should be reachable from inside Blender in a small, fixed number of actions, so that a user can iterate on geometry without leaving the modeling environment. Fourth, every one of these operations should degrade safely: a geometrically invalid or unsupported configuration should be reported to the user with an explicit, actionable message rather than silently producing an incorrect mesh or crashing the pipeline outright.

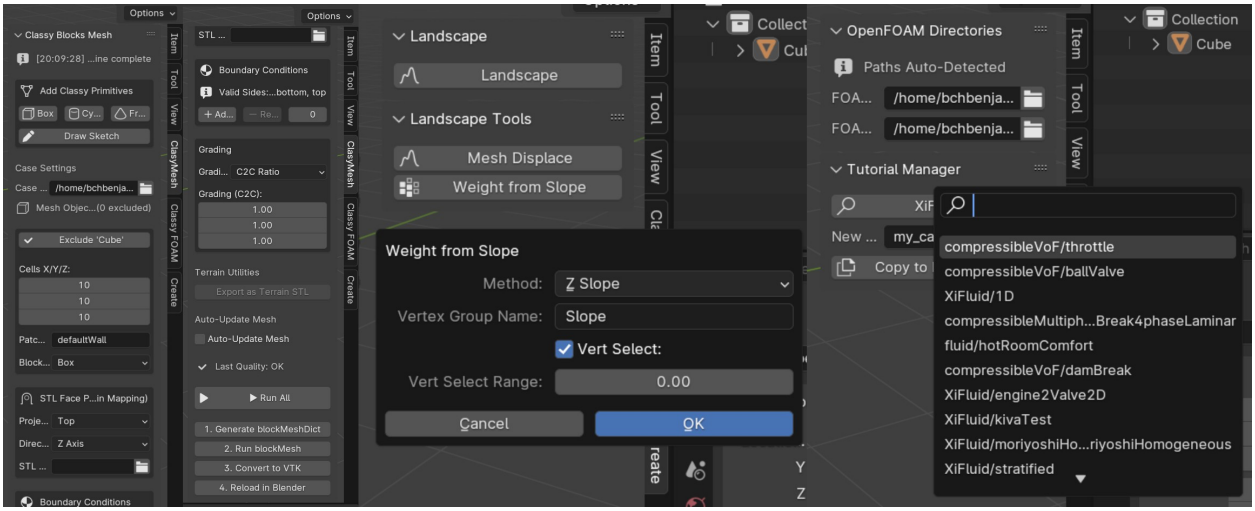

Figure 1: The add-on's sidebar panel in Blender, showing the primitive, sketch, and pipeline controls.

These requirements also shaped what was deliberately left out of scope. The add-on does not attempt to be a general-purpose parametric CAD system, does not solve geometric constraint systems, and does not replace OpenFOAM's own snappyHexMesh for unstructured or fully automatic meshing around complex geometry; its scope is specifically structured, block-based meshing of the kind blockMesh already supports, made faster and less error-prone to author.

2 Governing Equations

The add-on does not solve a flow-physics PDE itself; the geometric and topological correctness of the generated mesh instead rests on three pieces of computational geometry that were implemented

and validated during development.

2.1 Object transform decomposition

Each Blender object's world matrix M is decomposed into translation t, rotation (axis a, angle θ), and scale $s = (s_x, s_y, s_z)$, and re-applied to the underlying Classy Blocks shape in the same order Blender composes it:

$$M = T(t) \cdot R(a, \theta) \cdot S(s)$$

with the corresponding shape-level operations applied as scale, then rotate, then translate, so that a block's meshed position matches its position in the Blender viewport.

The rotation component $R(a, \theta)$ is computed from the axis-angle pair returned by Blender's matrix decomposition using Rodrigues' rotation formula, which expresses the rotation matrix for a unit axis $a = (a_x, a_y, a_z)$ and angle θ as:

$$R(a, \theta) = I + \sin(\theta) \cdot [a]_\times + (1 - \cos\theta) \cdot [a]_\times^2$$

where I is the 3×3 identity matrix and $[a]_\times$ is the skew-symmetric cross-product matrix of a. Rather than building this matrix explicitly, the add-on applies the equivalent sequence of Classy Blocks operations — scale, rotate about the given axis by the given angle, then translate — directly to the shape object, since Classy Blocks exposes these as first-class operations and composing them in this order reproduces Equation 1 without requiring a general 4×4 matrix interface, which — as discussed in Section 5 — the underlying library does not accept directly.

2.2 Sketch profile winding normalization

For a profile swept by extrusion along direction vector d, the face normal is computed from the first three profile points p_0, p_1, p_2 as:

$$n = (p_1 - p_0) \times (p_2 - p_1)$$

If $n \cdot d < 0$, the point order is reversed before the block is built, preventing the inside-out hex blocks that OpenFOAM's blockMesh rejects at run time when a profile is clicked in an inconsistent order.

Formally, the profile point order (p_0, p_1, p_2, p_3) is replaced by its reverse whenever:

$$n \cdot d < 0$$

This check is applied identically for both Extrude, where d is the extrusion direction vector, and Revolve, where d is replaced by the local sweep direction at the profile's centroid, computed as the cross product of the revolution axis and the vector from the axis to the centroid. Because the check depends only on the relative orientation of the clicked points and the block's intended direction, it is independent of whether the user clicked the four profile points clockwise or counter-clockwise, which was the underlying cause of the defect described in Section 5.2.

