Module Interface Specification for AutoVox

Team #10, Five of a Kind
Omar Abdelhamid
Daniel Maurer
Andrew Bovbel
Olivia Reich
Khalid Farag

January 21, 2026

1 Revision History

Date	Version	Notes
November 13, 2025	1.0	Initial draft by All
January 21, 2026	2.0	Revision 0 by Daniel and Omar

2 Symbols, Abbreviations and Acronyms

See SRS Documentation.

Contents

3 Introduction

The following document details the Module Interface Specifications for AutoVox. AutoVox is a desktop-based CAD enhancement tool that enables researchers and engineers to assign magnetic and material properties to individual voxels within a 3D model. The system allows users to import CAD files (STL format), automatically convert them into voxel grids, and interactively select and modify voxels layer by layer to assign magnetization directions and material assignments. This tool streamlines the magnetization planning process for multi-material 3D printing workflows in research laboratory environments.

Complementary documents include the System Requirement Specifications and Module Guide. The full documentation and implementation can be found at https://github.com/OmarHassanAdelhamid/Five-of-a-Kind-capstone-project-

4 Notation

The structure of the MIS for modules comes from Hoffman and Strooper (1995), with the addition that template modules have been adapted from Ghezzi et al. (2003). The mathematical notation comes from Chapter 3 of Hoffman and Strooper (1995). For instance, the symbol := is used for a multiple assignment statement and conditional rules follow the form $(c_1 \Rightarrow r_1|c_2 \Rightarrow r_2|...|c_n \Rightarrow r_n)$.

The following table summarizes the primitive data types used by AutoVox.

Data Type	Notation	Description
character	char	a single symbol or digit
string	str	a sequence of characters
integer	$\mathbb{Z}$	a number without a fractional compone in (-∞, ∞)
natural number	$\mathbb{N}$	a number without a fractional compone in [1, ∞)
real	$\mathbb{R}$	any number in (-∞, ∞)
boolean	bool	True or False
floating-point (computer-based)	float	Floating-point representation of real nu ber
integer (computer-based)	int	Computer representation of integer
any type	Any	any data type is acceptable
list	list[T]	an ordered collection of objects of type
set	set[T]	an unordered collection of unique obje of type T
dictionary	dict[key] = value	data structure containing multiple k value pairs
tuple	tuple[T_1, T_2, ...] or tuple[T]	an ordered collection of values, potentia of different types
three-dimensional array	array3d[type]	an ordered, three-dimensional set wh each entry can be referenced via integ starting from 0
void	void	indicates no return value
current instance	self	a reference to the current instance o module

The specification of AutoVox uses some derived data types: sequences, strings, and tuples. Sequences are lists filled with elements of the same data type. Strings are sequences of characters. Tuples contain a list of values, potentially of different types. In addition, AutoVox uses functions, which are defined by the data types of their inputs and outputs. Local functions are described by giving their type signature followed by their specification.

Finally, AutoVoxuses some enums and refers to some derived types via other aliases for ease of notation and understanding. For instance, Vector3D and Voxel are both aliases for a tuple with 3 real values; UIEvent, VisualUpdate, Action, and propertyName are each enums.

5 Module Decomposition

The following table is taken directly from the Module Guide document for this project.

Table 1: Module Hierarchy

Level 1	**Level 2**
Hardware-Hiding Module	None
Behaviour-Hiding Module	Input Interpreter Module (M1) Voxel Slicing Module (M2) Display Partitioning Module (M3) Project Manager Module (M4) Interaction Controller Module (M5) Visualization State Manager Module (M6) Model Manager Module (M7) History Manager Module (M8) Voxel Tracking Module (M9) Highlight Manager Module (M10) Export Validation Module (M11) Export Manager Module (M12) Error Diagnostic Handler Module (M13)
Software Decision Module	Model Structure Module (M14) Graphics Adapter Module (M15) Export Structure Module (M16)

6 MIS of Input Interpreter (M1)

InputInterpreter

6.1 Module

The InputInterpreter module is responsible for importing and parsing CAD model files and normalizing them into a consistent internal format that downstream modules can process.

6.2 Uses

- `VoxelSlicing` for voxel generation from input CAD file.
- `DisplayPartitioning` to partition the resulting set of voxels.
- `ErrorDiagnosticHandler` for error handling and diagnostics.

6.3 Syntax

6.3.1 Exported Constants

- `SUPPORTED_FORMATS`: list[str] — supported file types.

6.3.2 Exported Access Programs

Name	In	Out	Exceptions
`loadFile`	`filePath: str`	`ModelStructure`	`FileNotFoundError`, `UnsupportedFileTypeError`
`detectFormat`	`filePath: str`	`str`	`FileNotFoundError`

6.4 Semantics

6.4.1 State Variables

None.

6.4.2 Environment Variables

- `FILE_PATH: str` — path to the input file.

6.4.3 Assumptions

- The outer folder of the file path exists, even if the file may not be found.
- Data structure of the file conforms to the file format.

6.4.4 Access Routine Semantics

`loadFile(filePath: str) -> ModelStructure`

- **Transition:**
  - loads file specified at passed file path
  - passes file to VoxelSlicing
  - passes resulting voxels to DisplayPartitioning
  - returns resulting ModelStructure
- **Output:** `ModelStructure` to be passed to ModelManager
- **Exceptions:** `FileNotFoundError` if file does not exist; `UnsupportedFileTypeError` if file type is not in `SUPPORTED_FORMATS`

`detectFormat(filePath: str) -> str`

- **Transition:** returns file type via header or extension inspection
- **Output:** `str` indicating file type
- **Exceptions:** `FileNotFoundError` if file does not exist

6.4.5 Local Functions

None.

7 MIS of Voxel Slicing (M2)

VoxelSlicing

7.1 Module

The VoxelSlicing module is responsible for converting the geometric model into a structured voxel grid. It divides the 3D model into discrete voxels based on a specified resolution for simulation and visualization.

7.2 Uses

- `ErrorDiagnosticHandler` for error handling and diagnostics.

7.3 Syntax

7.3.1 Exported Constants

- `DEFAULT_RESOLUTION: tuple(float, float, float)` — default voxel size in x, y, and z directions.
- `MAX_LAYERS: int` — maximum number of voxels in the z-dimension.

7.3.2 Exported Access Programs

Name	In	Out	Exceptions
`voxelizeModel`	`model: list[Vector3D]`, `resolution: tuple(float, float, float)`	`list[Vector3D]`	None

7.4 Semantics

7.4.1 State Variables

None.

7.4.2 Environment Variables

None.

7.4.3 Assumptions

- Input model geometry is well-defined and watertight.
- Resolution values are positive real numbers.
- The resulting number of voxels does not exceed system memory capacity.

