OBJ File Transformation Tool: Scaling, Rotation and Movement

Bachelor Thesis

Submitted in partial fulfillment of the requirements
for the degree of Bachelor of Science in Computer Science

Author: Dimitrios Makrogiannis (Student ID: 4676)
Supervisor: Dr. Xenophon Zabulis

Department of Computer Science
School of Sciences & Engineering
University of Crete
Heraklion, Greece

September 2025

Contents

Contents

List of Tables

List of Figures

Introduction

Context and Motivation

In the field of 3D graphics and simulation, geometric transformations play a fundamental role in modifying and aligning objects within a scene. The Wavefront OBJ file format, a widely used standard for storing 3D models, encodes information about vertices, faces, and other graphical elements in a human-readable text format.

Transforming such models—by moving, scaling, or rotating them—is often necessary in both automated pipelines and manual editing tasks. Despite the simplicity of the transformation concepts, implementing a robust tool that handles these operations correctly and efficiently requires careful attention to geometric and numerical details.

Problem Statement

OBJ files often need to be repositioned or resized to fit within specific bounding boxes or coordinate systems, particularly when integrating them into larger scenes or preparing them for physical simulation or 3D printing. Without proper tools, these transformations must be performed manually or with general-purpose software, which may not provide the fine control needed for automated workflows.

Project Objective

The aim of this thesis is to develop a lightweight transformation tool that:

- Parses and reads OBJ files containing 3D vertex and face data,
- Applies translation (movement) to reposition the object,
- Applies scaling to fit the object within a destination bounding box,
- Applies rotation around the three principal axes,
- Writes the transformed object back into a valid OBJ file.

Approach Summary

This tool was implemented using `C++`, with custom logic to handle geometry processing and linear algebra calculations. The transformations are applied through matrix-based operations and coordinate-wise manipulation. The system ensures that:

- Transformed objects remain fully enclosed within their target bounding boxes,
- Operations are precise and avoid floating-point drift where possible,
- The object is centered appropriately after scaling and rotation.

Thesis Structure

The remainder of this document is organized as follows:

- **Chapter 1** describes the input structure and parsing approach for OBJ files.
- **Chapter 2** outlines the output generation and how transformed files are written.
- **Chapter 3** focuses on movement algorithms and how translation is applied.
- **Chapter 4** discusses the scaling operation and its associated algorithm.
- **Chapter 5** details the rotation procedure and mathematical foundations.
- **Chapter 6** presents the tools and libraries used during development.
- **Chapter 7** concludes the thesis and discusses potential extensions.
- **Appendix** contains code listings, input/output examples, and diagrams.

Tools Used

This project was developed using a combination of libraries, and development tools aimed at enabling accurate and efficient geometric transformations on 3D models in OBJ format.

Programming Language

The transformation tool was implemented entirely in **Python 3.10**, chosen for its simplicity, readability, and strong built-in support for file handling, mathematical operations, and scripting.

Libraries and Modules

The following standard Python modules were used:

- `math` — used for trigonometric functions in rotation operations.
- `json` — used to parse the configuration input files.
- `os` and `shutil` — used for directory creation, file handling, and cleanup of temporary files.
- `sys` — used for reading command-line arguments and managing execution flow.

No external or third-party libraries were used, making the tool lightweight and portable.

Development Environment

Development and testing were performed on:

- **Operating System**: Windows 10
- **Editor/IDE**: Visual Studio Code with Python extension
- **Terminal**: Powershell for execution and debugging
- **Version Control**: Git

Visualization Tools

To inspect and validate the input and output 3D models, the following visualization tool was used:

- **MeshLab** — an open-source tool for viewing, transforming, and analyzing 3D meshes in OBJ and other formats.

Input/Output Tools

Configuration files were created manually using a standard text editor in `.json` format. Output files were saved in OBJ format and verified visually in MeshLab and functionally via inspection of coordinate values.

1 Input

Before performing any transformation, the tool must correctly read and interpret the input files. These files define the object and destination geometry, making them essential for guiding the entire process.