2.3 Revolve degeneracy detection

A profile can only be swept into a non-zero-volume solid by revolution if its plane is not perpendicular to the revolution axis. Given the profile's unit normal $\hat{n}$ and the revolution axis unit vector $\hat{a}$, the sweep is degenerate (zero volume) when:

$$|\hat{n} \cdot \hat{a}| > 1 - \epsilon$$

for a small tolerance ϵ, since this indicates the profile plane's normal is (near-)parallel to the revolution axis, meaning the profile lies in a plane perpendicular to the axis and simply rotates in place. The add-on checks this condition in the UI and warns the user before a pipeline run is attempted.

This condition follows directly from Pappus's centroid theorem, which states that the volume swept by revolving a plane figure of area A about an external axis in the same plane, through an angle ϕ, equals the product of the area and the distance travelled by the figure's centroid:

$$V = \phi \cdot \bar{y} \cdot A$$

where $\bar{y}$ is the perpendicular distance from the figure's centroid to the axis of revolution. When the profile's plane normal is parallel to the revolution axis, every point of the profile lies at a fixed distance from the axis but sweeps out no new volume in the axial direction, and the resulting solid degenerates to zero thickness regardless of $\bar{y}$; the check in Equation 5 detects exactly this configuration before it reaches blockMesh, where it would otherwise surface only as an opaque "inside-out" block error.

2.4 STL surface projection

To conform a block face to an external surface, each face vertex is projected along a chosen direction vector by ray-casting against the imported STL's triangle mesh; the first intersected point along the ray replaces the original vertex, with a nearest-surface-point fallback for rays that do not intersect the surface within its footprint.

Formally, a projection ray from origin O along direction D is parameterised as:

$$P(t) = O + t \cdot D, \quad t \geq 0$$

and intersection against each triangle of the imported STL mesh is tested using the standard Möller–Trumbore formulation, which expresses the intersection point in barycentric coordinates (u, v) of the triangle with vertices V_0, V_1, V_2:

$$P(t) = V_0 + u \cdot (V_1 - V_0) + v \cdot (V_2 - V_0), \quad u, v \geq 0, \quad u + v \leq 1$$

solved for the smallest non-negative t across all triangles to find the nearest intersection along the ray. When no triangle satisfies this system for a given ray — for example, when a vertex lies outside the STL surface's footprint — the projected point instead falls back to the closest point on the surface overall, found by minimising Euclidean distance over all triangles:

$$P^* = \text{argmin}_{M \in R} \|M - P\|$$

This fallback avoids leaving a vertex unprojected (which would produce a discontinuous, non-manifold block face) at the cost of no longer guaranteeing the projected point lies exactly along the original ray direction; this trade-off, and the warning issued for non-manifold source STL files, is discussed further in Section 5.4.

3 Simulation Procedure

3.1 Geometry and Mesh

Geometry is created in Blender in one of two ways: as a typed primitive (Box, Cylinder, Frustum, Wedge) added directly from the add-on panel, or as a hand-drawn 2D sketch, created by clicking points in the 3D viewport (with optional grid snapping and a choice of sketch plane), which is then tagged for Extrude or Revolve to become a 3D block. In both cases, objects are read by a geometry extraction stage that converts the Blender object into a structured specification dictionary, and a mesh-building stage that constructs the corresponding Classy Blocks shape (Box, Cylinder, Frustum, Wedge, Loft, Extrude, or Revolve) and writes the resulting blockMeshDict. Blocks expose per-edge cell counts and grading ratios from the add-on panel, and any block face may optionally be projected onto an imported STL surface for terrain or arbitrary-surface conformance, using the projection described in Section 3.

3.1.1 Cell Grading

Each block edge exposed in the add-on panel takes a cell count and an expansion ratio, matching blockMesh's own simpleGrading specification. The expansion ratio r along an edge is defined as the ratio of the last cell's size to the first cell's size along that edge:

$$r = h_n / h_1$$

and, for a geometric progression of n cells between the edge's two endpoints, the size of the i-th cell follows:

$$h_i = h_1 \cdot k^{(i-1)}, \quad \text{where } k = r^{1/(n-1)}$$

The add-on writes these two parameters directly into each block's simpleGrading entry in the generated blockMeshDict, in the same form OpenFOAM itself expects, so that a user already familiar with blockMesh's grading convention can predict the resulting cell distribution directly from the values entered in the panel, without the add-on introducing a separate, translated grading convention of its own.

3.1.2 Module Architecture

The add-on's source is organised into single-responsibility modules that mirror the spec-dict-in, Classy-Blocks-object-out pipeline described above. geometry_extractor.py reads Blender objects (primitives, tagged sketches, and curves) and converts each into a structured specification dictionary, applying the transform decomposition of Equation 1 and the winding normalisation of Equation 3 during extraction. mesh_builder.py consumes these dictionaries and constructs the corresponding Classy Blocks shape objects, assembling them into a Mesh and writing the final blockMeshDict. sketch_tool.py implements the interactive point-and-click sketch operator as a Blender modal operator, using the gpu and gpu_extras.batch modules for live on-screen drawing, and separately drives lightweight Screw and Solidify modifier previews so a user can see an approximate extruded or revolved shape without running the full OpenFOAM pipeline. stl_projector.py implements STL loading, validation, and the ray-casting projection of Equations 7–9. operators.py

defines the Blender operators exposed as buttons in the sidebar panel (defined in ui.py), each following a consistent pattern of input validation, a try/except boundary around the underlying operation, and an explicit, human-readable error report on failure rather than a raw exception. vtk_importer.py and case_setup.py handle, respectively, reloading a generated VTK mesh back into Blender and writing the supporting OpenFOAM case directory structure (system/, constant/, and the boundary/initial field skeleton) around the generated blockMeshDict.