7.4.4 Access Routine Semantics

`voxelizeModel(points: list[Vector3D], resolution: tuple(float, float, float)) -> list[Vector3D]`

- **Transition:** produces a set of 3D-coordinates representing voxel centres, given bounding vertexes of the model
- **Output:** `list[Vector3D]` of voxel centres
- **Exceptions:** None

Formalization of `voxelizeModel`
Adapted from ideas discussed in *An Accurate Method for Voxelizing Polygon Meshes*, Huang et al. (1998)
Definitions

- 3D point - a tuple containing three real values, indicating a point in 3D space: t = ($\mathbb{R}$, $\mathbb{R}$, $\mathbb{R}$)
- 3D triangle - a tuple containing three 3D points, uniquely defining a triangle: $p = (t_1, t_2, t_3)$
- 3D mesh - a set of triangles that form a closed surface, e.g.

$$M = \{p_1, p_2, ..., p_n\}, \text{ where } (\forall p | p \in M : (\forall t | t \in p : (\exists p_a, p_b | p_a \in M \wedge p_b \in M \\ \wedge p_a \neq p_b \wedge p_b \neq p \wedge p_a \neq p : t \in p_a \wedge t \in p_b)))$$

- voxel - a 3D cube, defined by its central 3D point: v = t
- resolution - $\mathbb{R} > 0$, defines unit length of voxels
- voxel grid - set of equally spaced voxels of unit length *resolution* and with centres separated by length *resolution* in each cardinal direction, thereby completely filling 3D space: $G = v_1, v_2, ..., v_n$

Voxelization Process

- *Generation of surface voxels:* - let S be the union of the three following sets, where all $v \in G$, $v \in G \implies (\exists n_1, n_2, n_3 | n_1, n_2, n_3 \in \mathbb{Z} : (\forall t | t \in v : t = (n_1 \cdot resolution, c_2 \cdot resolution, c_3 \cdot resolution)))$.
  S_v - set of all voxels that represent all polygon vertices in M.

$$S_v = \{v_1, v_2, ..., v_n\}, \text{ such that} \\ (\forall p | p \in M : (\forall t | t \in p : (\exists v | v \in S_v : ||v - t|| \leq radius) \wedge (\forall v | v \notin S_v : ||v - t|| > radius))),$$
where $radius$ is the radius of a bounding sphere about t.

S_e - set of all voxels that represent all polygon edges in M.

$$S_e = \{v_1, v_2, ..., v_n\}, \text{ such that}$$
$$(\forall p | p \in M : (\forall t_1, t_2 | t_1, t_2 \in p \wedge t_1 \neq t_2 : (\forall t_x | t_x = t_1 + \alpha(t_2 - t_1) : (\exists v | v \in S_e :$$
$$||v - t_x|| \leq radius) \wedge (\forall v | v \notin S_e : ||v - t_x|| > radius)))),$$
$$\text{where } \alpha = [0, 1].$$

S_b - set of all voxels that represent all polygon faces in M.

$$S_b = \{v_1, v_2, ..., v_n\}, \text{ such that}$$
$$(\forall p | p \in M : (\forall t_1, t_2, t_3 | t_1, t_2, t_3 \in p \wedge t_1 \neq t_2 \wedge t_2 \neq t_2 \wedge t_1 \neq t_3 : (\exists v | v \in S_b : v \in$$
$$T^+(t_1, t_2, t_3)) \wedge (\forall v | v \notin S_b : v \notin T^+(t_1, t_2, t_3)))),$$
$$\text{where } T(a, b, c) = \{a + \alpha(b - a) + \beta(c - a) | \alpha, \beta \geq 0, \alpha + \beta \leq 1\},$$
$$T^+(a, b, c) = \{t + xn | t \in T(a, b, c), -radius \leq x \leq radius, n = \frac{(b-a)\times(c-a)}{||(b-a)\times(c-a)||}\}$$

- *Generation of inner voxels:* given S, create set F of voxels such that:

$$F = \{v | (\forall t | t \in v : (\exists v', v'' | v', v'' \in S \wedge v' \neq v'' \wedge v' \neq v \wedge v'' \neq v : (\exists t^+, t^- | t^+ \in$$
$$v' \wedge t^- \in v'' : t + \mathbb{R} = t^+ \wedge t - \mathbb{R} = t^-)))\}$$

- Return $S \cup F$.

7.4.5 Local Functions

None.

8 MIS of Display Partitioning (M3)

DisplayPartitioning

8.1 Module

The DisplayPartitioning module provides core functionality for partitioning the model into distinct display segments.

8.2 Uses

- `ErrorDiagnosticHandler` for error handling and diagnostics.

8.3 Syntax

8.3.1 Exported Constants

- `MAX_PARTITIONS: int` — maximum number of partitions that can be created
- `MAX_VOXELS_PER_PARTITION: int` — maximum number of voxels per partition

8.3.2 Exported Access Programs

Name	In	Out	Exceptions
`getPartitions`	`voxels: list[Vector3D]`	`dict[str] = partitionItem`	None
`mapVoxels`	`voxels: list[Voxel]`, `partition: partitionItem`	list[Voxel]	None

8.4 Semantics

8.4.1 State Variables

None.

8.4.2 Environment Variables

None.

8.4.3 Assumptions

- the `list[Vector3D]` passed is the complete set of voxels in a given model.

8.4.4 Access Routine Semantics

`getPartitions(voxels: list[Vector3D]) -> dict[str] = partitionItem`

- **Transition:** creates set of model partitions based upon passed voxels
- **Output:** `dict[str] = partitionItem` that maps 2D spatial IDs to `partitionItem`s
- **Exceptions:** None

Formalization of `getPartitions`
Definitions

- 3D point - a tuple containing three real values, indicating a point in 3D space: t = ($\mathbb{R}$, $\mathbb{R}$, $\mathbb{R}$)
- partition - sector of 3D space that is finite in x and y, but extends infinitely in z and can therefore be defined by a pair of 2D coordinates: p = $((\mathbb{R}, \mathbb{R}), (\mathbb{R}, \mathbb{R}))$. For ease of notation, $p.x$ and $p.y$ refer to a partition's length in the x and y dimensions respectively.
- voxel - a 3D cube, defined by its central 3D point: v = t

Partition Definition
Given a set of all voxels V, create a minimal set of partitions P such that:

- No partitions overlap:

$$(\forall p, p' | p, p' \in P : (\forall t, t' | t \in p \wedge t' \in p' : t \neq t'))$$

- Partitions are all the same size, and square:

$$(\forall p, p' | p, p' \in P : p.x = p'.x \wedge p.y = p'.y) \wedge (\forall p | p \in P : p.x = p.y)$$

- The set of partitions cover all voxels in the space:

$$(\forall v | v \in V : (\exists p | p \in P : v \in P))$$

- The number of voxels in one partition is renderable:

$$(+v | v \in P : 1) \leq \text{MAX\_VOXELS\_PER\_PARTITION}$$

Assign a unique ID to each partition to allow traversal between partitions and return P.

`mapVoxels(voxels: list[Voxel], partition: PartitionItem) -> list[Voxel]`

- **Transition:** modifies set of voxels to map them to the passed partition
- **Output:** `list[Voxel]` - the same list passed in, with partition mappings modified
- **Exceptions:** None

8.4.5 Local Functions

None.

9 MIS of Project Manager (M4)

ProjectManager

9.1 Module

The ProjectManager module manages the creation, initialization, and persistence of project workspaces, handles project metadata storage, and ensures project resources are properly allocated.

9.2 Uses

- `ModelManager` to obtain and create the model structure and data representation.
- `ErrorDiagnosticHandler` for error handling and diagnostics.

