Article

Mandala3D: A Configurable Pipeline for Processing Textured 3D Scans in Cultural Heritage Documentation

Prashant Kaushik [*]

Posted Date: 21 April 2025

doi: 10.20944/preprints202504.1715.v1

Keywords: Mandala3D; 3D scanning; cultural heritage; point cloud processing; automated pipeline

Article

Mandala3D: A Configurable Pipeline for Processing Textured 3D Scans in Cultural Heritage Documentation

Prashant Kaushik

Chaudhary Charan Singh Haryana Agricultural University, Hisar 125001, India; prashantumri@gmail.com

Abstract: The digitization of cultural heritage assets using 3D scanning technologies offers unprecedented opportunities for preservation, analysis, and dissemination. However, raw 3D scan data often requires significant processing to become truly usable, involving steps like format conversion, cleaning, texturing, and metadata enrichment. Manual execution of these steps is time-consuming, error-prone, and lacks consistency, hindering large-scale digitization efforts. This manuscript introduces Mandala3D, a configurable, open-source Python pipeline designed to automate the conversion of textured 3D mesh models (primarily OBJ and PLY formats) into accurately colored, analysis-ready point clouds (PLY and LAZ formats). Mandala3D integrates functionalities from established libraries like Open3D and Trimesh, providing a cohesive workflow that includes mesh validation, optional pre-processing (healing, decimation), robust texture and vertex color application using barycentric interpolation, point cloud post-processing (outlier filtering, normal estimation), comprehensive metadata generation (including checksums and processing provenance), thumbnail creation, and automated reporting. By offering a modular, configurable, and automated approach, Mandala3D aims to enhance the efficiency, consistency, and data quality associated with processing 3D scans in cultural heritage documentation, thereby supporting more effective digital preservation and research.

Keywords: Mandala3D, 3D scanning, cultural heritage, point cloud processing, automated pipeline

1. Introduction

Cultural heritage encompasses the tangible and intangible legacy of past generations, holding immense value for societal identity, historical understanding, and education. The preservation of this heritage faces numerous threats, including environmental degradation, conflict, neglect, and the inherent fragility of materials over time. In response, digital technologies, particularly 3D scanning, have emerged as powerful tools for documenting, analyzing, and safeguarding these invaluable assets [1,2]. Techniques such as laser scanning and photogrammetry allow for the creation of high-fidelity digital surrogates of artifacts, structures, and sites, capturing intricate geometric detail and surface appearance. These 3D models serve diverse purposes, ranging from condition monitoring and virtual restoration planning to academic research, museum exhibits, and public engagement through virtual reality experiences [3,4].

Despite the potential of 3D digitization, the transition from raw scan data to usable, archive-ready digital assets presents significant challenges. Raw data often comes in various formats, may contain geometric noise (e.g., outliers, holes, non-manifold topology), and requires careful processing to accurately represent the object's appearance, particularly its color and texture [5]. Furthermore, associating comprehensive metadata – detailing the object's context, capture process, and subsequent processing steps – is crucial for the long-term usability and interpretation of the digital record, establishing essential provenance [6,7]. Manually performing these tasks using disparate software tools like MeshLab [8], CloudCompare [9], or Blender is feasible for individual models but quickly becomes inefficient and inconsistent when dealing with large collections or extensive site documentation projects. The lack of standardized, automated workflows can lead to variations in output quality, incomplete metadata, and significant bottlenecks in digitization projects.

There is a clear need for integrated software pipelines that automate the complex sequence of operations required to transform raw 3D scans into high-quality, well-documented digital assets. Such pipelines should be flexible enough to accommodate different input data types and quality levels, configurable to meet specific project requirements, and robust enough to handle potential errors gracefully. They should prioritize geometric accuracy, color fidelity, and the generation of comprehensive metadata, ensuring the resulting data is suitable for both immediate analysis and long-term archival.

This manuscript introduces Mandala3D, a novel open-source pipeline developed in Python to address these challenges. Mandala3D provides an end-to-end workflow specifically designed for processing textured 3D mesh scans commonly encountered in cultural heritage documentation. It integrates functionalities for loading common mesh formats, automatically detecting and applying texture maps, performing optional mesh cleaning and simplification, sampling points accurately onto the surface, applying colors based on texture coordinates or vertex data, optionally filtering the resulting point cloud, estimating normals, and generating standardized outputs (PLY, LAZ point clouds), thumbnails, detailed JSON metadata, and summary reports. By automating these steps within a configurable framework, Mandala3D aims to significantly improve the efficiency, reproducibility, and quality of 3D data processing in the cultural heritage domain. This work details the architecture, methodology, implementation, and potential applications of the Mandala3D pipeline, highlighting its contribution to facilitating more effective digital heritage practices.