Table 1: Supported Classy Blocks shape types, their corresponding Blender authoring methods, and non-uniform scaling support.

Block type	Authored via	Edge representation	Non-uniform scale
Box	Primitive	Straight	Supported
Cylinder	Primitive	Circular arc	Not supported
Frustum	Primitive	Circular arc	Not supported
Wedge	Primitive	Circular arc	Not supported
Loft	Primitive	Spline	Supported
Extrude	Sketch + Extrude	Straight / spline	Supported
Revolve	Sketch + Revolve	Circular arc	Not supported

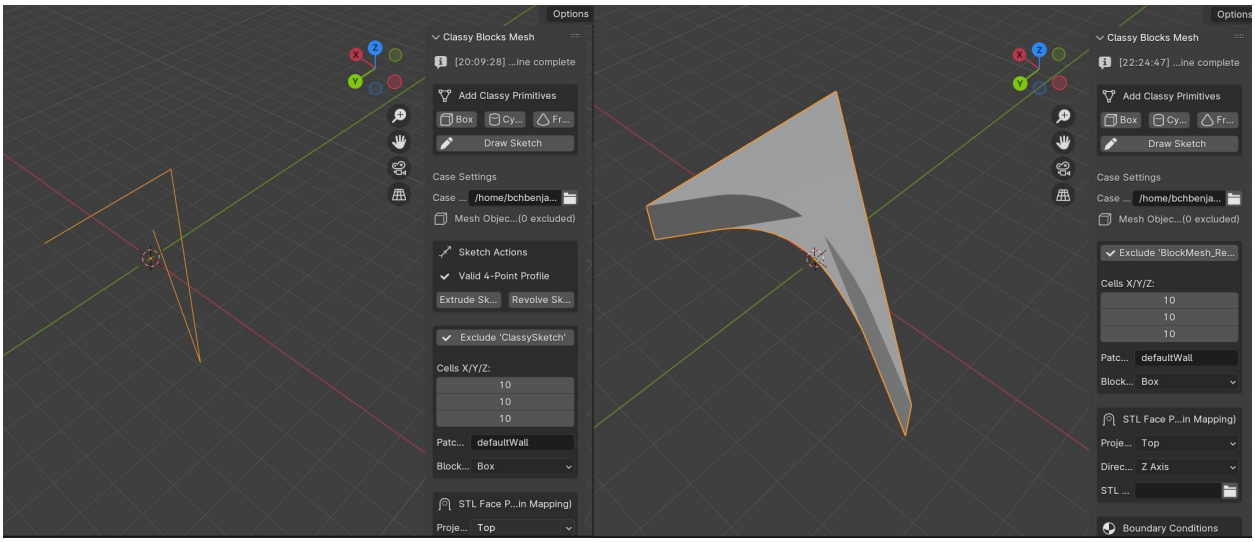

Figure 2: A profile sketch drawn by clicking points in the viewport, alongside its extruded 3D block.

3.2 Initial and Boundary Conditions

As a mesh-generation tool, the add-on does not set field initial conditions; it instead exposes OpenFOAM boundary patch assignment at the block-face level. Each block face can be labelled with a patch name and type (e.g. wall, inlet, outlet, empty) directly from the object's properties panel, and these labels are carried through into the generated blockMeshDict's boundary section, so that

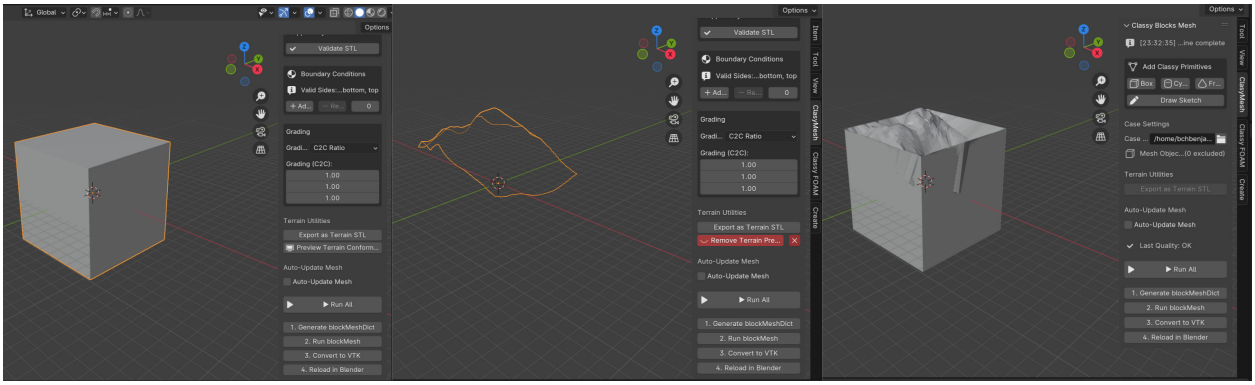

Figure 3: A block face projected onto an imported terrain STL, before and after projection, with the live viewport preview option enabled.

the mesh produced is ready for a user to attach field boundary conditions in the case's 0 directory using standard OpenFOAM practice.

