Geometry to volume conversion tool

K. Jaklič[1]

[1]University of Ljubljana, Faculty of Computer and Information Science

Abstract

The goal of this seminar was to implement a tool for converting polygonal geometry to volumes and demonstrate its usability on a set of 3D models. The developed tool uses a process of ray casting to convert the selected region of a model to a volumetric representation. Tool is able to load OBJ files, output a raw volumetric data and a 3D visualization of it. Users can define a specific region of space for conversion through coordinate selection, adjust the resolution of the volume and specify material properties for conversion. We demonstrate the utility of our tool through a series of tests on different 3D models, showcasing the ease of integration into existing workflows and the enhancement of volumetric data manipulation.

CCS Concepts

•Computing methodologies $\rightarrow$ ***Volumetric models; Mesh geometry models;***

1. Introduction

In the field of computer graphics, the use of volumetric representations for three-dimensional objects is becoming increasingly popular. Volumes are increasingly present in video games, furthermore there are different operations that are much faster with voxel representations than with regular polygons. [obj] However, incorporating these models into standard workflows is often challenging due to the widespread reliance on traditional polygonal modeling techniques. This paper presents a tool aimed at addressing this issue by simplifying the process of converting geometric data into volumetric formats. The tool, developed in Python, specifically facilitates the conversion of OBJ files, including their material properties, into volumetric data. This approach helps integrate volumetric models more seamlessly into existing graphic practices.

2. Implementation

2.1. Tools

Conversion tool was developed using Python and it's libraries and packages. Python is an open-source, high-level programming language known for its simplicity and readability. It supports multiple programming paradigms, including procedural, object-oriented, and functional programming.

One of Python's key features is its extensive standard library and active community, which contribute a vast array of modules and packages. This makes Python exceptionally versatile, allowing it to be used in web development, data science, artificial intelligence, machine learning, and more. Its interpretative nature allows for rapid testing and iteration of code, which is beneficial during the development phase.

This program utilizes a selection of Python libraries and packages designed to handle complex data manipulations and visualizations:

- **NumPy** is a fundamental package for scientific computing which offers powerful N-dimensional array objects and tools for integrating C/C++ code into Python syntax. We mostly used it for numerical operations.
- **Trimesh** is an easy-to-use library for loading, processing, and visualizing triangular meshes. It provides functionalities to read and write various mesh formats, perform standard mesh operations, and conveniently handle mesh properties and geometry.
- **Matplotlib** is a plotting library that provides an object-oriented API for embedding plots into applications. It is particularly useful for creating static, interactive, and animated visualizations in Python. It also includes a toolkit for plotting 3D data, which enhances Matplotlib's capabilities by adding simple 3D plotting utilities, ideal for creating 3D scatter plots, surface plots, and wireframes.
- **Vedo** is a library designed for 3D visualization and analysis, especially of volumetric data, meshes, and point clouds. It is built on top of VTK and provides tools to streamline the visualization pipeline in scientific applications.
- **SciPy spatial's KDTree** as Part of the SciPy library is used for quickly finding nearest neighbors in multidimensional space. It is highly effective for spatial indexing and querying, which is crucial in many scientific and engineering applications.
- **H5py** lets you store huge amounts of numerical data, and easily

manipulate that data from NumPy. It is an interface to the HDF5 binary data format.

Each of these tools is integrated into the program to leverage their specific strengths, from data manipulation and storage to advanced plotting and visualization techniques.

2.2. Implementation algorithm

For our seminar project, we developed a Python script that begins by loading an OBJ file and reading the associated material file. This script efficiently stores material parameters and records the name of the texture file, which is later used to determine voxel colors. We utilized Trimesh for mesh loading.

User is asked to select a preferred resolution of the resulting volume, which can be 64x64x64, 128x128x128 or 256x256x256. With the loaded model, user inputs minimum and maximum coordinates to create a bounding box, on which a conversion to volume will be made. Depending on the coordinates and selected resolution, voxel dimensions are calculated for later processing.

