EscherWeb

Mehrad Haghshenas[a]

[a] *200 University Ave W, Waterloo, ON N2L 3G1*

August, 2025

Abstract

EscherWeb is an interactive web tool for Escherization. We present a system that turns arbitrary input shapes into mathematically valid *tileable* forms using established optimization methods. It pairs a canvas-based interface with the algorithm taken from Yuichi Nagata's *EscherTiling* C++ codebase [4], letting users search, draw, or upload shapes and obtain tiles that cover the plane. We also host an online gallery that showcases example runs using polygons from the *EscherTiling* repository - content that would otherwise require installation and running of the original binary to view. Conceptually, the work sits at the intersection of tiling theory and computation: it tackles the challenge of shaping user-supplied figures to satisfy isohedral tiling constraints while preserving visual similarity to the input, a problem known as *Escherization* and first formalized by Kaplan and Salesin [2]. Overall, we implement a two-tier architecture consisting of a front-end canvas interface and an Escherization algorithmic core powered by Yuichi Nagata's *EscherTiling* C++ codebase [4].

1. Background

Escherization is the problem of finding a tile shape that closely resembles a given input figure while allowing copies of that shape to fit together and *tile* the plane [2]. In other words, given an arbitrary closed shape, we seek a new shape of similar appearance that satisfies the constraints of a *isohedral periodic tiling*. Note that a *tiling* is a covering of the plane with tiles that leave no gaps or overlaps. An *isohedral tiling* is a tiling in which all tiles are congruent and related to each other by the symmetries of the tiling.

This intriguing challenge was first formalized by Kaplan and Salesin in 2000, who proposed a simulated annealing approach to deform the input shape into a tiling-compatible form. Subsequent research by Koizumi and Sugihara reformulated Escherization as an eigenvalue problem [3][1], providing an alternative optimization strategy. These algorithms demonstrate the role of computational methods in art and tiling theory, enabling the creation of M.C. Escher-like tessellations from arbitrary shapes.

Despite these advances in methodology, there has been a lack of tools to bring Escherization to general users. For example, Tiled.art [6] is an interactive browser-based platform to create and explore tessellations. It provides symmetry templates, allowing users to draw directly within a fundamental domain, so the result is tilable by design. While Tiled.art focuses on manual creativity within fixed symmetry templates, our work goes further by performing automatic Escherization - transforming an *arbitrary* input shape into a similar-shaped form that can tile the plane isohedrally. Furthermore, C. S. Kaplan's *Tactile* library [1] offers a way to represent and draw isohedral tilings (supporting 81 edge-to-edge types). Yuichi Nagata's EscherTiling on the other hand is a C++ program that implements an eigen-based approach to Escherization, transforming an input polygon into a shape that tiles the plane isohedrally. Distributed as a standalone binary, it produces tilings from user-supplied shapes but requires local installation and manual execution.

However, no single web application combines Escherization algorithm, interactive arbitrary shape design, and a full-featured tiling visualization interface. EscherWeb is designed to fill this gap. The goal of EscherWeb is to provide users a convenient platform to turn their own drawings or images into repeatable tiling patterns. In the following sections, we outline the architecture of EscherWeb, detail the steps taken to integrate a C++ Escherization engine into the web, demonstrate results obtained with the system, and discuss challenges and future extensions.

2. System Overview

EscherWeb employs an architecture that separates the user interface and the algorithmic core. The

[1] More recent work on this can be found in [5].

front-end is a responsive web application (built with React and HTML5 Canvas) that allows users to create or provide input shapes. Users can draw closed shapes freehand on a canvas. The interface also includes options to *search* for images (using integrated APIs for image libraries) or *upload* an image in the *PNG/JPG* format from the local computer. In the backend, *Unsplash*, *Pexels*, and *Pixabay* APIs are used to retrieve images. This functionality is rate-limited, and frequent requests may result in temporary unavailability.

Once an image is selected, the front-end uses **OpenCV.js** (a WebAssembly port of OpenCV) to remove any background and extract a clean contour outline of the shape. The user can specify the number of points for the polygonal approximation of the contour (trading off detail vs. complexity). For performance reasons, please select a maximum of 50 points (even though the web app technically supports up to 200, in steps of 5) to avoid potential crashes. This contour then serves as the *goal shape* for Escherization.