2. Background and Related Work

The digital representation of three-dimensional objects typically relies on two primary data structures: polygon meshes and point clouds. Polygon meshes, often composed of triangles, represent surfaces as a collection of connected vertices, edges, and faces. They excel at representing continuous surfaces efficiently and often store additional information like texture coordinates (UVs) and vertex colors, crucial for capturing surface appearance [10]. Common mesh formats include OBJ, which stores geometry and material/texture references in human-readable text files, and PLY (Polygon File Format), which can store geometry, color, normals, and other attributes in either ASCII or binary formats [11]. Point clouds, conversely, represent objects as a discrete set of points in 3D space, each potentially associated with attributes like color, intensity, or normal vectors [12]. Point clouds are often the direct output of laser scanners or the intermediate result of photogrammetry pipelines (Structure from Motion). While less explicit in representing surface topology, point clouds can capture immense detail and are well-suited for certain types of analysis and large-scale environment representation. The LAZ format, a compressed version of the LAS (LASer) standard, is widely used for efficiently storing large point clouds, particularly from LiDAR surveys [13]. Mandala3D focuses on converting input meshes (OBJ, PLY) into processed point clouds (PLY, LAZ), leveraging the strengths of meshes for initial representation and texturing, and the utility of point clouds for analysis and visualization.

Several powerful software tools exist for manipulating 3D data. MeshLab [8] is a widely used open-source system for processing and editing unstructured 3D triangular meshes, offering a vast array of filters for cleaning, simplification, and measurement. CloudCompare [9] is another prominent open-source tool, primarily focused on point cloud processing, offering functionalities for registration, comparison, segmentation, and filtering. Blender is a comprehensive open-source 3D creation suite with advanced modeling, texturing, and rendering capabilities. Libraries like Open3D [14] and Trimesh [15] provide Python programmers with extensive functionalities for 3D data processing, including file I/O, geometry manipulation, filtering, registration, and visualization. Open3D offers efficient C++ implementations exposed through Python bindings, particularly strong in point cloud processing and basic mesh operations. Trimesh provides a pure Python interface with a focus on mesh analysis, boolean operations, and interaction with mesh topology and visual attributes, including robust handling of UV coordinates.

While these tools and libraries offer essential functionalities, they often require significant user interaction for complex workflows. Applying a consistent sequence of operations (e.g., load, heal, decimate, sample, color, filter, save) across multiple models typically involves manual scripting or repetitive GUI actions. Few existing solutions provide a readily available, configurable, end-to-end *pipeline* specifically tailored for converting textured meshes from heritage scans into accurately colored, metadata-rich point clouds with built-in provenance tracking. Mandala3D aims to fill this gap by orchestrating functionalities, largely drawn from Open3D and Trimesh, into a cohesive, automated workflow managed through a single interface (CLI and configuration file).

A critical aspect of processing textured meshes is accurately transferring color information to the sampled point cloud. Simple approaches might only transfer existing vertex colors, which are often sparse or represent averaged colors, losing texture detail. Accurate texture mapping requires identifying the location of each sampled point on the original mesh surface, determining its corresponding UV coordinate through interpolation (typically using barycentric coordinates relative to the triangle it falls on), and then sampling the color from the texture image at that UV coordinate [10,16]. This process requires robust handling of mesh topology and UV data, a capability well-supported by libraries like Trimesh, which Mandala3D leverages for its core coloring logic. Handling potential issues like missing UVs, malformed UVs, or the absence of texture files requires fallback strategies, such as using vertex colors or applying a default color, which are incorporated into the Mandala3D design.

Finally, the importance of metadata and provenance cannot be overstated in cultural heritage [6,7]. Knowing the origin of the data, the capture parameters, and every processing step applied is essential for assessing data quality, ensuring reproducibility, and enabling long-term interpretation. Standard metadata schemas like CIDOC CRM [17] provide conceptual models, but practical implementation often requires tailored solutions. Mandala3D addresses this by automatically generating detailed JSON metadata for each processed artifact, recording source file information, applied parameters, processing timestamps, output file details (including checksums), and basic geometry statistics, thus embedding crucial provenance information directly alongside the processed data.