9.3 Syntax

9.3.1 Exported Constants

- `AUTOSAVE_INTERVAL: int` - number of seconds between automatic saves

9.3.2 Exported Access Programs

Name	In	Out	Exceptions
`ProjectManager`	`workspaceRoot: str`	`self`	`IOError`
`createNewProject`	`projectPath: str, config: dict[str] = str`	`bool`	`IOError`, `ValueError`
`loadProject`	`projectPath: str`	`bool`	`IOError`, `FileNotFoundError`
`saveCurrentProject`	None	`bool`	`IOError`
`getCurrentProjectMetadata`	None	`dict[str] = str`	None
`changeWorkspace`	`workspaceRoot: str`	`bool`	`IOError`

9.4 Semantics

9.4.1 State Variables

- `currentProjectPath: str` - Path to the current project directory
- `workspaceRoot: str` - Root directory of the current workspace
- `projectMetadata: dict[str] = str` - Dictionary containing metadata of the current project
- `model: ModelManager` - Reference to the model of the current project

9.4.2 Environment Variables

- `FILE_SYSTEM`: The file system where the workspaces exist within
- `FILE_PERMISSIONS`: Permissions the module has (read, write) within the current workspace

9.4.3 Assumptions

- The file system has sufficient space for project/file creation
- Read/write permissions are available for the workspace directory
- The provided project paths exist within the current workspace path, or can be created

9.4.4 Access Routine Semantics

`ProjectManager(workspaceRoot: str) -> self`

- **Transition:** Creates new ProjectManager object and initializes workspace
- **Output:** `self` - a reference to itself
- **Exceptions:** `IOError` if passed workspace path does not exist or cannot be created

`createNewProject(projectPath: str, config: dict[str] = str) -> bool`

- **Transition:** creates a new project at the specified relative path within the current workspace. `config` notably contains information such as whether the project shall begin with a CAD model or not, and the path of the CAD model to be processed by `ModelManager`.
  - calls `validateConfig` to verify correctness
  - calls `createProjectStructure` to create and initialize project folder
  - uses `ModelManager` to create a model, passing path to the CAD model to be converted if needed

    - calls `writeMetadataFile` to write initial metadata to the project folder

- **Output:** `bool` indicating process success
- **Exceptions:** `IOError` if project path could not be created, `ValueError` if configuration invalid

`loadProject(projectPath: str) -> bool`

- **Transition:** updates current project path, loads the existing model with ModelManager, and loads metadata file into internal dictionary via `readMetadataFile`
- **Output:** `bool` indicating process success
- **Exceptions:** `IOError` if project path cannot be resolved, `FileNotFoundError` if any needed file within the project directly does not exist.

`saveCurrentProject() -> bool`

- **Transition:** saves all project components via a call to `ModelManager` to save the model and a call to `writeMetadataFile`
- **Output:** `bool` indicating process success
- **Exceptions:** `IOError` if any file cannot be saved

`getCurrentProjectMetadata() -> dict[str] = str`

- **Transition:** returns current metadata from related state variable
- **Output:** `dict[str] = str` the requested metadata
- **Exceptions:** None

`changeWorkspace(workspaceRoot: str) -> bool`

- **Transition:** changes workspace, and saves/closes any open project
- **Output:** `bool` indicating process success
- **Exceptions:** `IOError` if new workspace path cannot be resolved

9.4.5 Local Functions

`validateConfig(config: dict[str] = str) -> bool`

- **Transition:** validates the configuration dictionary contains required fields
- **Output:** `bool` indicating validity
- **Exceptions:** None

`createProjectStructure(path: str) -> bool`

- **Transition:** Creates directory and its structure for a new project
- **Output:** `bool` indicating process success
- **Exceptions:** `IOError` if structure cannot be created

`readMetadataFile(path: str) -> dict[str] = str`

- **Transition:** reads and parses project metadata from passed file
- **Output:** `dict[str] = str` - the metadata read
- **Exceptions:** `FileNotFoundError` if file cannot be found

`writeMetadataFile(metadata: dict[str] = str) -> bool`

- **Transition:** writes project metadata to file
- **Output:** `bool` indicating process success
- **Exceptions:** `IOError` if path cannot be resolved

10 MIS of Interaction Controller (M5)

InteractionController

10.1 Module

The InteractionController module manages the process of raw events from interaction with the UI to associated actions within internal code.

10.2 Uses

- `ProjectManager` to handle user intent related to workspace switching, project creation, saving or loading.
- `ModelManager` to handle user intent related to model modification.
- `HistoryManager` to record user's model modifications.
- `ExportManager` to handle user intent related to export.
- `ErrorDiagnosticHandler` for error handling and diagnostics.

10.3 Syntax

10.3.1 Exported Constants

- `UI_EVENTS: list[UIEvent]` - identifies all supported UI events that can be detected from user interaction.
- `UI_ACTIONS: : list[UIAction]` - identifies all supported interactions that can be derived from UI events.

10.3.2 Exported Access Programs

Name	In	Out	Exceptions
`interpretEvent`	`event: UIEvent`	`list[VisualUpdate]`	None

10.4 Semantics

10.4.1 State Variables

- `currentView: View` - identifies what view is currently displayed during the interaction.

10.4.2 Environment Variables

None.

10.4.3 Assumptions

- User intent does not exceed the scope of supported actions that can be derived.
- There can only be one current active view at a given moment.
- `UIEvent`s passed are always contained within `UI_EVENTS`.
- Actions are always routed to the relevant internal function.

10.4.4 Access Routine Semantics

`interpretEvent(event: UIEvent) -> list[VisualUpdate]`

- **Transition:** translates a `UIEvent` into a set of `VisualUpdate`s to be carried out by the `VisualizationStateManager`, and orchestrates/begins any processes on the model, project, or related to exporting.
  - parses `UIEvent`
  - if no model/project actions are being taken, returns a list of `VisualUpdates`
  - if edits to the model are made, additionally calls `handleModelUpdate`
  - if requests to save/load a project or workspace are made, additionally calls `handleProjectUpdate`
  - if requests to export model are made, additionally calls `handleExportUpdate`
- **Output:** `list[VisualUpdate]` for the `VisualizationStateManager` to execute
- **Exceptions:** None

10.4.5 Local Functions

`handleModelUpdate(action: Action) -> bool`

- **Transition:** translates `Action` into calls to `ModelManager` and `HistoryManager`
- **Output:** `bool` indicating process success
- **Exceptions:** None

`handleProjectUpdate(action: Action) -> bool`

- **Transition:** translates Action into calls to `ProjectManager`
- **Output:** `bool` indicating process success

- **Exceptions:** None

`handleExportUpdate(action: Action) -> bool`

- **Transition:** translates `Action` into calls to `ExportManager`
- **Output:** `bool` indicating process success
- **Exceptions:** None

11 MIS of Visualization State Manager (M6)

VisualizationManager

11.1 Module

The VisualizationManager module maintains visualization-related state associated with UI views and records visualization update requests corresponding to changes in the underlying model and highlight state.

11.2 Uses

- `GraphicsAdapter` for issuing rendering requests corresponding to visualization state updates.
- `HighlightManager` for obtaining semantic highlight state information.
- `InteractionController` as a source of interaction-triggered visualization state change notifications.
- `ErrorDiagnosticHandler` for error handling and diagnostics.

11.3 Syntax

11.3.1 Exported Constants

- `DEFAULT_VIEWS`: list[str] — list of identifiers corresponding to the default views generated upon project initialization.