The set of patch types exposed matches the boundary condition types most commonly required at the meshing stage — wall, patch, symmetry, and empty (the last being required for the pseudo-2D front-and-back faces of thin extruded cases, following standard OpenFOAM convention) — with any patch left unassigned defaulting to a generic patch type that a user can further specialise later in their case setup. Patch names are propagated verbatim into the blockMeshDict's boundary block, so that field files created afterward in the case's 0 directory can reference them directly without renaming.

3.3 Solver

The add-on does not itself run a flow solver. Its single-click pipeline instead automates the pre-processing chain around blockMesh: it writes the blockMeshDict, invokes the blockMesh utility as a subprocess, converts the resulting mesh to VTK using foamToVTK for visual inspection, and reloads that VTK mesh back into Blender as a new object. Running an actual solver (e.g. simpleFoam, icoFoam) on the resulting case is left to the user via standard OpenFOAM invocation once the mesh has been generated and verified.

3.4 Testing Methodology

Because bpy, Blender's Python API, is only importable from inside a running Blender process, modules that need to be tested outside Blender — geometry_extractor.py, mesh_builder.py, and stl_projector.py — are written without a top-level import bpy, and are loaded in the test suite via importlib.util against a mocked bpy module, allowing them to run under plain pytest in a continuous-integration-style environment. Modules that are inherently Blender-only, such as the modal sketch operator's viewport interaction, are instead verified through a combination of narrower headless unit tests of their non-Blender-dependent logic and manual verification inside a real Blender session, since aspects such as GPU draw-handler cleanup on operator cancellation or add-on disable cannot be exercised by a mocked pytest run.

A recurring lesson from development, discussed further in Section 5.2, was that unit tests asserting only the structure of an intermediate specification dictionary were insufficient to catch defects that only manifested in the final mesh geometry. The test suite was accordingly extended with end-to-end tests that carry a specification through the full extraction-to-blockMeshDict pipeline and, where practical, invoke the real blockMesh subprocess on the result and assert on its exit status and the resulting mesh's vertex coordinates, rather than stopping at the specification dictionary. At the time of writing, the suite comprises over one hundred tests spanning geometry extraction, mesh building for every supported block type, STL projection, sketch winding and revolve degeneracy, and full pipeline round-trips, run alongside a small number of known, pre-existing failures in an older test file that are caused by a stale mocked attribute reference unrelated to this project's own changes, and are documented as such rather than silently ignored.

Table 2: Automated test suite categories, coverage details, and execution modes.

Test category	What is exercised	Execution mode
Geometry extraction	Primitive and sketch classification, transform decomposition	Headless, mocked bpy
Mesh building	Spec-dict to Classy Blocks shape construction, all block types	Headless, mocked bpy
STL projection	Ray-casting, fallback, manifold validation	Headless, mocked bpy
Winding & degeneracy	Extrude/Revolve point-order and axis-plane checks	Headless, real blockMesh subprocess
Transform correctness	Translation, rotation, non-uniform scale, end-to-end	Headless, real blockMesh subprocess
Sketch tool logic	Point ordering, curve-to-spec classification fast-path	Headless, mocked bpy
Full pipeline	Generate, run blockMesh, convert VTK, reload	Manual, real Blender session

3.5 Error Handling Philosophy

Every operator in the add-on follows the same three-part error-handling pattern: an explicit guard that validates its required inputs (a set case path, a selected object, an existing STL file) before doing any work; the core operation wrapped in a try/except boundary; and, on failure, a human-readable message reported through Blender's own operator reporting mechanism rather than an unhandled traceback reaching the user. Two consequences of this pattern are visible in Section 5.2's defect descriptions. First, because mesh_builder.py is not wrapped in a blanket exception handler that silently skips a failing block, a genuinely invalid block — for example, one with a self-intersecting or otherwise degenerate profile that OpenFOAM itself does not reject — still causes the pipeline to fail loudly rather than silently omitting that block from the mesh and reporting success; an earlier version of this behaviour that did swallow such errors per-block was deliberately reverted

for this reason. Second, because failures are attributed to the specific stage that produced them (extraction, mesh building, blockMesh execution, or VTK conversion), the STL file availability defect in Section 5.2 was straightforward to isolate to the generation stage rather than blockMesh itself, despite blockMesh being where the failure was originally observed.

3.6 Worked Example: A Terrain-Conforming Extruded Block

As a concrete illustration of the full workflow, consider constructing a block whose top face should conform to an imported terrain surface. The user first selects the Draw Sketch tool from the sidebar and chooses a drawing plane appropriate to the intended profile; clicking four points in the viewport, optionally with grid snapping enabled, defines the profile, which is finalised by pressing Enter. The finished sketch is tagged for Extrude with a chosen direction and distance, at which point the add-on attaches a lightweight Solidify-modifier preview so the user can immediately see an approximate 3D shape in the viewport without leaving Blender or invoking OpenFOAM. The user then opens the STL projection panel for the relevant face, selects a previously exported terrain STL file, and runs the panel's validation step, which reports the file's triangle count and manifold status per Section 5.1. With the projection configured, the user clicks Run All: the add-on writes the blockMeshDict (automatically ensuring the referenced STL is present in the case's constant/triSurface directory, per the fix described in Section 5.2), invokes blockMesh, converts the result to VTK, and reloads the mesh into Blender as a new object, at which point the projected face's conformance to the terrain surface is directly visible and inspectable in the viewport, alongside the still-available, human-readable blockMeshDict on disk.