3. The Mandala3D Pipeline: Architecture and Methodology

Mandala3D is designed as a modular and configurable pipeline, implemented in Python, to automate the processing of 3D mesh data into colored point clouds. Its architecture emphasizes robustness, flexibility, and the generation of comprehensive metadata. The core workflow proceeds through a series of distinct stages, each configurable via a YAML file and command-line arguments.

The pipeline commences with **Input and Configuration**. Users specify a base directory containing input models or a single model file. The pipeline searches for models based on a configurable glob pattern (e.g., **/*.obj), supporting common mesh formats like OBJ and PLY. Processing parameters are loaded from a YAML configuration file (mandala3d_config.yaml by default), which defines settings for sampling density, pre/post-processing steps, output formats, project metadata, and logging behavior. Command-line arguments provide a mechanism to override any setting defined in the configuration file, offering flexibility for specific runs or scripting. A merged configuration dictionary, prioritizing CLI arguments over the configuration file, governs the execution of all subsequent steps. Comprehensive logging, configurable to different levels (DEBUG, INFO, WARNING, ERROR) and directed to both the console and an optional log file within the output directory, records the pipeline's operations and any issues encountered.

The next stage involves **Model Discovery and Loading**. Based on the specified base directory and search pattern, the pipeline identifies potential input model files. For each discovered model, it attempts loading using both the Open3D and Trimesh libraries. Open3D is primarily used for its efficient point sampling and basic geometry operations, while Trimesh is favored for its robust handling of mesh topology, UV coordinates, and visual attributes, which are critical for accurate texture mapping and pre-processing steps like healing. The loading process checks for the presence of essential data components like triangles, texture coordinates (UVs), and vertex colors, logging the findings. If a model cannot be loaded successfully by either library or lacks fundamental geometry (triangles), it is flagged, and processing for that specific file is skipped, preventing pipeline failure due to corrupted input. A checksum (SHA256 by default) is calculated for the input file to ensure data integrity tracking.

Following successful loading, **Texture Handling** is performed, primarily for OBJ files which commonly reference external texture maps. If a specific texture file is not provided via configuration or CLI, Mandala3D attempts to auto-detect a corresponding texture map based on common naming conventions (e.g., a PNG or JPG file with the same base name as the OBJ file, or common names like 'texture.png', located in the same directory). If a texture file is identified, it is loaded using the Pillow library. The pipeline validates the image, converts it to a standard RGB format (handling grayscale or RGBA inputs), and stores it as a NumPy array for efficient color sampling later. A checksum is also calculated for the texture file, if found. If no texture is found or the input is not an OBJ file, this stage is bypassed, and subsequent coloring will rely on vertex colors or defaults.

Before point sampling, optional **Mesh Pre-processing** steps can be applied, primarily utilizing Trimesh's capabilities. This stage aims to improve mesh quality and potentially reduce complexity. An optional mesh healing step (mesh_heal) attempts to fix common issues like small holes (using mesh.fill_holes()), remove unreferenced vertices or duplicate faces (mesh.process()), and optionally retain only the largest connected component of the mesh (mesh.split()). This can improve the robustness of subsequent sampling and coloring. Additionally, an optional mesh decimation step (mesh_decimation_target) can simplify the mesh to a target number of faces using Open3D's quadric decimation algorithm. This is useful for reducing the computational load when dealing with overly dense input meshes, although it may affect fine details and potentially alter UV coordinates or vertex colors, requiring careful consideration. The pipeline logs which pre-processing steps were applied and the resulting changes in vertex/face counts.

The core conversion step is **Point Sampling**. The pipeline samples a user-defined number of points (points parameter) from the surface of the (potentially pre-processed) Open3D mesh. It preferentially uses Poisson Disk sampling (sample_points_poisson_disk) if vertex normals are available, as this method generally produces a more uniform point distribution compared to simple uniform sampling [18]. If normals are unavailable or the mesh is too small, it falls back to uniform sampling (sample_points_uniformly). The actual number of points sampled might differ slightly from the target due to the nature of the algorithms.

