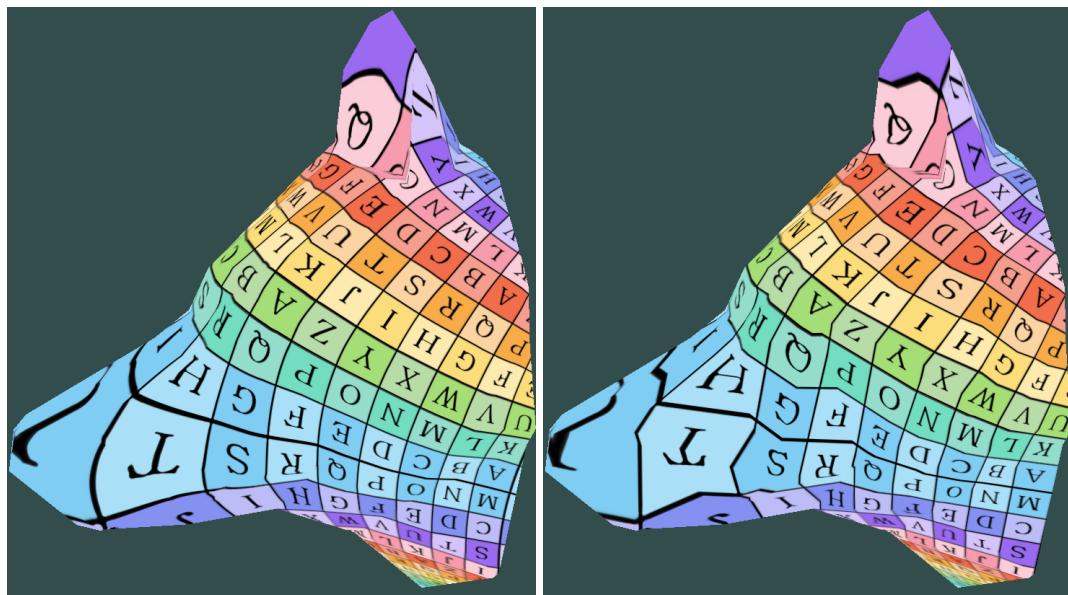


236328 Research Project in Computer Graphics

C++/OpenGL implementation of

BPM: Blended Piecewise Möbius Maps

Rorberg, Vaxman, Ben-Chen



Submitted to: Prof. Miri Ben-Chen

Submitted By: Ehud Gordon

Contents

1	Technical Report	1
1.1	Algorithm Implementation	1
1.1.1	Summary	3
1.2	Highlights	4
2	Results	6
3	Speckling	8
3.1	Speckling Solution	9

1 Technical Report

I've implemented the BPM algorithm in C++ using OpenGL shaders. I will first describe the implementation, and then highlight key points.

1.1 Algorithm Implementation

Input: The input is a 3D mesh \mathcal{M} with a discrete texture map $F : \mathcal{V} \rightarrow W$. Specifically, an OBJ file, with 'mtllib' and 'usemtl' commands to specify texture W , and with 'vt' coordinates to specify $F : \mathcal{V} \rightarrow W$. An example is given in file 'wolf_head.obj' in the [github repository](#).

Output: The code renders the given mesh using the texture map. The user can choose between Linear-Piecewise and BPM interpolation scheme by pressing the T key. For more info, refer to the [user guide](#).

Implementation:

1. Normalize the texture coordinates in the following way: calculate the minimal and maximal values in each axes of the texture coordinates. For each axes, find its range, and denote the maximum of the two ranges by s . The normalized coordinate is then:

$$\text{vt} := \frac{\text{vt}_{\text{old}} - [x_{\min}, y_{\min}]^T}{s}$$

2. Rotate vertices to be co-planar. In the GPU, using a compute shader, for each triangle t of \mathcal{M} :
 - (a) find t 's three adjacent triangles u, v, w (Shown in Fig. 1) if they exist.

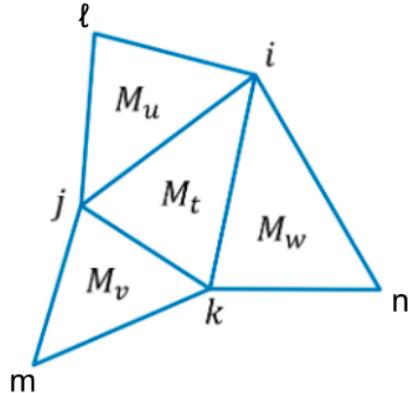


Figure 1: Triangle star notation.

Let v_i, v_j, v_k denote t 's three oriented vertices. Compute the change-of-axis transformation T that takes points in \mathbb{R}^3 to t 's plane, with the following convention: The normalized vector $v_j - v_i$ is the first axis. The line orthogonal to it, lying in t 's plane, pointing from v_i to v_k , is the second axis. Using the right-hand rule, the third axis is computed.

- (b) Transform t 's vertices and each of the vertices v_ℓ, v_m, v_n to t 's coordinates system.