11.3.2 Exported Access Programs

Name	In	Out	Exceptions
`getCurrentView`	None	str	None
`setView`	state: str	void	None

11.4 Semantics

11.4.1 State Variables

- `currentView`: str — identifies which UI view is currently shown to the user.
- `viewStatus`: dict[str] = ViewItem — stores all data encapsulated within a UI view.

- `updateBundle`: dict[str] = UpdateItem — stores all data needed to describe a visualization update request.
- `pendingUpdates`: list[UpdateKey] — visualization update requests waiting to be forwarded for rendering.
- `renderStatus`: bool — records the status of the most recent rendering request.
- `supportedViews`: list[str] — identifiers corresponding to all available project views.

11.4.2 Assumptions

- Visualization state references only initialized and valid view identifiers.
- ViewItem is properly formatted in accordance with ViewItem specification.
- UpdateItem is properly formatted in accordance with UpdateItem specification.
- `currentView`, `ViewState`, and `DEFAULT_VIEWS` are elements of `supportedViews`.

11.4.3 Access Routine Semantics

getCurrentView() → str

- Transition: retrieves the identifier of the currently active visualization view.
- Output: str representing the current view identifier.
- Exceptions: None.

setView(state: str) → void

- Transition: updates the current visualization view to the specified view identifier.
- Output: None.
- Exceptions: None.

12 MIS of Model Manager (M7)

ModelManager

12.1 Module

The ModelManager module manages and mutates the voxel-based 3D model. It provides operations to add, remove, and modify voxels, update material and magnetization properties, and explicitly persist model state through save operations.

12.2 Uses

- `ModelStructure` to access and update the model's internal data representation.
- `VoxelTracking` to track changes made to voxel data.
- `ErrorDiagnosticHandler` for error handling and diagnostics.

12.3 Syntax

12.3.1 Exported Constants

- `MAX_VOXEL_COUNT`: int — maximum allowed voxel entries per project.

12.3.2 Exported Access Programs

Name	In	Out	Exceptions
`addVoxel`	coord: Any, material: Any, mag: Any	void	InvalidInputError
`removeVoxel`	coord: Any	void	NotFoundError
`modifyVoxel`	coord: Any, newData: Any	void	NotFoundError
`saveModel`	None	bool	IOError
`forceSave`	None	bool	IOError
`loadModel`	filePath: str	bool	DeserializationError

12.4 Semantics

12.4.1 State Variables

- `voxelGrid`: VoxelGrid — stores all voxel elements and layer mapping.
- `metadata`: dict — contains file name, author, timestamp, etc.
- `unsavedChanges`: bool — true if edits have occurred since last save.

12.4.2 Environment Variables

- `SAVE_PATH`: str — default save location for serialized project data.

12.4.3 Assumptions

- Voxel coordinates fall within the defined model boundaries.
- Data supplied for voxel modifications are valid according to schema.
- Save operations are explicitly invoked and do not interrupt ongoing edits.

12.4.4 Access Routine Semantics

addVoxel(coord: Any, material: Any, mag: Any) → void

- Transition: inserts a new voxel into the model at the specified coordinate with the given material and magnetization properties.
- Output: None.
- Exceptions: InvalidInputError if the provided data is invalid.

removeVoxel(coord: Any) → void

- Transition: removes the voxel at the specified coordinate from the model.
- Output: None.
- Exceptions: NotFoundError if the voxel does not exist.

modifyVoxel(coord: Any, newData: Any) → void

- Transition: updates the properties of the voxel at the specified coordinate using the provided data.
- Output: None.
- Exceptions: NotFoundError if the voxel does not exist.

saveModel() → bool

- Transition: serializes and persists the current model state to storage.
- Output: bool indicating whether the save operation was successful.
- Exceptions: IOError if the save operation fails.

forceSave() → bool

- Transition: immediately serializes and persists the current model state, regardless of unsaved change state.
- Output: bool indicating whether the save operation was successful.

- Exceptions: IOError if the save operation fails.

loadModel(filePath: str) → bool

- Transition: loads and deserializes model data from the specified file path, replacing the current model state.
- Output: bool indicating whether the load operation was successful.
- Exceptions: DeserializationError if loading fails due to invalid data.

13 MIS of History Manager (M8)

HistoryManager

13.1 Module

The HistoryManager module maintains the complete change history of the 3D voxel model. It allows undoing and redoing edits by tracking incremental changes (referred to here as deltas) triggered by user modification.

13.2 Uses

- `ErrorDiagnosticHandler` for error handling and diagnostics.

13.3 Syntax

13.3.1 Exported Constants

- `MAX_HISTORY_SIZE`: int — maximum number of undo states stored.

13.3.2 Exported Access Programs

Name	In	Out	Exceptions
`recordChange`	`delta: modelDelta`	None	None
`undoRequest`	None	`delta: modelDelta`	`EmptyHistoryError`
`redoRequest`	None	`delta: modelDelta`	`EmptyHistoryError`
`clearHistory`	None	None	None

13.4 Semantics

13.4.1 State Variables

- `historyStack: list[modelDelta]` - sequence of changes to the model.
- `redoStack: list[modelDelta]` - sequence of undone model states available for redo.

13.4.2 Environment Variables

None.

13.4.3 Assumptions

- Changes are discrete and atomic.
- No new change is made while undo or redo is in progress.
- The number of stored changes does not exceed `MAX_HISTORY_SIZE`; when another change is added, the oldest change is lost.
- `clearHistory` is called when a new project is created or switched to (changes are not saved with projects).

13.4.4 Access Routine Semantics

`recordChange(delta: modelDelta) -> None`

- **Transition:** adds passed change to `historyStack`, and calls `clearRedo()`
- **Output:** None
- **Exceptions:** None

`undoRequest() -> modelDelta`

- **Transition:** moves most recent `modelDelta` on `historyStack` to `redoStack` and returns it
- **Output:** `modelDelta` - most recent change
- **Exceptions:** `EmptyHistoryError` if `historyStack` is empty

`redoRequest() -> modelDelta`

- **Transition:** moves most recent `modelDelta` on `redoStack` to `historyStack` and returns It
- **Output:** `modelDelta` - most recent change

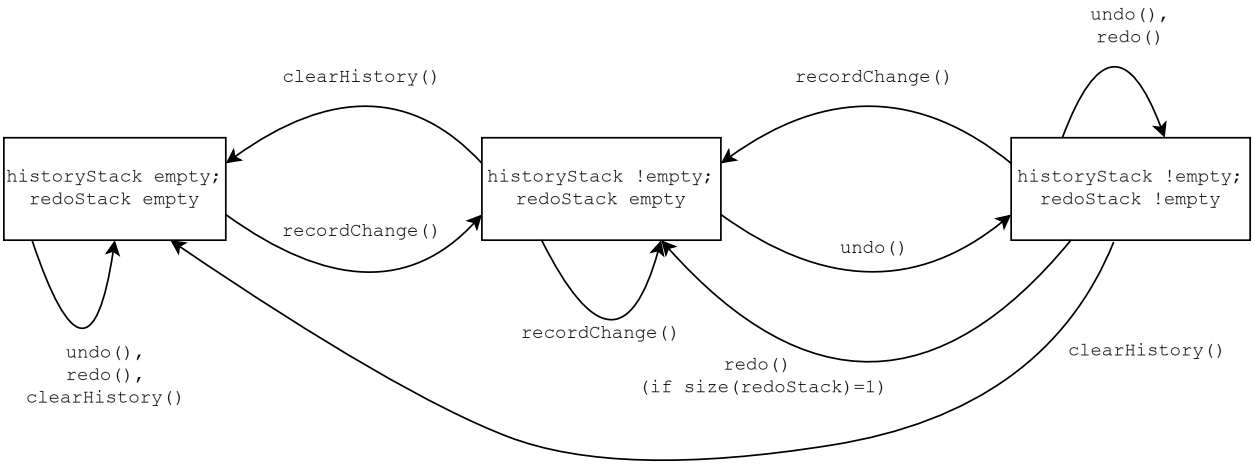

Figure 1: Finite State Machine for HistoryManager.

