FOSSEE, IIT Bombay
OpenFOAM GUI Project
June 5, 2025

Development of a Graphical Interface for SnappyHexMesh Dictionary Generation in OpenFOAM

Kumar Ashmit Ranjan
Dual Degree - Aerospace Engineering
IIT Kharagpur

Acknowledgement

I'm truly grateful to **Mr. Diptangshu Dey** and **Mr. Rajdeep Adak**, and **Prof. Chandan Bose** for their constant support and thoughtful mentorship throughout my internship on the OpenFOAM GUI Project at FOSSEE, IIT Bombay. Their deep knowledge of open-source development not only guided me through the technical aspects of the project but also helped me grow as a learner. Their patience, clarity, and encouragement made a big difference at every step.

I especially appreciate their understanding during my academic exams and tests. Balancing coursework with project deadlines was challenging at times, but their flexibility and support helped me manage both without feeling overwhelmed.

A heartfelt thank you as well to the entire FOSSEE team for creating such a welcoming and collaborative environment. Being part of a community that's so passionate about free and open-source software was inspiring and made this internship a truly enriching experience.

This report reflects many conversations, corrections, and small moments of learning, each made possible by the generous guidance of everyone I've mentioned. I'm genuinely thankful for the experience and the impact it has had on both my academic and professional journey.

Contents

List of Figures

1 Introduction

1.1 The SnappyHexMesh Challenge

SnappyHexMesh is a powerful mesh generation utility in OpenFOAM that creates high-quality hexahedral-dominant meshes through a multi-stage process: background mesh refinement, geometry snapping, and boundary layer addition. However, configuring it requires authoring a complex `snappyHexMeshDict` file with numerous interdependent parameters arranged in a deeply nested structure.

This configuration complexity creates several practical challenges:

- **Complex Parameter Relationships**: Many parameters have interdependencies that aren't immediately obvious from the dictionary syntax
- **Error-Prone Manual Editing**: Text editing offers minimal validation, leading to syntax errors and invalid parameter combinations
- **Difficult Troubleshooting**: When mesh generation fails, determining which parameter combination caused the issue is time-consuming
- **Steep Learning Curve**: New users must simultaneously learn the dictionary structure and understand the effect of numerous parameters

The Venturial SnappyHexMesh GUI directly addresses these issues by providing a structured interface that represents parameter relationships visually while maintaining full compatibility with OpenFOAM's underlying dictionary format.

1.2 Implementation Objectives

The development of the SnappyHexMesh dictionary UI was guided by several key technical objectives:

- **Parameter Organization**: Group related parameters logically to better represent their functional relationships rather than just their position in the dictionary hierarchy
- **Real-time Validation**: Implement validation logic that prevents invalid parameter combinations before they cause mesh generation failures
- **Syntax Generation**: Create a robust system to generate properly formatted dictionary syntax regardless of parameter complexity
- **Dynamic UI**: Develop contextual interface elements that adapt to parameter selections, showing only relevant options
- **Integrated Documentation**: Embed parameter documentation directly in the interface via tooltips to reduce constant referencing of external materials
- **Blender Integration**: Leverage Blender's Python API and property system while ensuring the UI remains focused on OpenFOAM-specific functionality

2 Design and Architecture of the GUI Tool

2.1 Technical Strategy

The SnappyHexMesh GUI implements a specialized version of the Model-View-Controller pattern specifically tailored to Blender's property and operator system:

- **Model**: Implemented using Blender's PropertyGroup classes that store and validate parameter data
- **View**: Created using Blender's Panel system with custom UI list implementations for collections
- **Controller**: Built with Blender Operators that modify property data in response to UI events

This architecture leverages Blender's built-in property system to handle undo/redo functionality and property validation without requiring custom implementations of these complex features.

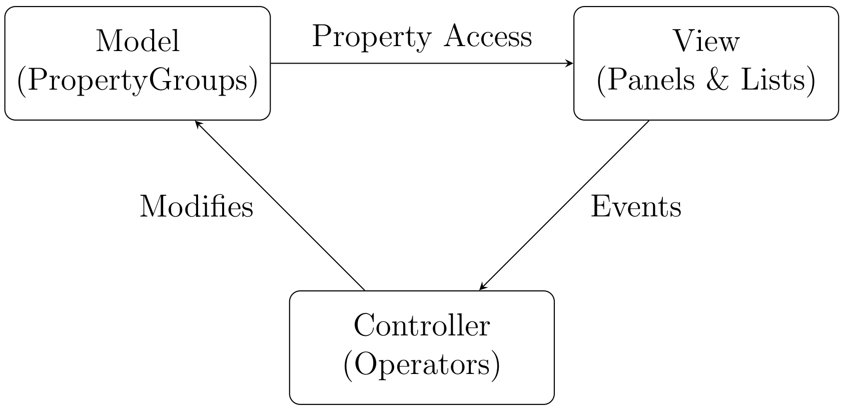


Figure 1: Architecture of the SnappyHexMesh GUI

2.2 Property System Implementation

The implementation represents SnappyHexMesh's complex dictionary structure using Blender's PropertyGroup system. A key design challenge was handling the deeply nested nature of the SnappyHexMesh dictionary. The implementation uses a combination of:

- **Top-level control flags** as direct Scene properties for major sections
- **PropertyGroups** for logically related parameters
- **CollectionProperty** for lists of similar items (features, regions, etc.)

Here's an example of how mesh quality parameters are implemented:

```
    class MeshQualityProperties(PropertyGroup):
        """Standard mesh quality settings"""

        # External dictionary inclusion option
        includeMeshQualityDict: BoolProperty(
            name="Use External Dictionary",
            description="Include external mesh quality dictionary file",
            default=False
        )

        meshQualityDictPath: StringProperty(
            name="Mesh Quality Dict Path",
            description="Path to external mesh quality dictionary file",
            default="meshQualityDict"
        )

        # Primary quality metrics
        maxNonOrtho: FloatProperty(
            name="Max Non-Orthogonality",
            description="Maximum non-orthogonality allowed. 0=orthogonal, 90=bad. Values over 70-80 may lead to robustness issues.",
            default=65.0,
            min=0.0,
            max=180.0
        )

        maxBoundarySkewness: FloatProperty(
            name="Max Boundary Skewness",
            description="Maximum boundary face skewness allowed. Lower is better. Values over 4-5 may affect stability.",
            default=20.0,
            min=0.0,
            soft_max=20.0
        )

        # Additional constraints
        minVol: FloatProperty(
            name="Minimum Volume",
            description="Minimum cell volume allowed",
            default=1e-13,
            min=0.0,
        )

        minTwist: FloatProperty(
            name="Minimum Twist",
            description="Minimum face twist allowed",
            default=0.02,
            min=0.0,
            max=1.0
        )

        # Relaxed settings section - properties for relaxed quality
           constraints
        relaxed_enabled: BoolProperty(
            name="Use Relaxed Settings",
            description="Enable relaxed mesh quality criteria after
               specified iterations",
            default=False
        )

        relaxed_maxNonOrtho: FloatProperty(
            name="Relaxed Max Non-Orthogonality",
            description="Relaxed maximum non-orthogonality allowed
               after iterations",
            default=75.0,
            min=0.0,
            max=180.0
        )
```

Listing 1: Implementation of Mesh Quality Properties

For collections of items like refinement regions or features, the implementation uses CollectionProperty with custom PropertyGroup classes:

```
    class FeatureEdge(PropertyGroup):
        """Feature edge settings for snappyHexMesh"""
        file: StringProperty(
            name="eMesh File",
            description="Path to feature edge mesh file (.eMesh)",
            default=""
        )

        level: IntProperty(
            name="Refinement Level",
            description="Level of refinement along feature edges",
            default=0,
            min=0,
            max=10
        )

    # Register the collection in the Scene
    bpy.types.Scene.cast_features = CollectionProperty(type=
        FeatureEdge)
    bpy.types.Scene.cast_features_index = IntProperty()
```

Listing 2: Collection Implementation for Features

2.3 File Organization

The SnappyHexMesh GUI implementation is organized into multiple Python modules with specific responsibilities:

```
venturial/
├── models/
│   └── snappyhexmesh/
│       ├── castellated_operators.py .................... Castellated mesh controls
│       ├── dictionary_operators.py ................ Dictionary generation operators
│       ├── file_operators.py ................................. File I/O operations
│       ├── geometry_operators.py ............................. Geometry handling
│       ├── layer_operators.py ............................. Layer addition controls
│       ├── mesh_quality_operators.py .................. Quality control parameters
│       ├── snap_operators.py ................................. Snapping controls
│       ├── snappydict_writer.py ......................... Dictionary writing utility
│       ├── tooltips.py .................................... Parameter descriptions
│       └── tooltip_updater.py ....................... Dynamic tooltip management
├── views/
│   └── mainpanel/
│       └── meshing_tools/
│           └── snappyhexmesh.py ............................ UI panel definitions
└── __init__.py ................................ Registration and property definitions
```

The directory layout enforces a clear separation of responsibilities. All of the SnappyHexMesh logic lives under `models/snappyhexmesh`: each Python module there encapsulates a distinct aspect of the mesh-generation workflow (e.g., geometry setup, castellated mesh controls, snapping routines, layer addition, mesh-quality checks, dictionary assembly, and file I/O). By contrast, the GUI definitions—i.e., the panels and widget layouts that expose those parameters to the user—are contained in `views/mainpanel/meshing_tools/snappyhexmesh.py`. Finally, the top-level `__init__.py` file handles registration and property definitions so that the entire SnappyHexMesh GUI integrates smoothly into the larger application framework.