1.1 Description of Input File Format

The transformation system uses two types of inputs:

- Two **Wavefront OBJ files**, which contain the 3D models to be transformed and used as spatial references.
- A **JSON configuration file**, which specifies which models to process, how to transform them, and whether any rotation is to be applied.

Wavefront OBJ Format

The Wavefront OBJ format is a simple, human-readable file format used to describe the geometry of 3D models. It supports vertices, normals, texture coordinates, and faces.

Each line in an OBJ file starts with a keyword, followed by numeric values depending on the element being defined. The most relevant elements used by the transformation tool are summarized in Table 1.

Keyword	Meaning	Example
`v`	Vertex (X, Y, Z coordinates)	`v 1.0 2.0 3.0`
`vt`	Texture coordinate (U, V)	`vt 0.5 1.0`
`vn`	Vertex normal (X, Y, Z)	`vn 0.0 1.0 0.0`
`f`	Face made of vertex indices	`f 1 2 3` or `f 1/1/1 2/2/2 3/3/3`

Table 1: Common elements in the OBJ file format

```
# Simple cube OBJ
v -1.0 -1.0 -1.0
v  1.0 -1.0 -1.0
v  1.0  1.0 -1.0
v -1.0  1.0 -1.0
v -1.0 -1.0  1.0
v  1.0 -1.0  1.0
v  1.0  1.0  1.0
v -1.0  1.0  1.0
f 1 2 3 4
f 5 6 7 8
f 1 5 8 4
f 2 6 7 3
f 1 2 6 5
f 4 3 7 8
```

Figure 1: Raw OBJ structure of a simple cube model

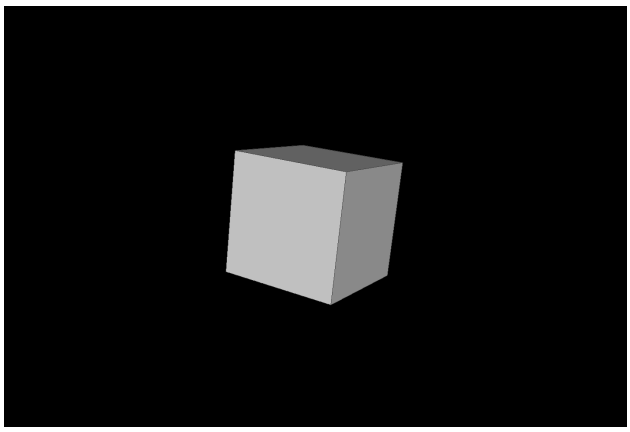

Figure 2: The same cube visualized in MeshLab

Only `v` (vertices) are used by the tool. Texture coordinates, faces, and normals are ignored during transformation but preserved in the output for compatibility.

1.2 Preprocessing and Validation

Before transformations are applied, the tool performs several validation checks:

- Checks if the OBJ file paths in the JSON config exist and are readable.

- Confirms that the OBJ file contains valid ‘v‘ entries (i.e., 3 floating-point values).
- Validates the JSON structure and required fields.
- Verifies that the object-to-move is not degenerate (non-zero size in X, Y, Z).

If validation fails, the process halts and an error is displayed to the user.

1.3 Parsing Strategy

OBJ files are read line-by-line using Python's built-in file handling. The program:

- Parses lines starting with ‘v‘ to extract and store vertex coordinates.
- Calculates bounding boxes and centers from these vertex values.
- Copies all other lines (‘f‘, ‘vn‘, ‘vt‘, etc.) to preserve geometry.

This simple but effective parsing strategy is sufficient for most standard OBJ models.

1.4 Data Structures Used

The tool uses native Python data structures:

- `float` — to store min, max, and transformed coordinates.
- `dict` — for bounding boxes and rotation parameters.
- `tuple` — to represent the 3D center of an object.
- `list of strings` — for reading/writing lines to files.

No external 3D or graphics libraries are used to keep the tool lightweight and minimal.

1.5 JSON Input Configuration

The JSON input defines which objects to transform and how. It must include:

- `"object_to_move"` — path to the OBJ file that will be transformed.
- `"destination"` — path to a reference OBJ file that defines the destination bounding box.
- `"rotation"` — **optional** dictionary specifying angles (in degrees) to rotate around X, Y, Z axes.

