

CONVERSION OF VOXEL MODELS INTO POLYGONAL MESHES WITH ENSURING STRUCTURAL INTEGRITY AND EDITABILITY

Semen Duvanov, Iryna Baranova

Sumy State University, Department of Information Technologies, Sumy, Ukraine

Abstract. Voxel representation is one of the natural ways of describing three-dimensional objects as a set of cubic elements with coordinates. It is widely used in medical visualization, geology, computer simulations, and games. At the same time, most modern 3D editors and engines operate with polygonal models, which creates the need for tools capable of converting voxel scenes into polygonal meshes. The aim of this work is to develop an efficient algorithmic approach for converting voxel-based 3D models into polygonal meshes, taking into account position, rotation, and scaling, while preserving model integrity, minimizing polygon count, and enabling export to the *.obj format. The proposed algorithm has been implemented in the software tool VoxToObjConverter, developed in C# using the geometry3Sharp library. The algorithm consists of several sequential stages. First, data is read from input *.vox files. Then, voxel structure optimization is performed by extracting only surface elements and grouping them using the Flood Fill algorithm, which helps to eliminate redundant elements and noise. In the next stage, a polygonal mesh is generated by converting voxels into triangular polygons using the Face Culling method, which ensures that faces are added only in the absence of neighbouring voxels and significantly reduces the polygon count. The initially generated triangular mesh can optionally be merged into a quadrilateral topology during export, providing improved editability and suitability for subdivision-based modelling workflows. Global transformations (translation, rotation, scaling) are then applied to the mesh, and the resulting model parts are merged into a unified structure. Finally, the model is exported into the *.obj format, ensuring compatibility with most modern 3D editors and engines. Testing on various voxel models confirmed the accuracy of conversion, geometric consistency, and efficiency of the converter. The developed algorithm and software provide a complete pipeline for voxel-to-mesh conversion and have practical significance for visualization, game development, and 3D modelling tasks, where fast and precise conversion of voxel data into render-ready polygonal models is required.

Keywords: voxel modelling, conversion algorithm, quadrilateral mesh generation, solid element, mesh optimization

KONWERSJA MODELI WOKSELOWYCH NA SIATKI POLIGONOWE Z ZACHOWANIEM INTEGRALNOŚCI STRUKTURALNEJ I MOŻLIWOŚCI EDYCJI

Streszczenie. Reprezentacja woksłowa jest jednym z naturalnych sposobów opisywania obiektów trójwymiarowych jako zbioru sześciennych elementów o określonych współrzędnych. Znajduje szerokie zastosowanie w wizualizacji medycznej, geologii, symulacjach komputerowych oraz grach. Jednocześnie większość nowoczesnych edytorów i silników 3D opiera się na modelach wielokątnych, co rodzi zapotrzebowanie na narzędzia umożliwiające konwersję scen woksłowych na siatki wielokątne. Celem niniejszej pracy jest opracowanie wydajnego algorytmicznego podejścia do konwersji modeli 3D opartych na woksłach na siatki wielokątne, z uwzględnieniem położenia, obrotu i skalowania, przy jednoczesnym zachowaniu integralności modelu, zminimalizowaniu liczby wielokątów oraz umożliwieniu eksportu do formatu *.obj. Proponowany algorytm został zaimplementowany w narzędziu programowym VoxToObjConverter, opracowanym w języku C# z wykorzystaniem biblioteki geometry3Sharp. Algorytm składa się z kilku kolejnych etapów. Najpierw odczytywane są dane z plików wejściowych *.vox. Następnie przeprowadzana jest optymalizacja struktury woksli poprzez wyodrębnienie wyłącznie elementów powierzchniowych i zgrupowanie ich za pomocą algorytmu Flood Fill, co pomaga wyeliminować zbędne elementy i szum. W kolejnym etapie generowana jest siatka wielokątna poprzez przekształcenie woksli w trójkątne wielokąty przy użyciu metody Face Culling, która gwarantuje, że powierzchnie są dodawane wyłącznie w przypadku braku sąsiednich woksli, co znacznie zmniejsza liczbę wielokątów. Wstępnie wygenerowana siatka trójkątna może opcjonalnie zostać przekształcona w topologię czworokątną podczas eksportu, co zapewnia lepszą edytowalność i przydatność w procesach modelowania opartych na podziale siatki. Następnie do siatki stosowane są transformacje globalne (przesunięcie, obrót, skalowanie), a powstałe części modelu są łączone w jednolitą strukturę. Na koniec model jest eksportowany do formatu *.obj, co zapewnia kompatybilność z większością nowoczesnych edytorów i silników 3D. Testy przeprowadzone na różnych modelach woksłowych potwierdziły dokładność konwersji, spójność geometryczną oraz wydajność konwertera. Opracowany algorytm i oprogramowanie zapewniają kompletny proces konwersji modeli woksłowych na siatki i mają praktyczne znaczenie w zakresie wizualizacji, tworzenia gier oraz modelowania 3D, gdzie wymagana jest szybka i precyzyjna konwersja danych woksłowych na gotowe do renderowania modele wielokątne.