2.4 Data Flow and Validation

The data flow within the system follows a consistent pattern:

1. User interacts with UI controls in panel or list components
2. Events trigger operators that modify property data

3. Property updates may trigger validation callbacks (currently not completely implemented)
4. UI is refreshed to reflect updated property state
5. Dictionary generation functions convert properties to OpenFOAM syntax when requested

Validation occurs at multiple levels:

```python
# 1. Property-level validation using min/max constraints
expansionRatio: FloatProperty(
    name="Expansion Ratio",
    description="Expansion ratio for layer mesh",
    default=1.2,
    min=1.0,  # Physical constraint: must expand, not contract
    max=2.0   # Practical upper limit for stable layers
)

# 2. Update callback validation for parameter relationships
def update_layer_thickness(self, context):
    """Ensure layer thickness parameters are consistent"""
    scene = context.scene

    # First layer must be thicker than final layer when using
       expansion ratio > 1
    if scene.firstLayerThickness <= scene.finalLayerThickness
       and scene.expansionRatio > 1.0:
        scene.firstLayerThickness = scene.finalLayerThickness *
            scene.expansionRatio

    # Minimum thickness must be less than both first and final
       layer
    min_thickness = min(scene.firstLayerThickness, scene.
       finalLayerThickness)
    if scene.minThickness >= min_thickness:
        scene.minThickness = min_thickness * 0.5

# Register property with update callback
bpy.types.Scene.firstLayerThickness = FloatProperty(
    name="First Layer Thickness",
    description="Thickness of first layer next to surface",
    default=0.1,
    min=0.001,
    update=update_layer_thickness
)

# 3. Export-time validation in dictionary generation
def validate_before_export(scene):
    """Final validation before dictionary export"""
    errors = []

    if scene.addLayers and len(scene.layers) == 0:
        errors.append("Layer addition enabled but no layers
           defined")

    if scene.snap and scene.nSolveIter < 10:
        errors.append("nSolveIter value too low for reliable
           snapping")

    return errors
```

Listing 3: Multi-level Validation Example

3 SnappyHexMesh Dictionary Structure

3.1 Dictionary Format Analysis

The SnappyHexMesh dictionary has a complex hierarchical structure with several major sections:

```
    // Control flags for major phases
    castellatedMesh true;
    snap true;
    addLayers true;

    // Geometry definition section
    geometry { ... };

    // Castellated mesh controls
    castellatedMeshControls { ... };

    // Snapping controls
    snapControls { ... };

    // Layer addition controls
    addLayersControls { ... };

    // Mesh quality settings
    meshQualityControls { ... };

    // Miscellaneous settings
    mergeTolerance 1e-6;
```

Listing 4: Top-Level Dictionary Structure

Each section contains numerous parameters with varying types:

- Boolean flags (true/false)
- Numeric values (integers and floating point)
- 3D vectors represented as (x y z)
- Nested sub-dictionaries
- Lists of items (with parentheses instead of braces)
- Lists of dictionaries (for features, regions, etc.)

This variety of data types presents a significant challenge for both UI representation and dictionary generation.

3.2 Mapping Dictionary to UI Elements

A key design task was mapping different dictionary elements to appropriate UI controls:

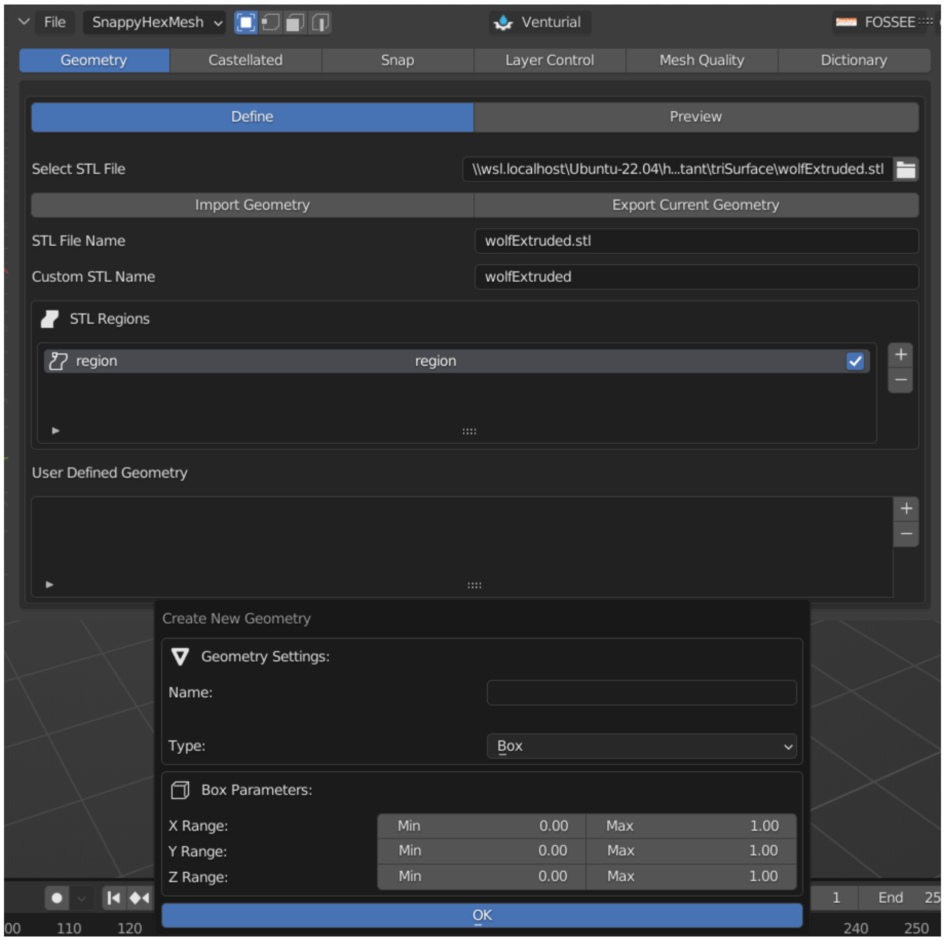
Dictionary Element	**UI Component**	**Example Parameters**
Boolean flags	Checkbox	`castellatedMesh`, `snap`, `addLayers`
Integer values	Integer field with slider	`nSmoothPatch`, `nSolveIter`
Float values	Float field with slider	`tolerance`, `expansionRatio`
Vectors	Three numeric fields	`locationInMesh`, box `min/max`
String identifiers	Text field	file paths, region names
Enumerations	Dropdown or radio buttons	refinement modes
Lists of dictionaries	Custom UIList	`features`, `refinementRegions`
Nested dictionaries	Collapsible panels	`layers`, `relaxed` settings

Table 1: Mapping Dictionary Elements to UI Components

3.3 Dictionary Sections in Detail

The implementation handles each major section of the dictionary with specialized UI components and property groups:

3.3.1 Geometry Section



Select STL File: \\wsl.localhost\Ubuntu-22.04\h...tant\triSurface\wolfExtruded.stl

STL File Name: wolfExtruded.stl

Custom STL Name: wolfExtruded

region region ✓

Name:

Type: Box

X Range: Min 0.00 Max 1.00

Y Range: Min 0.00 Max 1.00

Z Range: Min 0.00 Max 1.00

Figure 2: Geometry Tab

The Geometry Section in Venturial provides comprehensive tools for creating, importing, managing, and visualizing geometric elements used in OpenFOAM simulations. This section is essential for defining the physical domains and boundaries upon which the computational fluid dynamics (CFD) calculations will be performed.

Overview The Geometry Section serves as the foundation for mesh generation in OpenFOAM, allowing users to define and manipulate the geometric components that will be used during the snappyHexMesh process. This interface bridges the gap between CAD models and computational domains, providing visual feedback and intuitive controls for geometry definition.

Interface Organization The Geometry Section is organized into two main tabs:

Define Tab The Define tab provides tools for:

- Importing external STL files from CAD software
- Creating primitive geometrical shapes directly within Blender
- Managing STL regions and their custom naming
- Exporting geometry for use in OpenFOAM

Preview Tab The Preview tab offers:

- Visualization of the generated geometry dictionary entries
- A formatted display of how geometry will be represented in the snappyHexMeshDict

Geometry Types The Geometry Section supports multiple geometry definition methods:

External STL Files

- Import pre-created geometry from CAD software
- Associate custom names with STL files for easier identification
- Define and manage STL regions for fine-grained boundary condition application
- Control visibility of imported STL geometry

Primitive Shapes Users can create and configure basic geometric primitives directly within the interface to create regions to set region-specific parameters.

Box (searchableBox)

- Define minimum and maximum coordinates (X, Y, Z)

Sphere (searchableSphere)

- Define center coordinates (X, Y, Z)
- Adjust radius

Region Management The STL region management system allows:

- Custom naming of regions for boundary condition application
- Enabling/disabling regions as needed for simulation setup
- Mapping between original STL region names and user-defined names

Dictionary Preview The preview functionality shows:

- Formatted representation of geometry definitions as they will appear in OpenFOAM dictionaries
- Live updates as geometry parameters are modified

Integration with OpenFOAM The Geometry Section automatically generates the appropriate entries for:

- The `snappyHexMeshDict` geometry subdictionary
- Property definitions for STL files and primitive shapes
- Region mapping for boundary conditions
- Proper formatting for OpenFOAM compatibility

Workflow Typical workflow in the Geometry Section includes:

1. Either importing external STL files or creating primitive shapes
2. Defining and naming regions for boundary condition application
3. Previewing the generated dictionary entries
4. Proceeding to other settings once geometry is properly defined

The Geometry Section serves as the first step in the meshing process, providing the foundation upon which all subsequent meshing operations will be performed.

```
    geometry
    {
        // STL file with named regions
        "propeller.stl"
        {
            type triSurfaceMesh;
            name propeller;
            regions
            {
                blade { name blade; }
                hub { name hub; }
            }
        }

        // Primitive shape for refinement
        refinementBox
        {
            type box;
            min (1 -0.5 -0.5);
            max (3 0.5 0.5);
        }
    }
```

