Mesh-to-Line: A Vectorization Tool For Converting 3D Meshes to Line Graphics

Spencer Lin
University of Southern California
Los Angeles, USA
linspenc@usc.edu

David Faizi
University of Southern California
Los Angeles, USA
dfaizi@usc.edu

Chunyao Jiang
University of Southern California
Los Angeles, USA
chunyaoj@usc.edu

Derek Santolo
University of Southern California
Los Angeles, USA
santolo@usc.edu

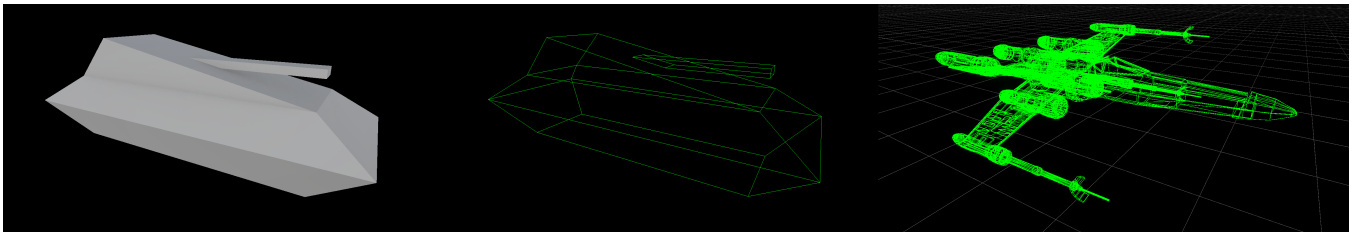

Figure 1: Simple mesh object (left), converted to Vector Line representation using Mesh-To-Line tool (center), stress-test of Mesh-To-Line on a complex mesh (right)

ABSTRACT

This paper introduces Mesh-to-Line, a vectorization tool designed to convert 3D meshes into vector line representations. The tool integrates several algorithms for Silhouette Edge Detection, Crease Edge Detection, and Bézier curve interpolation to create smooth and visually accurate representations. This paper analyzes several algorithmic approaches, challenges with noisy or complex meshes, and potential optimizations for performance and scalability. Downstream applications include recreating retro aesthetics, generating cartoon-like visuals, and supporting vector-style art in real-time rendering.

ACM Reference Format:
Spencer Lin, David Faizi, Chunyao Jiang, and Derek Santolo. 2024. Mesh-to-Line: A Vectorization Tool For Converting 3D Meshes to Line Graphics. In *Proceedings of Extended Abstracts of the CSCI 580 2024 Class (CSCI 580 '24).* ACM, New York, NY, USA, 5 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

In recent years, there has a been a keen interest in game development and research to recreate retro aesthetics. Projects such as B99, Ark-Ade, Holoball, and Immersive Archive [8] are among many which attempt to recreate a vector-inspired visual style. Despite this interest, there remains a research gap in modern authoring tools for creating vector line graphics in a practical manner. Though it is technically possible to manually draw vector lines in a modern game engine, it is often an unscalable and time-intensive process. To address this gap, we created Mesh-to-Line, a generalized method for vectorizing meshes which integrates into modern 3D mesh-based pipelines. It empowers creatives and developers to automatically generate vector line graphics from meshes and without the need for manually specified vertices with the added benefit of being able to recreate other line-based art styles such as cel-shading.

CSCI 580 '24, Decemeber 03–05, 2024, Los Angeles, CA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/XXXXXXX.XXXXXXX

2 RELATED WORKS

2.1 Intuitive Approaches

Beginning with an intuitive approach, one might believe a thinly dimensioned mesh to be an appropriate approximation of a vector line. However, the pitfall of this approach is that meshes will appear thicker/thinner across varying depths whereas vector lines will maintain a uniform dimension. This is illustrated in Figure 9 where meshes closer to the camera appear thicker such as those in the center. Furthermore, it is still very time-intensive to have to manually 3D model objects by what is effectively inserting one rectangular prism for each edge of a mesh.