- (c) rotate v_ℓ, v_m, v_n so they will lie in t 's plane. See Fig. 3. This is explained in highlights.
3. On the CPU, compute the unique normalized Möbius map M_t between v_i, v_j, v_k (after rotation) and their corresponding texture coordinates w_i, w_j, w_k . To compute the normalized Möbius map:

$$M = \begin{pmatrix} \hat{a} & \hat{b} \\ \hat{c} & \hat{d} \end{pmatrix}$$

Compute the unnormalized coefficients using:

$$\begin{aligned} a &= \det \begin{pmatrix} z_i w_i & w_i & 1 \\ z_j w_j & w_j & 1 \\ z_k w_k & w_k & 1 \end{pmatrix} \\ b &= \det \begin{pmatrix} z_i w_i & z_i & w_i \\ z_j w_j & z_j & w_j \\ z_k w_k & z_k & w_k \end{pmatrix} \\ c &= \det \begin{pmatrix} z_i & w_i & 1 \\ z_j & w_j & 1 \\ z_k & w_k & 1 \end{pmatrix} \\ d &= \det \begin{pmatrix} z_i w_i & z_i & 1 \\ z_j w_j & z_j & 1 \\ z_k w_k & z_k & 1 \end{pmatrix} \end{aligned}$$

and then normalize a, b, c, d by the factor $\sqrt{ad - bc}$. Similarly compute M_u, M_v and M_w . We normalize the matrices so they'll be in the Lie Group $\text{SL}(2, \mathbb{C})$. This will be used later in interpolation.

Note - transform and rotate only the vertices positions, not the texture coordinates. Then compute the Möbius ratio:

$$\delta_{tu} := M_t^{-1} M_u$$

Order is important, as $\text{SL}(2, \mathbb{C})$ is non-abelian. $\delta_{tu} \in \text{SL}(2, \mathbb{C})$ as it is closed under inversion and multiplication.

When a neighboring triangle does not exist, set its Möbius ratio to be the 2×2 zero matrix.

4. For each neighboring triangle u , compute the log Möbius ratio:

$$\ell_{tu} := \log(\text{Sign}(\text{Tr}(\mathcal{R}(\delta_{tu}))) \cdot \delta_{tu})$$

When the neighboring triangle u does not exist, $\ell_{tu} = \text{Id}$.

Taking the log takes δ_{tu} from the Lie group $\text{SL}(2, \mathbb{C})$ to the Lie algebra $\mathfrak{sl}(2, \mathbb{C})$.

5. For each triangle t , store on the GPU the 3D-to-2D transformation T , the Möbius map M_t and log Möbius ratios ℓ_{tu}, ℓ_{tv} and ℓ_{tw} . Use for example Shader Storage Buffer Objects (SSBOs). The computation in the previous steps is performed only once, and used only in the GPU. Storing the results on the GPU keeps them closest to where they'll be used, saving I/O operations.

Rendering loop

In the fragment shader, for a pixel associated to a mesh point $z' \in \mathbb{R}^3$ inside triangle $t = ijk$, provide as input the triangle ID, the 2D locations of transformed v_i, v_j, v_k , the Möbius map M_t and the Möbius log ratios $\ell_{tu}, \ell_{tv}, \ell_{tw}$.

6. Transform z' to t plane using the transformation T . Let's denote the result by z , a point \mathbb{R}^2 . Next, compute the edge barycentric coordinates. This is a triplet coordinate system $(\gamma_{ij}, \gamma_{jk}, \gamma_{ki})$, each coordinate specifying distance from an edge. For point z inside a triangle, we have $\gamma \geq 0$, and $\sum \gamma = 1$. The computation is as follows:

- (a) if z is close up to epsilon to a vertex, denote the adjacent edges a, b , and set $\gamma_a(z) = \gamma_b(z) = 0.5, \gamma_c(z) = 0$.
- (b) else, if z is close up to epsilon to an edge a , set $\gamma_a(z) = 1$.

- (c) else, compute the euclidean distance $d_{ij}(z), d_{jk}(z), d_{ki}(z)$ from z to each edge. Then the normalized weights are $\gamma_{ij}(z) = \frac{d_{jk}(z)d_{ki}(z)}{s(z)}$, where $s(z)$ is the normalization factor:

$$s(z) = d_{ij}(z)d_{jk}(z) + d_{jk}(z)d_{ki}(z) + d_{ki}(z)d_{ij}(z)$$

and similarly for $\gamma_{jk}(z)$ and $\gamma_{ki}(z)$. The weights are visualized in Fig. 2

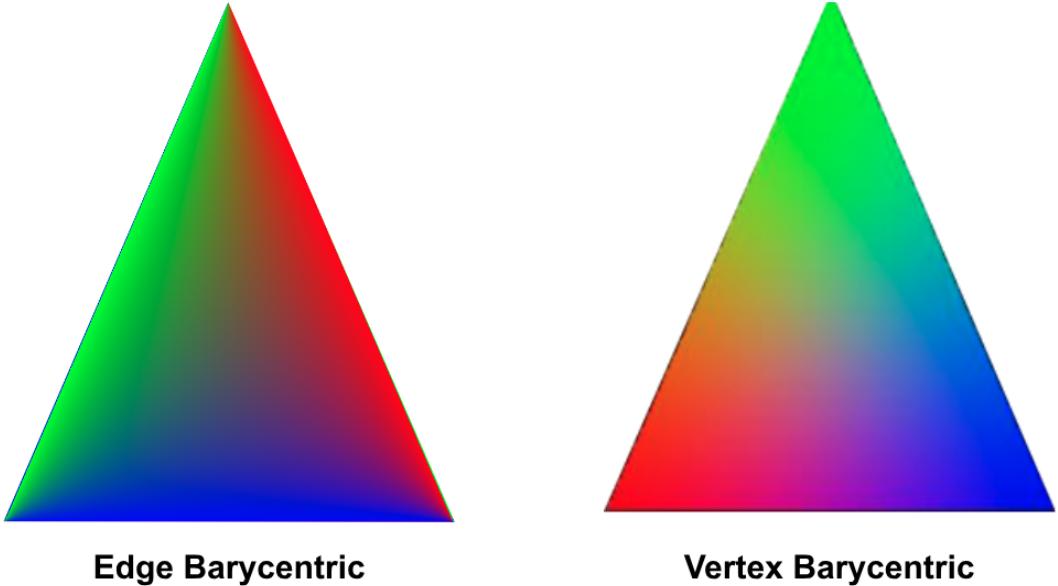


Figure 2: (left) edge barycentric coordinates, as used in the paper. Edges are assigned Red/Green/Blue color.
(Right) vertex barycentric coordinates