Słowa kluczowe: modelowanie woksłowe, algorytm konwersji, siatka czworokątna, element bryłowy, optymalizacja siatki

Introduction

In modern computer graphics, various approaches are used to represent three-dimensional scenes, among which voxel-based representation occupies a special place. It provides a natural way to describe volumetric structures by discretizing space into cubic elements – voxels. Due to its intuitiveness and simplicity, voxels are widely applied in medical visualization, geology, scientific modelling of physical processes, as well as in computer games and procedural content generation systems [18].

Despite the advantages of voxel-based representation, most modern 3D editors (Blender, 3ds Max) and rendering engines (Unity, Unreal Engine) primarily operate with polygonal models. This creates compatibility issues when integrating voxel content into traditional graphics pipelines. The key challenge lies in developing efficient conversion methods that can produce editable, optimized polygonal meshes while maintaining the structural integrity of the original voxel models.

Existing conversion algorithms such as Marching Cubes [17], Dual Contouring [5], and Surface Nets [12] provide certain accuracy levels but have specific limitations. They often exhibit high computational complexity, produce only triangular meshes, and may generate artefacts or non-solid geometry. Additionally, standard export tools from voxel editors (e.g., MagicaVoxel

Export, VoxEdit Export) frequently produce low-quality models with disconnected polygons or excessive element counts, complicating subsequent editing workflows.

The research focuses on three-dimensional voxel models and voxel-based scenes as its primary object of investigation. The subject matter encompasses conversion algorithms that transform voxel structures into polygonal mesh representations while maintaining structural coherence and cross-platform compatibility. This work aims to create an optimized algorithmic solution and corresponding software tool that maintains model fidelity and geometric transformations, reduces polygonal complexity via Face Culling techniques, and enables seamless *.obj format export for integration with industry-standard platforms such as Blender, Unity, and 3ds Max.

1. Problem statement

The development of an efficient voxel-to-mesh converter must address the following technical requirements:

- Generation of optimized polygonal meshes with quadrilateral or triangular topology;
- Preservation of global transformation matrices (position, rotation, scale);
- Ensuring solid geometry suitable for professional 3D editing;

- Minimization of face count through efficient adjacency checks (Face Culling);
- Universal compatibility with the *.obj format for cross-platform integration.

Input data requirements:

- Three-dimensional scene in *.vox format;
- Cubic voxels with defined global transformations (position, rotation, scale matrices).

Output data specifications:

- Optimized polygonal mesh in *.obj format;
- Suitable for further editing in 3D editors;
- Compatible with graphics engine integration.

Quality evaluation criteria:

- Geometric integrity. Models must form solid, connected structures without holes or disconnected elements;
- Polygon count optimization. Minimize face count without loss of detail through Face Culling;
- Transformation accuracy. Correctly preserve global transformations (position, rotation, scale) from source scene.

Constraints and assumptions:

- Source models consist exclusively of cubic voxels;
- All transformations defined by position-rotation-scale matrices;
- Resulting mesh composed of quadrilateral polygons (preferred) or triangular polygons (alternative).

2. State of the art

Voxel modelling has become popular in computer graphics since it allows an efficient representation of the occupancy of three-dimensional objects. This approach is widely used in scientific computing [1], computer games [11], and medical data visualization [3].

In a previous study [7], the main algorithms for converting voxel 3D models into polygonal ones, such as Marching Cubes, Dual Contouring, Surface Nets, Voxel Carving, Sparse Voxel Octree, and others, were described and compared, with a focus on their features, advantages, and drawbacks.

In this section, we will focus on the technologies and methods that were directly used or adapted in the practical implementation of the VoxToObjConverter, particularly the geometry3Sharp library, the Face Culling and Flood Fill algorithms, as well as handling the *.vox file structure to preserve transformations.

Geometry3Sharp. The geometry3Sharp library [15] is a modern tool for working with three-dimensional geometry in the .NET environment. It provides data types for representing meshes (DMesh3), vectors (Vector3d), quaternions (Quaternion), and matrices, as well as tools for performing transformations, mesh merging, topology checks, and export. Compared to other .NET libraries, geometry3Sharp ensures high performance in generating and processing polygonal models, which justifies its use in the project as the core for constructing, transforming, and exporting polygonal models.

Face culling. One of the key steps in polygonal mesh optimization is the Face Culling method, which performs discarding of invisible faces between adjacent voxels. When two neighbouring voxels are located side by side, the face between them is not added to the final mesh. Unlike basic approaches that generate polygons for all faces, Face Culling reduces the overall polygon count, optimizing the mesh structure and improving processing and rendering performance. In the implemented VoxToObjConverter, this approach is applied in the CreateTrianglesMesh method, where each voxel is checked against its neighbours in six orthogonal directions.