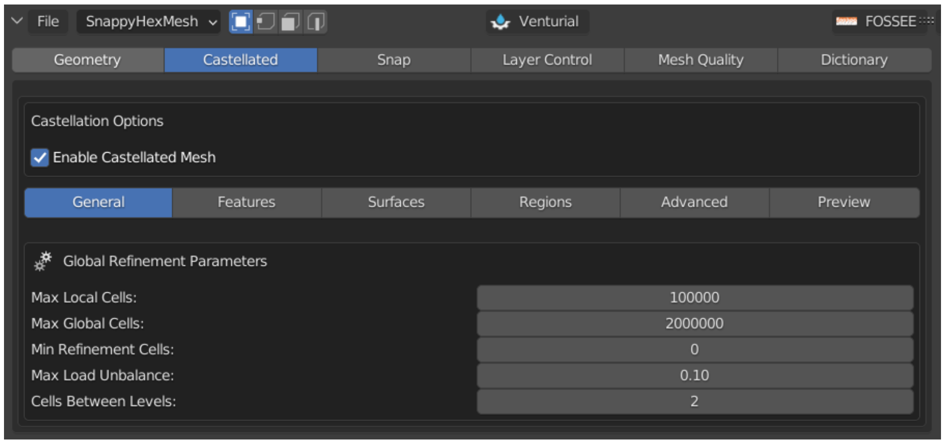
Listing 5: Geometry Section Example

This is implemented using a combination of file selection controls for STL files and specialized property groups for primitive shapes:

```
    class GeometryItem(PropertyGroup):
        """Geometric item for refinement or boundaries"""
        name: StringProperty(
            name="Name",
            description="Identifier for this geometry item",
            default="geometry"
        )

        type: EnumProperty(
            name="Type",
            description="Type of geometry primitive",
            items=[
                ('box', "Box", "Rectangular box defined by min/max
                   points"),
                ('sphere', "Sphere", "Sphere defined by center and
                   radius"),
                ('cylinder', "Cylinder", "Cylinder defined by points
                    and radius")
            ],
            default='box'
        )

        # Box parameters
        min_x: FloatProperty(name="Min X", default=0.0)
        min_y: FloatProperty(name="Min Y", default=0.0)
        min_z: FloatProperty(name="Min Z", default=0.0)
        max_x: FloatProperty(name="Max X", default=1.0)
        max_y: FloatProperty(name="Max Y", default=1.0)
        max_z: FloatProperty(name="Max Z", default=1.0)

        # Sphere parameters
        center_x: FloatProperty(name="Center X", default=0.0)
        center_y: FloatProperty(name="Center Y", default=0.0)
        center_z: FloatProperty(name="Center Z", default=0.0)
        radius: FloatProperty(name="Radius", default=1.0, min=0.001)
```

Listing 6: Geometry UI Implementation

3.3.2 Castellated Mesh Controls



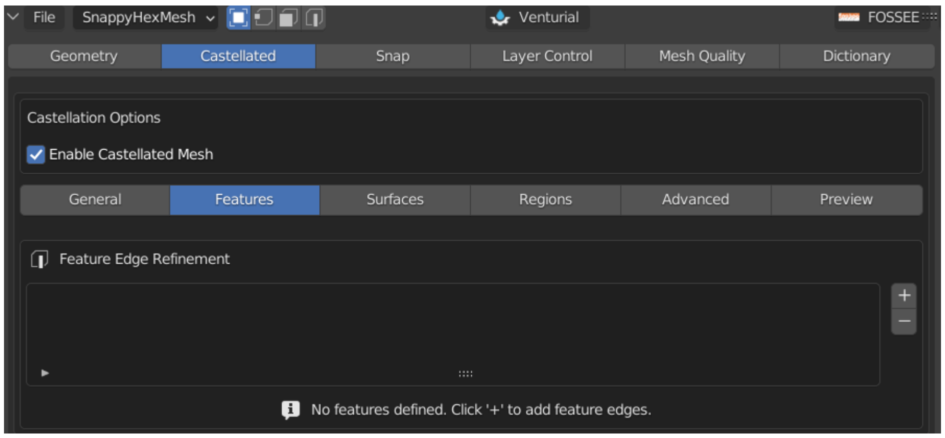
Castellation Options

Enable Castellated Mesh

Global Refinement Parameters

Max Local Cells:	100000
Max Global Cells:	2000000
Min Refinement Cells:	0
Max Load Unbalance:	0.10
Cells Between Levels:	2

Figure 3: Castellated-General Tab



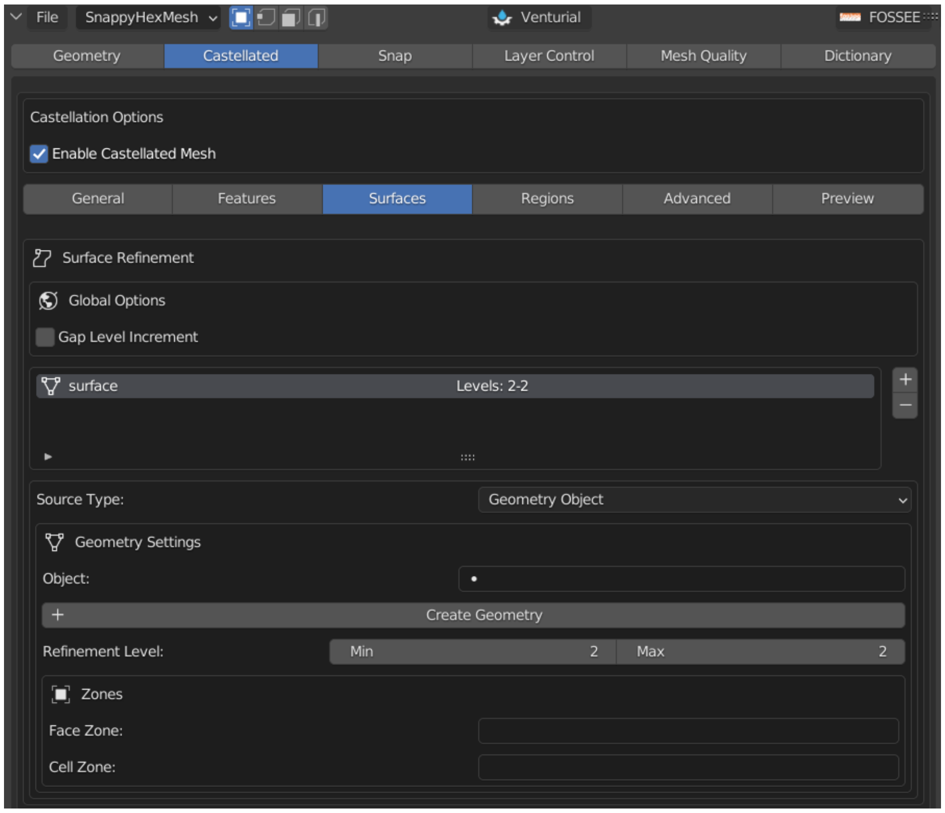
Castellation Options

Enable Castellated Mesh

Feature Edge Refinement

No features defined. Click '+' to add feature edges.

Figure 4: Castellated-Features Tab



Castellation Options

✓ Enable Castellated Mesh

Surface Refinement

Global Options

Gap Level Increment

surface Levels: 2-2

Source Type: Geometry Object

Geometry Settings

Object:

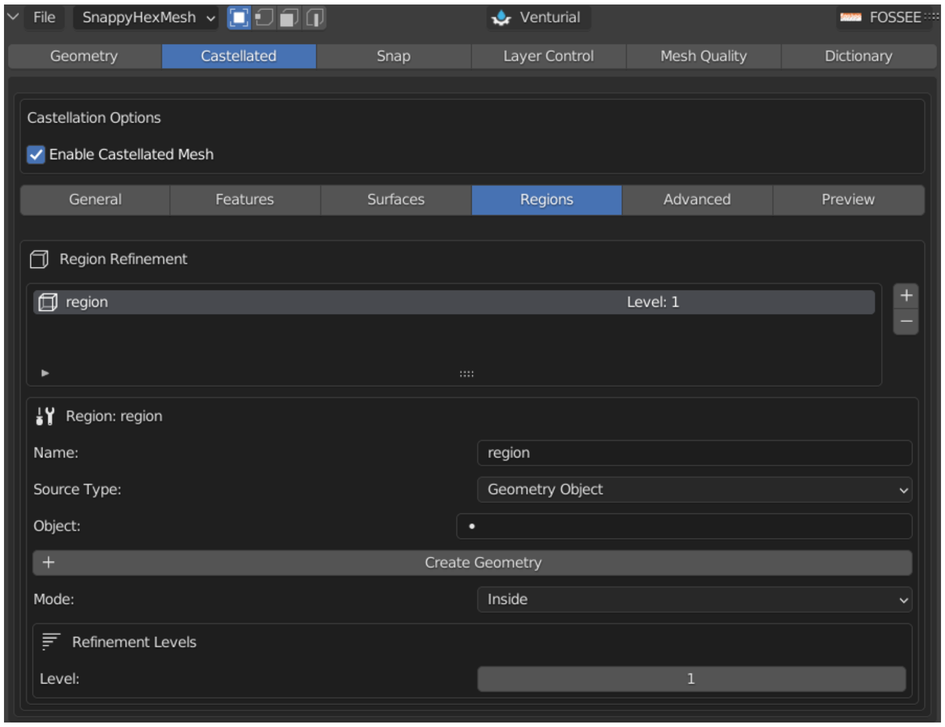
Refinement Level: Min 2 Max 2

Zones

Face Zone:

Cell Zone:

Figure 5: Castellated-Surfaces Tab



Castellation Options

✓ Enable Castellated Mesh

Region Refinement

region Level: 1

Region: region

Name: region

Source Type: Geometry Object

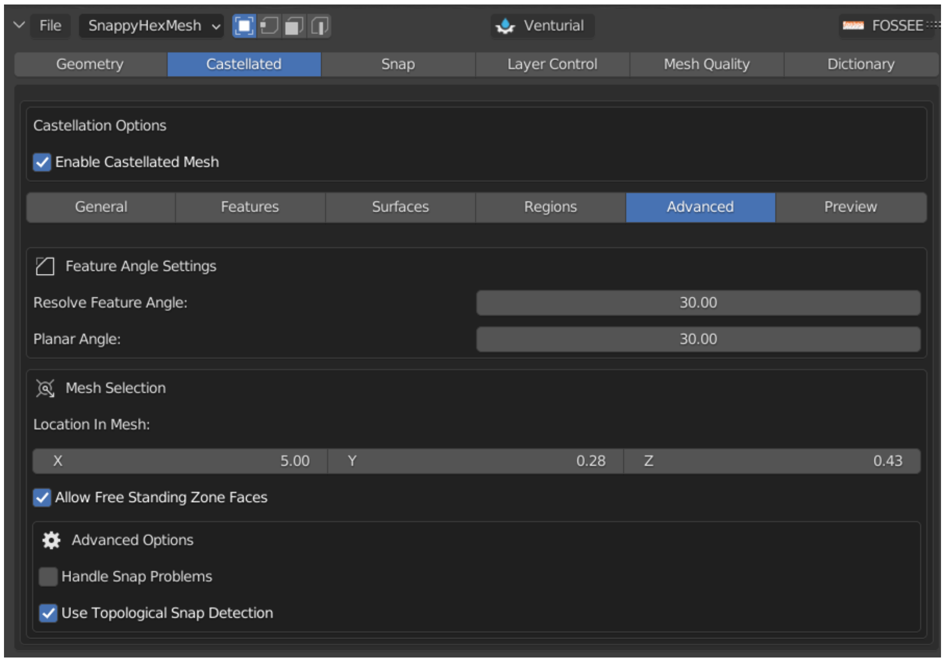
Object:

Mode: Inside

Refinement Levels

Level: 1

Figure 6: Castellated-Regions Tab


File SnappyHexMesh Venturial FOSSEE
Geometry | Castellated | Snap | Layer Control | Mesh Quality | Dictionary

Castellation Options
✓ Enable Castellated Mesh

General | Features | Surfaces | Regions | Advanced | Preview

Feature Angle Settings
Resolve Feature Angle: 30.00
Planar Angle: 30.00

Mesh Selection
Location In Mesh:
X 5.00 Y 0.28 Z 0.43
✓ Allow Free Standing Zone Faces

Advanced Options
☐ Handle Snap Problems
✓ Use Topological Snap Detection

Figure 7: Castellated-Advanced Tab

The castellated mesh generation represents the first phase of the snappyHexMesh process, where the initial block mesh is refined according to user-specified criteria to capture geometry features. This section describes the parameters that control the castellated meshing phase within Venturial.

Basic Controls The castellated mesh phase can be enabled or disabled using the main toggle switch. When enabled, the following global parameters control the overall mesh refinement process:

maxLocalCells Maximum number of cells per processor during refinement. Constrains memory usage for parallel operation. Default: 100000.

maxGlobalCells Maximum total cells in the mesh after refinement. Prevents excessive refinement. Default: 2000000.

minRefinementCells Minimum threshold of cells to be refined before refinement stops. Useful for limiting over-refinement in small features. Default: 0.

maxLoadUnbalance Maximum allowable load imbalance between processors. Values between 0.0-1.0, with lower values enforcing better balance. Default: 0.1.

nCellsBetweenLevels Required number of buffer cells between refinement levels. Controls smoothness of transition between different refinement levels. Default: 2.

Feature Edge Refinement Feature edge refinement allows explicit control over mesh resolution along sharp geometry edges:

Features Collection Interface for defining multiple feature edge sets for refinement using `.eMesh` files. The dictionary used for generating these files using the surfaceFeatures operation of OpenFOAM is not yet implemented.

Refinement Modes Each feature edge can use one of three refinement approaches:

- **Uniform**: Apply constant refinement level to all features
- **Single Distance**: Define one distance-level pair to control gradual refinement from edge
- **Multiple Distances**: Define multiple distance-level pairs for fine control over refinement gradient

Distance-Level Pairs For gradient refinement, each pair defines:

- **Distance**: Distance from feature edge (in mesh units)
- **Level**: Refinement level at that distance

Surface Refinement Controls the cell refinement along geometry surfaces:

Gap Level Increment Additional refinement level to apply in narrow gaps between surfaces (boolean toggle with associated increment value).

Surface Sources Surface refinement can be applied to:

- Geometry objects from the current Blender scene
- External STL files imported into the interface

Refinement Levels Each surface has:

- **Minimum Level**: Base refinement level guaranteed across entire surface
- **Maximum Level**: Additional refinement applied to features and high-curvature areas

Surface Regions Each surface can define specific regions with:

- Custom name for identification
- Region-specific min/max refinement levels
- Optional patch information for boundary condition specification:
  - Patch type (patch, wall, symmetry, empty, wedge)
  - Optional patch group for organization
- Cell and face zone specifications for specialized physics regions

Volume Refinement Controls refinement in specific regions of the domain volume:

Region Definition Each region has:

- Unique name identifier
- Source geometry (Blender object or STL file)

Refinement Modes Two approaches to volumetric refinement:

- **Inside Mode**: Applies uniform refinement to all cells inside a closed volume
- **Distance Mode**: Refines cells based on distance from surface, with options for:
  - Single distance-level pair
  - Multiple distance-level pairs for gradual refinement transitions

Advanced Settings Fine control over feature detection and mesh selection:

resolveFeatureAngle Angle threshold for feature detection. Features with angle exceeding this value get refined. Default: 30 degrees.

planarAngle Angle threshold for treating surfaces as planar (for patch splitting). Default: 30 degrees.

locationInMesh Point coordinates (X, Y, Z) inside the desired fluid region. Used to mark cells for keeping vs. removing. Critical for correct domain selection.

allowFreeStandingZoneFaces Allow mesh faces that are not connected to cells. Typically enabled to prevent issues with specialized zones. Default: True.

handleSnapProblems Keep cells likely to cause snapping problems. Default: False.

useTopologicalSnapDetection Use topological test for cells to be squashed. If disabled, uses geometric test instead. Default: True.

The interface organizes these controls across five tabs: General, Features, Surfaces, Regions, and Advanced, with a sixth Preview tab showing the generated dictionary entries. This organization provides a logical workflow for setting up the castellated mesh phase, starting with basic parameters and progressively configuring more specialized refinement options.

```
    castellatedMeshControls
    {
        maxLocalCells 100000;
        maxGlobalCells 2000000;
        minRefinementCells 10;
        maxLoadUnbalance 0.1;

        features
        (
            {
                file "propeller.eMesh";
                level 2;
            }
        );

        refinementSurfaces
        {
            propeller
            {
                level (1 2);
                regions
                {
                    blade { level (2 3); }
                    hub { level (1 2); }
                }
            }
        }

        refinementRegions
        {
            refinementBox { mode inside; level 2; }
        }

        locationInMesh (0 0 0);
    }
```

Listing 7: Castellated Mesh Controls Example

The UI implements this with basic input fields for scalar values and custom list controls for features and refinement regions:

```python
    def draw_castellated_general(self, context):
        """Draw general castellated mesh controls"""
        layout = self.layout
        scene = context.scene

        box = layout.box()
        col = box.column(align=True)

        # Basic parameters
        col.prop(scene, "maxLocalCells")
        col.prop(scene, "maxGlobalCells")
        col.prop(scene, "minRefinementCells")
        col.prop(scene, "maxLoadUnbalance")
        col.prop(scene, "nCellsBetweenLevels")

        # Location in mesh with vector components
        box = layout.box()
        box.label(text="Location In Mesh")
        row = box.row()
        row.prop(scene, "locationInMesh_x")
        row.prop(scene, "locationInMesh_y")
        row.prop(scene, "locationInMesh_z")
```

Listing 8: Castellated Mesh UI Elements

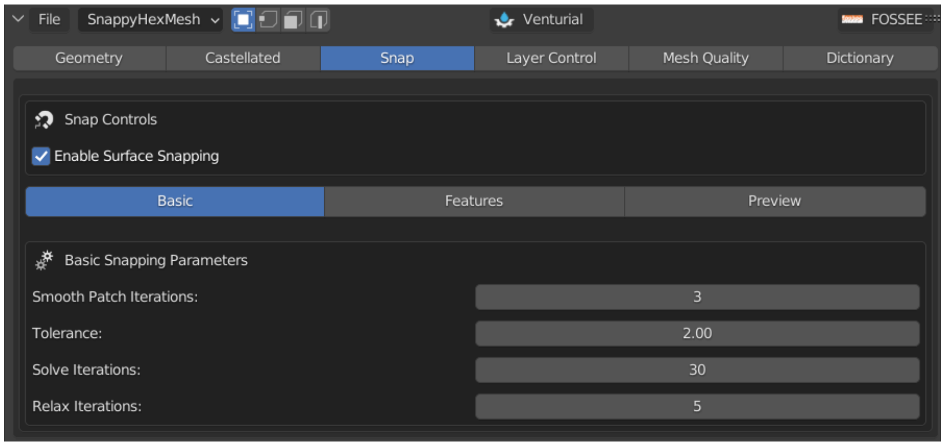
3.3.3 Feature and Region Lists

Custom UIList implementations manage collections of features and refinement regions:

```
    class CAST_UL_features_list(UIList):
        """Display list of feature edges for refinement"""

        def draw_item(self, context, layout, data, item, icon,
           active_data, active_propname):
            feature = item

            # Create multi-column layout for each feature
            row = layout.row(align=True)

            # File path column
            split = row.split(factor=0.6)
            split.prop(feature, "file", text="", emboss=False)

            # Level column
            split.prop(feature, "level", text="Level")

    # Operator to add a new feature to the list
    class VNT_OT_add_feature(Operator):
        """Add a feature edge for refinement"""
        bl_idname = "vnt.add_feature"
        bl_label = "Add Feature"

        def execute(self, context):
            context.scene.cast_features.add()
            context.scene.cast_features_index = len(context.scene.
               cast_features) - 1
            return {'FINISHED'}

    # Operator to remove a feature
    class VNT_OT_remove_feature(Operator):
        """Remove selected feature edge"""
        bl_idname = "vnt.remove_feature"
        bl_label = "Remove Feature"

        def execute(self, context):
            if context.scene.cast_features:
                context.scene.cast_features.remove(context.scene.
                   cast_features_index)
                context.scene.cast_features_index = min(max(0,
                   context.scene.cast_features_index - 1),
                                                      len(context.
                                                         scene.
                                                         cast_features
                                                         ) - 1)
            return {'FINISHED'}
```