- **Exceptions:** `EmptyHistoryError` if `redoStack` is empty

`clearHistory() -> None`

- **Transition:** makes both `historyStack` and `redoStack` empty
- **Output:** None
- **Exceptions:** None

13.4.5 Local Functions

`clearRedo() -> None`

- **Transition:** makes only `redoStack` empty
- **Output:** None
- **Exceptions:** None

14 MIS of Voxel Tracking (M9)

VoxelTracking

14.1 Module

The VoxelTracking module interprets the voxel data structure to locate and track voxels that satisfy particular property criteria, such as selection state, material type, or user-defined rules. It provides read-only access to voxel query results without modifying model state or interpreting user interaction.

14.2 Uses

- `ModelStructure` to get the model structure and data representation.
- `ErrorDiagnosticHandler` for error handling and diagnostics.

14.3 Syntax

14.3.1 Exported Constants

- `MAX_QUERY_RESULTS`: int — maximum number of voxels returned in a single query result.
- `SUPPORTED_PROPERTIES`: list[str] — list of property names that can be queried (e.g., "material", "magnetization", "selected", "layer").

14.3.2 Exported Access Programs

Name	In	Out	Exceptions
`queryByMaterial`	materialType: Any	list[Any]	InvalidMaterialError
`queryBySelection`	isSelected: bool	list[Any]	None
`queryByRegion`	bounds: Any	list[Any]	InvalidBoundsError
`queryByLayer`	layerZ: Z	list[Any]	IndexError
`queryByProperty`	propertyName: str, value: Any	list[Any]	InvalidPropertyError
`getVoxelProperties`	coord: Any	dict[str] = Any	NotFoundError

14.4 Semantics

14.4.1 State Variables

- `queryCache`: dict[str] = list[VoxelCoord] — cached results of recent read-only queries for performance optimization.
- `activeSelections`: set[VoxelCoord] — cached snapshot of selection state derived from model state.

14.4.2 Environment Variables

None

14.4.3 Assumptions

- Voxel coordinates provided in queries are within valid grid boundaries.
- Property names used in queries match those defined in `SUPPORTED_PROPERTIES`.
- The voxel grid structure remains consistent during query operations.

14.4.4 Access Routine Semantics

queryByMaterial(materialType: Any) → list[Any]

- Transition: retrieves all voxel identifiers whose material property matches the specified material type.
- Output: list[Any] containing voxel identifiers.
- Exceptions: InvalidMaterialError if the material type is not recognized.

queryBySelection(isSelected: bool) → list[Any]

- Transition: retrieves all voxel identifiers that match the specified selection state.
- Output: list[Any] containing voxel identifiers.
- Exceptions: None.

queryByRegion(bounds: Any) → list[Any]

- Transition: retrieves all voxel identifiers that lie within the specified spatial bounds.
- Output: list[Any] containing voxel identifiers.
- Exceptions: InvalidBoundsError if the bounds are invalid.

queryByLayer(layerZ: Z) → list[Any]

- Transition: retrieves all voxel identifiers that belong to the specified layer.
- Output: list[Any] containing voxel identifiers.
- Exceptions: IndexError if the layer index is out of range.

queryByProperty(propertyName: str, value: Any) → list[Any]

- Transition: retrieves all voxel identifiers whose specified property matches the given value.
- Output: list[Any] containing voxel identifiers.
- Exceptions: InvalidPropertyError if the property name is not supported.

getVoxelProperties(coord: Any) → dict[str] = Any

- Transition: retrieves all stored properties associated with the specified voxel identifier.
- Output: dict[str] = Any containing voxel properties.
- Exceptions: NotFoundError if the voxel identifier does not exist.

15 MIS of Highlight Manager (M10)

HighlightManager

15.1 Module

The HighlightManager module manages the voxel highlights in accordance with visual semantic meaning.

15.2 Uses

- `ErrorDiagnosticHandler` for error handling and diagnostics.

15.3 Syntax

15.3.1 Exported Constants

- `DEFAULT_HIGHLIGHT_MAP`: dict[str] = str — mapping from semantic keys to their default highlight colours.

15.3.2 Exported Access Programs

Name	In	Out	Exceptions
`getHighlight`	semanticKey: str	ColourValue	None
`editPalette`	semanticKey: str, newColour: ColourValue	void	None
`setHighlight`	semanticKey: str	void	None
`resetPalette`	None	void	None

15.4 Semantics

15.4.1 State Variables

- `highlightColourMap`: dict[str] = str — current mapping from semantic keys to active highlight colours.

15.4.2 Environment Variables

None

15.4.3 Assumptions

- Semantic keys must correspond to valid entries maintained by the TrackingManager.
- Colour values are valid colour representations.

15.4.4 Access Routine Semantics

getHighlight(semanticKey: str) → str

- Transition: retrieves the highlight colour associated with the specified semantic key.
- Output: str representing the highlight colour.
- Exceptions: None.

editPalette(semanticKey: str, newColour: str) → void

- Transition: updates the highlight colour mapping for the specified semantic key.
- Output: None.
- Exceptions: None.

setHighlight(semanticKey: str) → void

- Transition: applies the highlight associated with the specified semantic key to relevant voxels.
- Output: None.
- Exceptions: None.

resetPalette() → void

- Transition: restores all highlight colours to their default values.
- Output: None.
- Exceptions: None.

16 MIS of Export Validation (M11)

ExportValidation

16.1 Module

The ExportValidation module validates export readiness by checking export file format requirements, verifying completeness of voxel properties (material and magnetization), and ensuring all export constraints are met.

16.2 Uses

- `ErrorDiagnosticHandler` for error handling and diagnostics.

16.3 Syntax

16.3.1 Exported Constants

- `MAX_VOXELS`: int = 13996800000 - Maximum number of voxels allowed
- `MAX_LAYERS`: int = 518400 - Maximum number of layers allowed

16.3.2 Exported Access Programs

Name	In	Out	Exceptions
`validateExportReadiness`	model: Any	Any	None
`checkPrinterCompatibility`	model: Any	bool	None
`checkPropertyCompleteness`	model: Any	list[str]	None
`validateFileFormat`	filePath: str	bool	IOError

16.4 Semantics

16.4.1 State Variables

None

16.4.2 Environment Variables

None

16.4.3 Assumptions

- The model structure provided is valid and properly initialized
- All voxels in the model have consistent property structures
- File paths provided are accessible and readable

16.4.4 Access Routine Semantics

validateExportReadiness(model: Any) → Any

- Transition: evaluates whether the provided model satisfies all export constraints, including size limits, property completeness, and format requirements.
- Output: Any representing the result of export readiness validation.
- Exceptions: None.

checkPrinterCompatibility(model: Any) → bool

- Transition: checks whether the provided model satisfies printer-specific constraints and limitations.
- Output: bool indicating printer compatibility.
- Exceptions: None.

checkPropertyCompleteness(model: Any) → list[str]

- Transition: verifies that all required voxel properties are present and returns identifiers of any missing properties.
- Output: list[str] containing names of missing properties, if any.
- Exceptions: None.

validateFileFormat(filePath: str) → bool

- Transition: validates that the export file format at the specified file path meets supported format requirements.
- Output: bool indicating whether the file format is valid.
- Exceptions: IOError if the file cannot be accessed.