Subsequently, **Color Application** assigns color information to each sampled point. This is a critical stage for achieving visual fidelity. The primary strategy involves using the loaded texture map and the mesh's UV coordinates. For each sampled point, its closest point on the Trimesh surface is found (mesh.nearest.on_surface). The barycentric coordinates of this closest point within its corresponding triangle are calculated. These barycentric coordinates are then used to interpolate the UV coordinates of the triangle's vertices (obtained from mesh_trimesh.visual.uv). The resulting interpolated UV coordinate is mapped to pixel coordinates in the texture image (correcting for the V-axis orientation difference between UV space and image space), and the color is sampled from the texture NumPy array. If texture mapping is not possible (no texture loaded, no UV coordinates on the mesh, or an error occurred during interpolation), the pipeline attempts a fallback strategy using vertex colors, if available. This involves a similar process of finding the closest point, calculating barycentric coordinates, and interpolating the vertex colors (mesh_trimesh.visual.vertex_colors) of the corresponding triangle's vertices. If neither texture nor vertex colors can be applied, a default gray color is assigned to the points. The source of the applied color ("texture", "vertex", or "default") is recorded in the metadata.

After color application, optional **Point Cloud Post-processing** steps can refine the generated point cloud using Open3D functions. Statistical Outlier Removal (SOR) (pointcloud_sor) can be enabled to remove points whose average distance to their neighbors is statistically significantly larger than the average distance in the point cloud, effectively removing sparse outliers [9,14]. Radius Outlier Removal (ROR) (pointcloud_ror) removes points that have fewer than a specified number of neighbors within a given radius, useful for removing smaller, isolated noise clusters. Parameters for these filters (number of neighbors, standard deviation ratio for SOR, radius and minimum points for ROR) are configurable. Furthermore, point cloud normals can be estimated (pointcloud_normals) using methods like estimate_normals, which considers the local neighborhood of each point to determine surface orientation. Normals can be optionally oriented consistently, often towards a heuristic camera location, improving visualization and downstream analysis.

The penultimate stage is **Output Generation**. The processed point cloud is saved in standard formats. A binary compressed PLY file is always generated. Optionally (save_laz), a LAZ file can also be created using the laspy library, providing efficient compression valuable for large datasets. The LAZ writer populates standard fields including coordinates, RGB colors (converted to the required 16-bit scale), and potentially normals if they were estimated and the point format supports them. Additionally, a PNG **Thumbnail** image of the point cloud is generated using Open3D's off-screen rendering capabilities, providing a quick visual preview of the artifact. Checksums are calculated for all generated output files.

Throughout the process, **Metadata and Provenance Tracking** are paramount. For each successfully processed artifact, a detailed JSON metadata file is created. This file includes identifiers, paths and checksums for source files (mesh and texture), paths and checksums for all generated output files (PLY, LAZ, thumbnail), a record of the specific processing parameters used for that artifact (point count, filter settings, etc.), the determined color source, flags indicating which pre/post-processing steps were applied, key geometry statistics of the final point cloud (point count, bounding box, center), and the total processing time. This detailed record ensures traceability and reproducibility. Furthermore, an overall project metadata JSON file aggregates information about the pipeline run, the project settings (name, location, curator, license), and a summary list of all processed artifacts.

Finally, an optional **Reporting** stage (generate_report) compiles information from the project metadata into a human-readable Markdown report. This report summarizes the project details, lists the processed artifacts with key statistics (point count, color source, processing time), and includes embedded thumbnails, providing a convenient overview of the processing batch.

4. Implementation Details

Mandala3D is implemented entirely in Python 3, leveraging several high-quality open-source libraries for its core functionalities. The choice of Python allows for rapid development, easy integration of existing libraries, and broad accessibility for users and potential contributors.

The primary dependencies include Open3D [14] and Trimesh [15]. Open3D is utilized for its efficient implementations of point cloud sampling algorithms (Poisson Disk, Uniform), point cloud filtering (Statistical Outlier Removal, Radius Outlier Removal), normal estimation, file I/O for PLY meshes and point clouds, and off-screen visualization for thumbnail generation. Trimesh is crucial for its robust handling of mesh topology and visual attributes. It is used for loading meshes (providing fallback and detailed attribute access), accessing and validating UV coordinates and vertex colors, performing mesh pre-processing steps like healing (fill_holes, process, split), and, critically, for the accurate determination of surface points (nearest.on_surface) and barycentric coordinates (triangles.points_to_barycentric) required for interpolating UVs and vertex colors during the color application stage.