At the core of EscherWeb is an integration of Yuichi Nagata's *EscherTiling* C++ codebase [4]. By leveraging Nagata and Imahori's recent work, our system benefits from a solver that can handle complex shapes robustly. The algorithm layer is used as a WebAssembly module (for in-browser execution). Specifically, the goal shape is passed to the Escherization algorithm, compiled from C++ to WebAssembly via `em++`, and executed directly in the client-side. The algorithm outputs a modified polygon that meets isohedral tiling constraints while retaining close visual similarity to the original. The tiling plane can be viewed in two ways: 1) The EscherTiling by Nagata and Imahori provides a tiling display which has been extensively modified to work in our web interface. 2) rendered in the browser using tactile-js.

3. Technical Contributions

Yuichi Nagata's *EscherTiling* is a C++ implementation of optimization algorithms for the Escherization problem. The software accepts a *goal figure* in either *polygonal* or *mesh* format and applies exhaustive or heuristic search to produce isohedral tilings that closely match the target shape, supporting multiple distance metrics (Euclidean, AD, E_I, E_{IR}). Distribution is as a standalone binary requiring compilation on a Unix-like environment with Eigen and X11. Output can be displayed through the included X11-based viewer. In our work, we first replace the X11 viewer with a web-based tiling renderer. Then we compile *EscherTiling* to WebAssembly with `em++` and integrate it into an interactive React front-end. This allows shapes to be processed entirely in-browser, and displayed live without installing or running the original binary. To recapitulate, integrating the *EscherTiling* C++ codebase required several technical steps to extend its functionality. We highlight two main contributions: (1) modifying the rendering subsystem of the code to enable headless image generation, (2) compiling the code to WebAssembly for browser use. Note that in this case the *jikken_E.js* in the *src/components* directory was automatically generated from the modified C++ codes of [4] repository.

3.1. Native Apple Silicon Build of EscherTiling

The EscherTiling codebase was originally developed on Linux (Ubuntu) and depends on the Eigen linear algebra library and X11 for display. To use this code on a Mac with Apple Silicon (M1 chip), we created a custom Bash build script and made some code changes. The build script sets the include paths for Eigen (installed via Homebrew) and the X11 libraries (via XQuartz). Running this script will produce a native executable called `jikken_E`. We obtained a working Apple Silicon-native binary of Nagata's solver that once compiled, the `jikken_E` program can be invoked with command-line arguments to perform Escherization on a given input shape data file and produce a solution. The script can be found in the appendix.

3.2. WebAssembly (WASM) Compilation for Browser

To move toward a fully in-browser solution, we set up an Emscripten build of the EscherTiling code. Emscripten compiles C++ into WebAssembly, allowing it to run within a web page's JavaScript environment. We configured Emscripten with appropriate flags to compile the core solver logic (excluding the X11-dependent parts). The output of this process was a `jikken_E.wasm` module accompanied by a `jikken_E.js` code, which our React app loads asynchronously. Through this WebAssembly integration, the browser can execute the Escherization algorithm directly on the client side. Note that there are important limitations. The computational intensity of the exhaustive search algorithm means that running it in the browser can be slow, especially for shapes with many vertices. Browsers impose restrictions on execution time, so long runs risk freezing or being aborted. In practice, we found the WebAssembly version useful for moderate complexity shapes (up to 50 points), but not ideal for very complex cases.

3.3. Headless Rendering of Tilings

Yuichi Nagata's original EscherTiling program was designed as a command-line solver that outputs

numerical results and, if enabled, launches an interactive X11 window to display the tiling solutions. A technical contribution of our project was modifying this display mechanism to enable **headless** rendering of the tilings. We achieved this by editing the program's source (notably `display.cpp` and `display.h`) to remove X11 dependencies. In our modified version, when the solver finds a tiling solution, it uses an image library to draw the tile edges. At the end of the run, it writes the result to a PNG image file. This automated image generation allows EscherWeb to obtain the tiling visualization without any intervention. Note that the *EscherTiling* algorithm only draws tilings for isohedral types 1, 4, 5, and 6).

4. How to use

This project uses **Node.js** and **Vite** so be sure to install them.

1. Clone the repository in repository.
2. Install Dependencies *npm install*.
3. Run in Development Mode *npm run dev*.

The app will be available at *http://localhost:5174/* by default. The *favicon* of the website looks like the following:

Figure 1: Favicon used in the EscherWeb interface. Image taken from Kaplan & Salesin (2000).