If the ‘rotation‘ field is not provided, the object is only moved and scaled.

Input Example: With Rotation

Figure 3: JSON input specifying rotation around X, Y, and Z axes

Input Example: Zero Rotation

Figure 4: JSON input file with all rotation angles set to zero

Minimal Input Example

Figure 5: JSON input with only object paths — no rotation specified

2 Output

After all transformations are applied, the final result is written to a new OBJ file. The output must reflect the updated geometry while preserving the structure and integrity of the original model.

2.1 Output File Format

The output produced by the transformation tool is a modified Wavefront OBJ file, structurally identical to the original input format but with altered vertex data. This transformed file is saved in a dedicated output directory named `patched/`.

The output file:

- Retains the original OBJ structure.
- Includes updated `v` (vertex) entries reflecting the applied transformations.
- Preserves all other elements such as faces (`f`), texture coordinates (`vt`), and vertex normals (`vn`) exactly as they appeared in the input file.

The filename of the final output is derived from the original file, with the suffix `_moved.obj` appended. For example:

- Input: `bunny.obj`
- Output: `bunny_moved.obj`

2.2 Output Specification

The final output reflects the cumulative effect of the following geometric transformations, applied in a fixed order:

1. **Rotation** (optional): If rotation angles are specified in the JSON configuration file, the object is rotated around the X, Y, and Z axes in sequence. The rotation is applied using standard rotation matrices.
2. **Scaling**: The object is scaled uniformly so that its bounding box fits entirely within the bounding box of the destination object. The smallest dimension ratio between the destination and source objects is selected as the scale factor to ensure full containment.

3. **Movement (Translation)**: After scaling, the object is translated so that its geometric center aligns with the center of the destination object.

The order of operations is critical to achieving accurate and predictable positioning of the object. Rotating or scaling after translation would lead to incorrect placement and possibly clipping outside the destination bounds.

Throughout this process, only the vertex coordinates are modified. All other data — including face definitions, texture coordinates, and normals — are left untouched to preserve the original topological and visual characteristics of the object.

2.3 Post-processing

Once the transformations are complete and the output file is written, the tool performs a series of post-processing steps:

- **File Cleanup:** Any temporary files created during intermediate stages (such as rotated or scaled versions) are deleted.
- **Directory Management:** The final output is moved to the `patched/` directory. The temporary directory (`temp/`) is removed entirely to keep the working environment clean.
- **Output Notification:** A terminal message is printed indicating the scale factor used and the final output file location.

These post-processing steps ensure that users are only left with the final transformed result, without clutter from intermediate files, while preserving data integrity and traceability.

3 Movement and Algorithm Analysis

Movement (Translation) ensures that the object is correctly positioned relative to the destination model. By aligning their centers, the tool prepares both models for consistent scaling and placement.

3.1 Objective of Movement

In 3D geometry, "moving" an object refers to a **translation** — shifting every point of the object by the same vector so that its position in space changes without altering its size, shape, or orientation.

In this project, movement is used to **align the center of the object to move with the center of the destination object**. This ensures that once scaling and rotation are applied, the object is positioned correctly within the destination's bounding volume.

3.2 Translation Algorithm

Translation is achieved by computing the center point of both objects (source and destination) and adjusting all vertex coordinates of the source so that its center matches that of the destination.

Let:

- $C_s = (x_s, y_s, z_s)$ be the center of the object to move
- $C_d = (x_d, y_d, z_d)$ be the center of the destination object
- $v = (x, y, z)$ be a vertex in the source object

The new translated vertex v' is computed as:

$$v' = v - C_s + C_d$$

This formula ensures that all vertices are shifted consistently so the object is repositioned but remains geometrically intact.

Implementation

This operation is applied to each vertex during a line-by-line pass of the OBJ file. For each line starting with `v`, the tool:

- Extracts x, y, z
- Applies the translation using the formula above
- Writes the updated vertex line to the output file

3.3 Edge Cases

The translation algorithm handles all typical cases, including:

- **Negative Coordinates:** Objects with vertices in negative coordinate space are translated correctly since subtraction and addition are handled uniformly.
- **Degenerate Objects:** If the source object has all vertices at the same point (i.e., zero-size bounding box), the center computation results in a fixed point, and translation has no meaningful visual effect.
- **Floating-Point Precision:** Very large or small coordinates can cause small numerical errors during subtraction/addition, but these are generally negligible.