Moving towards working with actual vector graphics, Schmidt Workshops describes a brute-force method of converting meshes into vector lines by simply iterating through all the vertices and drawing lines between them [9]. However, this approach is far from complete, often resulting in hidden lines being rendered or important lines missing altogether. This makes sense as this approach

blindly renders edges without taking into account the relationship between each vertex.

2.2 Algorithmic Approaches

Several algorithms exist to selectively render important edges in a mesh. Hertzmann [7] describes the two most important edge types for polygonal meshes: silhouette edges and crease edges. Silhouette edges are defined as edges which connect back-facing polygons to front-facing polygons whereas crease edges are any discontinuities in an otherwise smooth surface. Finally, we represent curves in meshes through the use of bézier curves. We dive into the specifics of our implementation of these algorithms in our Methodologies section.

Silhouette Edge Detection (SED). Buchannan and Sousa's so-called "Edge Buffer" algorithm [2] iterates over polygons in a mesh to check for silhouette edges. We chose to base our SED implementation on this algorithm for its efficient design suitable for realtime processing by utilizing fast XOR operations. It analyzes all edges that comprise the outermost planes of a starting mesh. If an edge connects a front-facing and back-facing triangle, it is considered a silhouette edge.

Crease Edge Detection (CED). For potentially noisy meshes, it may be necessary to perform CED to detect important edges such as folds, corners, and edges that may have rapid geometric changes. The foundational Normal Vector Voting algorithm introduced by Page et al. [10] is a robust option for this purpose. It works by comparing the normal vectors across a local region of the mesh to identify significant features based on deviations between the normals.

Bézier curves. It may be difficult to represent certain meshes with curves using purely lines. The de Casteljau Algorithm [1] is an algorithm which lays the groundwork for producing bézier by linearly interpolating between control points. We choose this algorithm for its stability and ease of implementation.

3 METHODOLOGY

We build a mesh's "vectorization" by drawing lines that represent its silhouette and crease edges, and then use Bezier curves to properly represent curvature. The methods described are potentially generalizable to any modern mesh pipeline. Here, we leverage the Unity game engine [14] for its flexibility and extensibility in downstream applications. The Unity APIs used include the Unity base API for accessing mesh information (vertices, transforms, etc.) and the Graphics Library (GL) API for making low-level graphics library calls for drawing vector lines.

3.1 Silhouette Edge Detection

Mathematically, a silhouette edge for a polygonal model can be detected using three principles as specified by Hartner et al. [6] so long as the normals of the input polygon face outward from the surface:

- A polygon is front-facing if the dot product of the normal and eye vector is less than 0.
- A polygon is back-facing if the dot product of the normal and eye vector is greater than 0.
- A polygon is perpendicular to the view direction if the dot product of the normal and eye vector is equal to 0.

The main process of Edge Buffer SED is to check each edge in the mesh using the following steps:

- Get the triangles that contain the edge's vertices
- Compare the eye vector against the triangle normals to classify triangles as front-facing or back-facing
- Check if the list of triangles contains both front-facing and back-facing triangles

If the edge connects front-facing and back-facing triangles, it is a silhouette edge. Also, to ensure that edges are not processed multiple times per frame, we maintain a hash set of the already-processed edges.

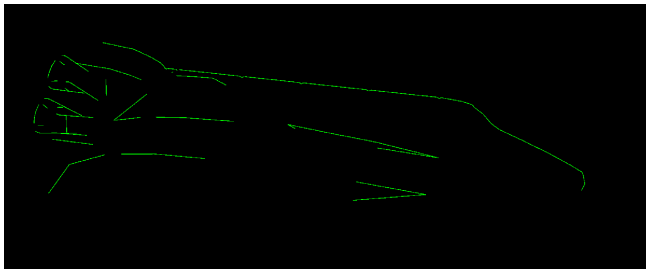

Figure 2: Silhouette rendered from silhouette edge detection algorithm. Missing some edges due to low-poly mesh, not enough edges to represent a perfect silhouette.

The outcome of running this algorithm alone is shown in Figure 2. A perfect silhouette depends on the complexity of the mesh, where the existing edges on the mesh determine the silhouette drawn. Therefore, more edges available to work with allow for a more accurate representation of a silhouette without breaks/discontinuous edge connections.