Volume conversion takes the specified bounds and creates empty voxel and color grid, which are filled in the next few steps. The code iterates through each voxel's position using nested loops for each dimension (x, y, z) to first calculate each voxel's center. For each voxel center program checks if it is located inside or outside the mesh. This is done by using ray casting algorithm, which checks for intersections of ray or multiple rays with the mesh. The determination if voxel is inside or outside of the mesh is based on the odd/even rule. If there is an even number of intersections, the voxel is outside and if the number is odd, the voxel is inside the mesh and should be processed further. [GG17]

Raycasting is implemented by specifying basic parameters such as the origin and direction of the ray. Furthermore, our program uses pre-processing, which by using bounding boxes checks for potential candidates, where the intersections might occur. This pre-processing significantly speeds up the process of converting polygon to volume and is important for optimizing the ray-triangle intersection tests by quickly eliminating rays that don't intersect the mesh's bounding volume. It is important to check if ray is parallel to the x-axis, since the parallel ray either never intersect the bounding box in the direction or is always intersecting, depending on the x-coordinate relative to the bounds. Intersection points in pre-processing are calculated using intersection times and ray properties - origin and direction. We then filter the resulted intersections to only valid ones and the ones within the mesh, using barycentric coordinates for precise checking. We also ensure that the intersection occurs in the direction of the ray and not behind the ray origin. [MT05]

In the next step, precise points of intersections of rays and planes are calculated. Vector from ray origin to each plane origin is calculated. This vector is essential for determining how far a ray origin is from its corresponding plane along the direction of the plane's normal. We then identify the valid intersections again as non parallel and calculate coordinates on the plane, where ray hits the surface.

After the ray casting, voxel grid and output array are updated, and program continues to calculate the color of the processed voxel. If the voxel is inside the mesh, the function looks up the nearest vertex with the help of precomputed tree and fetches the texture coordinates from the mesh's UV map. It uses these coordinates to get a color from a texture image, assigning this color to the voxel. A user must then decide whether or not to use ambient and diffuse properties from material file and the color grid is updated accordingly.

The last step of the tool is saving a file. Our volume data is saved as raw data to RAW file format, which can be later viewed in different programs, for example ParaView or VPT.

2.3. Results

We tested the implemented tool by converting various OBJ files with associated materials and textures. In this chapter we present tests of an OBJ model Suzanne with different resolutions and subsections selected. As mentioned before, our tool lets user specify the preferred resolution of the resulting volume. With higher resolutions we get more detailed and smooth volumes, but their conversion takes a bit longer.

Visualizations and time measurements are shown in the figures bellow.

In figure 1 our model 'Suzanne' is shown as original OBJ file on the left, meanwhile the image on the right shows Suzanne after converting to a volume. The resolution was set to 64^3 and the whole model was selected for conversion. The voxel dimensions achieved through this conversion were 0.043 x 0.031 x 0.027. The computational time required to generate the volumetric representation was approximately 78.00939 seconds.

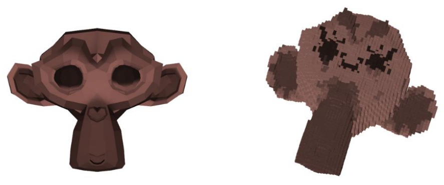

Figure 1: *Starting OBJ model and resulted volume with* 64^3 *resolution.*

In the subsequent analysis, presented in Figure 2, the model 'Suzanne' was converted to a higher resolution of 128^3. This adjustment resulted in a more refined volumetric representation, as it can be seen in the image on the right. The voxel dimensions for this resolution were reduced to 0.021 x 0.015 x 0.013. The conversion process required significantly more time, totaling approximately 551.17712 seconds. For comparative purposes, Figure 2 also includes a reference image depicting the previously discussed volume at a resolution of 64^3.

In an additional test, we evaluated our tool's capability to select and convert only a subsection of a model into a volume. Figure 3 illustrates this test; the left side of the image displays the base model of 'Suzanne' with the selected subsection clearly marked.

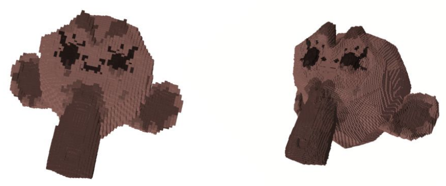

Figure 2: *Converted volumes from the model with resolutions* 64^3 *and* 128^3.