Each stage of the pipeline is exposed as its own operator and button (Generate blockMeshDict, Run blockMesh, Convert to VTK, Reload in Blender) rather than being collapsed into a single opaque action, so that a failure at any stage is immediately attributable: a malformed specification is caught at Generate time, a geometrically invalid mesh (for example, an inside-out block reaching blockMesh despite the checks in Section 3, or a cell count that makes the case impractically large) is caught at Run blockMesh time with OpenFOAM's own diagnostic output surfaced directly in Blender's info log, and a missing or unreadable VTK file is caught at Convert or Reload time. A single Run All action chains all four stages for the common case where no intermediate inspection is needed.

4 Results and Discussions

The pipeline was validated end-to-end on three representative cases: a default box primitive, a sketch-derived extruded block, and a box with one face projected onto a terrain STL. In each case, Generate blockMeshDict, Run blockMesh, Convert to VTK, and Reload completed successfully, with the reloaded mesh in Blender visually matching the source geometry's position and orientation, including after non-uniform scaling and rotation of the source object.

4.1 Quantitative Mesh Statistics

For the default box primitive at a $10\times10\times10$ cell grading, the generated blockMeshDict was consistently on the order of 1.5 kilobytes, growing to roughly 9–10 kilobytes once a sketch-derived

Extrude or Revolve block and its associated boundary patch definitions were added to the same case. Terrain STL files used during projection testing ranged up to approximately 32,000 triangles; the add-on's STL validation step reports both triangle count and a manifold (watertight) check for every imported file, since a non-manifold source surface can produce inconsistent projection results at the surface's boundary, as discussed in Section 5.4.

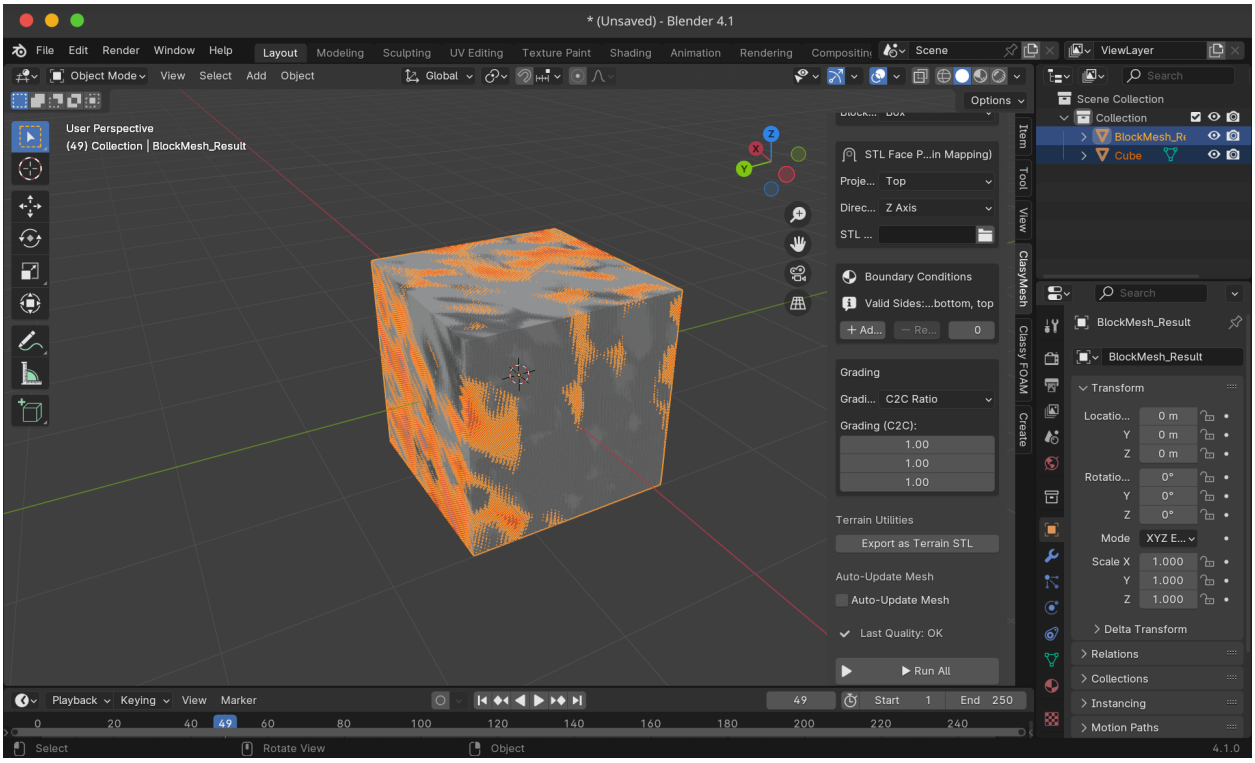

Figure 4: The default box primitive after the full generate → blockMesh → VTK → reload pipeline.

Two correctness issues surfaced during manual testing and were resolved during development. First, sketch profiles clicked in an inconsistent point order produced inside-out hex blocks that blockMesh rejected outright; this was resolved by the winding-normalization check in Section 3 and is now covered by regression tests that build a profile in both point orders and confirm both mesh successfully. Second, object transforms (translation, rotation, and non-uniform scale) were silently dropped by an incorrect call into the Classy Blocks transform API, causing every block to be meshed at its local, untransformed position regardless of where it was placed in the Blender scene; this was traced to the underlying library expecting its own transform objects rather than a raw 4×4 matrix, and was resolved by decomposing and reapplying the transform explicitly, per Section 3, with end-to-end tests added that assert the final meshed vertex coordinates match the expected world-space position for translated, rotated, and non-uniformly scaled objects. Both fixes highlight the value of testing against the actual meshing output rather than only the intermediate specification dictionary, since both defects were invisible at the specification level and only manifested in the final mesh.