3.2 Crease Edge Detection

With more complex meshes, SED alone causes outputs to appear oversimplified. For instance, in Figure 3, because certain facial features such as the eyes, nose, and mouth don't lie on the outermost planes of the mesh, they are not shown at all by our SED implementation. So, in these cases we perform CED using Normal Vector Voting to retain more of the essence of the original mesh in its vectorization.

The Normal Vector Voting algorithm involves tensor-based analysis. The algorithm begins by initializing the voting tensors for each vertex in the mesh. These tensors are 4x4 matrices that encode information about the local surface orientation. We firstly initialize a zero tensor for each vertex to prepare for the upcoming process. The entire process consists of several iterations:

- For each vertex, we create tensor vote based on the face normals of the adjacent triangles
- These votes are weighted according to spatial relationships between vertices
- The accumulated votes form a tensor field that codes local surface structure
- Confidence measures are computed from eigenvalue analysis

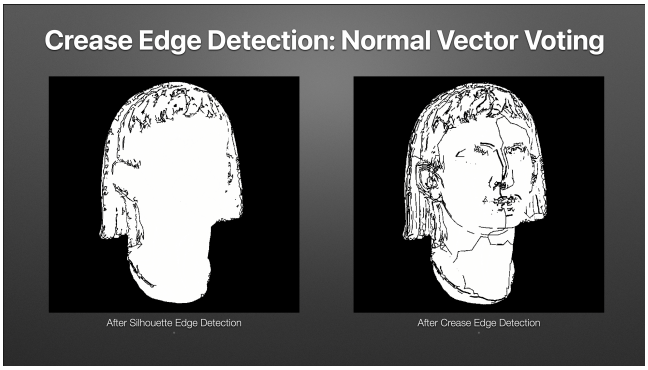

Figure 3: The silhouette edges of a statue mesh drawn by themselves on the left, and then in combination with the crease edges on the right. Used contour styling to depict a roman statue [5] with an unlit shader material (removes shadows on the 3D model).

The process for a number of iterations that is configurable to refine the results.

After the voting process, we use the following to determine if an edge is a crease:

- The angle between vertex normals on either side of the edge
- Confidence values derived from the tensor voring
- The edge's alignment with primary coordinate axes

As for the confidence calculation, we use eigenvalue analysis of the accumulated tensors to determine the reliability of edge classifications. This involves:

- Calculate the dominant eigenvalue through power iteration
- Analyzing the relationship between the primary and the secondary eigenvalues
- Use this information to establish confidence in edge classification decisions

In order to achieve the 'retro-style' look, edges will only be rendered if they meet both the crease detection criteria and alignment constraints. We check for alignment to ensure that only edges which are predominantly aligned with main coordinate axes are rendered, thereby avoiding the lines which would not fit to the classic vector graphic style. The following figure is the outcome of applying the crease edge detection algorithm + the existing silhouette edges:

3.3 Bézier Curves

Simply combining SED and CED may not represent rounded shapes well. In these cases, we create Bézier curves, using the de Casteljau's algorithm [1]. Our implementation checks every pair of edges within the collection of SED and CED edges and determines if the edges are curved based on the delta of the angle between them. If it falls within a significant range between 110-175 degrees, the two edge pairs are considered curved. To form a unison curvature for both edges, the curvature is assumed to be a line with 3 vertices, a combination of both edges. If instead we assigned a curvature individually to both edges, this would cause unnatural behavior of curvatures, not accurately representing the curvature of the mesh. Therefore, having a 3-vertex line is a more accurate way to depict

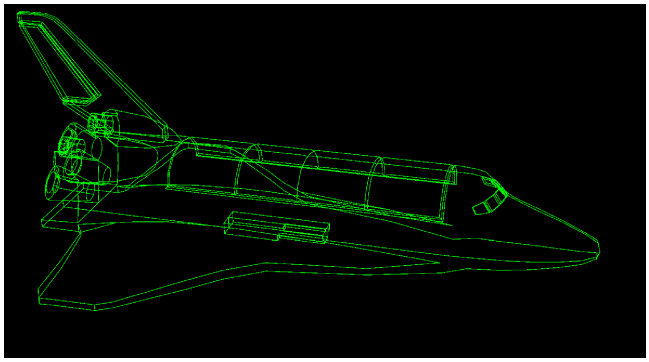

Figure 4: Crease edges drawn on top of existing silhouette edges from SED applied before.