Texture image loading and manipulation are handled by Pillow (a fork of PIL), the standard Python imaging library. It allows reading various image formats and converting them into NumPy arrays suitable for direct pixel access during color sampling. The laspy library [13] is employed for writing point cloud data to the compressed LAZ format, providing control over the LAS header and point record data, including coordinates, RGB colors, and potentially normals.

Configuration management is implemented using PyYAML, allowing users to define pipeline parameters in a human-readable YAML file (mandala3d_config.yaml). Python's built-in argparse module is used to handle command-line arguments, providing a standard interface for users and allowing CLI arguments to override settings specified in the YAML file. This layered configuration approach (defaults < YAML < CLI) offers significant flexibility.

Progress indication for potentially long-running batch operations is provided using tqdm, which displays informative progress bars in the console. Standard Python libraries like os, pathlib, glob, json, datetime, hashlib, and logging are used extensively for file system interaction, data serialization, timekeeping, checksum calculation, and robust logging, respectively. The logging framework is configured to provide informative output to both the console and a persistent log file, facilitating debugging and tracking of the pipeline's execution.

The color application logic, a core part of the pipeline, relies heavily on the synergy between Trimesh and NumPy. Trimesh efficiently finds the closest surface point and corresponding triangle for each sampled 3D point. It then computes the barycentric coordinates, which represent the sampled point's position within that triangle as a weighted average of the triangle's vertices. These same weights are then applied, using efficient NumPy array operations (specifically einsum or broadcasting multiplication followed by summation), to interpolate either the UV coordinates or the vertex colors associated with the triangle's vertices. The interpolated UV coordinate is then scaled and converted to integer pixel indices to sample the color from the texture image (stored as a NumPy array). This vectorized approach using NumPy is crucial for achieving reasonable performance when processing hundreds of thousands or millions of points.

Error handling is incorporated throughout the pipeline. File loading, processing steps, and output saving are wrapped in try...except blocks to catch potential exceptions (e.g., file not found, corrupted data, library errors). Errors related to a single model typically result in that model being skipped, allowing batch processing to continue, with the error being logged. Critical errors, such as the inability to create the output directory, may halt the entire pipeline. Checksum calculations using hashlib (SHA256) provide a mechanism for verifying the integrity of input and output files over time or after transfer.

5. Evaluation and Use Cases

Mandala3D is designed to address practical needs within the cultural heritage digitization workflow. Its effectiveness can be evaluated based on several criteria: efficiency, output quality (geometric fidelity and color accuracy), data completeness (metadata and provenance), and usability.

Efficiency: By automating the sequence of processing steps, Mandala3D offers significant time savings compared to manual workflows involving multiple software packages. Batch processing capabilities allow users to process large collections of scans overnight or in the background. While processing time depends heavily on factors like input mesh complexity (face count), number of points sampled, and enabled pre/post-processing steps, the use of efficient libraries like Open3D and vectorized NumPy operations ensures reasonable performance. Performance analysis (potentially using Matplotlib plots generated by future enhancements) could quantify time savings compared to manual methods across different datasets. The progress bars provided by tqdm offer immediate feedback during batch runs.

Output Quality: The pipeline prioritizes accurate color representation. By using Trimesh's robust geometry queries and barycentric interpolation for UV mapping, Mandala3D achieves more accurate texture color transfer to the point cloud than simpler methods relying solely on vertex colors or nearest vertex mapping. The optional point cloud filtering steps (SOR, ROR) help remove noise inherent in raw scan data, improving the clarity of the final point cloud. Normal estimation provides additional geometric information valuable for rendering and analysis. The quality of the output is, of course, dependent on the quality of the input mesh and texture map. Mandala3D provides tools to condition the data but cannot fundamentally fix severely flawed input. Qualitative comparisons of point clouds generated by Mandala3D versus those processed manually or by other tools would demonstrate its effectiveness in preserving detail and color fidelity.

Data Completeness: A key contribution of Mandala3D is its focus on metadata and provenance. The automatic generation of detailed JSON metadata for each artifact, including source information, processing parameters, checksums, timestamps, and geometry statistics, ensures that crucial contextual information is captured alongside the data. The overall project metadata file and the optional Markdown report provide accessible summaries. This embedded provenance is critical for long-term archival, data sharing, and ensuring the scientific validity of research based on the processed data.