7. Compute blended log Möbius ratio:

$$\ell_t(z, M) = \gamma_{ij}(z)\ell_{tu} + \gamma_{jk}(z)\ell_{tv} + \gamma_{ki}(z)\ell_{tw}$$

Note: $\ell_t \in \mathfrak{sl}(2, \mathbb{C})$ since it is a vector space.

8. Compute the blended Möbius map:

$$M_z = M_t \exp\left(\frac{1}{2}\ell_t(z, M)\right)$$

Note the order of multiplication.

The exponential map takes ℓ_t from the Lie algebra to the Lie group. $M_z \in \text{SL}(2, \mathbb{C})$ as its closed under multiplication.

9. The interpolated texture coordinates for z are:

$$f(z) = M_z(z)$$

1.1.1 Summary

The table below summarizes the operations done and their host.

Operation	CPU/GPU
Parse OBJ file	CPU
Normalize Texture	CPU
Find Triangle's Neighbors	GPU
Rotate and Transform Vertices	GPU
Compute Möbius Maps	CPU
Compute Log Ratios	CPU
Store Maps on GPU	GPU
Compute Edge Barycentric Coordinates	GPU
Sample Texture	GPU

1.2 Highlights

Numerical Issues when Computing Determinant. Initially I've attempted to compute the Möbius maps (Item 3) on the CPU. This incurred numerical problems related to performing floating-point calculations using OpenGL on the GPU. Specifically, the computation of the determinant, required for computing the Möbius map, gave wildly inaccurate results, compared to the same computation, line for line, performed on the CPU, using GLM. Changing precision from float to double, which improved the results, had the effect of increasing the runtime to impractical levels.

Computing Matrix Exponent using Taylor Series. The blended log Möbius ratio (Item 8) involves computing the exponent of a matrix. As OpenGL does not natively supports matrix exponentiation on the GPU, initially I've used a naive implementation of the formula:

$$\exp(M) = M + \frac{M^2}{2!} + \frac{M^3}{3!} + \dots$$

I've also implemented the closed-formula [given by the Laurent series](#). However, the Taylor formula turned out to be sufficient. Taking only the first three terms produced visually-indistinguishable results from using the more complex closed formula.

Efficiency of Computing Neighbors on GPU vs CPU. In Item 2, when computing the neighbors of each triangle, I've split the task to the GPU cores. This approach is critical to the runtime of the program. For large mesheses, the GPU finished in around 2 seconds, compared to 10 minutes on the CPU.

Computing Rotation Angle for Isometric Embedding As for the angle needed to rotate a neighboring angle, I've implemented the following approach: Let triangles ijk and jiv' be as in Fig. 3. We wish to compute a rotation matrix around line $vj - vi$, so that \mathbf{v}_2 , the normal to triangle jiv' (shown in green) will point in the positive z -direction. Let's denote the angle from \mathbf{v}_2 to the positive z -axis by θ . Denote $\mathbf{e}_1 = \frac{v_j - v_i}{\|v_j - v_i\|}$, and define $\mathbf{v}_1 := \mathbf{e}_1 \times \hat{\mathbf{z}}$. Then:

$$\theta = \arctan(-(\mathbf{v}_2)_2, (\mathbf{v}_2)_1)$$

where $(\mathbf{v}_2)_2$ is the component of \mathbf{v}_2 in the \mathbf{v}_1 direction, and $(\mathbf{v}_2)_1$ is the component of \mathbf{v}_2 in the positive z -axis.

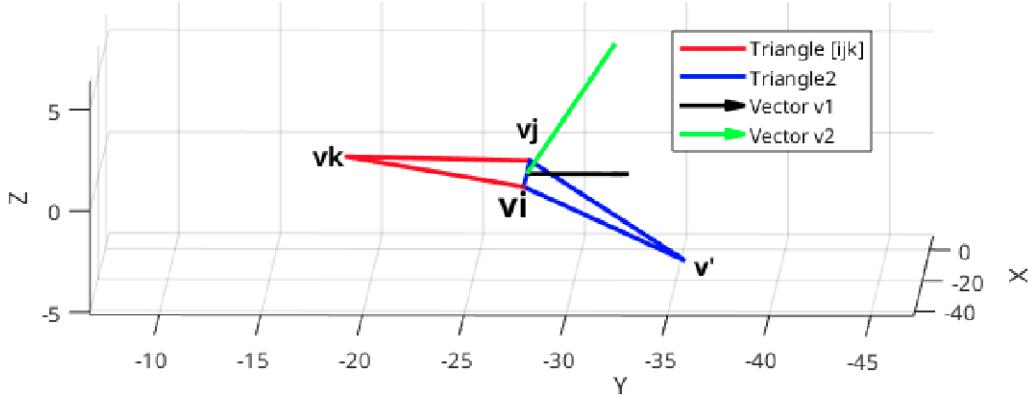


Figure 3: Triangle ijk (in red) and neighbor triangle jiv' (in blue) after transformation to t 's plane. The normal to jiv' is shown in green. Triangle jiv' is rotated so it'll be co-planar to ijk .