3.4 Performance Considerations

The movement (translation) algorithm is highly efficient, with a time complexity of:

$$\mathcal{O}(n)$$

where n is the number of vertices. Each vertex is processed exactly once with a fixed number of arithmetic operations.

In practice, even large models (e.g., 100,000+ vertices) can be translated in milliseconds. The tool uses Python's built-in file I/O and arithmetic operations, which are optimized for such linear passes.

4 Scaling and Algorithm Analysis

Scaling adjusts the size of the object so that it fits entirely within the destination's bounding box. It ensures proper proportions are maintained while avoiding any overlap or clipping.

4.1 Objective of Scaling

Scaling in 3D graphics refers to the process of resizing an object by multiplying each of its vertex coordinates by a scale factor. The purpose of scaling in this project is to ensure that the object to move can fully fit inside the destination bounding volume, without intersecting or exceeding its boundaries.

Uniform scaling applies the same factor across all axes (X, Y, Z), preserving the object's proportions. **Non-uniform scaling**, on the other hand, applies different scale factors to each axis, which can distort the object.

This tool uses **uniform scaling**, selecting a single factor to maintain the object's shape while ensuring it fits within the destination's bounding box.

4.2 Scaling Algorithm

To compute the scaling factor, the tool first determines the axis-aligned bounding box of both the object to move and the destination object. This involves finding the minimum and maximum coordinates in all three axes.

Let:

- $S_{obj} = (\Delta x_o, \Delta y_o, \Delta z_o)$ be the size of the source object
- $S_{dest} = (\Delta x_d, \Delta y_d, \Delta z_d)$ be the size of the destination object

The uniform scale factor s is calculated as:

$$s = \min\left(\frac{\Delta x_d}{\Delta x_o}, \frac{\Delta y_d}{\Delta y_o}, \frac{\Delta z_d}{\Delta z_o}\right)$$

This ensures that after scaling, the object fits completely within the destination bounding box in all three dimensions.

Centering Before and After Scaling

Scaling is always performed about the origin (0,0,0). To ensure correct placement:

1. The object is first centered at the origin (by subtracting its center).

2. The scaling factor is applied to all vertices.

3. The object is then moved (translated) to match the destination's center.

This avoids the common problem of "scaling offset," where the object appears shifted after scaling.

Axis	**Source Size**	**Destination Size**	**Ratio**
X	3.2	1.8	0.56
Y	2.1	1.5	0.71
Z	4.0	2.2	0.55

Table 2: Bounding box comparison and computed ratios per axis

4.3 Ensuring Fit Within Bounds

To prevent clipping or overflow, the tool uses the minimum of the three axis-wise scale ratios as the global uniform scale factor. This guarantees that:

- The scaled object remains fully enclosed within the destination bounding box.
- No part of the object exceeds the destination's dimensions in any axis.

The example below shows the result of scaling a human model into a cylinder-shaped bounding box.

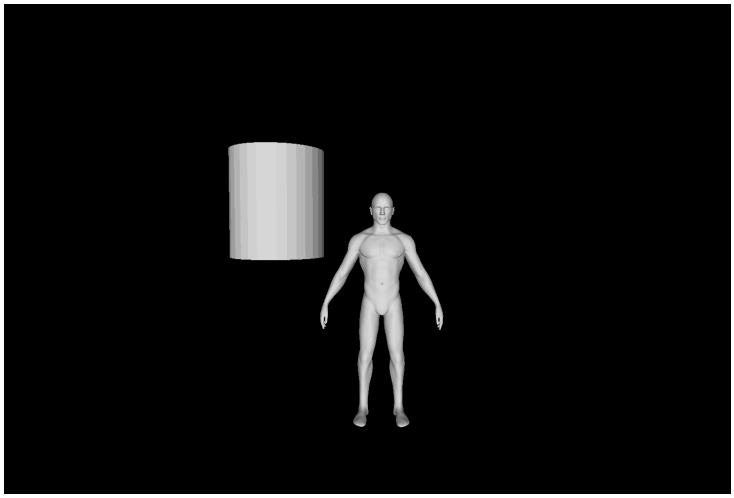

Figure 6: Before scaling: human model and destination cylinder (not fitted)