Greedy meshing. The Greedy Meshing algorithm is widely considered the "gold standard" for voxel mesh optimization. This technique iteratively merges adjacent coplanar voxel faces into larger rectangular polygons, potentially reducing face count by orders of magnitude. The algorithm operates by scanning the voxel grid in multiple passes, greedily expanding polygons as far as possible before committing them to the final mesh.

However, Greedy Meshing introduces computational complexity: it requires additional data structures to track merged regions and exhibits $O(n^2)$ worst-case complexity for n voxels, compared to $O(n)$ for basic Face Culling. For large scenes or real-time conversion scenarios, this overhead becomes significant. The current implementation employs Face Culling as the primary optimization method due to its favourable performance-quality trade-off: it provides substantial polygon reduction (4-6x decrease) with linear complexity. Future versions may integrate Greedy Meshing as an optional post-processing step for scenarios where conversion time is less critical than achieving maximum polygon reduction.

Flood fill. The Flood Fill algorithm is used to traverse connected regions in a three-dimensional voxel array. Recent optimizations for connected component labelling in 3D, such as GPU-based approaches [2], significantly improve processing speed for large voxel datasets. In this project, it is used to identify separate components and mark processed voxels, which prevents redundant processing and enables efficient formation of distinct polygonal meshes. Flood Fill plays a key role in converting voxels into polygons, ensuring the detection of connected regions. This is especially important when a clean, optimized polygonal mesh must be obtained from a voxel representation.

Vox file structure. The *.vox format, developed in MagicaVoxel, has a hierarchical structure that includes a colour palette, voxel models, and a scene with transformations and instances. For proper reading, the VoxReader parser is used, which deserializes the information considering the chunk file structure (MAIN, SIZE, XYZI, nTRN, nGRP, nSHP) [8]. Understanding this structure is critical for correctly reflecting transformations in the polygonal model.

Recent advances in voxel-to-mesh conversion. Recent developments in voxel-to-mesh conversion have focused on deep learning approaches and GPU acceleration. Wickramasinghe et al. [19] introduced Voxel2Mesh, an end-to-end trainable architecture that converts 3D volumetric images directly to surface meshes without post-processing, demonstrating improved accuracy in bio-medical segmentation tasks. For applications in medical imaging and radiation transport, Brown et al. [4] developed a robust voxel-to-polygon conversion algorithm that efficiently handles complex anatomical structures while maintaining high geometric accuracy. Luo et al. [14] recently proposed a differentiable voxelization algorithm enabling GPU-accelerated mesh morphing and optimization tasks, opening new possibilities for real-time applications.

The listed algorithms, technologies, and tools form the basis of the core functionality of the implemented VoxToObjConverter, designed to convert voxel 3D models into polygonal ones.

3. Implementation of the voxel-to-polygon conversion algorithm

3.1. Input data format and structure

The *.vox format, developed for MagicaVoxel, employs a hierarchical structure consisting of the following components: SIZE chunks defining grid dimensions, XYZI chunks containing voxel coordinates, RGBA chunks storing the colour palette, and transformation nodes (nTRN, nGRP, nSHP) that define object hierarchies, positions, rotations, and scales. Understanding this structure is critical for correctly extracting and applying transformations in the polygonal model [8].

3.2. Implementation platform and libraries

For the practical realization of the proposed approach, the C# programming language was chosen because of its strong support for object-oriented programming, convenient memory management, and wide ecosystem of libraries suited for algorithmic and graphical tasks. In particular, the geometry3Sharp library was utilized, as it provides a rich set of tools for working

with three-dimensional meshes, including methods for generating and editing polygonal structures, applying geometric transformations such as translation, rotation, and scaling, as well as ensuring export in formats compatible with common 3D editors.

The developed algorithm was implemented as part of a dedicated software tool named VoxToObjConverter, which was fully written in C# and integrated with geometry3Sharp for handling the core geometry processing stages. Within this program, geometry3Sharp plays a central role: it manages mesh construction from voxel data, supports the removal of redundant faces through Face Culling, applies global transformations to the resulting structures, and prepares the final optimized model for export into the *.obj format.

Thus, the combination of C# and geometry3Sharp enabled not only the reliable implementation of the conversion pipeline but also the creation of a flexible and extensible framework for further improvements of the converter.

3.3. Algorithm stages

The main logic consists of the following stages: reading the voxel scene, voxel optimization, polygonal mesh construction, applying transformations, merging meshes, and exporting to the *.obj format.

The program is composed of the following core components:

- VoxModelParser – reads the *.vox file and deserializes the scene;
- VoxModelOptimizer – optimizes the voxel model by filtering out only surface voxels;
- MeshBuilder – constructs a polygonal mesh based on the outer voxel faces (faces adjacent to empty space);
- MeshWelder – optimizes the mesh by removing duplicates, merges all generated polygons into a solid model, and recalculates polygon normals;
- TransformMeshUtility – applies global transformations to the polygonal model so that the model's position and rotation after import match the voxel scene;
- QuadMeshToObjExporter – exports the final model in quadrilateral polygon format to an *.obj file;
- TriangleMeshToObjExporter – exports the final model in triangular polygon format to an *.obj file;
- VoxModelConverter – performs the entire export cycle using the above-mentioned components in the appropriate order (top to bottom).