curvature within the mesh, by having the outer vertices as the anchor points of the Bézier curve, and the curvature of the line must pass through the center vertex, the vertex that connected both edges. Also, the curvature must be along the plane of the two edges, to prevent cases where curvature extends away from the original line edge. Therefore, this gives us curves for edges along the crease and silhouette edges where it falls within the mentioned angle range. The following figure (Figure 5) depicts the outcome of applying Bézier curves to the line representation.

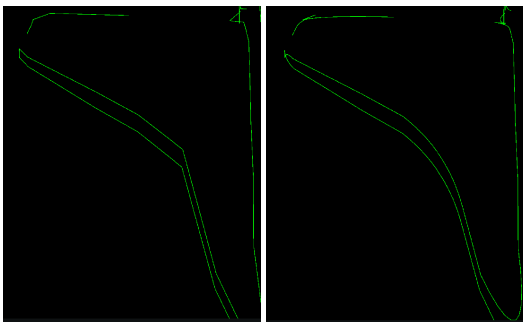

Figure 5: Left image: The spaceship's right wing rendered solely with straight lines. Right image: The same wing after applying Bézier curves for smoother, more elegant contours.

A downside to this implementation is that curves are applied to all edge pairs that fall within the indicated angle range. Therefore, corners that fall within this range but are not supposed to have a curved edge are also included as a curved edge within our mesh-to-line process. As a result, a perfectly accurate depiction of the mesh is not always reflected by our tool in these cases.

4 DISCUSSION

4.1 Simple Meshes

Initial testing of our vectorization tool yielded promising results. A simplified representation of the starting mesh was achieved, while obtaining the unique aspects of vector graphics such as scalability (no pixelization as the camera approaches object). We found that in all cases silhouette edges are necessary for rendering a complete

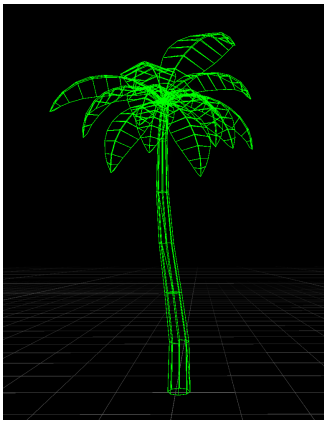

Figure 6: Simple mesh conversion of a palm tree model illustrating complete silhouette and crease edge detection

vector line representation of a mesh. We also found that only SED was necessary when applied to primitives and simple low-poly meshes such as a BattleZone Tank model [3] in Figure 1, though CED was necessary for slightly more complex models such as the palm tree [13] in Figure 6.

Silhouette edges depend on the complexity of the mesh; the less complexity the mesh has, the less accurate the silhouette line representation is rendered. Therefore, in cases where the mesh is a low-poly 3D model, the SED provides an incomplete silhouette, as seen in 2. However, applying CED makes up for the edges missing from the silhouette, outputting the expected simple line representation of the mesh without including all edges within the mesh, as seen in 6 and 4. In addition to accuracy, the real-time performance of Mesh-To-Line for these low-poly models is adequate, with negligible impact to computational load, according to Unity's internal performance profiler 10. Including Bézier curves, this reduces performance even more due to the addition of line segments for each line changed to a curved representation.