16.4.5 Local Functions

- `countVoxels(model: ModelStructure) -> int`: Counts the total number of voxels in the model
- `countLayers(model: ModelStructure) -> int`: Counts the total number of layers in the model
- `checkPrinterSpecs(model: ModelStructure) -> bool`: Validates model against printer-specific constraints

17 MIS of Export Manager (M12)

ExportManager

17.1 Module

The ExportManager module coordinates the export process, transforms internal model data into export-compatible formats, and serializes project data including voxel grids, metadata, material properties, and magnetization information according to export specifications.

17.2 Uses

- `ModelStructure` to obtain the model structure and data representation.
- `ExportValidation` to validate export validity.
- `ExportStructure` to define the export data structure format.
- `ErrorDiagnosticHandler` for error handling and diagnostics.

17.3 Syntax

17.3.1 Exported Constants

- `DEFAULT_EXPORT_FORMAT`: str = "CSV" - Default export file format

17.3.2 Exported Access Programs

Name	In	Out	Exceptions
`ExportManager`	-	self	-
`exportProject`	model: ModelStructure, exportPath: str, format: str	None	IOError, ValueError
`transformToExportFormat`	model: ModelStructure, format: str	ExportData	-
`serializeData`	data: ExportData, format: str	str	ValueError

17.4 Semantics

17.4.1 State Variables

- `exportFormat`: str - Current export format being used
- `exportConfig`: dict - Configuration settings for export operations

17.4.2 Environment Variables

- `FILE_SYSTEM`: The file system where export files are written

17.4.3 Assumptions

- The model structure provided has been validated for export readiness
- The export path directory exists or can be created
- Write permissions are available for the export directory
- The export format is supported

17.4.4 Access Routine Semantics

ExportManager() → self

- Transition: initializes the export manager with default configuration values.
- Output: self.
- Exceptions: None.

exportProject(model: Any, exportPath: str, format: str) → void

- Transition: coordinates the export process by validating the model, transforming model data into the specified export format, and writing the serialized data to the given export path.
- Output: None.
- Exceptions: IOError if writing fails; ValueError if the export format is invalid.

transformToExportFormat(model: Any, format: str) → Any

- Transition: transforms the provided model data into an intermediate export-compatible data representation based on the specified format.
- Output: Any representing export-formatted data.
- Exceptions: None.

serializeData(data: Any, format: str) → str

- Transition: serializes the provided export data into a string representation according to the specified format.
- Output: str representing serialized export data.
- Exceptions: ValueError if serialization fails due to unsupported format.

17.4.5 Local Functions

- `groupVoxelsByLayer(model: ModelStructure) -> dict`: Groups voxels by their layer Z-coordinate
- `encodeCSV(exportData: ExportData) -> str`: Encodes export data into CSV format string
- `validateExportPath(path: str) -> bool`: Validates that the export path is writable
- `creatExportDirectory(path: str) -> None`: Creates the export directory if it does not exist

18 MIS of Error Diagnostic Handler (M13)

ErrorDiagnosticHandler

18.1 Module

The ErrorDiagnosticHandler module detects, diagnoses, and handles errors that occur during model operations, graphics rendering, and file interactions. It provides error classification, logging, and recovery mechanisms to ensure system stability and user feedback.

18.2 Uses

None

18.3 Syntax

18.3.1 Exported Constants

- `ERROR_MODEL_UNRESPONSIVE`: str = "MODEL_UNRESPONSIVE" - Error code for unresponsive model
- `ERROR_GRAPHICS_UPDATE`: str = "GRAPHICS_UPDATE_FAILURE" - Error code for graphics update failure
- `ERROR_FILE_ISSUE`: str = "FILE_OPERATION_ERROR" - Error code for file operation errors

18.3.2 Exported Access Programs

Name	In	Out	Exceptions
`detectError`	errorContext: dict	ErrorDiagnostic	-
`classifyError`	errorCode: str, errorContext: dict	ErrorType	-
`logError`	errorDiagnostic: ErrorDiagnostic	None	IOError
`handleErrorRecovery`	errorDiagnostic: ErrorDiagnostic	RecoveryAction	-
`getErrorSource`	errorDiagnostic: ErrorDiagnostic	str	-

18.4 Semantics

18.4.1 State Variables

- `errorLog`: list[ErrorDiagnostic] - History of detected errors
- `errorPatterns`: dict - Patterns for error classification

18.4.2 Environment Variables

- `LOG_FILE`: File system location for error logging

18.4.3 Assumptions

- Error context dictionaries contain sufficient information for diagnosis
- Log file location is writable
- Error codes follow the defined constants

18.4.4 Access Routine Semantics

detectError(errorContext: dict[str] = Any) → Any

- Transition: analyzes the provided error context to detect and generate an error diagnostic record.
- Output: Any representing the detected error diagnostic.
- Exceptions: None.

classifyError(errorCode: str, errorContext: dict[str] = Any) → Any

- Transition: classifies the error based on the provided error code and contextual information.
- Output: Any representing the determined error classification.
- Exceptions: None.

logError(errorDiagnostic: Any) → void

- Transition: records the provided error diagnostic information in persistent error logs.
- Output: None.
- Exceptions: IOError if the error log cannot be written.

handleErrorRecovery(errorDiagnostic: Any) → Any

- Transition: determines and returns an appropriate recovery action based on the provided error diagnostic.
- Output: Any representing the recovery action.
- Exceptions: None.

getErrorSource(errorDiagnostic: Any) → str

- Transition: identifies the source component associated with the provided error diagnostic.
- Output: str indicating the source of the error.
- Exceptions: None.

18.4.5 Local Functions

- `analyzeErrorPattern(errorContext: dict) -> str`: Analyzes error context to identify error patterns
- `determineErrorSource(context: dict) -> str`: Determines the source component from error context
- `formatErrorMessage(errorCode: str, context: dict) -> str`: Formats a human-readable error message
- `suggestRecoverySteps(errorType: ErrorType) -> list[str]`: Generates recovery step suggestions based on error type

19 MIS of Model Structure (M14)

ModelStructure

19.1 Module

The ModelStructure module stores and organizes all voxel and layer data for the model, including per-voxel properties, metadata, and structural dimensions.

19.2 Uses

- `DisplayPartitioning` to remake partitions should the need arise.
- `ErrorDiagnosticHandler` for error handling and diagnostics.