Compatible Orientation of vertex and texture triangles Inevitably, when flattening a triangle in 3D to 2D, we endow it with an orientation. A convenient choice is to give it a right-hand orientation. Note that the triangles of the texture coordinates also have an orientation. If they don't have a right-handed orientation, then the Möbius matrix, mapping between the flattened triangle and the texture triangle, will map points inside the source triangle to points *outside* the target triangle. This is shown in Fig. 4.

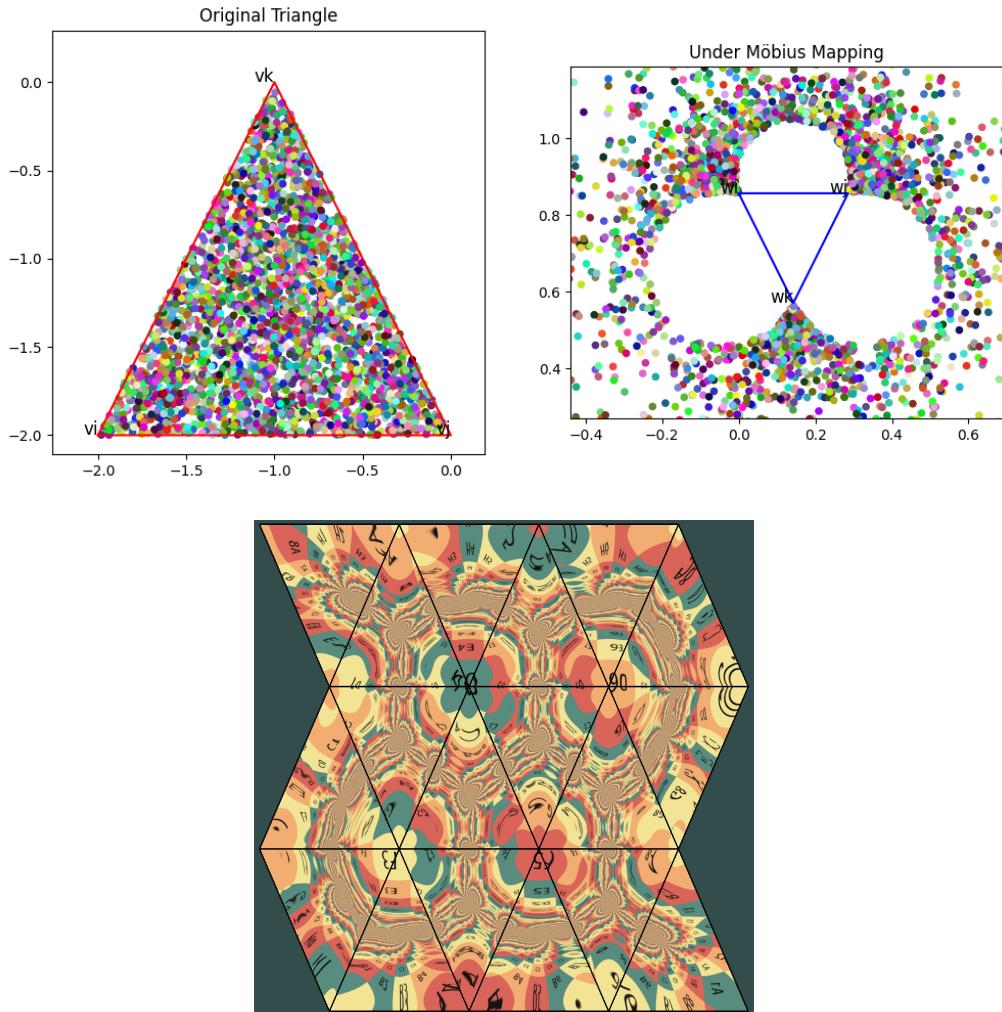
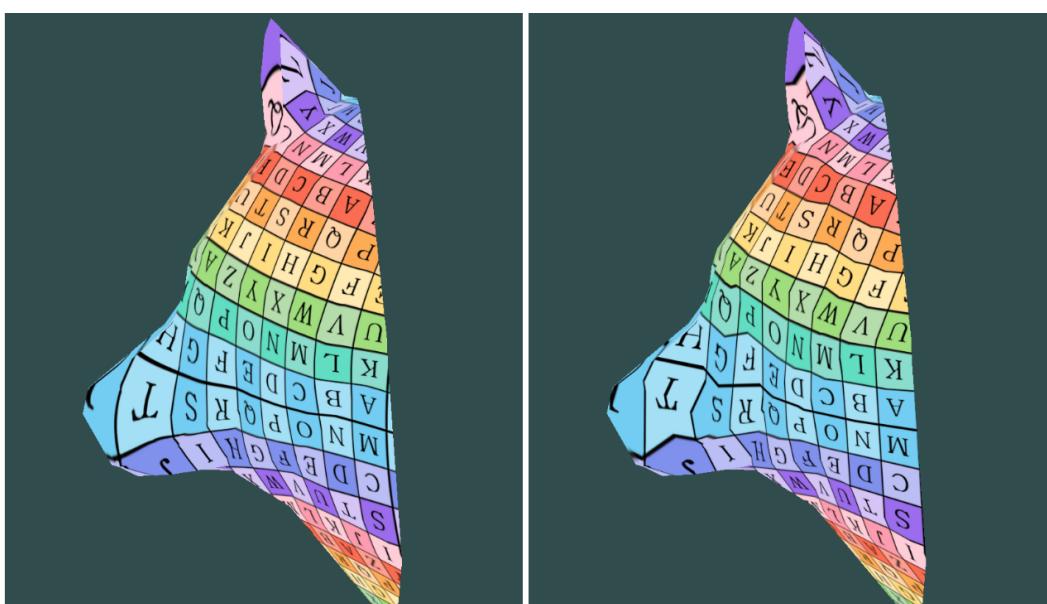
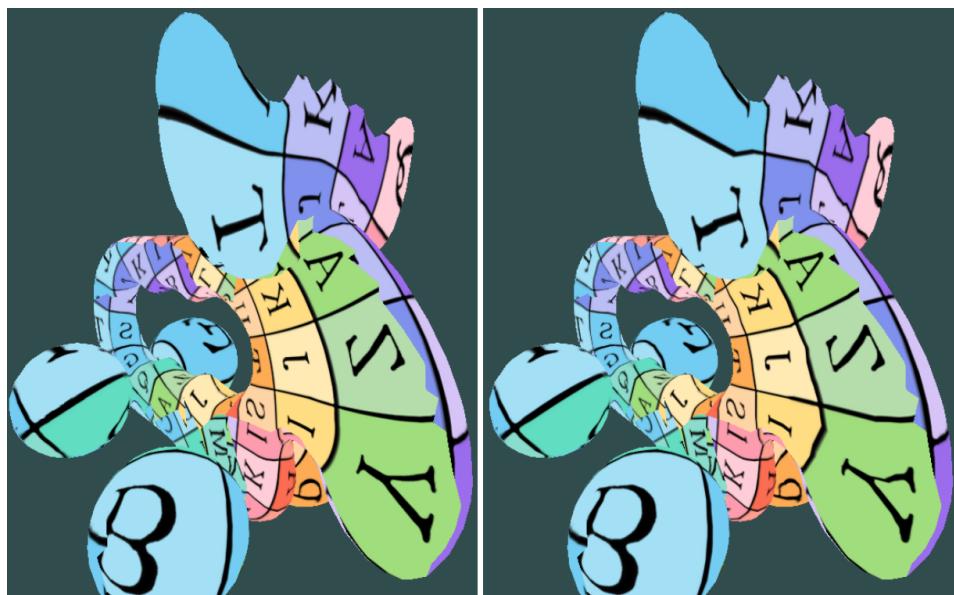
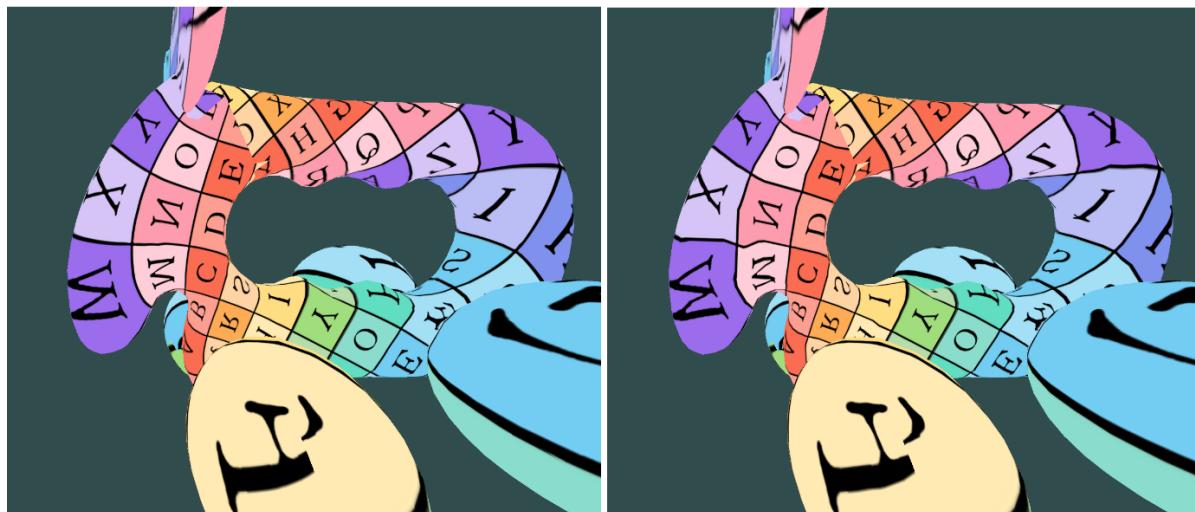