4.2 Stress-Testing

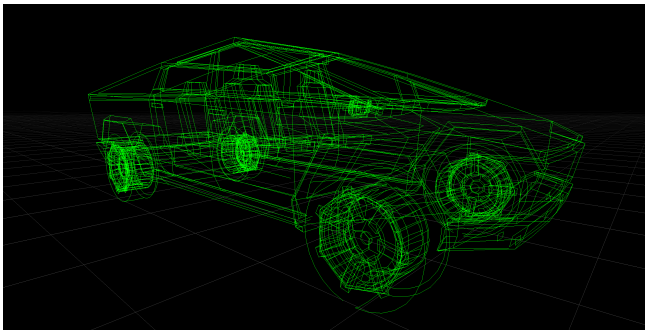

Figure 7: Advanced mesh conversion of a Tesla Cyber Truck model with a much higher poly-count

To stress-test our system further, we applied it to vastly more complex models in the range of tens of thousands of triangles including a Tesla Cyber Truck [11] (Figure 7), an X-Wing fighter [12] (Figure 1), and a Roman statue [5] (Figure 3). Compared to simple 3D models, a more accurate silhouette is generated. However, extraneous edges are included due to the complexity of our mesh; however, this does not take away from the final outcome when applying the CED which adds crease edges to the line render to get the final expected result.

In the matter of performance, it takes a noticeable hit due to lines rendered and deleted every frame in real-time. Inclusion of Bézier curves, similar to low-poly meshes, reduces performance even more due to added complexity of the line render. Therefore, in the matter of using this application of algorithms to produce line representations of the mesh, the more 3D models in the scene, the more it impacts the performance by a substantial amount. This method must be optimized or paired with another method to be viable, if used in real-time, like a video game created within the Unity engine.

4.3 Optimizations

To reduce the workload with the current implementation, we could further improve the tool by pairing it with other tools, such as Unity's Shader tool, and by optimizing the CED and SED scripts.

When optimizing the existing CED and SED scripts, we can limit the number of lines rendered each frame to a certain amount. We could also implement a Level of Detail (LOD) system based on the distance from the camera: if the 3D model is closer to the camera, more lines are rendered; the farther the model is from the camera, the fewer lines are rendered. Additionally, we could include depth culling for each individual mesh to remove unnecessary lines; however, this depends on the chosen style for the scene, as depth culling is not necessary when adopting the "retro" style depicted in [11] (Figure 7).

By incorporating other tools like Unity's Shader tool alongside scripting, we can optimize this process even further. Applying a shader to a mesh using a material allows for a more accurate depiction of the 3D model's silhouette, where the complexity of the mesh does not considerably affect the generated silhouette. One such method to extract the silhouette of a mesh is using a Sobel filter, where silhouette extraction is performed at the pixel level based on the camera's perspective. However, this may reduce the amount of control over the styling of rendered lines in exchange for better performance. Since the Sobel filter produces a texture representation of the silhouette, it cannot be used directly with our vectorization tool, complicating the production of a vector graphic (via an SVG file) of the 3D model. Therefore, while using shaders may yield better results and performance in some instances, they are limited in styling individual lines or producing a vector graphic representation of the 3D render.

Our implementation works directly with the mesh data and does not generate vertices or edges that do not exist, which may occur when using a shader alongside our script. In summary, we are trading performance for quality depending on the complexity of the 3D mesh. This method must be optimized or paired with another approach to be viable for real-time applications, such as video games created within the Unity engine.

4.4 Broad Applications

Although the focus and inspiration of the project are the "retro" styles depicted in the above figures, such as Figure 7, our project can be applied more broadly since silhouette and crease edges are used in various styles. For instance, cartoon and animated styles use silhouettes a lot to bring the focus of characters and items within the scene to the foreground. For simple styles, crease edges are not needed. For example, in Figure 8, only SED is used to produce silhouettes of the objects within the scene, creating a simpler cartoon-style look from a low-poly scene [4].

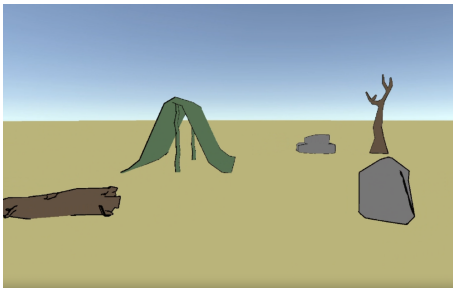

Figure 8: Depicts a toon-style rendering of a camp scene using only silhouette edges around objects within the scene. A shader was applied to all objects within the scene, removing shadows and applying uniform coloring.