19.3 Syntax

19.3.1 Exported Constants

- `VOXEL_SIZE_X`: float - Voxel size in X dimension
- `VOXEL_SIZE_Y`: float - Voxel size in Y dimension
- `VOXEL_SIZE_Z`: float - Voxel size in Z dimension

19.3.2 Exported Access Programs

Name	In	Out	Exceptions
ModelStructure	voxelPositions: list[Vector3D]	self	None
getLayer	z: int	list[Vector3D]	IndexError
getProperty	index: tuple(int, int, int), property: propertyName	Any	IndexError, KeyError
setProperty	index: tuple(int, int, int), property: propertyName, value: Any	None	IndexError, ValueError
hasProperty	index: tuple(int, int, int), property: propertyName	bool	IndexError

19.4 Semantics

19.4.1 State Variables

- voxelGrid: array3D[Voxel] - Three-dimensional grid storing voxel data
- partitions: list[partitionItem] - List of all current partitions

19.4.2 Environment Variables

None.

19.4.3 Assumptions

- Property names are consistent across voxels
- Property values conform to expected formats
- Passed partitions are validated and do not overlap
- All passed voxel centres are contained within exactly one partition and all partitions cover all voxels

19.4.4 Access Routine Semantics

`ModelStructure(positions: list[Vector3D], partitions: list[partitionItem]) -> self`

- **Transition:** creates a 3D array of blank voxels and initializes partition list
- **Output:** `self` - reference to created `ModelStructure` object
- **Exceptions:** None

`getLayer(z: int) -> list[Vector3D]`

- **Transition:** returns all voxel centres at a specific vertical index
- **Output:** `list[Vector3D]` - the requested layer
- **Exceptions:** `IndexError` if index is out of bounds

`getProperty(index: tuple(int, int, int), property: propertyName) -> Any`

- **Transition:** returns the requested property of the voxel at the given index
- **Output:** `Any` as the property may be represented in various ways
- **Exceptions:** `IndexError` if index is out of bounds

`setProperty(index: tuple(int, int, int), property: propertyName, value: Any) -> None`

- **Transition:** sets the property specified to the value specified of the voxel at the given index
- **Output:** None
- **Exceptions:** `IndexError` if index is out of bounds

`hasProperty(index: tuple(int, int, int), property: propertyName) -> bool`

- **Transition:** returns true if the voxel at the given index has an initialized non-blank value of the specified property
- **Output:** `bool` - indicating property initialization
- **Exceptions:** `IndexError` if index is out of bounds

19.4.5 Local Functions

None.

20 MIS of Graphics Adapter (M15)

GraphicsAdapter

20.1 Module

GraphicsAdapter handles all communication with the graphics API to enable visual rendering and generate a model on the UI.

20.2 Uses

- `ErrorDiagnosticHandler` for error handling and diagnostics.

20.3 Syntax

20.3.1 Exported Constants

- None

20.3.2 Exported Access Programs

Name	In	Out	Exceptions
`requestRender`	key: str, update: dict[str] = Any	void	NetworkError
`getUpdateStatus`	key: str	bool	None
`checkServerStatus`	None	void	NetworkError

20.4 Semantics

20.4.1 State Variables

- `updateStatus`: bool — tracks status of rendering completion.

20.4.2 Environment Variables

- `API_BASE_URL`: *str* — external base URL for establishing environment configurations and enabling backend API requests.

20.4.3 Assumptions

- Backend service is reachable and operational.
- Connection is available for API.
- Valid configurations exist for necessary API operations.

20.4.4 Access Routine Semantics

requestRender(key: str, update: dict[str] = Any) → void

- Transition: sends a rendering request identified by the given key along with the associated update data to the graphics backend.
- Output: None.
- Exceptions: NetworkError if the rendering request cannot be transmitted.

getUpdateStatus(key: str) → bool

- Transition: retrieves the current rendering status associated with the specified update key.
- Output: bool indicating whether rendering has completed successfully.
- Exceptions: None.

checkServerStatus() → void

- Transition: checks connectivity and availability of the graphics backend service.
- Output: None.
- Exceptions: NetworkError if the backend service cannot be reached.

21 MIS of Export Structure (M16)

ExportStructure

21.1 Module

The ExportStructure module defines and implements the internal data structure for representing exported files, including field ordering, data types, and encoding format. It ensures consistency between internal model representations and external file layouts used during export operations.

21.2 Uses

- `ErrorDiagnosticHandler` for error handling and diagnostics.

21.3 Syntax

21.3.1 Exported Constants

- `EXPORT_SCHEMA_VERSION`: str — current version of the export structure schema.
- `FIELD_ORDER`: list[str] — ordered list of field names in export format: ["x", "y", "z", "layer", "materialId", "magnetizationX", "magnetizationY", "magnetizationZ"].
- `DEFAULT_ENCODING`: str — default character encoding for export files (e.g., "UTF-8").

21.3.2 Exported Access Programs

Name	In	Out	Exceptions
`defineStructure`	format: str	dict[str] = Any	UnsupportedFormatError
`getFieldOrder`	format: str	list[str]	UnsupportedFormatError
`getFieldType`	fieldName: str	Any	InvalidFieldError
`validateStructure`	data: Any	bool	StructureMismatchError
`createHeader`	metadata: dict[str] = Any	dict[str] = Any	None

21.4 Semantics

21.4.1 State Variables

- `exportSchema`: ExportSchema — current export structure schema definition.

- `fieldTypes`: dict[str] = DataType — mapping from field names to their data types.

21.4.2 Environment Variables

None

21.4.3 Assumptions

- Export format identifiers match supported formats ("CSV").
- Field names in export data match those defined in `FIELD_ORDER`.
- Data types conform to the schema defined by `fieldTypes`.
- Export structure remains consistent across export operations.

21.4.4 Access Routine Semantics

defineStructure(format: str) → dict[str] = Any

- Transition: selects and defines the export structure schema corresponding to the specified export format and stores it as the active export schema.
- Output: dict[str] = Any representing the defined export schema.
- Exceptions: UnsupportedFormatError if the export format is not supported.

getFieldOrder(format: str) → list[str]

- Transition: retrieves the ordered list of field names associated with the specified export format.
- Output: list[str] representing the field order.
- Exceptions: UnsupportedFormatError if the export format is not supported.

getFieldType(fieldName: str) → Any

- Transition: retrieves the expected data type for the specified export field name.
- Output: Any representing the field's data type.
- Exceptions: InvalidFieldError if the field name is not part of the export schema.

validateStructure(data: Any) → bool

- Transition: checks whether the provided export data conforms to the active export schema and field ordering.

- Output: bool indicating whether the structure is valid.
- Exceptions: StructureMismatchError if the data does not match the export schema.

createHeader(metadata: dict[str] = Any) → dict[str] = Any

- Transition: generates an export header using the provided metadata in accordance with the export schema.
- Output: dict[str] = Any representing the export header.
- Exceptions: None.

21.4.5 Local Functions

- `mapInternalToExport(internalData: VoxelData) -> ExportData`: Maps internal voxel representation to export format structure.
- `validateFieldValue(fieldName: str, value: Any) -> bool`: Validates that a field value matches its expected data type.
- `encodeFieldValue(fieldName: str, value: Any) -> str`: Encodes a field value according to the export format specification.

References

Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, Upper Saddle River, NJ, USA, 2nd edition, 2003.

Daniel M. Hoffman and Paul A. Strooper. *Software Design, Automated Testing, and Maintenance: A Practical Approach*. International Thomson Computer Press, New York, NY, USA, 1995. URL http://citeseer.ist.psu.edu/428727.html.