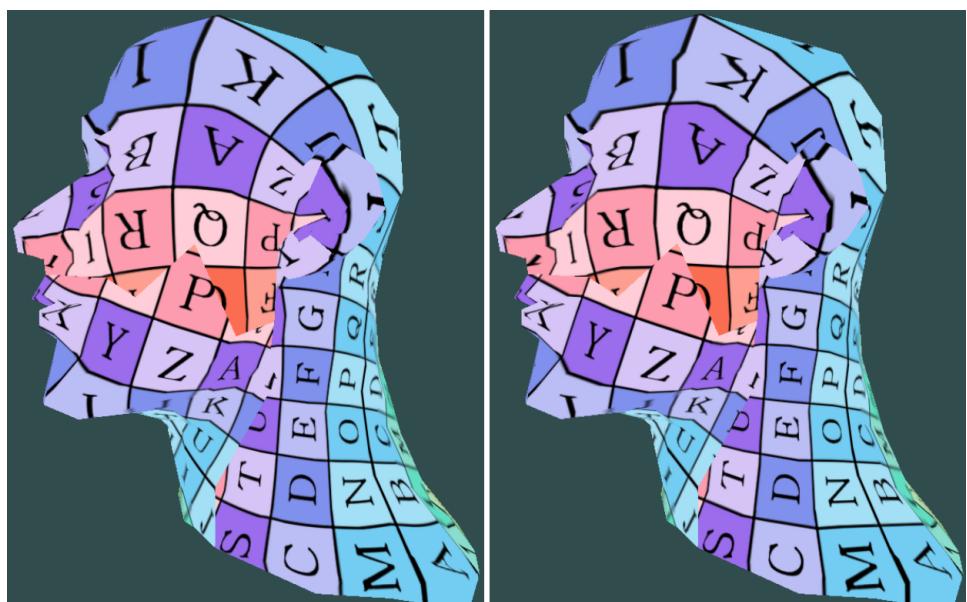
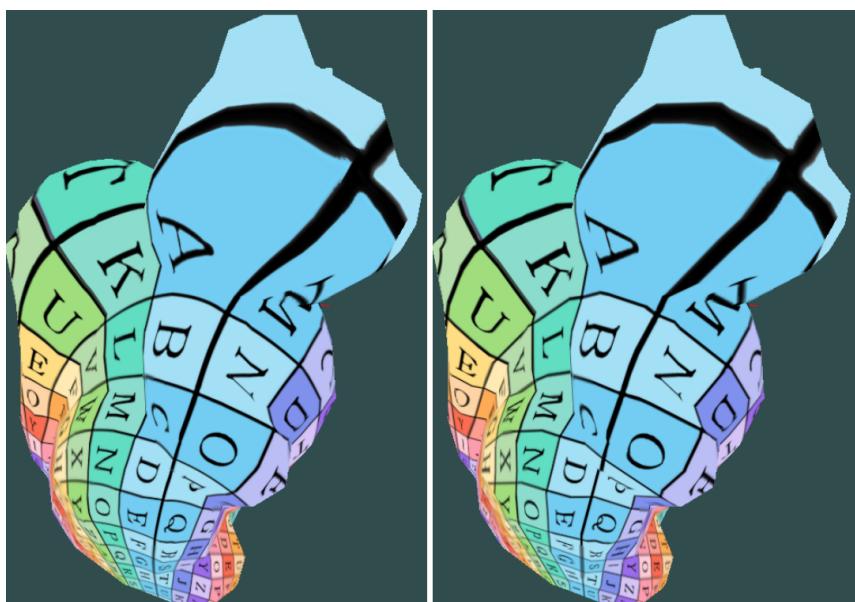
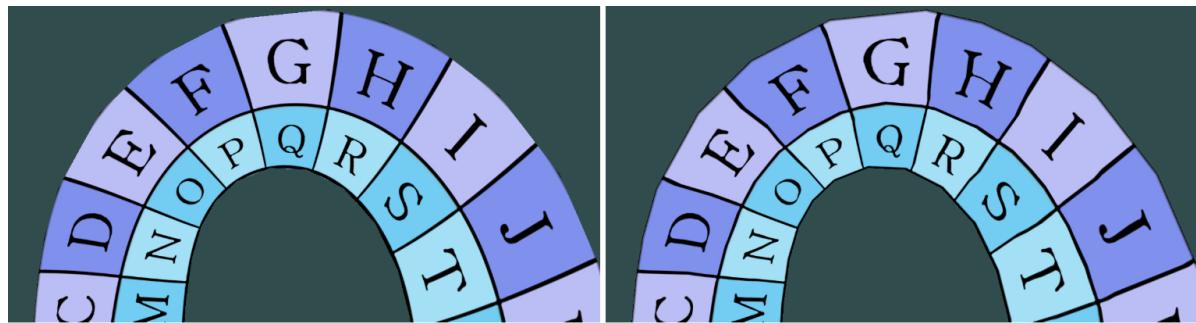
Figure 4: (Top) Möbius map when source and target triangle have reverse orientation. (Bottom) Result of applying BPM in such cases. Each triangle is mapped to an infinite amount of texture.