Listing 9: Feature List Implementation

3.3.4 Snapping Controls



Enable Surface Snapping ✓

Smooth Patch Iterations:	3
Tolerance:	2.00
Solve Iterations:	30
Relax Iterations:	5

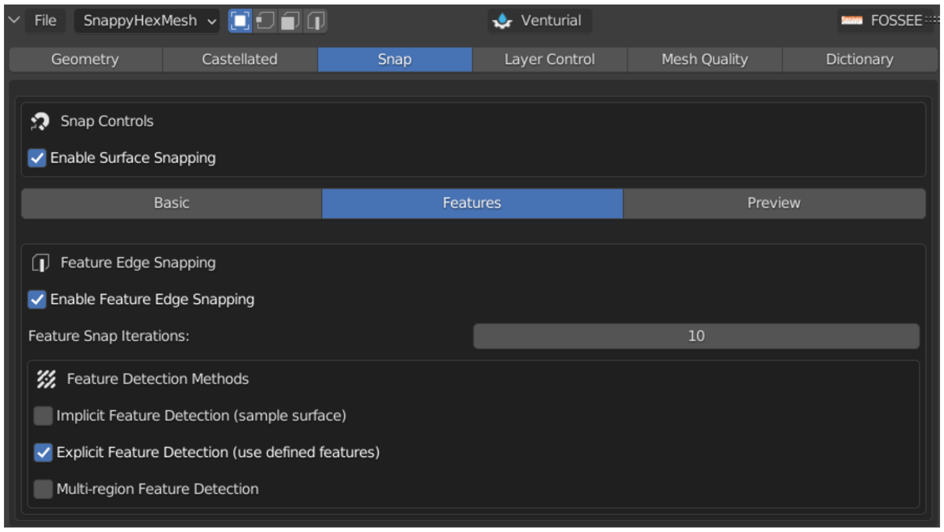
Figure 8: Snap-Basic Tab

The snapping phase of the snappyHexMesh process adjusts the mesh vertices to better conform to the input geometry surfaces. This section provides controls to fine-tune this critical phase where the castellated mesh is modified to more accurately represent the underlying geometry features.

Basic Parameters The snapping process is governed by several fundamental parameters that control the mesh relaxation and surface attraction:

nSmoothPatch Integer specifying the number of patch smoothing iterations performed before finding correspondence to the final surface. Higher values improve mesh quality but may reduce geometric accuracy. Default: 3.

tolerance Maximum relative distance for points to be attracted by surface. Measured relative to local cell size. Values above 1.0 allow attraction from further away; decreasing improves boundary adherence at risk of poor quality cells. Default: 2.0.



Snap Controls

✓ Enable Surface Snapping

Feature Edge Snapping

✓ Enable Feature Edge Snapping

Feature Snap Iterations: 10

Feature Detection Methods

☐ Implicit Feature Detection (sample surface)

✓ Explicit Feature Detection (use defined features)

☐ Multi-region Feature Detection

Figure 9: Snap-Features Tab

nSolveIter Number of mesh displacement relaxation iterations controlling mesh motion and smoothing during snapping. Higher values produce better quality but increase computation time. Default: 30.

nRelaxIter Maximum number of snapping relaxation iterations for surface attraction. Controls how aggressively points move toward surfaces. Default: 5.

Feature Edge Snapping Special attention is given to sharp edges and corners through feature snapping controls:

useFeatureSnap Master toggle enabling feature edge snapping functionality. When enabled, mesh points are attracted to feature edges in addition to surfaces. Default: True.

nFeatureSnapIter Number of iterations specifically for feature edge snapping. This controls how precisely the mesh conforms to sharp edges. Default: 10.

When feature snapping is enabled, three detection methods are available, which can be used individually or in combination:

implicitFeatureSnap Detects feature edges by sampling the surface for sharp angle changes. This method works without explicit feature edge definitions but may miss some features. Default: False.

explicitFeatureSnap Uses the feature edges explicitly defined in the castellatedMeshControls section, requiring proper `.eMesh` files. This provides more precise control over which features are captured. Default: True.

multiRegionFeatureSnap Detects feature edges at the intersection of multiple surface regions. Essential for complex multi-region geometries where different patches meet. Default: False.

Advanced Configuration The interface organizes snapping controls into logical tabs for better workflow:

Basic Tab Contains fundamental smoothing and relaxation parameters that affect the overall snapping behavior.

Features Tab Groups feature edge snapping parameters with their detection methods.

Preview Tab Displays the generated OpenFOAM dictionary entries for immediate feedback on configuration changes.

Integration with OpenFOAM The snapping phase settings are integrated into the snappyHexMeshDict file under the `snapControls` subdictionary. The interface automatically formats all parameters according to OpenFOAM syntax requirements:

```
    snapControls
    {
        nSmoothPatch 3;
        tolerance 2.0;
        nSolveIter 30;
        nRelaxIter 5;

        // Feature handling
        nFeatureSnapIter 10;
        implicitFeatureSnap false;
        explicitFeatureSnap true;
        multiRegionFeatureSnap false;
    }
```

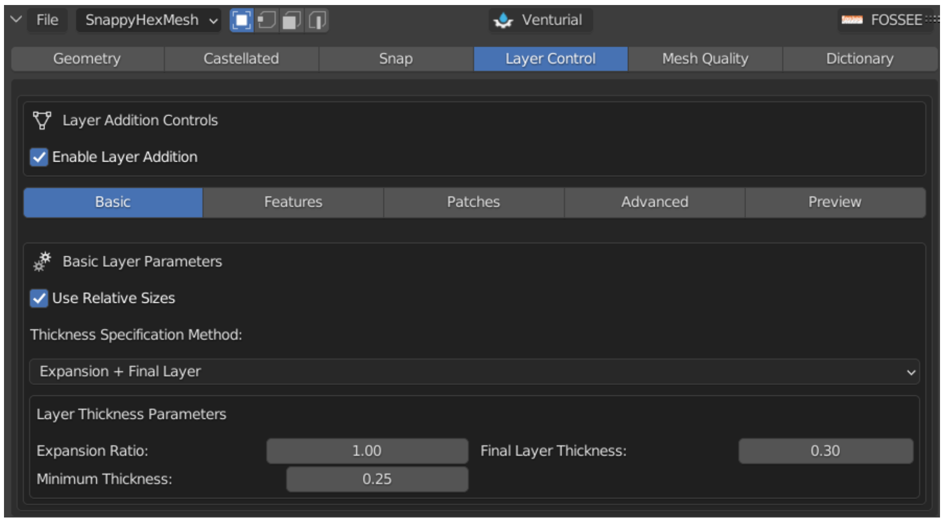
Computational Impact The snapping phase is often the most computationally expensive part of the snappyHexMesh process. Higher values for iterations and tolerance parameters typically improve mesh quality but at the cost of increased processing time. The interface provides direct control over this quality-performance tradeoff through intuitive parameter adjustment.

Workflow Integration The snap controls section interacts with both the castellated mesh phase (which provides the initial mesh and potentially explicit features) and the layer addition phase (which depends on a properly snapped surface mesh). The master toggle for snapping (`snap`) determines whether this phase is included in the overall meshing process, allowing for rapid iteration during case setup.

```python
    def draw_snapping_panel(self, context):
        """Draw snapping controls panel"""
        layout = self.layout
        scene = context.scene

        # Main enable/disable toggle
        row = layout.row()
        row.prop(scene, "snap")

        # Only show controls if snapping is enabled
        if scene.snap:
            box = layout.box()

            col = box.column(align=True)
            col.prop(scene, "nSmoothPatch")
            col.prop(scene, "tolerance")
            col.prop(scene, "nSolveIter")
            col.prop(scene, "nRelaxIter")

            # Feature snapping subsection
            box = layout.box()
            box.label(text="Feature Snapping")

            col = box.column()
            row = col.row()
            row.prop(scene, "implicitFeatureSnap")
            row.prop(scene, "explicitFeatureSnap")

            # Only show multi-region if explicit feature snapping enabled
            if scene.explicitFeatureSnap:
                col.prop(scene, "multiRegionFeatureSnap")
```

Listing 10: Snapping Controls UI

3.3.5 Layer Addition Controls



Layer Addition Controls

✓ Enable Layer Addition

Basic Layer Parameters

✓ Use Relative Sizes

Thickness Specification Method:

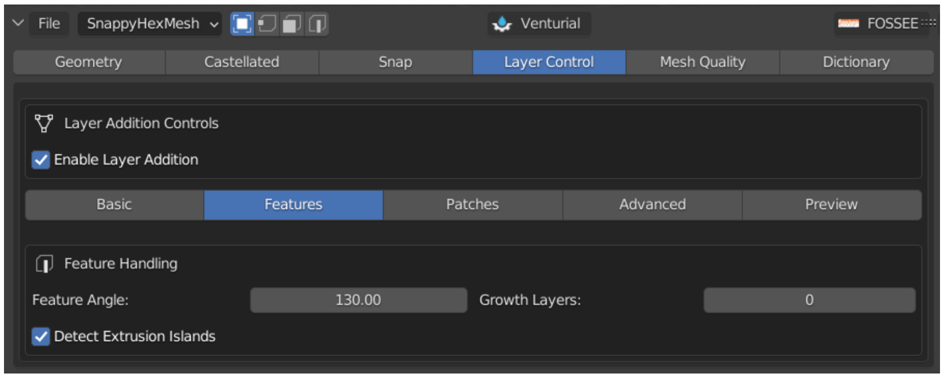
Expansion + Final Layer

Layer Thickness Parameters

Expansion Ratio: 1.00 Final Layer Thickness: 0.30

Minimum Thickness: 0.25

Figure 10: Layer Control-Basic Tab



Layer Addition Controls

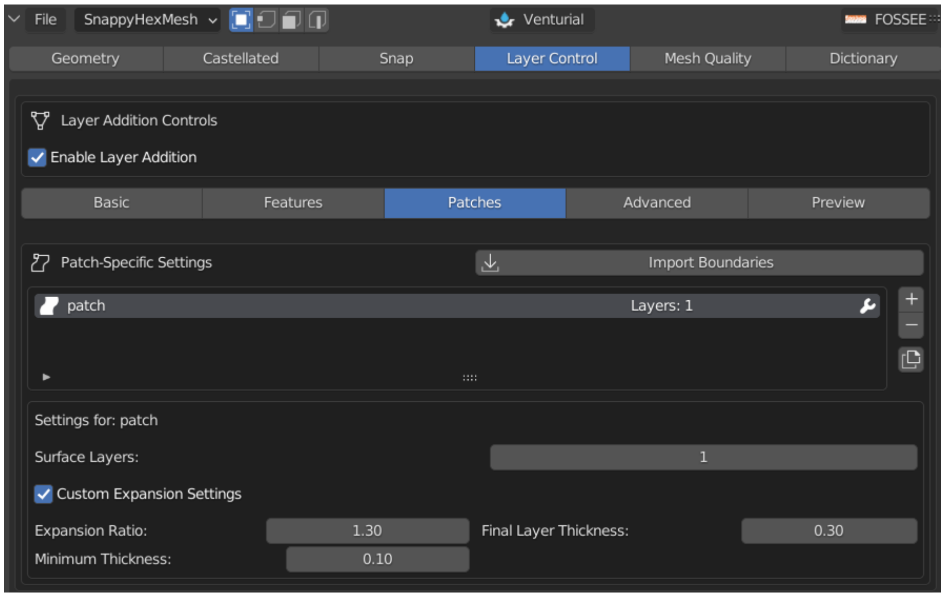
✓ Enable Layer Addition

Feature Handling

Feature Angle: 130.00 Growth Layers: 0

✓ Detect Extrusion Islands

Figure 11: Layer Control-Features Tab



Layer Addition Controls

✓ Enable Layer Addition

Patch-Specific Settings

patch Layers: 1

Settings for: patch

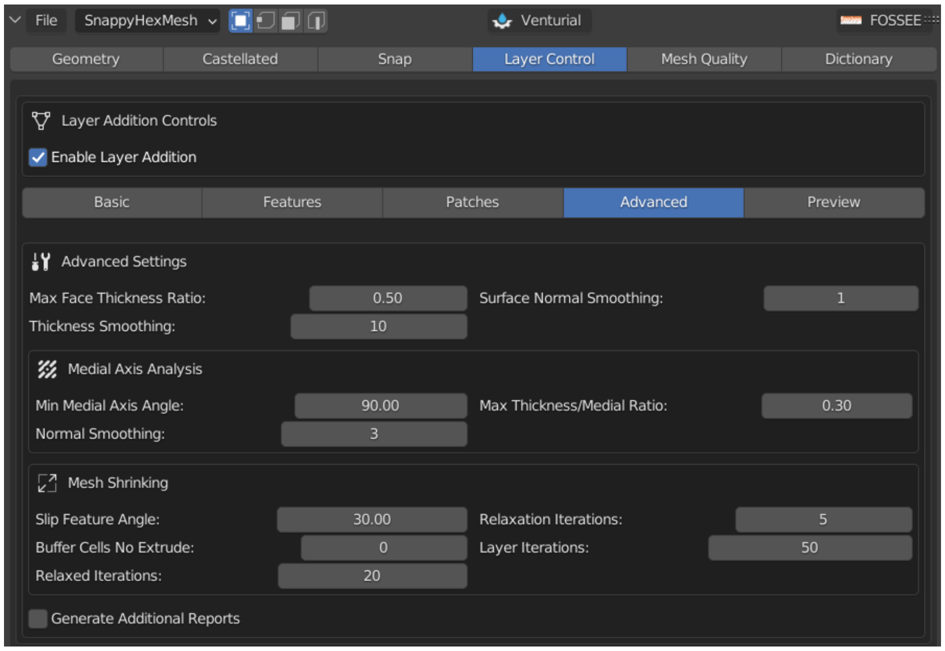
Surface Layers: 1

✓ Custom Expansion Settings

Expansion Ratio: 1.30 Final Layer Thickness: 0.30

Minimum Thickness: 0.10

Figure 12: Layer Control-Patches Tab



Layer Addition Controls

☑ Enable Layer Addition

Advanced Settings

Max Face Thickness Ratio: 0.50
Surface Normal Smoothing: 1
Thickness Smoothing: 10

Medial Axis Analysis

Min Medial Axis Angle: 90.00
Max Thickness/Medial Ratio: 0.30
Normal Smoothing: 3

Mesh Shrinking

Slip Feature Angle: 30.00
Relaxation Iterations: 5
Buffer Cells No Extrude: 0
Layer Iterations: 50
Relaxed Iterations: 20

☐ Generate Additional Reports

Figure 13: Layer Control-Advanced Tab

The layer addition phase represents the final stage of the snappyHexMesh process, where prismatic cell layers are added to boundary surfaces to properly capture boundary layer effects. This section documents the parameters controlling the layer addition process.

Basic Layer Parameters The fundamental controls governing the overall layer addition strategy:

relativeSizes Boolean toggle determining whether layer thicknesses are specified relative to local cell size (true) or as absolute sizes (false). Relative sizing provides more consistent layers across mesh refinement regions. Default: True.

expansionRatio Growth ratio for layer thickness between consecutive layers moving away from the wall. Values typically range from 1.0 (uniform) to 1.5 (aggressive growth). Controls the smooth transition from fine near-wall cells to core mesh. Default: 1.2.

finalLayerThickness Controls the thickness of the layer furthest from the wall, either as a relative multiplier of near-wall cell size (when relativeSizes is true) or as an absolute value. Ensures proper transition to core mesh. Default: 0.7.

minThickness Minimum acceptable thickness for any layer as a fraction of local cell size, preventing excessively thin layers in tight geometries. Default: 0.1.

nGrow Number of layers of cells added in the normal direction where layers terminate. Creates a smoother transition between layered and unlayered regions. Default: 0.

Layer Generation Controls Parameters that control the layer generation algorithm behavior:

nSmoothSurfaceNormals Number of iterations of surface normal smoothing. Higher values create more uniform layer orientation across patch faces. Default: 1.

nSmoothNormals Number of iterations of interior mesh normal smoothing. Improves layer quality in regions with changing geometry. Default: 3.

nSmoothThickness Number of iterations to smooth layer thickness. Higher values help maintain more consistent layer thickness across complex geometries. Default: 10.

maxFaceThicknessRatio Maximum ratio of layer thickness to medial axis distance. Prevents layer collision in narrow regions. Default: 0.5.

maxThicknessToMedialRatio Maximum ratio of layer thickness to medial distance. Another control to prevent layers from colliding in narrow sections. Default: 0.3.

Feature Controls Parameters for controlling layer behavior near geometric features:

featureAngle Angle threshold (in degrees) for detecting sharp features where layer generation requires special handling. Smaller angles detect more features. Default: 30.

slipFeatureAngle Angle above which boundary layers ”slip” (are not generated) when relativeSizes is true. Prevents layer generation on sharp features. Default: 30.

nRelaxIter Number of relaxation iterations when generating layers, smoothing out potential mesh distortions. Default: 5.

Region-Specific Controls Layer parameters can be customized for specific boundary patches through region-specific settings:

Patch Inclusion/Exclusion Controls which patches receive layers through explicit lists:

- **layers**: Dictionary mapping patch names to layer specifications
- **nSurfaceLayers**: Number of layers on each specified patch

Layer Reduction Controls for reducing layer counts in problematic areas:

- **Maximum thickness factors**: Controls thickness in narrow regions
- **Reduction factors**: Gradually reduces layer count in difficult regions

Interface Organization The layer controls interface is organized into two main tabs for better usability:

Basic Tab Contains fundamental layer parameters including sizing, growth ratio, and general behavior controls.

Features Tab Houses feature-specific controls for managing layer generation near geometric features, including angle thresholds and patch-specific settings.

Integration with OpenFOAM The layer addition settings are compiled into the `addLayersControls` subdictionary in the snappyHexMeshDict file. The interface automatically formats all parameters according to OpenFOAM syntax requirements:

```
addLayersControls
{
    relativeSizes true;
    expansionRatio 1.2;
    finalLayerThickness 0.7;
    minThickness 0.1;

    // Feature handling
    featureAngle 30;
    nRelaxIter 5;

    // Patch specifications
    layers
    {
        "wall.*"
        {
            nSurfaceLayers 3;
        }
    }
}
```

Computational Considerations Layer addition significantly increases cell count near boundaries but is essential for accurate simulation of boundary layer effects. The interface provides direct control over the thickness distribution and count, allowing users to balance computational cost with simulation accuracy for wall-bounded flows where boundary layer resolution is critical.

```python
def draw_layer_panel(self, context):
    """Draw layer addition controls"""
    layout = self.layout
    scene = context.scene

    # Main toggle
    row = layout.row()
    row.prop(scene, "addLayers")

    # Only show if layers enabled
    if scene.addLayers:
        box = layout.box()

        # Size specification mode
        col = box.column()
        col.prop(scene, "relativeSizes")
```

```

            # Layer thickness parameters
            col = box.column(align=True)
            col.prop(scene, "expansionRatio")

            # Different controls based on relative sizing
            if scene.relativeSizes:
                col.prop(scene, "finalLayerThickness")
                col.prop(scene, "minThickness")
            else:
                col.prop(scene, "firstLayerThickness")
                col.prop(scene, "thickness")

            # Layer settings per patch
            box = layout.box()
            box.label(text="Layer Settings")

            # Layer collection with add/remove controls
            row = box.row()
            row.template_list("LAYER_UL_patch_layers", "", scene, "
               layers",
                             scene, "layers_index", rows=3)

            col = row.column(align=True)
            col.operator("vnt.add_layer", icon='ADD', text="")
            col.operator("vnt.remove_layer", icon='REMOVE', text="")

            # Settings for selected layer
            if len(scene.layers) > 0 and scene.layers_index >= 0:
                layer = scene.layers[scene.layers_index]
                col = box.column()
                col.prop(layer, "name")
                col.prop(layer, "nSurfaceLayers")
```