Jian Huang, Roni Yagel, Vassily Filippov, and Yair Kurzion. An accurate method for voxelizing polygon meshes. pages 119–126, 11 1998. ISBN 0-8186-9180-8. doi: 10.1109/SVV.1998.729593.

Appendix — Reflection

The purpose of reflection questions is to give you a chance to assess your own learning and that of your group as a whole, and to find ways to improve in the future. Reflection is an important part of the learning process. Reflection is also an essential component of a successful software development process.

Reflections are most interesting and useful when they're honest, even if the stories they tell are imperfect. You will be marked based on your depth of thought and analysis, and not based on the content of the reflections themselves. Thus, for full marks we encourage you to answer openly and honestly and to avoid simply writing "what you think the evaluator wants to hear."

Please answer the following questions. Some questions can be answered on the team level, but where appropriate, each team member should write their own response:

1. What went well while writing this deliverable?

Omar: What went well for me during this deliverable was working with a clearer system structure than in previous revisions. The module hierarchy and relationships were much more defined, which made it easier to understand how each component fit into the overall architecture. This helped me focus on implementing and documenting simpler modules efficiently while still contributing meaningfully to the project. Communication within the team was also smoother, which made coordination and integration less stressful and more productive.

Daniel: Overall, the two major issues I foresaw with this deliverable - the restructuring of some modules and the formalization of the more complex pieces of our software - went smoother than I expected. The existing module decomposition made sense to me even though I did not work on the initial revision besides reviewing it, and the only major changes were altering the hierarchy. The formalization of how voxels are created also went better after some research that provided a base inspiration into how to define the algorithm's output.

Andrew: Integrating the UI and backend from Daniel went really well this deliverable. Working with him in unison was very helpful, and we got everything done in expert time.

Olivia: Overall, I think that this deliverable really emphasized understanding the structure of the code prior to execution. As someone who mainly focused on the documentation during Rev 0 and then took a more active role in the code development afterwards, the time spent on developing these design documents forced me to take a step back and understand how the system would ultimately fit together. This helped guide the process of writing the code, as I felt I had a better idea of what was expected of each module.There was also more clarity on how to structure things based on what the system would require, well ensuring all responsibilities and functionalities would be met.

Khalid: In terms of the code, the things that went well were the splitting the structure into multiple layers so they can be edited at the same time without interfering with each other. I was also able to find a way to convert the web application to a desktop application using Electron if needed.

2. What pain points did you experience during this deliverable, and how did you resolve them?

Omar: One of the main pain points I experienced was keeping track of all the module dependencies and ensuring consistency between the Module Guide and the MIS documents as changes were made. Small inconsistencies could easily propagate and cause confusion. I resolved this by frequently cross-referencing the architecture diagram and updating documentation incrementally instead of in large batches. Asking for clarification early and verifying assumptions with teammates also helped prevent larger issues later on.

Daniel: A pain point I experienced was maintaining consistency with my partner. While the split between modules allowed us to work in parallel effectively, there was some overlap in types, which required us to examine our work and ensure we named things in the same fashion. Additionally, ensuring that the functions each module possessed needed to exist and what module would call it posed difficulty at times, especially for more abstract modules that are ultimately more implementation specific in their required functions.

Andrew: I was part of the code team for this deliverable, so like Khalid - I did not have many point points regarding this MIS. Of my own exploration and work, I built the MVP UI and made sure to include an initial renderer, export and import manager using javascript frameworks and previous implementations of STL renderers. The pain point here was to find a suitable renderer and have it work in unison with our other modules, but it was not too much of a headache.

Olivia: At first it was overwhelming to decompose the system in a way that clearly captured responsibilities and interactions without integrating premature design decisions. To help combat this issue, I decided to use the initial module decomposition as a guide. This ensured that when I went into more detail for the modules, I didn't feel stuck on trying to develop a module that didn't work. By keeping this open-minded approach during document development, it placed less pressure on the initial decomposition, and ensured that the modules chosen in our final document were included because they actually hold value.

Khalid: I was part of the code team for this deliverable, so I didn't have any pain points regarding the document. However, in terms of the pain points I experienced for the code, I worked on the splitting the structure into multiple layers so they can be edited at the same time without interfering with each other. The pain point I had was finding an algorithm that

will efficiently split the structure into multiple layers. I ended up using a simple algorithm that will split the structure into multiple layers based on the z-coordinate.

3. Which of your design decisions stemmed from speaking to your client(s) or a proxy (e.g. your peers, stakeholders, potential users)? For those that were not, why, and where did they come from?

Several key design decisions were influenced by discussions with teammates acting as proxies for end users, as well as guidance provided by the TA during feedback and review sessions. The TA's input helped reinforce the importance of clear module boundaries, low coupling, and well defined responsibilities, which directly shaped how the system was decomposed into modules. Feedback from peers also emphasized the need for maintainability and clarity over unnecessary architectural complexity. Design decisions that did not stem directly from client or TA input were informed by course material, software architecture principles discussed in lectures, and the team's prior experience with similar systems.

4. While creating the design doc, what parts of your other documents (e.g. requirements, hazard analysis, etc), it any, needed to be changed, and why?

During the creation of the Design Docs, a major changed revolved around a missing requirement identified - the ability to manually save projects. The SRS required an update throughout many of its sections, as well as the VnV Plan which required an additional functional system test. Many of the other changes were actually in the reverse direction; consistency changes in the SRS or VnV that trickled into changed in the Design Docs, however most of these changes were minor (e.g. the alteration of wording to align with wording in other documents).

5. What are the limitations of your solution? Put another way, given unlimited resources, what could you do to make the project better? (LO_ProbSolutions)

Given unlimited resources, there are a few ideas we have that would count as additions. First, the addition of a module allowing projects to be stored on a remote server (e.g. the cloud) would allow projects and workspaces to be transferred across end user devices seamlessly. This would be especially beneficial to our stakeholders, who may wish to work from (say) a personal laptop and a lab computer. A cloud-based storage system would open up the design to two other ideas; a collaborative editing system, where many users could work on the same model simultaneously. This would require an intermediate module to handle concurrent edits, and would further help speed up work. Second, by storing projects remotely, other tasks could also be made to run remotely; specifically, computation heavy tasks, for instance initial voxel processing, which is currently limited by the user's machine. A final addition made possible by a cloud-based system would involve a module to allow users to leave comments linked to parts of the model to allow for the documentation of potential iteration ideas kept within a project.

6. Give a brief overview of other design solutions you considered. What are the benefits and tradeoffs of those other designs compared with the chosen design? From all the potential options, why did you select the documented design? (LO_Explores)

The primary design pattern used in this project follows the Model View Controller (MVC) architecture. MVC was selected because it provides a clear separation of concerns: the model manages voxel data and application state, the view handles visualization and rendering, and the controller coordinates user interactions and system behavior. This structure aligns well with the project's interactive nature and made the system easier to develop, reason about, and document.

Several alternative design approaches were considered. A monolithic design, where most responsibilities are handled within a small number of modules, was deemed unsuitable because it would tightly couple components and make the system harder to maintain and extend. A layered architecture was also considered, but it did not provide sufficient flexibility for handling complex user interactions and visualization updates. Additionally, an event driven or publish subscribe architecture was explored; while powerful, it would have introduced unnecessary complexity and overhead for a project of this scale. Ultimately, MVC offered the best balance between modularity, clarity, and implementation feasibility, making it the most appropriate choice for this project.