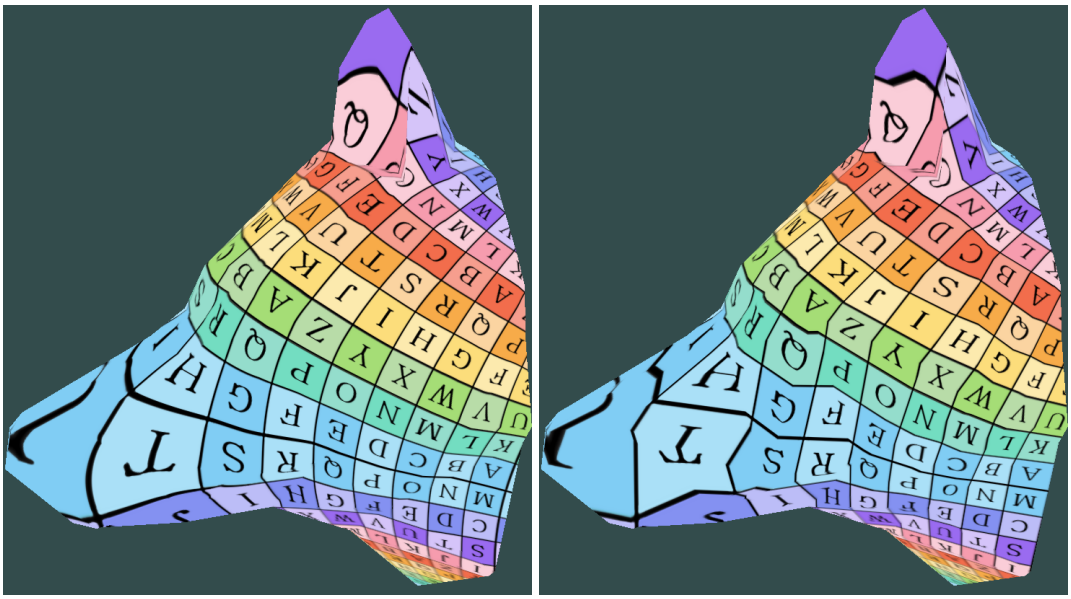
236328 Research Project in Computer Graphics

C++/OpenGL implementation of

BPM: Blended Piecewise Möbius Maps

Rorberg, Vaxman, Ben-Chen



Submitted to: Prof. Miri Ben-Chen

Submitted By: Ehud Gordon

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 $F : \mathcal{V} \to W$. Specifically, an OBJ file, with 'mtllib' and 'usemtl' commands to specify texture W, and with 'vt' coordinates to specify $F : \mathcal{V} \to W$. An example is given in file 'wolf_head.obj' in the github directory.

Output: The code renders the given Mesh using the texture map. The user can choose between Linear-Piecewise and BPM interpolation scheme. For more, 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.

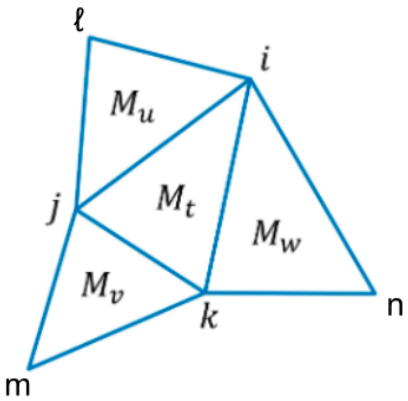


Figure 1: Triangle star notation.

   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 Mobius map:

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

Compute the un-normalized coefficients using:

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

$$b = \det \begin{pmatrix} z_i w_i & z_i & w_i \\ z_j w_j & z_j & w_j \\ z_k w_k & z_k & w_k \end{pmatrix}$$

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

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

and then normalize a, b, c, d by the factor $\sqrt{ad - bc}$. Similarly compute M_u, M_v and M_w.

Transform and rotate only the vertices positions, not the texture coordinates. Then compute the Möbius ratio $\delta_{ut} := M_u M_t^{-1}$. When a neighboring triangle, for example u, does not exist, set δ_{ut} to be the 2×2 zero matrix.

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

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

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

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_{ut}, ℓ_{vt} and ℓ_{wt}. 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 $\ell_{ut}, \ell_{vt}, \ell_{wt}$.