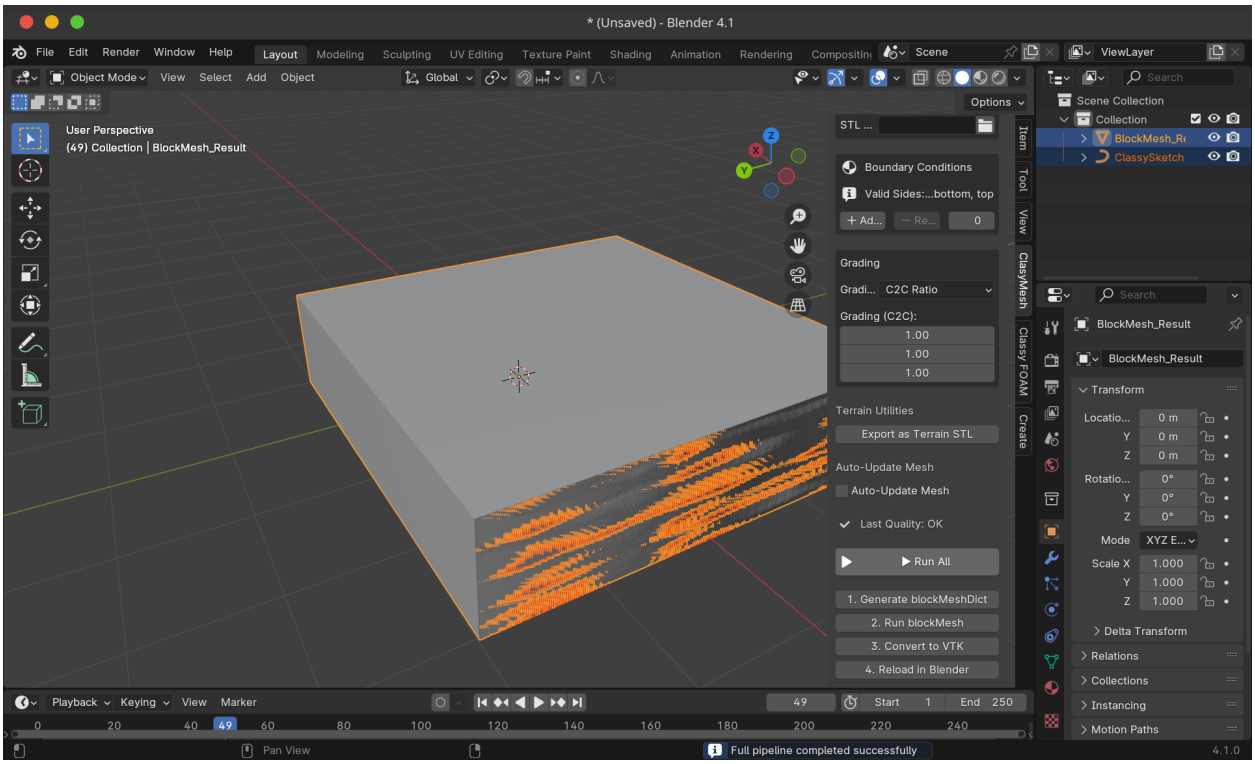

Figure 5: A sketch-derived extruded block after meshing and reload.

4.2 Defect Analysis

Beyond the two defects already described, three further defects were identified through manual end-to-end testing in a real Blender session, each illustrating a different way an automated test suite can miss a real failure mode.

Sketch dispatch regression. After the winding and transform fixes above were merged, a later change to the geometry extraction module's fast-path logic caused any sketch object to be routed to a generic, untyped sketch handler regardless of whether it had actually been tagged for Extrude or Revolve, reproducing the original "Unknown block type'sketch'" failure this fast-path had originally been written to prevent. The regression was traced to the fast-path's fallback branch, rather than its primary dispatch condition, having been altered independently of the branch that had originally been tested; the fix restored the fallback to skip untagged sketches rather than emitting an unbuildable specification, and a dedicated regression test was added that asserts specifically on the fallback branch's behaviour, rather than only on the already-covered primary path, so that a similar future regression would be caught in the branch it actually occurred in.

STL file availability at generation time. Configuring an STL projection through the panel's validation control recorded the file path on the object but did not copy the source STL into the case directory's constant/triSurface and constant/geometry locations that blockMesh expects a referenced surface file to exist in; only a separate, easily-missed "Export as Terrain STL" action performed that copy. The practical effect was that blockMesh could fail with an OpenFOAM-level "cannot find triSurface file" error on a case that had otherwise generated a syntactically valid blockMeshDict. The fix makes blockMeshDict generation itself responsible for guaranteeing that any referenced