Listing 11: Layer Addition UI

4 Dictionary Generation

4.1 Dictionary Writer Implementation

The core functionality of the SnappyHexMesh GUI is to generate valid OpenFOAM dictionary syntax from the UI parameter values. This functionality is implemented in the `snappydict_writer.py` module. It uses the functions that were used to generate the preview of the various subdictionary stored in the `dictionary_writers.py` module. Aside from being modular, this ensures that the preview displayed in the tabs and the generated dictionary have identical subdictionaries.

```python
    def generate_snappy_dict(scene):
        """Generate complete snappyHexMeshDict from scene properties"""
        lines = []

        # Header
        lines.append("/*---------------*- C++ -*------------------*\\")
        lines.append("| =========                 |                                                 |")
        lines.append("| \\\\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |")
        lines.append("|  \\\\    /   O peration     | Version:  v2306                                 |")
        lines.append("|   \\\\  /    A nd           | Website:  www.openfoam.com                      |")
        lines.append("|    \\\\/     M anipulation  |                                                 |")
        lines.append("\\*---------------------------------------------------*/")
        lines.append("FoamFile")
        lines.append("{")
        lines.append("    version     2.0;")
        lines.append("    format      ascii;")
        lines.append("    class       dictionary;")
        lines.append("    object      snappyHexMeshDict;")
        lines.append("}")
        lines.append("// * * * * * * * * * * * * * * * * * * * * //")
        lines.append("")

        # Main control flags
        lines.append(f"castellatedMesh {str(scene.castellatedMesh).lower()};")
        lines.append(f"snap {str(scene.snap).lower()};")
        lines.append(f"addLayers {str(scene.addLayers).lower()};")
        lines.append("")

        # Add each main section
        lines.extend(generate_geometry_section(scene))
        lines.append("")

        if scene.castellatedMesh:
            lines.extend(generate_castellated_section(scene))
            lines.append("")

        if scene.snap:
            lines.extend(generate_snap_section(scene))
            lines.append("")

        if scene.addLayers:
            lines.extend(generate_layers_section(scene))
            lines.append("")

        lines.extend(generate_quality_section(scene))
        lines.append("")

        # Add footer settings
        lines.append(f"mergeTolerance {scene.mergeTolerance};")
        lines.append("")
        lines.append("// *************************************************** //
")

        return "\n".join(lines)
```

Listing 12: Dictionary Generation Functions

4.2 Section-Specific Generators

Each section of the dictionary has a dedicated generation function that handles its specific formatting requirements:

```
    def generate_geometry_section(scene):
        """Generate the geometry section of the dictionary"""
        lines = []
        lines.append("geometry")
        lines.append("{")

        # Add STL file entries
        if scene.stl_file_path:
            basename = os.path.basename(scene.stl_file_path)
            lines.append(f'    "{basename}"')
            lines.append("    {")
            lines.append("        type triSurfaceMesh;")
            name_base = os.path.splitext(basename)[0]
            lines.append(f"        name {name_base};")

            # Add regions if defined
            if len(scene.stl_regions) > 0:
                lines.append("        regions")
                lines.append("        {")
                for region in scene.stl_regions:
                    lines.append(f"            {region.name} {{ name {region.name}; }}")
                lines.append("        }")

            lines.append("    }")

        # Add primitive geometry items
        for item in scene.geometry_items:
            lines.append(f"    {item.name}")
            lines.append("    {")

            if item.type == 'box':
                lines.append("        type box;")
                lines.append(f"        min ({item.min_x} {item.min_y} {item.min_z});")
                lines.append(f"        max ({item.max_x} {item.max_y} {item.max_z});")
            elif item.type == 'sphere':
                lines.append("        type sphere;")
                lines.append(f"        centre ({item.center_x} {item.center_y} {item.center_z});")
                lines.append(f"        radius {item.radius};")

            lines.append("    }")

        lines.append("}")
        return lines
```

Listing 13: Geometry Section Generator

4.3 Handling Complex Collections

Collections like features and refinement regions require special handling to generate the correct nested list structure:

```
    def generate_features_list(scene):
        """Generate the features list section for castellatedMeshControls"""
        if not scene.cast_features:
            return []

        lines = []
        lines.append("    features")
        lines.append("    (")

        for feature in scene.cast_features:
            lines.append("        {")
            file_path = feature.file
            if not os.path.isabs(file_path):
                # Handle relative paths
                lines.append(f'            file "{file_path}";')
            else:
                # Handle absolute paths - extract filename only
                lines.append(f'            file "{os.path.basename(file_path)}";')

            lines.append(f"            level {feature.level};")
            lines.append("        }")

        lines.append("    );")
        return lines
```

Listing 14: Features List Generator

4.4 Handling Conditional Sections

Some dictionary sections should only be included based on property values or other conditions:

```
    def generate_quality_section(scene):
        """Generate the mesh quality controls section"""
        lines = []
        lines.append("meshQualityControls")
        lines.append("{")

        # If using an external file, just include it
        if scene.includeMeshQualityDict:
            lines.append(f'    #includeEtc "{scene.meshQualityDictPath}"')
        else:
            # Add main quality parameters
            lines.append(f"    maxNonOrtho {scene.maxNonOrtho};")
            lines.append(f"    maxBoundarySkewness {scene.maxBoundarySkewness};")
            lines.append(f"    maxInternalSkewness {scene.maxInternalSkewness};")
            lines.append(f"    maxConcave {scene.maxConcave};")

            if scene.minFlatness > 0:
                lines.append(f"    minFlatness {scene.minFlatness};")

            lines.append(f"    minVol {scene.minVol};")
            lines.append(f"    minTwist {scene.minTwist};")
            lines.append(f"    minDeterminant {scene.minDeterminant};")
            lines.append(f"    minFaceWeight {scene.minFaceWeight};")

            # Add relaxed section only if enabled
            if scene.relaxed_enabled:
                lines.append(f"    nRelaxIter {scene.nRelaxIter};")
                lines.append("    relaxed")
                lines.append("    {")
                lines.append(f"        maxNonOrtho {scene.
                   relaxed_maxNonOrtho};")
                lines.append("    }")

        lines.append("}")
        return lines
```

Listing 15: Conditional Section Generation

5 User Interface Implementation

5.1 Panel Organization

The UI is organized into a hierarchical panel structure with tabs for major sections:

```
    class MESH_PT_snappyhexmesh(Panel):
        """Panel for SnappyHexMesh dictionary generation"""
        bl_label = "SnappyHexMesh"
        bl_idname = "MESH_PT_snappyhexmesh"
        bl_space_type = "VIEW_3D"
        bl_region_type = "UI"
        bl_category = "OpenFOAM"

        def draw(self, context):
            layout = self.layout
            scene = context.scene

            # Main section tabs
            row = layout.row()
            row.prop(scene, "snappy_section_tab", expand=True)

            # Show appropriate section based on tab selection
            if scene.snappy_section_tab == 'MAIN':
                self.draw_main_controls(context)
            elif scene.snappy_section_tab == 'GEOMETRY':
                self.draw_geometry_panel(context)
            elif scene.snappy_section_tab == 'CASTELLATED':
                self.draw_castellated_panel(context)
            elif scene.snappy_section_tab == 'SNAP':
                self.draw_snapping_panel(context)
            elif scene.snappy_section_tab == 'LAYERS':
```

```python
                self.draw_layer_panel(context)
            elif scene.snappy_section_tab == 'QUALITY':
                self.draw_quality_panel(context)
            elif scene.snappy_section_tab == 'PREVIEW':
                self.draw_dictionary_preview(context)
```

Listing 16: Panel Registration and Organization

5.2 Progressive Disclosure Pattern

The implementation uses a progressive disclosure pattern to manage complexity:

```python
    def draw_castellated_panel(self, context):
        """Draw castellated mesh controls with sub-tabs"""
        layout = self.layout
        scene = context.scene

        # Enable/disable for entire section
        row = layout.row()
        row.prop(scene, "castellatedMesh")

        # Only show contents if enabled
        if scene.castellatedMesh:
            # Sub-tabs for section organization
            row = layout.row()
            row.prop(scene, "castellated_tab", expand=True)

            # Show appropriate sub-panel
            if scene.castellated_tab == 'GENERAL':
                self.draw_castellated_general(context)
            elif scene.castellated_tab == 'FEATURES':
                self.draw_castellated_features(context)
            elif scene.castellated_tab == 'SURFACES':
                self.draw_castellated_surfaces(context)
            elif scene.castellated_tab == 'REGIONS':
                self.draw_castellated_regions(context)
            elif scene.castellated_tab == 'ADVANCED':
                self.draw_castellated_advanced(context)
```

Listing 17: Progressive Disclosure Implementation

5.3 Context-Sensitive Controls

The implementation shows or hides controls based on parameter values:

```python
    def draw_layer_thickness_controls(self, context, box):
        """Draw layer thickness controls that adapt to the sizing
           mode"""
        scene = context.scene
        col = box.column()

        # Mode selection affects which controls are shown
        col.prop(scene, "relativeSizes")

        # Always show expansion ratio
        col = box.column(align=True)
        col.prop(scene, "expansionRatio")

        # Different controls based on relative sizing mode
        if scene.relativeSizes:
            # Relative sizing mode controls
            col.prop(scene, "finalLayerThickness")
            col.prop(scene, "minThickness")
        else:
            # Absolute sizing mode controls
            col.prop(scene, "firstLayerThickness")
            col.prop(scene, "thickness")

        # Additional controls
        box.separator()
        col = box.column(align=True)
        col.prop(scene, "nGrow")

        # Advanced features in their own section
        box = layout.box()
        col = box.column()
        col.prop(scene, "featureAngle")
```