The user can choose between two export options: a model with quadrilateral polygons or a model with triangular polygons.

Figure 1 shows the program interface. The program can export several selected voxel models sequentially.

Next, we will examine the stages of the algorithm in more detail.

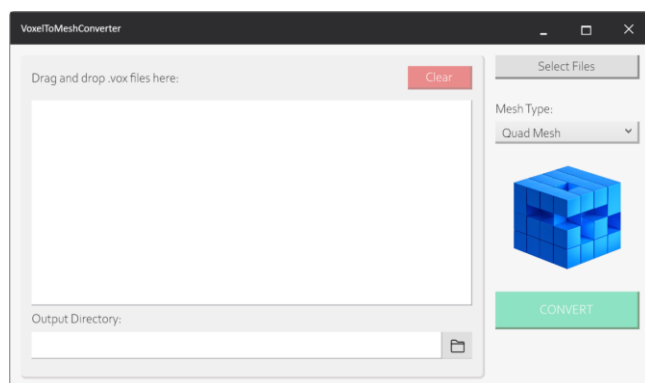


Fig. 1. Converter program interface

Reading the *.vox file. The *.vox format has a hierarchical structure and consists of the following blocks: SIZE (grid size), XYZI (voxels), RGBA (colors). The VoxModelParser parser uses the VoxReader library, which applies a step-by-step traversal of chunks. For each model, an array of voxels and transformation matrices (position, rotation) is generated and stored inside the IModel object.

```
public IModel[] ReadModel(string voxFilePath)
{
    var voxFile =
    VoxReader.VoxReader.Read(voxFilePath);
    return voxFile.Models;
}
```

Voxel optimization before geometry generation is performed in two steps: detection of boundary voxels (those that have at least one open face, implemented in the VoxModelOptimizer component) and application of Flood Fill for voxel grouping (implemented in the MeshBuilder component).

The Flood Fill implementation employs 6-connectivity (considering only face-adjacent neighbours) rather than 26-connectivity (which would include edge and corner neighbours). This design choice prevents corner-touching objects from being incorrectly merged into a single component. For example, two voxel cubes sharing only a corner vertex remain as separate connected components, ensuring proper object separation in the final mesh.

The Flood Fill algorithm helps during conversion by:

- Identifying connected components: Flood Fill determines which voxels belong to the same object, helping avoid unnecessary polygons and improving mesh accuracy;
- Noise removal: isolated voxels not connected to the main structure can be excluded;
- Preparation for surface reconstruction: after Flood Fill, polygonal mesh generation algorithms can be applied.

The goal of the optimization stage is to reduce the number of computations and avoid creating unnecessary polygons.

Polygonal mesh generation initially creates two triangular polygons for each voxel face (arranged along the face diagonal). During export, the user can choose between two output modes: (1) Quadrilateral Mesh, where adjacent triangle pairs are merged back into quadrilaterals, providing topology preferred for manual editing and subdivision surface modelling; or (2) Triangle Mesh, where triangles are exported directly, optimized for game engine import. The mesh generation process uses the Face Culling method to ensure faces are added only where neighbouring voxels are absent.

The first step is performed as follows: for each active voxel, six directions are checked (X+, X-, Y+, Y-, Z+, Z-). If there is no filled neighbouring voxel, the corresponding face is added. Each face consists of two triangles (along the diagonal).

```
if (!voxelMap.ContainsKey(pos + Vector3Int.Right))
{
    AddQuad(...); // Adding a face to the right
}
```

The next step in implementing Face Culling involves removing the internal faces of the polygonal mesh. To achieve this, all six neighbouring positions of each voxel are checked, and if another voxel is present in the neighbouring position, the face between them is not added to the final mesh. As a result, only the outer surfaces that enclose the empty space remain.

This way, the internal faces between adjacent voxels are removed, significantly reducing the total number of polygons, while the mesh interior remains hollow, and the model's external appearance is perceived as a solid object. In addition, the problem of double walls is eliminated, and the model is formed exclusively from surface polygons, which optimizes construction and rendering performance.

Such an approach ensures that the model will consist only of surface polygons.

Applying transformations. Each model in the scene has its own transformation, represented as a 3×3 matrix (rotation) and a position vector. These data are converted into a quaternion and applied to each vertex of the mesh:

```
public void ApplyTransformToMesh(IModel voxModel,
DMesh3 voxMesh)
{
    // Step 1: Calculate the final rotation by combining
    // model and fix rotations.
    Quaterniond rotation = GetFinalRotation(voxModel);
    // Step 2: Convert model position into g3's
    // coordinate space.
    Vector3d translation =
    GetTransformedPosition(voxModel);
    // Step 3: Apply rotation and translation to every
    // mesh vertex.
    TransformMeshVertices(voxMesh, rotation,
    translation);
}
```

Transformations allow all scene objects to be correctly placed in the global coordinate system.