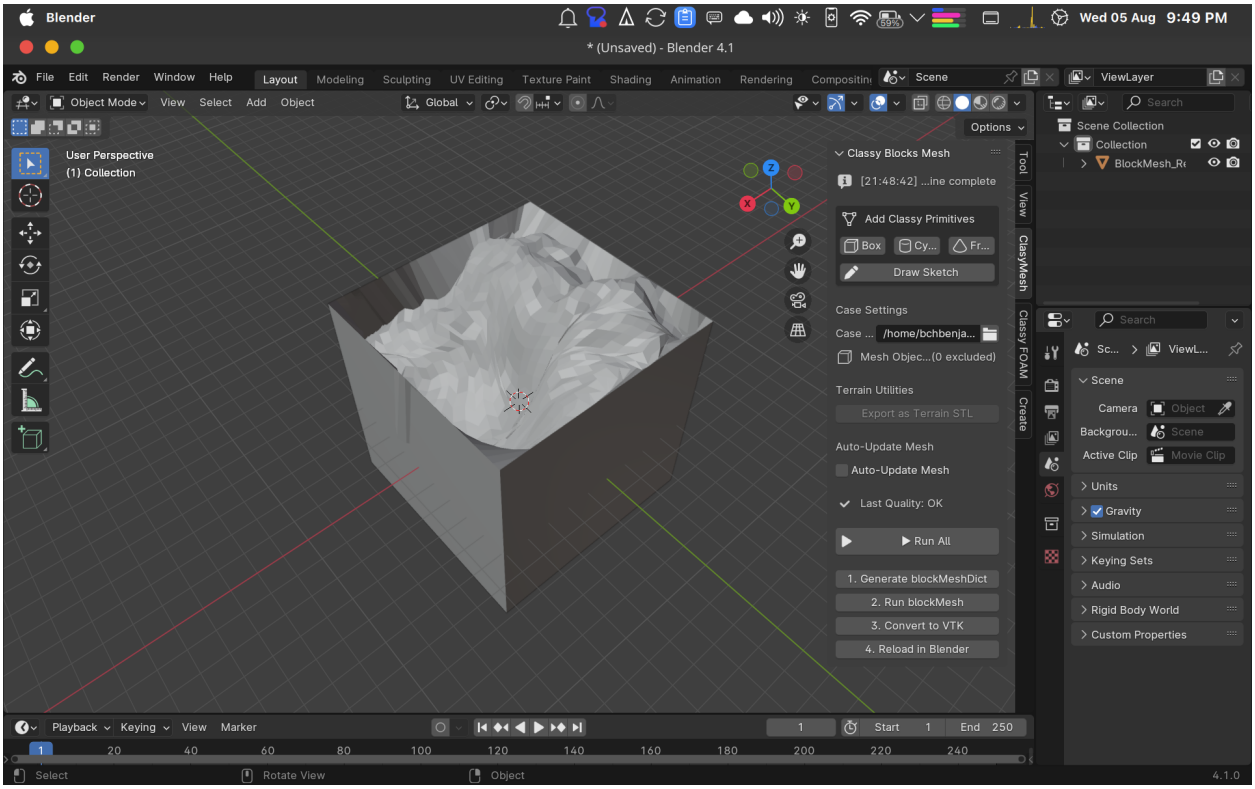

Figure 6: A terrain-projected block face after meshing, showing the mesh conforming to the imported STL surface.

STL is present in the case directory, copying it automatically if needed and failing early, with an explicit message, if the source file cannot be found at all, rather than allowing an unrelated-looking failure to surface later inside blockMesh's own output.

Revolve plane/axis mismatch. Even with the winding and degeneracy checks of Section 3 in place, a user could still draw a valid, correctly-wound sketch on the default ground plane and select a revolution axis perpendicular to that plane, producing a geometrically degenerate zero-volume sweep by construction, independent of point order. This is not a defect in the degeneracy check itself — Equation 5 correctly flags this case — but a workflow gap: the sketch tool's default drawing plane and the set of revolution axes a user might reasonably choose were not connected in the interface. The resolution added an explicit sketch-plane selector (ground, front, or side plane) to the sketch tool itself, so that a profile intended for revolution can be authored in a plane that contains the intended revolution axis from the outset, alongside the existing inline warning that fires when an already-drawn profile and the selected axis are geometrically incompatible.

Overall, the add-on successfully replaces hand-authored blockMeshDict construction for the geometry classes it supports, while remaining a thin, inspectable layer over Classy Blocks and blockMesh rather than a black box: the generated blockMeshDict is always available for direct inspection, and every add-on operation maps onto an explicit, documented Classy Blocks or OpenFOAM operation.

4.3 Design Decisions and Trade-offs

Native projection versus pre-warped geometry. STL surface conformance (Section 3.4) could in principle be implemented either by pre-warping a block's vertex positions in Python before writing the blockMeshDict, or by relying on OpenFOAM's own searchableSurface projection mechanism at blockMesh run time. The add-on takes the latter approach where the underlying Classy Blocks API supports it, since native projection respects OpenFOAM's own edge and face curvature handling during mesh generation rather than baking in a fixed, pre-computed approximation; the Python-side ray-casting projection of Equations 7–9 is retained primarily for validation — confirming a chosen STL and projection direction actually intersect the target geometry — and as a fallback for shape/face combinations the native mechanism does not cover.

Inline warnings versus restrictive controls. When a user's chosen combination of sketch plane and revolution axis is geometrically invalid, the interface could either omit the invalid axis options from the control entirely or allow any selection while warning inline when the current combination is invalid. The add-on takes the latter approach, since restricting the available options would require anticipating every way a sketch's plane could be edited afterward, whereas an inline check computed from the sketch's actual current geometry remains correct regardless of how that geometry was produced.

Approximate live preview versus authoritative geometry. The Extrude and Revolve operators attach a lightweight Blender Solidify or Screw modifier to give immediate visual feedback in the viewport before a user commits to running the full OpenFOAM pipeline. This preview is explicitly labelled as such in the interface, since it approximates the intended shape using Blender's own modifier stack rather than Classy Blocks' hexahedral block topology, and is not guaranteed to match the final meshed geometry exactly, particularly for grading and edge curvature; it is positioned purely as a fast sanity check, with the generated and reloaded mesh remaining the authoritative result.