5. Features

Upon visiting the application (in a web browser), the user is presented with a navigation menu offering different modes: **Home**, **Search**, **Draw**, **Escher**, **Gallery**, and **About**. The typical workflow begins either in the Search or Draw mode. In the **Search** mode, users can enter a keyword to find an image of interest. The app connects to online image APIs (Unsplash, Pexels, Pixabay) and displays matching images (re-emphasizing that heavy usage may be rate-limited). Once the user selects an image, they can click on the *Use in Escher* button. This action transitions to the **Escher** page, carrying over the selected image. In the **Draw** mode, users can free-draw a shape on an HTML5 canvas. The drawing toolkit allows basic operations like dragging lines. After drawing, a similar button brings the user to the Escher page with their drawn shape.

On the **Escher** page, the user sees either the searched or drawn shape. Note that in this stage, the user can upload their local image as well. The first step here is to generate a polygonal contour of the image. The user selects the desired number of vertices for the contour approximation (for performance and simplicity, we allow up to 200 points, in increments of 5, but we recommend using around 50 or fewer points for best results). Upon clicking **Get Contour**, the system (via OpenCV.js in the browser) removes the background and computes the outline of the shape. The result is a simplified polygon capturing the essence of the input shape. This polygon is then displayed to the user, who can download it or inspect the coordinates if desired. Moving on, the user can initiate the Escherization process and the tiling results are presented. The output is shown in two ways for comparison: first, a rendered image generated by the EscherTiling backend itself, and second, a live tiling drawn using TactileJS on the canvas. The Gallery page of the app showcases a few pre-generated examples to inspire users, and the About page provides an overview of the project and credits (including academic references and acknowledgments).

6. Results

A short video demonstrating the typical usage of EscherWeb is available at Escherization of searched or uploaded images and Escherization of drawn images, and a gallery of tilings generated by users can be found at the gallery page of the website. A demo of the gallery can be found in the following video gallery.[2] In the following you can see a screenshot of the *gallery* page:

The following example provides a clearer representation of this pipeline. The example is a silhouette of a bat. From this shape, we extract its contour, perform an Escherization step to generate a valid prototile, and finally apply an isohedral tiling algorithm to cover the plane with repeated copies of the resulting shape.

7. Challenges

Developing EscherWeb required overcoming some challenges: **Integration of Code:** Integrating

[2]The tiling output from the C++ backend and the `tactile-js`–rendered tiling differed. We believe this is either due to improper use of the `tactile-js` API or to `tactile-js` always including a fixed background motif when generating the tiling?

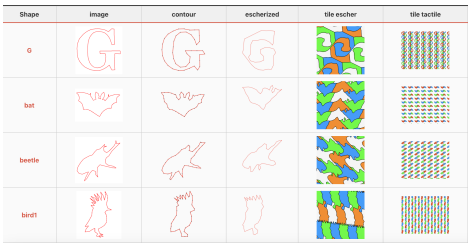

Figure 2: Gallery rows illustrating the input shape, contour, its Escherization, and the resulting isohedral tiling.

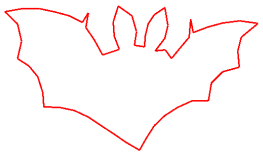

Figure 3: Input bat shape.

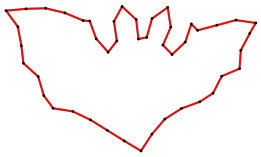

Figure 4: Extracted contour of the bat.

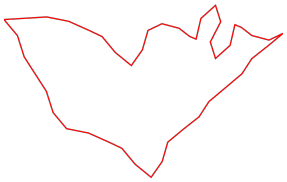

Figure 5: Escherized prototile of the bat.

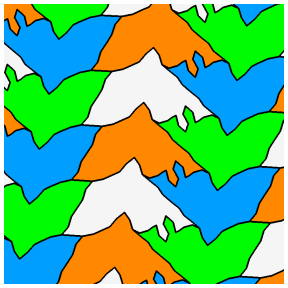

Figure 6: Isohedral tiling of the bat.