Merging model polygons. The MeshWelder class performs automated clean-up and optimization of a 3D mesh, which includes removing duplicate triangles, merging geometrically coincident polygon edges, compacting the internal mesh structure, recalculating normals for correct lighting, and checking geometry integrity. This ensures high-quality mesh preparation for further use in rendering, physical simulation, or editing, improving processing performance and stability.

In addition, since a *.vox file may contain several models, each voxel model is first converted into a separate polygonal one, after which all resulting meshes are merged into a single unified mesh.

Instance handling. The *.vox format supports instancing through nSHP (shape) and nGRP (group) transformation nodes, allowing a single voxel model to appear multiple times with different transformations. The current implementation creates independent geometry for each instance by applying the appropriate transformation matrix to each copy. For a scene containing 100 instances of the same voxel model, this results in 100 transformed mesh copies in the final .obj file. While this approach increases file size, it ensures maximum compatibility since the .obj format does not natively support geometry instancing. Future versions could implement a mesh library system with external references to reduce redundancy, though this would require export to formats with instancing support such as .fbx or .gltf.

The GetCombinedSolidTriangleMesh method of the main VoxModelConverter class performs the merging of multiple voxel models into one unified triangular mesh.

```
private DMesh3 GetCombinedSolidTriangleMesh(IModel[]
voxelModels)
{
    DMesh3 combinedMesh = new DMesh3();
    var transformMeshUtility = new
    TransformMeshUtility();

    foreach (IModel voxelModel in voxelModels)
    {
        IEnumerable<Voxel> optimizedVoxels =
        OptimizeVoxelModel(voxelModel);
        DMesh3 originalMesh =
        CreateTrianglesMesh(optimizedVoxels);
        WeldTrianglesIntoSolidMesh(originalMesh);
        DMesh3 transformedMesh = new
        DMesh3(originalMesh);

        transformMeshUtility.ApplyTransformToMesh(voxelModel,
        transformedMesh);
        AddMeshToParent(combinedMesh, transformedMesh);
    }

    return combinedMesh;

    static void AddMeshToParent(DMesh3 parentMesh,
    DMesh3 meshToAdd)
```

```
{
    MeshEditor editor = new MeshEditor(parentMesh);
    editor.AppendMesh(meshToAdd);
}
```

The result is a single unified DMesh3 mesh that combines all input voxel models into one coherent 3D geometry, fully prepared for further export.

Export to obj. The exporter provides two options for exporting the resulting polygonal mesh: Quad Mesh (consisting of quadrilateral polygons) and Triangle Mesh (consisting of triangular polygons). The preferred option is the former, as it produces a model that is most convenient for further editing.

```
private void ExportMeshToObjFile(string filePath,
DMesh3 mesh)
{
    IMeshToObjExporter exporter =
    _convertingOptions.MeshType switch
    {
        MeshType.Quads => new QuadMeshToObjExporter(),
        => new TriangleMeshToObjExporter()
    };
    string outputPath = GetOutputPath(filePath);
    exporter.ExportToFile(mesh, outputPath);
}
```

The final object is exported in *.obj format, which stores vertex coordinates and a set of faces.

Thus, the implemented algorithm provides a complete cycle for converting a voxel scene into a polygonal mesh suitable for further import into most popular 3D editors and game engines.

Among other tools used, it is worth mentioning WPF (Windows Presentation Foundation), which enables the creation of the converter's graphical user interface.

4. Testing

To evaluate the correctness and efficiency of the algorithm, scenes created in MagicaVoxel were used, including both simple cubic objects and complex multicomponent compositions with repeated use of the same model in different positions, possibly with scaling or rotation. This approach allows for effective testing of the algorithm on different types of objects and scene complexities. The conversion results are presented in Table 1.

Table 1. Results of exporting voxel models to polygonal meshes

Scene name	Dimensions (voxels)	Number of polygons	Conversion time
monu1.vox	126 × 126 × 118	65 900	2.4 s
monu2.vox	72 × 72 × 126	100 528	1.8 s
road with street light.vox	40 × 40 × 40	38 528	0.5 s
room.vox	118 × 121 × 60	177 068	2.8 s
Table Lamp.vox	256 × 256 × 256	24 744	0.7 s
teapot.vox	126 × 80 × 61	109 128	1.9 s
cars.vox	34 × 32 × 14	7 812	0.1 s
castle.vox	21 × 21 × 21	7 392	0.1 s
model.vox	40 × 40 × 40	21 252	0.3 s

The build time is linearly dependent on the number of active voxels. Thanks to Face Culling, the total number of polygons was reduced to approximately 15–25% of the naive approach. Even for a scene with over 100,000 polygons, the total export time did not exceed 4 seconds (on a system with 11th Gen Intel(R) Core(TM) i5-11400H @ 2.70GHz (2.69 GHz), 16GB RAM).