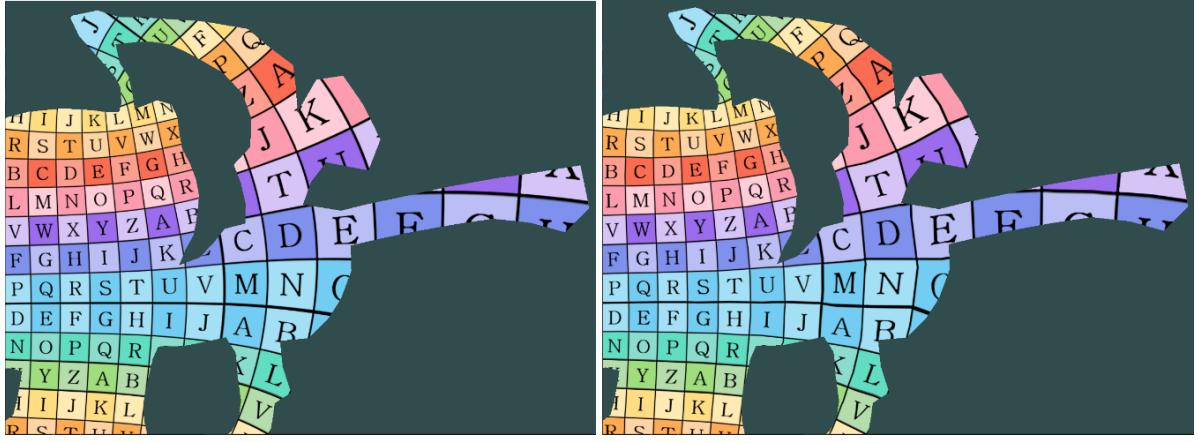
Therefore, when computing the flattening of the mesh triangles, a check is done to see whether the matching texture triangle has a right-hand orientation. If not, an odd permutation is done to fix the orientation.