Listing 18: Context-Sensitive UI Controls

5.4 Advanced UI Components

Custom UI list implementations are used for collections like features, regions, and layers:

```
    class MESH_UL_refinement_regions(UIList):
        """Custom UI list for refinement regions"""

        def draw_item(self, context, layout, data, item, icon,
           active_data, active_propname):
            region = item

            # Create row with multiple columns for region properties
            row = layout.row(align=True)

            # Name column
            split = row.split(factor=0.3)
            split.prop(region, "name", text="", emboss=False)

            # Mode and level in second column
            subsplit = split.split(factor=0.4)
            subsplit.prop(region, "mode", text="")

            if region.mode != 'distance':
                # Simple level for inside/outside modes
                subsplit.prop(region, "level", text="")
            else:
                # Just show indicator for distance mode (managed in
                   details panel)
                subsplit.label(text="Multiple Levels")

    # Draw detailed controls for the selected region
    def draw_region_details(layout, region):
        """Draw detailed controls for a refinement region"""
        col = layout.column()

        # Basic properties
        col.prop(region, "name")
        col.prop(region, "mode")

        # Different UI based on region mode
        if region.mode == 'distance':
            # Distance refinement needs list of distance-level pairs
            row = col.row()
            row.label(text="Distance Levels:")

            row = col.row()
            row.template_list("MESH_UL_distance_levels", "", region,
                "distance_levels",
                             region, "distance_level_index", rows=3)

            col2 = row.column(align=True)
            col2.operator("mesh.add_distance_level", icon='ADD',
               text="")
            col2.operator("mesh.remove_distance_level", icon='REMOVE
               ', text="")
        else:
            # Simple level for inside/outside modes
            col.prop(region, "level")
```

Listing 19: Custom UI List for Refinement Regions

5.5 Export Functionality

Dictionary export is implemented with file selection:

```python
    class MESH_OT_export_dictionary(Operator, ExportHelper):
        """Export the snappyHexMeshDict to a file"""
        bl_idname = "mesh.export_dictionary"
        bl_label = "Export Dictionary"

        filename_ext = ""
        filter_glob: StringProperty(default="*", options={'HIDDEN'})

        def execute(self, context):
            from venturial.models.snappyhexmesh.snappydict_writer import generate_snappy_dict

            # Generate dictionary
            dictionary = generate_snappy_dict(context.scene)

            # Get target path
            path = self.filepath

            # Ensure path ends with correct filename if not specified
            if not os.path.basename(path) or '.' in os.path.basename(path):
                path = os.path.join(path, 'snappyHexMeshDict')

            # Ensure directory exists
            os.makedirs(os.path.dirname(path), exist_ok=True)

            # Write file
            try:
                with open(path, 'w') as f:
                    f.write(dictionary)
                self.report({'INFO'}, f"Dictionary exported to {path}")
            except Exception as e:
                self.report({'ERROR'}, f"Export failed: {str(e)}")
                return {'CANCELLED'}

            return {'FINISHED'}
```

Listing 20: Dictionary Export Implementation

6 Documentation and Help System

6.1 Tooltip Implementation

The Venturial interface incorporates a sophisticated tooltip system that enhances user experience by providing detailed explanations of OpenFOAM parameters directly within the Blender UI.

6.1.1 Design Philosophy

The tooltip implementation follows several key design principles:

- **Centralized Management:** All tooltips are defined in dedicated dictionary structures
- **Categorization:** Tooltips are organized by functional domain (castellated mesh, snapping, layers, etc.)
- **Dynamic Integration:** Tooltip content is injected into Blender property descriptions at runtime
- **Separation of Concerns:** Documentation is maintained separately from implementation code

6.1.2 Technical Implementation

The tooltip system uses a dedicated module (`tooltip_updater.py`) that handles tooltip synchronization through:

- Category-specific update functions targeting functional domains
- A general property update mechanism that modifies Blender property descriptions
- Integration with Blender's property system during addon registration
- Robust error handling to prevent UI issues if tooltip updates fail

6.1.3 Content Organization

Tooltips are organized in dictionary structures imported from a dedicated tooltips module containing:

- `CASTELLATED_TOOLTIPS`: Descriptions for castellated mesh parameters
- `SNAP_TOOLTIPS`: Descriptions for surface snapping parameters
- `LAYER_TOOLTIPS`: Descriptions for boundary layer addition parameters
- `QUALITY_TOOLTIPS`: Descriptions for mesh quality parameters
- `DICTIONARY_TOOLTIPS`: Descriptions for dictionary generation parameters

The tooltip system significantly enhances Venturial's usability by bridging the gap between OpenFOAM's technical complexity and practical mesh generation workflows, supporting both novice and experienced users. The implementation includes a comprehensive tooltip system that provides context-specific help for each parameter:

```python
# Dictionary of detailed tooltips for parameters
TOOLTIPS = {
    "maxNonOrtho": """
Maximum non-orthogonality allowed. 0 is fully orthogonal, 90 is bad.

Non-orthogonality measures the angle between the line connecting cell centers
and the face normal. Values over 70-80 can lead to numerical diffusion and
solver stability issues. For complex geometries, you may need to relax this
constraint (up to 75-80), while for simple geometries, aim for lower values (50-60).

Default: 65.0
Range: 0.0 to 180.0
    """,

    "expansionRatio": """
Expansion ratio for layer addition.

Controls how quickly layers grow away from the wall:
- 1.0 = uniform thickness (no growth)
- >1.0 = each layer is thicker than the previous one
- Typical values: 1.1-1.3
- Values >1.5 can cause poor cell quality

Default: 1.2
Range: 1.0 to 2.0
    """,

    # Many more tooltips...
}

def update_property_tooltips():
    """Apply detailed tooltips to properties"""
    for prop_path, tooltip in TOOLTIPS.items():
        parts = prop_path.split('.')

        # Handle Scene properties
        if len(parts) == 1 and hasattr(bpy.types.Scene, parts[0]):
                prop = getattr(bpy.types.Scene, parts[0])
                if hasattr(prop, "__annotations__"):
                    prop.__annotations__[parts[0]].__dict__["description"] = tooltip.strip()

            # Handle PropertyGroup properties
            elif len(parts) == 2:
                cls_name, prop_name = parts
                for cls in [c for c in dir(bpy.types) if c.endswith(cls_name)]:
                    prop_group = getattr(bpy.types, cls)
                    if hasattr(prop_group, "__annotations__") and prop_name in prop_group.__annotations__:
                        prop_group.__annotations__[prop_name].__dict__["description"] = tooltip.strip()
```

Listing 21: Tooltip System

7 Technical Assessment and Future Development

7.1 Implementation Challenges

Several technical challenges were encountered during implementation:

- **Dict Structure Representation**: Representing OpenFOAM's deeply nested dictionary structure within Blender's property system required careful design to maintain both usability and accurate dictionary generation
- **Collection Management**: Managing ordered collections of items (features, regions, etc.) required custom UIList implementations and index tracking
- **Conditional Parameters**: Many parameters are only relevant when certain options are enabled, requiring conditional UI rendering and validation
- **Dictionary Format Compatibility**: Ensuring generated dictionaries conform to OpenFOAM's specific syntax requirements for different data types and nested structures
- **Parameter Dependencies**: Maintaining consistency between interdependent parameters required validation callbacks and careful UI design

7.2 Current Limitations

The current implementation has several technical limitations:

- **Limited Dictionary Parsing**: While the tool can generate dictionaries, its ability to parse existing dictionaries for modification is limited.

- **No Visual Feedback**: The interface does not provide visual representation of refinement regions or mesh previews.
- **Limited Error Recovery**: When validation detects issues, error reporting is basic and doesn't always provide clear resolution guidance.
- **More Parameters and Settings**: The parameters and options present in the current implementation is limited.

7.3 Future Development Opportunities

Several enhancements can significantly expand the functionality and usability of Venturial:

- **Extended File Import Support:** Enhance the importer to support not only STL files but also all file formats compatible with snappyHexMesh, including OBJ, OFF, PLY, and VTK.
- **Region Display by Name:** Add rendering functionality in the Blender viewport that visually highlights regions defined by name within STL or other supported files.
- **Multi-Point LocationInMesh for Castellated Mesh:** Extend the `LocationInMesh` control to accept multiple points, enabling accurate meshing of both internal and external regions simultaneously.
- **Layer Addition from blockMesh Patches:** Implement parsing of patches defined in the blockMesh dictionary and automatically generate corresponding layer patches within the Layer Addition controls.
- **UI for surfaceFeaturesDict Generation:** Develop a user interface to create and manage the surfaceFeaturesDict, streamlining the entire snappyHexMesh generation workflow.

8 Conclusion

The SnappyHexMesh dictionary GUI significantly improves the mesh generation workflow in OpenFOAM by transforming a complex text-based configuration into an intuitive visual interface. The implementation leverages Blender's property system and UI capabilities while maintaining full compatibility with OpenFOAM's dictionary format.

Key technical achievements include:

- A modular architecture that separates UI components from dictionary generation logic
- Multi-level validation that prevents invalid parameter combinations
- Dynamic UI components that adapt to parameter selections
- Comprehensive tooltip documentation integrated directly into the interface
- Robust dictionary generation that handles complex nested structures

References

1. OpenFOAM Foundation. *Mesh Generation with the snappyHexMesh Utility.* Retrieved from `https://www.openfoam.com/documentation/user-guide/4-mesh-generation-and-conversion/4.4-mesh-generation-with-the-snappyhexmesh-utility`

2. OpenFOAM Foundation. *User Guide: snappyHexMesh.* Retrieved from `https://www.openfoam.com/documentation/guides/latest/doc/guide-meshing-snappyhexmesh.html`

3. Wolf Dynamics. *snappyHexMesh – Meshing in OpenFOAM.* Retrieved from `https://www.wolfdynamics.com/wiki/meshing_OF_SHM.pdf`

4. Blender Foundation. *Blender Python API Documentation.* Retrieved from `https://docs.blender.org/api/current/index.html`