Usability: Mandala3D offers a command-line interface combined with a YAML configuration file, providing flexibility for both interactive use and scripted automation. The default configuration provides sensible starting points, while the ability to override any parameter via the CLI caters to specific needs. Comprehensive logging aids in troubleshooting, and the generated report offers a quick overview of batch processing results.

Use Cases: Mandala3D is applicable to a wide range of cultural heritage scenarios:

- **Artifact Documentation:** Processing scans of pottery, sculptures, tools, or other museum objects to create detailed, colored point clouds for digital archives, virtual exhibits, or condition analysis.
- **Architectural and Site Scanning:** Handling scans of building facades, intricate carvings, archaeological excavations, or rock art panels, converting potentially large mesh models into manageable point clouds for measurement, monitoring, or reconstruction studies.
- **Research:** Providing researchers with consistently processed, well-documented 3D datasets suitable for geometric morphometrics, surface analysis, or comparative studies.
- **Conservation:** Generating accurate point clouds to aid conservators in planning interventions, documenting treatment progress, or creating digital records before and after conservation efforts.
- **Education and Outreach:** Creating assets suitable for integration into virtual tours, educational applications, or online collections, making heritage more accessible.

While Mandala3D provides a robust framework, empirical evaluation on diverse datasets representing different object types, scales, and scanning modalities would be valuable to further refine default parameters and assess performance characteristics across various scenarios.

6. Discussion and Future Work

Mandala3D represents a significant step towards streamlining and standardizing the processing of textured 3D scans within the cultural heritage domain. By integrating essential functionalities from powerful libraries like Open3D and Trimesh into an automated, configurable pipeline, it addresses key bottlenecks associated with manual processing. Its emphasis on accurate color transfer using UV interpolation, optional data conditioning steps, and comprehensive metadata generation enhances the quality, usability, and long-term value of the resulting digital assets. The provision of both PLY and compressed LAZ outputs caters to different storage and analysis needs, while the open-source nature encourages adoption, modification, and community contribution.

Despite its capabilities, Mandala3D has limitations and ample scope for future development. Currently, mesh healing is basic; incorporating more advanced mesh repair algorithms (potentially via libraries like PyMeshFix) could handle more severely damaged input models. The texture handling assumes a single texture map per mesh; extending support for multi-tile texturing systems like UDIMs would increase its applicability to very high-resolution models, although this presents significant implementation complexity. The point sampling currently uses standard methods; implementing curvature-aware sampling could lead to more efficient representation of geometrically complex objects.

A major area for future enhancement involves integrating more sophisticated analysis and visualization, particularly using libraries like Matplotlib. As previously brainstormed, generating histograms of point density, color distributions (RGB, HSV), and normal orientations would provide valuable quantitative insights into the output data quality. These plots could be automatically saved and embedded within the Markdown report, creating a much richer documentation package. Furthermore, plotting processing performance metrics (time vs. complexity) after batch runs could help users optimize parameters and understand pipeline behavior.

Further development could also focus on enhancing input validation. More rigorous checks on mesh topology (watertightness, orientability, self-intersections) and UV coordinate validity could preemptively identify issues with source data and provide more informative warnings or errors to the user. Aligning the generated metadata structure more closely with established standards like CIDOC CRM or Linked Art would improve interoperability with larger heritage information systems, although this requires careful mapping of concepts.

Usability could be improved by offering selective execution of pipeline stages, allowing users to run only specific steps (e.g., only perform mesh healing, or only generate metadata for existing files). While currently CLI-based, a graphical user interface (GUI) could make the pipeline accessible to a wider range of users less comfortable with command-line tools. Performance for very large datasets or extensive batch runs could potentially be improved by exploring parallel processing options, perhaps processing multiple models concurrently (though care must be taken regarding memory consumption and library thread-safety).

In conclusion, Mandala3D provides a solid foundation for automated 3D scan processing in cultural heritage. Its modular design, configurable nature, and focus on data quality and provenance make it a valuable tool for digitization projects. The planned future enhancements, particularly the integration of quantitative analysis and visualization with Matplotlib and more advanced data validation, promise to further increase its utility and contribution to the field of digital heritage preservation and research.