The converter code was posted on Github [6].

5. Results

Figure 2 shows examples of conversion results. The models were imported into 3ds Max for geometry verification and positioning: the left-hand image shows the scene in MagicaVoxel, while the right-hand image shows the same scene after conversion and import.

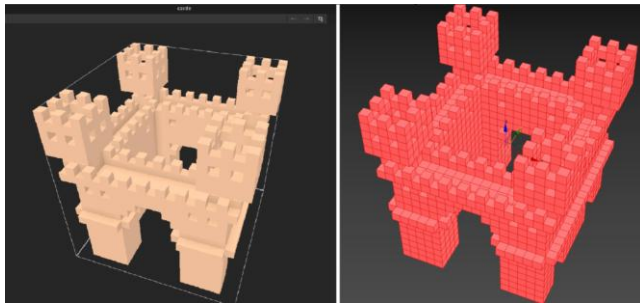


Fig. 2. Original voxel model (left), and converted model in 3Ds Max (right)

As the figure demonstrates, after import, the model is holistic and consists of quadrilateral polygons, which is a much more convenient option for further editing compared to triangular meshes.

The geometric structure of the models after transformation precisely corresponds to the original arrangement in the *.vox scene, including rotation and displacement.

Analysis of the polygonal mesh topology confirmed the absence of holes or incorrect normals, which validates the correctness of the algorithm's operation.

To evaluate the obtained results, it is appropriate to compare the developed approach with existing alternative solutions. Among them, notable examples include the web applications ImageToStl [13] and FabConvert [10], as well as voxel model editors MagicaVoxel [9] and VoxEdit [16]. Table 2 provides a comparative analysis of these open-source tools, and Figure 3 illustrates the results of converting a voxel model into a polygonal mesh using different converters. As shown in Figure 3, alternative solutions often produce meshes consisting of disconnected polygon elements. Such structural deficiencies complicate further editing, subdivision, and topology refinement in professional 3D modelling environments.

Table 2. Comparison of alternative converters

Tool	Advantages	Disadvantages
ImageToStl	-	– the model is difficult to edit, since it consists of triangles; – contains artefacts in the form of separate polygon elements not connected to other polygons in the mesh.
FabConvert	-	identical to the result of ImageToStl
MagicaVoxel Export	the mesh is optimized for the number of polygons.	– the model is difficult to edit, since it consists of triangles; – each face of the model is a separate element (the mesh is not solid).
VoxEdit Export	– the mesh is optimized for the number of polygons; – the model consists entirely of rectangular polygons.	the model is not solid, since each polygon is a separate element, which complicates further editing and modelling.

The developed converter is free from the defects typically found in similar tools and ensures the creation of a model that is easy to edit thanks to the use of quadrilateral polygons. The mesh contains no artefacts such as isolated elements, and all polygons are integrated into a single cohesive structure. This significantly simplifies subsequent editing and modification of the model.

Table 3 shows the comparison of the counts of the elements inside polygon models, obtained by different converters. Our developed converter shows the best result – 1 element per model, which means that the model is fully solid.

The obtained results demonstrate that the proposed approach provides accurate and efficient conversion of voxel scenes into polygonal meshes while preserving the main geometric and semantic characteristics of the original.

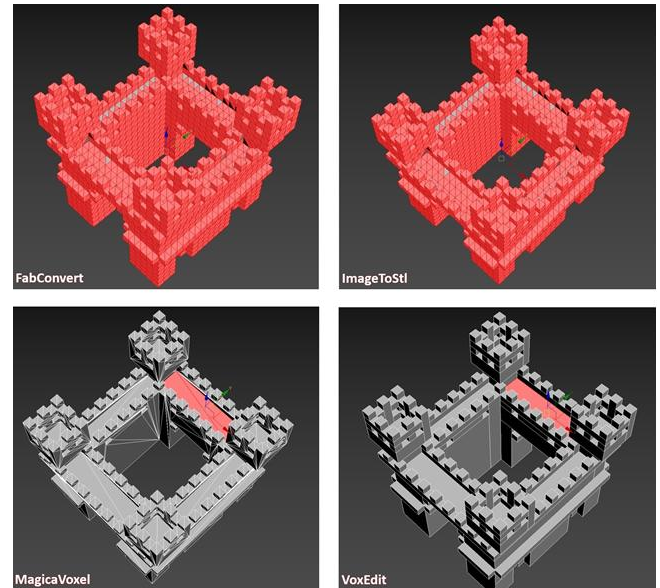


Fig. 3. Results of converting a voxel model into a polygonal model using alternative tools

Table 3. Elements count in polygonal model

Tool	Model			
	castle	model	monu1	monu2
ImageToStl	145	1	3	538
FabConvert	145	1	3	538
MagicaVoxel Export	893	23	535	1976
VoxEdit Export	1310	44	811	3128
VoxToObjConverter	1	1	1	1

The system successfully converts *.vox files into fully functional *.obj models that can be imported into Blender, Unity, 3ds Max, and other editors. Additionally, the program supports working with multi-component scenes, accepting multiple objects as input and producing a single combined *.obj file with correctly positioned and oriented polygonal meshes.