4.4 Limitations

Several limitations remain at the time of writing. Non-uniform scaling is only supported for block types whose edges are represented as straight or spline lines (Box, Extrude, Loft); shapes that rely on OpenFOAM's circular arc edges (Cylinder, Frustum, Wedge, Revolve) cannot represent a non-uniformly scaled, elliptical cross-section as a valid arc, and the add-on reports this explicitly rather than silently applying only part of the requested scale. Sketch-derived Extrude and Revolve blocks currently require exactly four profile points, matching a single hexahedral face; generalising to arbitrary polygon counts would require either decomposing a profile into multiple hex blocks or extending Classy Blocks' shape primitives beyond what is currently used. STL projection validates surface manifoldness and reports non-manifold input, but does not itself repair a non-manifold surface, leaving projection results at surface boundaries dependent on the quality of the imported file. Finally, the add-on does not perform any post-generation mesh quality assessment (skewness, non-orthogonality, or aspect ratio checks); a user must still consult blockMesh's own diagnostic output or a separate tool such as checkMesh for that analysis.

4.5 Future Work

Natural extensions of this work include generalising the sketch tool beyond four-point profiles by decomposing arbitrary polygons into multiple hexahedral blocks automatically; extending non-uniform scale support to arc-based shapes by substituting spline edges for arcs where a user's scaling makes a true circular arc impossible; wiring STL projection through to OpenFOAM's native searchableSurface mechanism for every supported block type rather than the subset currently covered; and surfacing basic mesh quality metrics directly in the Blender panel after a successful blockMesh run, so that a user can identify a poor-quality region without leaving the modelling environment to run checkMesh separately.

5 Conclusion

This report has described a Blender add-on, developed under the FOSSEE OpenFOAM GUI internship at IIT Bombay, that replaces hand-authored blockMeshDict construction with a visual, interactive workflow built on the open-source Classy Blocks library. The add-on supports typed primitives, an interactive sketch-to-3D workflow via Extrude and Revolve, STL/terrain surface projection, and a single, inspectable pipeline from geometry to a reloaded, meshed result inside Blender. Three pieces of computational geometry — transform decomposition, profile winding normalisation, and surface projection — were identified as correctness-critical, formalised, implemented, and covered by end-to-end tests that assert on the actual meshed output rather than only an intermediate specification. Development surfaced five real defects, each resolved and each, in different ways, reinforcing the same lesson: for a geometry-generation tool, correctness has to be verified against the geometry that is actually produced, not only against the data structures that describe an intention to produce it. The result is a tool that keeps the underlying OpenFOAM blockMeshDict and blockMesh model fully visible and inspectable throughout, while removing the most tedious and error-prone parts of authoring it by hand.

References

Fossee. (2026). OpenFOAM GUI: Classy Blocks integration for Blender [Software]. https://github.com/bchbenjar

Classy Blocks contributors. (n.d.). Classy Blocks: A Python library for parametric, structured, multi-block meshing [Software]. https://github.com/damogranlabs/classy_blocks

OpenCFD Ltd. (n.d.). blockMesh. OpenFOAM User Guide. https://www.openfoam.com/documentation/user-guide/4-mesh-generation-and-conversion/4.3-mesh-generation-with-the-blockmesh-utility

Blender Foundation. (n.d.). Blender Python API documentation. https://docs.blender.org/api/current/

Möller, T., & Trumbore, B. (1997). Fast, minimum storage ray-triangle intersection. Journal of Graphics Tools, 2(1), 21–28.

Weisstein, E. W. (n.d.). Pappus's centroid theorem. MathWorld—A Wolfram Web Resource. https://mathworld.wolfram.com/PappussCentroidTheorem.html

OpenCFD Ltd. (n.d.). Mesh generation with the blockMesh utility: Grading. OpenFOAM User Guide. https://www.openfoam.com/documentation/user-guide/4-mesh-generation-and-conversion/4.3-mesh-generation-with-the-blockmesh-utility

The pytest development team. (n.d.). pytest documentation. https://docs.pytest.org/

Appendix A: Illustrative Generated blockMeshDict Excerpt

The following excerpt, reproduced from an add-on-generated case, illustrates the structure of blockMeshDict the tool emits for a single box block with a projected top face and per-edge grading, corresponding to the pipeline described in Section 4.6. Vertex coordinates and cell counts have been abbreviated for brevity.

```
convertToMeters 1;

vertices
(
    (0 0 0)   (1 0 0)   (1 1 0)   (0 1 0)
    (0 0 1)   (1 0 1)   (1 1 1)   (0 1 1)
);

blocks
(
    hex (0 1 2 3 4 5 6 7) ClassyBox (10 10 10)
        simpleGrading (1 1 1)
);

boundary
(
    terrainTop
    {
        type wall;
        faces ((4 5 6 7));
```

```
    }
);
```

The terrainTop patch's face is written using OpenFOAM's own vertex indexing convention; when STL projection is configured for this face (Section 3.4), the corresponding vertices are either replaced with their ray-projected positions before this file is written, or, where the native projection path of Section 5.3 is used, the block is instead written referencing a searchableSurface geometry entry that OpenFOAM's own blockMesh resolves against the imported STL directly at mesh-generation time.

DISCLAIMER: This project reproduces the results from an existing work, which has been acknowledged in the report. Any query related to the original work should not be directed to the contributor of this project.