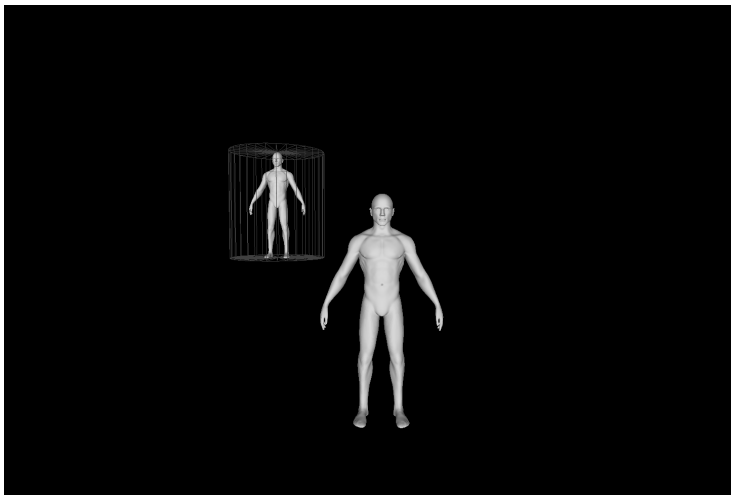

Figure 7: After scaling: human model scaled to fit inside the cylinder

These images clearly show how the tool resizes the object proportionally to fully enclose it within the destination geometry.

4.4 Performance Considerations

Scaling is a linear operation applied to each vertex in the OBJ file. Its time complexity is:

$$\mathcal{O}(n)$$

where n is the number of vertices.
The process involves:

- One pass to calculate the bounding box of the object
- One pass to apply the scale factor to each vertex

Since only basic arithmetic operations are used, and no external libraries are needed, memory usage remains low, and the operation is fast even on large files.

5 Rotation and Algorithm Analysis

Rotation changes the orientation of the object to match the target alignment. It allows the object to be positioned correctly in space before scaling and movement are applied.

5.1 Objective of Rotation

Rotation in 3D space refers to turning an object around one or more of the principal axes: X, Y, and Z. These transformations are common in spatial alignment tasks, where an object must be oriented to match the pose of a target structure.

In this project, rotation is applied before scaling and movement to ensure the object's orientation is corrected prior to fitting and placement.

5.2 Rotation Algorithms

To rotate a 3D object, each vertex (x, y, z) is transformed using rotation matrices about the X, Y, and Z axes. These matrices are applied in the following order: **Z $\rightarrow$ Y $\rightarrow$ X**. This order was chosen to reduce compound rotation effects and to preserve spatial consistency.

The rotation matrices used are:

Rotation around X-axis:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta_x & -\sin\theta_x \\ 0 & \sin\theta_x & \cos\theta_x \end{bmatrix}$$

Rotation around Y-axis:

$$\begin{bmatrix} \cos\theta_y & 0 & \sin\theta_y \\ 0 & 1 & 0 \\ -\sin\theta_y & 0 & \cos\theta_y \end{bmatrix}$$

Rotation around Z-axis:

$$\begin{bmatrix} \cos\theta_z & -\sin\theta_z & 0 \\ \sin\theta_z & \cos\theta_z & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Each vertex is transformed in sequence, resulting in a rotated mesh.

5.3 Centering Before Rotation

To ensure the object rotates around its geometric center (and not the origin), the object is first translated so that its center aligns with the origin. After rotation, it is translated back to its original center.

This prevents undesired offsets and orbital rotation effects. The process ensures that the object remains spatially stable and predictable.

5.4 Precision/Accuracy Issues

Rotating floating-point vertex coordinates can introduce minor precision errors. This tool minimizes these issues by:

- Applying only one transformation pass per object,
- Using `math.radians()` to ensure angle consistency,
- Preserving vertex order and precision when writing output.