6. Discussion

The proposed algorithmic approach proved to be effective for the rapid transformation of voxel scenes into polygonal meshes. An important advantage is the use of quadrilateral polygons, which makes the model more convenient for editing and further processing.

The mesh is solid, as all polygons are integrated into a single structure, and the global scene parameters – position, rotation, and scale – are correctly preserved after conversion. This ensures the possibility of directly importing scenes from MagicaVoxel into common 3D editors and rendering engines without losing structural consistency.

The use of the geometry3Sharp library greatly simplified working with geometry, transformations, and exporting to the *.obj format. The approach provides accurate reproduction of global transformations: rotation, scaling, and positioning of models from the *.vox scene are precisely transferred to the final polygonal model, allowing full scenes to be imported without structural loss.

The implementation of the algorithm demonstrated that sequential processing of voxel faces with Face Culling allows efficient generation of polygonal meshes with an adequate level of detail. Mesh optimization using Face Culling significantly reduces the number of polygons.

Despite the identified advantages, the developed converter has certain limitations. First and foremost, it does not support the Level of Detail (LOD) mechanism, which limits its use in large scenes where dynamic geometry simplification is important.

The program employs a basic polygonization approach that does not consider more advanced methods such as Dual Contouring or Surface Nets. This affects the quality of smooth surfaces and complex geometric shapes.

A particular drawback is the occurrence of non-manifold edges when four polygons intersect along a single edge, which can create difficulties during further processing.

In addition, the resulting polygonal mesh requires further optimization, as the number of polygons is relatively high and not always justified, which reduces performance in complex scenes.

Future development of the converter should focus on integrating mesh simplification algorithms, in particular full Greedy Meshing, which allows merging adjacent planes of the same colour into larger polygons, significantly reducing the number of faces without loss of quality.

The current version of the VoxToObjConverter focuses primarily on the extraction and optimization of geometric structures and topological integrity. Colour data contained in the RGBA chunks of the *.vox file is currently bypassed during the generation of the *.obj mesh to maintain maximum performance and focus on solid geometry construction. Future iterations are planned to implement colour transfer via vertex colours (XYZRGB extension), which would allow for colour preservation without the overhead of creating complex texture atlases or multiple material files.

Moreover, more advanced smoothing algorithms (e.g., Laplacian smoothing) could be introduced, making the mesh more suitable for rendering and animation.

Thus, the developed converter provides a balance between implementation simplicity, geometric accuracy, and performance. Its structure allows easy expansion of functionality according to user needs.

7. Conclusions

In this work, an efficient algorithmic approach for converting voxel scenes into optimized polygonal meshes was developed and implemented. The research led to the following conclusions:

Structural Integrity. The use of a 6-connectivity Flood Fill algorithm and automated mesh welding ensures the generation of solid, cohesive models (1 element per model), eliminating the issue of disconnected polygons found in existing alternatives.

Optimization Efficiency. The implementation of the Face Culling method allows for a significant reduction in polygon count (to 15–25% of the naive approach) while maintaining linear computational complexity, making the algorithm suitable for complex voxel scenes.

Editability. The generation of quadrilateral-based topology provides a significant advantage for further professional editing, subdivision, and integration into industry-standard 3D editors such as Blender and 3ds Max.

Transformation Accuracy. The developed pipeline ensures precise preservation of global scene parameters (position, rotation, and scale), allowing for seamless data exchange between voxel editors and rendering engines.

Practical Value. The VoxToObjConverter tool provides a complete, high-performance solution for voxel-to-mesh conversion, balancing geometric accuracy with cross-platform compatibility via the universal .obj format.

Future work will focus on implementing Greedy Meshing for further polygon reduction and integrating vertex colour (XYZRGB) support to preserve original voxel palette data.

Acknowledgements

The work was carried out within the framework of the R&D project "Models, Methods, and Information Technologies for Digital Content Generation and Decision Support in Complex Information Systems" (No. 0125U003432), conducted at the Department of Information Technologies, Sumy State University.