7. Conclusion

The increasing use of 3D scanning in cultural heritage necessitates efficient, reliable, and reproducible methods for processing raw scan data into usable digital assets. Mandala3D addresses this need by providing an integrated, open-source Python pipeline that automates the conversion of textured 3D meshes into accurately colored point clouds, complete with comprehensive metadata and provenance tracking. By leveraging the strengths of libraries like Open3D and Trimesh and incorporating features such as configurable pre/post-processing, checksum verification, and automated reporting, Mandala3D significantly reduces the manual effort and potential inconsistencies associated with traditional workflows. It enhances data quality through accurate texture mapping and optional cleaning steps, and ensures long-term data usability through detailed metadata generation. While opportunities for future enhancement exist, particularly in advanced analysis, visualization, and support for more complex data types, Mandala3D currently offers a valuable and practical tool for researchers, conservators, museum professionals, and anyone involved in the critical task of digitally documenting and preserving our shared cultural heritage. Its open-source nature invites collaboration and further development, promising continued relevance and impact within the digital heritage community.

References

[1] Pieraccini, M., Guidi, G., & Atzeni, C. (2001). 3D digitizing of cultural heritage. *Journal of Cultural Heritage*, 2(1), 63-70.

[2] Remondino, F. (2011). Heritage recording and 3D modeling with photogrammetry and 3D scanning. *Remote Sensing*, 3(6), 1104-1138.

[3] Ioannides, M., Fink, E., Brumana, R., Patias, P., Doulamis, A., Martins, J., & Wallace, M. (Eds.). (2016). *Digital Heritage. Progress in Cultural Heritage: Documentation, Preservation, and Protection*. Springer.

[4] Barsanti, S. G., Guidi, G., & Remondino, F. (2014). 3D visualization of cultural heritage artifacts: The state of the art. *Virtual Archaeology Review*, 5(10), 69-79.

[5] Boehler, W., & Marbs, A. (2004). 3D scanning and photogrammetry for heritage recording: A comparison. *Proceedings of the 12th International Conference on Geoinformatics*, 291-298.

[6] Dallas, C. (2007). Metadata, description, and digital provenance: traces and trust in the networked information landscape. *Journal of the American Society for Information Science and Technology*, 58(12), 1813-1828.

[7] Doerr, M. (2003). The CIDOC CRM–an ontological approach to semantic interoperability of metadata. *AI magazine*, 24(3), 75-75.

[8] Cignoni, P., Callieri, M., Corsini, M., Dellepiane, M., Ganovelli, F., & Ranzuglia, G. (2008). MeshLab: an open-source mesh processing tool. *Eurographics Italian Chapter Conference*, 129-136.

[9] CloudCompare (Software). GPL V2 license. Retrieved from http://www.cloudcompare.org/

[10] Botsch, M., Kobbelt, L., Pauly, M., Alliez, P., & Levy, B. (2010). *Polygon mesh processing*. CRC Press.

[11] Turk, G. (1994). The PLY polygon file format. *Georgia Institute of Technology*.

[12] Vosselman, G., & Maas, H. G. (Eds.). (2010). *Airborne and terrestrial laser scanning*. Whittles Publishing.

[13] LAS Specification. American Society for Photogrammetry and Remote Sensing (ASPRS). Retrieved from https://www.asprs.org/divisions-committees/lidar-division/laser-las-file-format-exchange-activities (Accessed via laspy documentation reference). See also: laspy documentation: https://laspy.readthedocs.io/

[14] Zhou, Q. Y., Park, J., & Koltun, V. (2018). Open3D: A modern library for 3D data processing. *arXiv preprint arXiv:1801.09847*. See also: Open3D documentation: http://www.open3d.org/docs/release/

[15] Dawson-Haggerty, M., et al. (2019). Trimesh (Software). MIT License. Retrieved from https://trimsh.org/

[16] Floater, M. S. (1997). Parameterization and smooth approximation of surface triangulations. *Computer Aided Geometric Design*, 14(3), 231-250. (Example of relevant theory for interpolation).

[17] Le Boeuf, P., Doerr, M., Ore, C. E., & Stead, S. (Eds.). (2015). Definition of the CIDOC Conceptual Reference Model. *CIDOC CRM Special Interest Group*, Version 6.2.

[18] Corsini, M., Cignoni, P., & Scopigno, R. (2012). Efficient and flexible sampling with blue noise properties of triangular meshes. *IEEE Transactions on Visualization and Computer Graphics*, 18(6), 914-924. (Example paper on Poisson Disk sampling on meshes)