We can also represent 2D art styles in a 3D representation using our line rendering tool, such as contour depictions of complex meshes, shown in Figure 3. Silhouette edges are used to define the outline of the 3D model, while crease edges are used to add more detail in the statues face to make it more recognizable.

Thus, our project can be broadly applied since silhouette representations, both linear and curved lines, of characters and objects within a scene are utilized in many forms of media (video game graphics, 2D/3D animations, and 2D art), with the addition of crease edges to add more detail to the line representation.

REFERENCES

[1] Wolfgang Boehm and Andreas Müller. 1999. On de Casteljau's algorithm. *Computer Aided Geometric Design* 16, 7 (1999), 587–605. https://doi.org/10.1016/S0167-8396(99)00023-0

[2] John W. Buchanan and Mario C. Sousa. 2000. The edge buffer: a data structure for easy silhouette rendering. In *Proceedings of the 1st International Symposium on Non-Photorealistic Animation and Rendering* (Annecy, France) *(NPAR '00)*. Association for Computing Machinery, New York, NY, USA, 39–42. https://doi.org/10.1145/340916.340921

[3] Creazilla. n.d.. Atari Battlezone Low-Poly Tank. https://creazilla.com/media/3d-model/7851151/atari-battlezone-low-poly-tank.

[4] Fab. 2024. Low Poly Nature Pack Lite. https://www.fab.com/listings/d2c038a0-302b-4197-b22b-b6a1b21a703b.

[5] Fab. 2024. Roman Statue. https://www.fab.com/listings/42bb2be3-9b8c-4d6d-af7c-f0edef2637da.

[6] Ashley Hartner, Mark Hartner, Elaine Cohen, and Bruce Gooch. 2003. Object Space Silhouette Algorithms. (2003). Unpublished.

[7] Aaron Hertzmann. 1999. Introduction to 3D Non-Photorealistic Rendering: Silhouettes and Outlines. In *ACM SIGGRAPH 99 Course Notes. Course on Non-Photorelistic Rendering*, Stuart Green (Ed.). ACM Press/ACM SIGGRAPH, New York, Chapter 7. http://mrl.nyu.edu/publications/npr-course1999/

[8] Immersive Archive. 2024. Immersive Archive. https://immersivearchive.org/.

[9] IndieDB. 2019. Making a Modern Vector Graphics Game. https://www.indiedb.com/games/paradox-vector/tutorials/making-a-modern-vector-graphics-game.

[10] D.L. Page, Y. Sun, A.F. Koschan, J. Paik, and M.A. Abidi. 2002. Normal Vector Voting: Crease Detection and Curvature Estimation on Large, Noisy Meshes. *Graphical Models* 64, 3 (2002), 199–229. https://doi.org/10.1006/gmod.2002.0574

[11] Sketchfab. 2019. Tesla Cybertruck. https://sketchfab.com/3d-models/tesla-cybertruck-657e71b3e2ad468196668e9c9df708fb.

[12] Sketchfab. 2020. Low-Poly X-Wing. https://sketchfab.com/3d-models/low-poly-x-wing-9eb1841d367d431cbf1c17f7b29c546c.

[13] Unity Asset Store. 2017. Free Trees. https://assetstore.unity.com/packages/3d/vegetation/trees/free-trees-103208.

[14] Unity Technologies. 2024. Unity Game Engine. https://unity.com. Version 2024.1.0f1.

A APPENDIX

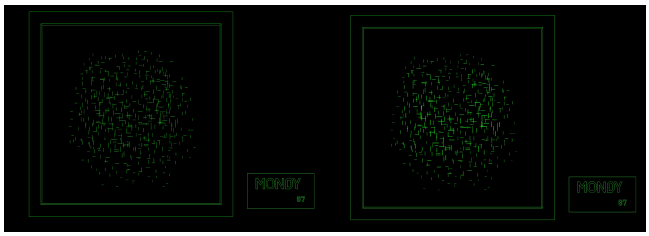

Figure 9: Comparison of true vector lines (left) to meshes (right) which appear thicker/thinner at varying depths

Figure 10: Unity performance profiler showing minimal performance impact when processing the BattleZone mesh, maintaining close to 200 FPS