References

- [1] Aleksandrov, M., Zlatanova, S., & Heslop, D. J. (2021). Voxelisation Algorithms and Data Structures: A Review. *Sensors*, 21(24), 8241. <https://doi.org/10.3390/s21248241>
- [2] Allegretti, S., Bolelli, F., & Grana, C. (2020). Optimized Block-Based Algorithms to Label Connected Components on GPUs. *IEEE Transactions on Parallel and Distributed Systems*, 31(2), 423–438. <https://doi.org/10.1109/TPDS.2019.2934683>
- [3] Angermann, C., Bereiter-Payr, J., Stock, K., Degenhart, G., & Haltmeier, M. (2024). Three-Dimensional Bone-Image Synthesis with Generative Adversarial Networks. *Journal of Imaging*, 10(12), 318. <https://doi.org/10.3390/jimaging10120318>
- [4] Brown, J. L., Furuta, T., & Bolch, W. E. (2019). A Robust Algorithm for Voxel-to-Polygon Mesh Phantom Conversion. In S. Makarov, M. Horner, & G. Noetscher (Eds), *Brain and Human Body Modeling* (pp. 317–327). Springer International Publishing. https://doi.org/10.1007/978-3-030-21293-3_17
- [5] Chen, Z., Tagliasacchi, A., Funkhouser, T., & Zhang, H. (2022). Neural dual contouring. *ACM Transactions on Graphics*, 41(4), 1–13. <https://doi.org/10.1145/3528223.3530108>
- [6] Duvanov, S. (2025). *VoxToObjConverter*. GitHub. <https://github.com/SamDuvanov/VoxToObjConverter>
- [7] Duvanov, S. S., & Baranova, I. V. (2023). Algorithms for Converting Voxel 3d Models into Polygonal Ones. *Computer Science and Applied Mathematics*, (2), 14–21. <https://doi.org/10.26661/2786-6254-2023-2-02>
- [8] Ephtracy. (2022). *MagicaVoxel file format .vox extension*. GitHub. <https://github.com/ephtracy/voxel-model/blob/master/MagicaVoxel-file-format-vox-extension.txt>
- [9] Ephtracy. (2025). *MagicaVoxel—Free voxel editor*. <https://ephtracy.github.io/>
- [10] FabConvert. (2025). *FabConvert—Online 3D File Conversion Tools*. <https://fabconvert.com/>
- [11] Fang, Y., Wang, Q., & Wang, W. (2025). Aokana: A GPU-Driven Voxel Rendering Framework for Open World Games. *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, 8(1), 1–17. <https://doi.org/10.1145/3728299>
- [12] Gibson, S. F. F. (1998). Constrained elastic surface nets: Generating smooth surfaces from binary segmented data. In W. M. Wells, A. Colchester, & S. Delp (Eds), *Medical Image Computing and Computer-Assisted Intervention—MICCAI'98* (Vol. 1496, pp. 888–898). Springer Berlin Heidelberg. <https://doi.org/10.1007/BFb0056277>
- [13] ImageToStl. (2025). *ImageToStl—Convert Images into 3D Models*. <https://imagetostl.com/>
- [14] Luo, Y., Wang, Y., Xiang, Z., Xiu, Y., Yang, G., & Yap, C. (2024). *Differentiable Voxelization and Mesh Morphing* (Version 2). arXiv. <https://doi.org/10.48550/ARXIV.2407.11272>
- [15] Schmidt, R. (2020). *Geometry3Sharp*. GitHub. <https://github.com/gradientspace/geometry3Sharp>
- [16] The Sandbox. (2025). *VoxEdit—3D Voxel Modeling and NFT Creation Tool*. <https://www.sandbox.game/en/create/vox-edit/>
- [17] Wang, X., Gao, S., Wang, M., & Duan, Z. (2021). *A Marching Cube Algorithm Based on Edge Growth* (Version 3). arXiv. <https://doi.org/10.48550/ARXIV.2101.00631>
- [18] Wang, Z. (2024). *3D Representation Methods: A Survey* (Version 1). arXiv. <https://doi.org/10.48550/ARXIV.2410.06475>
- [19] Wickramasinghe, U., Remelli, E., Knott, G., & Fua, P. (2020). Voxel2Mesh: 3D Mesh Model Generation from Volumetric Data. In A. L. Martel, P. Abolmaesumi, D. Stoyanov, D. Mateus, M. A. Zuluaga, S. K. Zhou, D. Racocanu, & L. Joskowicz (Eds), *Medical Image Computing and Computer Assisted Intervention – MICCAI 2020* (Vol. 12264, pp. 299–308). Springer International Publishing. https://doi.org/10.1007/978-3-030-59719-1_30

M.Sc. Semen Duvanov

e-mail: samduvanov@gmail.com

Master's degree in Computer Science from Sumy State University. Currently pursuing Ph.D. in Computer Science at Sumy State University, specializing in computer 3D graphics research.

<https://orcid.org/0000-0001-7766-6837>

Ph.D. Iryna Baranova

e-mail: i.baranova@cs.sumdu.edu.ua

Associate professor in Department Information Technology, Sumy State University, Ukraine. Author and co-author of more than 30 scientific papers. Research interests: application of information technologies for the development of systems of automated design, transportation and distribution of thermal energy, e-learning systems, 3D modelling, visualization and animation.

<https://orcid.org/0000-0002-3767-8099>