2 Results

The left column displays the BPM texture interpolation, and the right column the Linear Piecewise interpolation.







3 Speckling

When rendering a model with high distortion near a cone geometry, I've noticed speckling in the BPM interpolation (see Fig. 5).

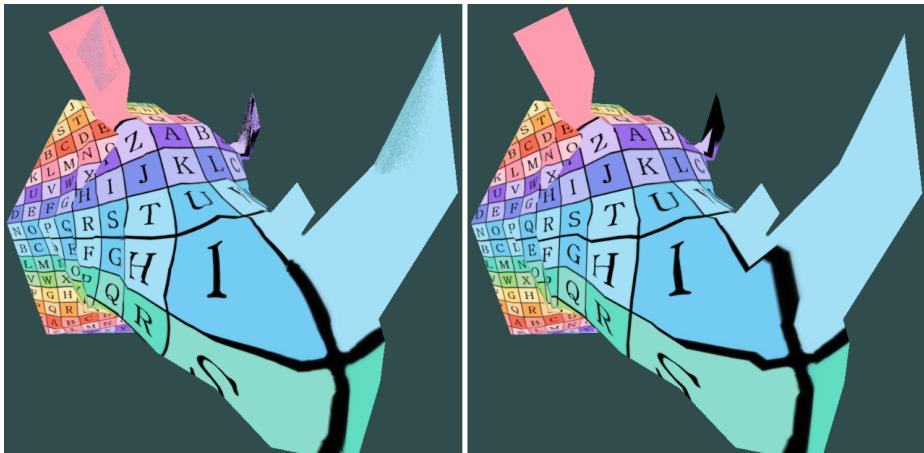


Figure 5: (Left) Speckling in the ears and horn, under BPM interpolation. (Right) Linear interpolation, without speckling.

First, I've tested whether this was due to the cone-like structure. Therefore, I've created a hexagonal-pyramid. As can be seen in Fig. 6, no speckling occurs.

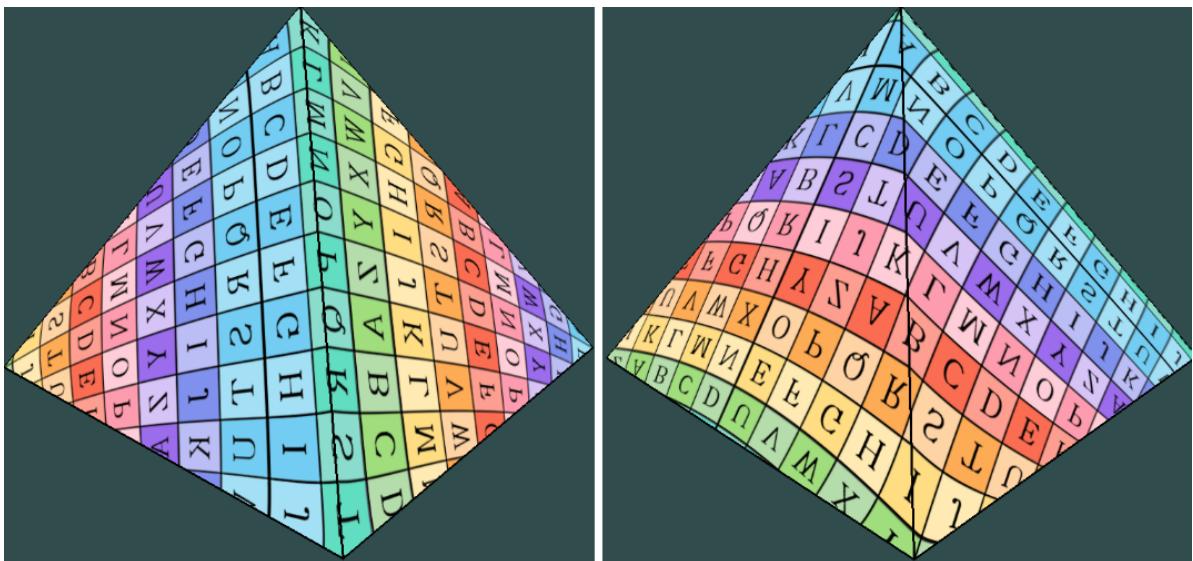


Figure 6: Two views of the hexagonal pyramid under BPM, with wireframe overlayed for visualization. No speckling occurs.

I've noticed that out of the six triangles comprising the rhino's horn, speckling occurs only in four of them. Therefore I took a closer look, and extracted only the horn, preserving the proportion of the original texture coordinates. As can be seen in Fig. 7, there is a significant difference between the mapping of the two triangles (left), and the other triangles.