EscherTiling C++ required careful reading of the code and understanding input/output formats. The solver expects input shapes in a specific text file format and outputs solution data in another format. We had to create adaptors to construct this input from a user's drawn shape or extracted contour, and then to parse the output to retrieve the tile's vertex coordinates and isohedral type. **Cross-Platform Compatibility:** We encountered issues in making the code portable across operating systems. For instance, the reliance on X11 for graphics meant we had to have the necessary tools. On macOS, we installed XQuartz but ultimately avoided using X11 by moving to headless rendering. **Performance:** Perhaps the most significant challenge was balancing the heavy computation. Running an exhaustive search Escherization in the browser can easily freeze the UI. We addressed some of this but even so, a long computation could lead users to think the app had stalled. **User Interface and Experience:** On the front-end side, one challenge was designing an intuitive workflow. Escherization involves several steps (choose or draw shape, simplify shape, run algorithm, view tiling) which could confuse users if not guided properly. We addressed this by clearly separating the steps into the modes/pages described in the Features section.

8. Future Work

EscherWeb demonstrates a working solution for interactive Escherization, but there are many avenues for future improvement. We outline a some of them here: **Interactive Tile Editing:** One limitation of the current system is that once the algorithm produces a tile, the user cannot easily adjust it except by restarting the process with a modified input. A enhancement would be to allow users to tweak the resulting tile's shape manually while maintaining the tiling constraints. **Performance and Scalability:** To handle more complex shapes or to make the tool more responsive, further optimization is needed. One route is to improve the WebAssembly performance. Additionally, one could implement a server backend: the shape data could be sent to a server that runs the heavy algorithm and returns the result. **Expanding to More Tiling Types:** Our focus was on isohedral tiling. There are other classes of tilings that could be incorporated. For example, *dihedral tilings* involve two shapes alternating. There has been research on generalizing Escherization to multiple tiles, which could be explored. **Multiple Escherization Algorithms:** A future version of EscherWeb could include multiple algorithmic backends – for example, Kaplan and Salesin's original simulated annealing method – and allow users to select which one to use. **UI Enhancements:** On the user experience side, future work can make the tool more fun to use. For example, by adding polygon tools drawing features to ensure that the shapes are closed. **Animation** that shows the transformation from the original shape to the final tile shape gradually, which could be an engaging visualization of the optimization process.

In conclusion, there remain many opportunities to improve both the power and usability of the system. We hope that this work will inspire further developments in continuing M.C. Escher's legacy. Last but not least, this project serves two purposes:

- generating escherizations and tilings for random images.
- hosting an online gallery of the polygons from the EscherTiling project.

Acknowledgement

***LLM Disclosure**: Portions of the text in this report were refined with the assistance of ChatGPT Language Model (LLM). However, all content was authored and verified by the human author, with the LLM used to improve clarity and coherence. Furthermore, the code was developed with the assistance of GitHub Copilot integrated into VS Code. So it is practically impossible to distinguish which portions were generated with the assistance of AI or otherwise. Lastly, the project was development on a mac M1 computer.*

References

[1] C. S. Kaplan, *TactileJS: a javascript library for representing, manipulating, and drawing isohedral tilings*, https://github.com/isohedral/tactile-js, Accessed: 2025-06-30, 2019.

[2] C. S. Kaplan and D. H. Salesin, «Escherization», in Proceedings of the 27th annual conference on computer graphics and interactive techniques (siggraph '00) (2000), pp. 499–510, 10.1145/344779.345022.

[3] H. Koizumi and K. Sugihara, «Maximum eigenvalue problem for escherization», Graphs and Combinatorics **27**, 431–439 (2011) 10.1007/s00373-011-1022-5.

[4] Y. Nagata, *Eschertiling*, https://github.com/nagata-yuichi/EscherTiling, Accessed: 2025-08-15.

[5] Y. Nagata and S. Imahori, «An efficient algorithm for the escherization problem in the polygon representation», arXiv preprint arXiv:1912.09605 (2019).

[6] *Tessellations, escher-style*, https://tiled.art/en/home/?id=HaveAParty, Accessed: 2025-08-15.

Appendix A. Build Script for *EscherTiling*

The following Bash script was used to compile *EscherTiling* C++ codebase with the correct include and library paths for Eigen and X11 on macOS:

```
#!/bin/bash

EIGEN_PATH=
"/opt/homebrew/Cellar/eigen/3.4.0_1/include/eigen3"
X11_INCLUDE="/opt/X11/include"
X11_LIB="/opt/X11/lib"

g++ -o jikken -O3 \
    -I$EIGEN_PATH \
    -I$X11_INCLUDE \
-L$X11_LIB \
main_E.cpp env_E.cpp search_E_EST.cpp
search_E_conf.cpp \
operator_base.cpp operator_E.cpp
operator_AD.cpp \
display.cpp Xwindow4.cc \
-lm -lX11 -std=c++11
```