The right side of the image features the same marked subsection, now converted into a volume at a resolution of 64^3. The conversion process for this partial model required 91.88958 seconds, and the resulting voxel dimensions were 0.021 x 0.015 x 0.027.

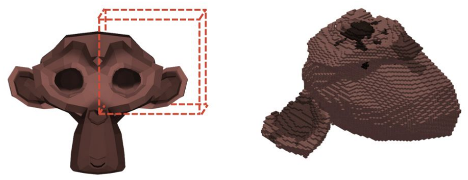

Figure 3: *Starting OBJ model with selected subsection and converted subsection volume with* 64^3 *resolution.*

Figure 4 extends the examination of the tool's capabilities by presenting the same subsection from Figure 3, now converted to a volumetric form at a higher resolution of 128^3. This increased resolution significantly enhanced the detail of the volume, resulting in voxel dimensions of 0.011 x 0.007 x 0.013. The conversion process for this enhanced resolution was considerably longer, requiring 740.612183 seconds to complete.

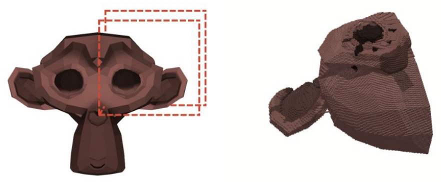

Figure 4: *Converted volume subsections from the model with resolutions* 64^3 *(on the left) and* 128^3 *(on the right).*

The volumetric output generated by the conversion can also be visualized using software capable of interpreting binary data, as our tool is designed to save the voxel grid data in RAW file format. Figure 5 demonstrates a straightforward visualization of this volume data, saved in a RAW file format, utilizing the Visualization Processing Tool (VPT). This illustration provides an effective display of the voxel-based structure, showcasing the practical applications of our conversion tool in rendering and analyzing 3D data.

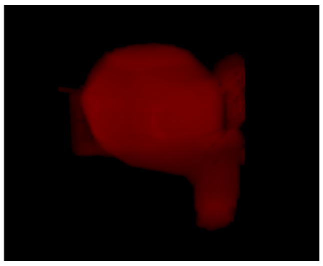

Figure 5: *Converted volume with resolutions* 64^3 *visualized in the VPT, pictured from the side.*

2.4. Conclusions

The tool developed serves as an efficient solution for converting OBJ files, along with their associated material and texture files, into volumetric data. Utilizing the ray casting algorithm, this tool achieves rapid conversion rates at lower resolutions, such as 64^3. However, when operating at higher resolutions, the conversion process extends in duration, indicating potential areas for optimization.

Upon completion of the conversion, the resultant volumes are stored in formats conducive to further analysis and manipulation. Due to the widespread compatibility of file formats such as RAW and HDF5, the saved volume data can be accessed using various software tools that support volume rendering or the visualization of binary data. This versatility underscores the utility of the implemented tool in facilitating diverse computational and visualization tasks.

The development of the tool on the macOS system with ARM architecture presented certain constraints, primarily due to compatibility issues with key libraries and software. Notably, libraries such as OpenVDB, which are ideally suited for intermediate conversions and for rendering or visualizing the final volume, were not fully compatible. Additionally, the choice to implement the tool in Python posed further limitations on its functionality. Python, while versatile, may not fully leverage the graphical, lighting, camera, and material capabilities that are important for advanced visualization tasks.

Considering these challenges, future enhancements could involve transitioning the tool to a more robust platform such as webGPU, which could improve performance and enable more sophisticated features in graphics, lighting, camera operations, and material handling, offering a more comprehensive solution for volumetric data manipulation and visualization.

References

[GG17] GALETZKA M., GLAUNER P.: A simple and correct even-odd algorithm for the point-in-polygon problem for complex polygons. *arXiv (Cornell University)* (Jan 2017). doi:https://doi.org/10.5220/0006040801750178. 2

[MT05] MÖLLER T., TRUMBORE B.: Fast, minimum storage ray/triangle intersection. 7–es. URL: https://doi.org/10.1145/1198555.1198746. 2

[obj] Voxelizing mesh data: Obj to voxels. URL: https://paulbourke.net/dataformats/obj2vox/. 1