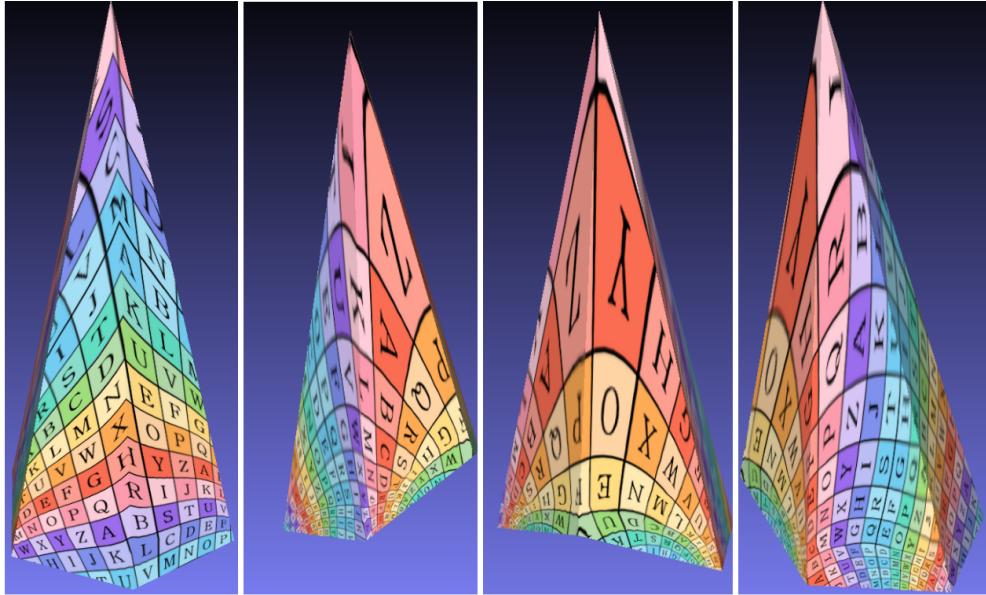


Figure 7: Rendering of the Rhino Horn. (left) the two triangles where no speckling occurs. (mid-right) triangles where rendering occurs.

The following attempts had no effect on speckling:

- Implementing different schemes for rotating the triangles in order to get co-planar triangles.
- Using different texture filtering methods (linear/bilinear/nearest-neighbor/anisotropic).
- Different textures (except for monochromatic colors).

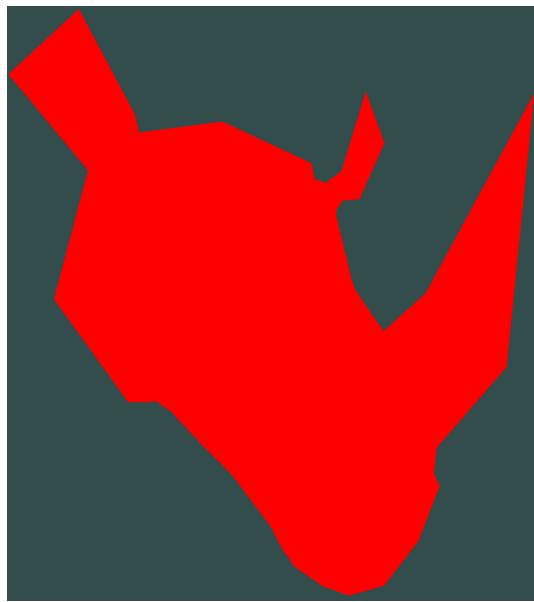


Figure 8: BPM interpolation with a monochromatic texture.

The speckling behavior reminded me of "z-fighting" that occurs in depth testing. Possibly, due to numerical reasons, some pixels are mapped to a different region, thereby displaying a different color.

3.1 Speckling Solution

My next attempt was to interpolate using only a single neighboring mobius map, instead of all three neighbors. Then I've noticed the behavior displayed in Fig. 9:



Figure 9: Infinitear mapping

Apparently, under conditions of large distortion, as happened at those triangles, the interpolation produced Möbius maps of the type that map the inside of triangles to the outside of their targets. This finding has led me to spend several hours implementing different interpolation schemes, including Slerp, Log-Euclidean, and Karcher Mean. During those attempts, I've changed the order of matrix multiplication, and noticed an improvement. This led to test changing the multiplication order, from:

$$M_z = \exp\left(\frac{1}{2} \log(M_u M_t^{-1})\right) M_t = \sqrt{M_u} \sqrt{M_t}$$

to (as correctly shown in the implementation section):

$$M_z = M_t \exp\left(\frac{1}{2} \log(M_t^{-1} M_u)\right) = \sqrt{M_t} \sqrt{M_u}$$

As the Lie group $\text{SL}(2, \mathbb{C})$ is non-abelian, these results differ. Using the second formula produced rendering without speckling. This highlights the importance of preserving properties of transformations when interpolating.

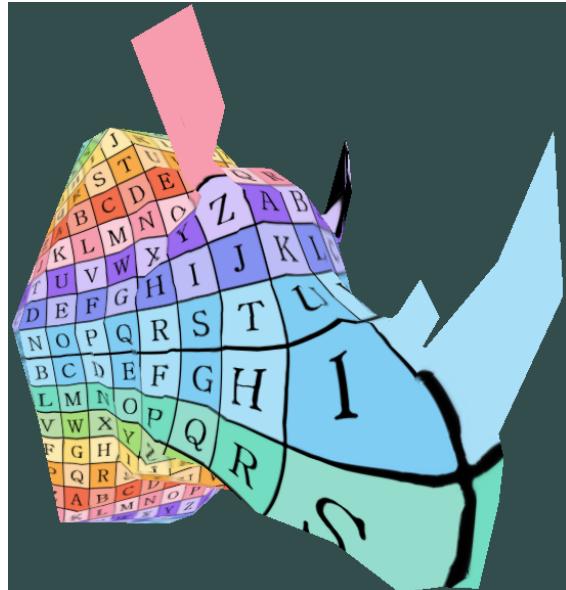


Figure 10: Happy Rhino with no speckles