These practices avoid drift and maintain compatibility with downstream 3D tools.

5.5 Performance Considerations

Rotation is applied only to lines beginning with `v` (vertex lines). The algorithm iterates linearly over all vertices, resulting in a time complexity of:

$$O(n) \quad \text{where } n = \text{number of vertices}$$

Since only three multiplications and additions are applied per vertex, the overhead is minimal. Even for large meshes, runtime remains negligible.

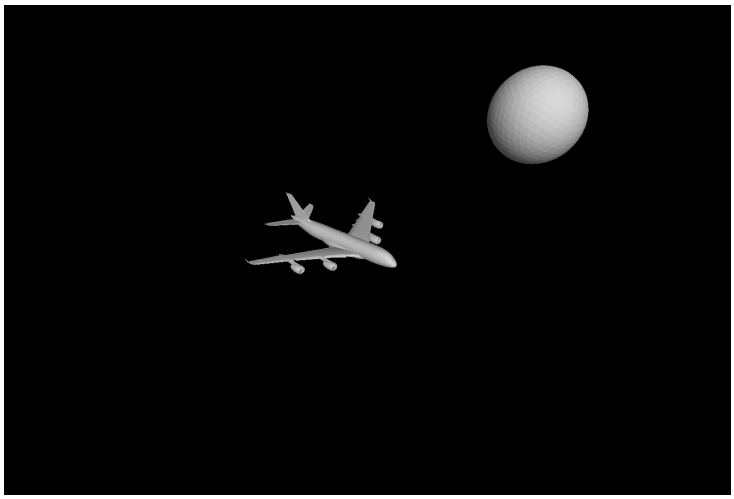

Figure 8: Original object before rotation

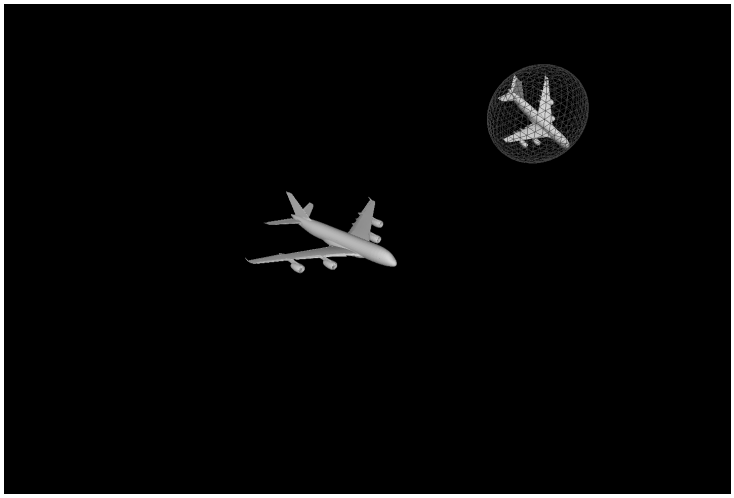

Figure 9: Object correctly rotated to match user's input

6 Conclusion

The objective of this thesis was to develop a lightweight and efficient transformation tool for 3D objects in the Wavefront OBJ format. The tool enables accurate translation, uniform scaling, and axis-based rotation of 3D models, allowing them to fit within and align to reference objects using a configuration-based approach.

A key contribution of this work is the ability to fully automate transformations using simple JSON input files. The system computes bounding boxes, applies matrix-based rotations, and aligns object centers with minimal user intervention. By preserving the OBJ structure, the tool remains compatible with standard graphics pipelines and 3D viewers like MeshLab.

The transformation results were verified through visual inspection using side-by-side comparisons of models before and after scaling, rotation, and movement. These examples demonstrated that the tool successfully fits models within destination volumes while maintaining geometry integrity and spatial alignment.

Although the system works reliably in controlled test cases, there are areas for improvement. Rotation angles currently rely on manual input and may benefit from a GUI or automatic alignment logic. Additionally, the tool could be extended to support non-uniform scaling, material handling, or other OBJ features. Future work may focus on expanding the transformation pipeline and integrating the tool into larger 3D processing systems or game engines.