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 visuzlied in Fig. 2

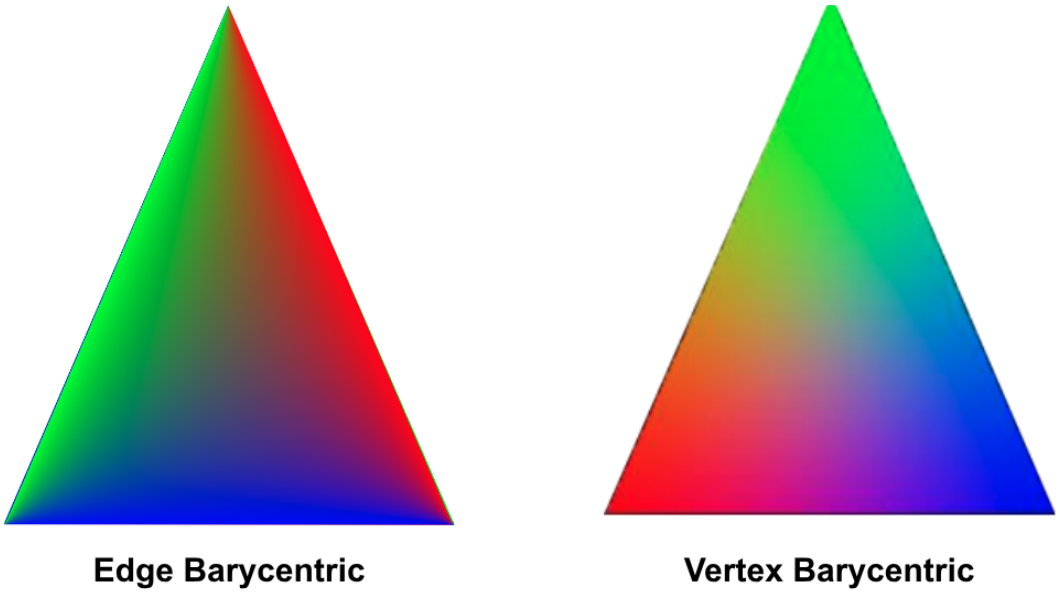


Figure 2: (left) edge barycentric coordinates, as used in the paper. Edges are assigned Red/Green/Blue color. (Right) vertex barycentric coordinates

7. Compute blended log Möbius ratio:

$$\ell_t(z, M) = \gamma_{ij}(z)\ell_{ut} + \gamma_{jk}(z)\ell_{vt} + \gamma_{ki}(z)\ell_{wt}$$

8. Compute the blended Möbius map:

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

9. The interpolated texture coordiantes 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 Mobius Maps	CPU
Compute Log Ratios	CPU
Store Maps on GPU	GPU
Compute Edge Barycentric Coordinates	GPU
Sample Texture	GPU

1.2 Highlights

Numerical Issuses 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, I've used a

naive implementation of the formula:

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

this turned out to be both efficient and sufficient, converging to satisfactory precision using only the three leading terms.

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 meshses, the GPU finished in around 2 seconds, comapred 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 $\boldsymbol{v}_2$, the normal to triangle jiv' (shown in green) will point in the positive z-direction. Let's denote the angle from $\boldsymbol{v}_2$ to the positive z-axis by θ. Denote $\boldsymbol{e}_1 = \frac{v_j - v_i}{\|v_j - v_i\|}$, and define $\boldsymbol{v}_1 := \boldsymbol{e}_1 \times \widehat{\boldsymbol{z}}$. Then:

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

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

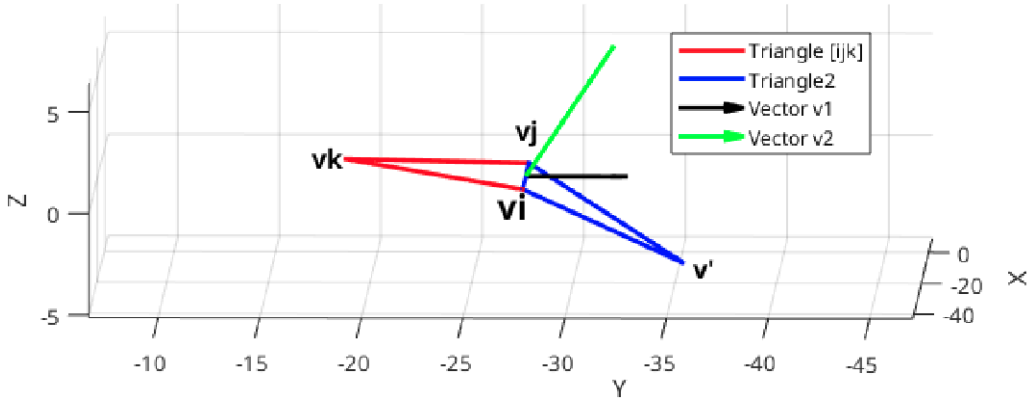


Figure 3: Triangle ijk (in red) and neighbor triangle jiv' (in blue) after transformation to t's plane. The normal to jiv' is shown in green. Triangle jiv' is rotated so it'll be co-planar to ijk.

While visually, the results look almost similar, the QC error differs, especially the max values. I've run it over four models. In one it had no effect, in two it improved results, and in one model it worsened the results.

Model Name	Wolf	Mannequin	Rhino
Previous theta	max=153, avg_qc=1.1851	max=293,955, avg_qc=1.3239	max=241,756, avg_qc=2.0283
New theta	max=12, avg_qc=1.1667	max=2,005, avg_qc=1.095	max=453,856, avg_qc=2.9455

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. In addition, the triangles of the texture coordinates have an orientation. If they don't have right-handed orientation, then the Möbius map between them will map points inside the source triangle to points *outside* the target triangle. This is shown in Fig. 4.

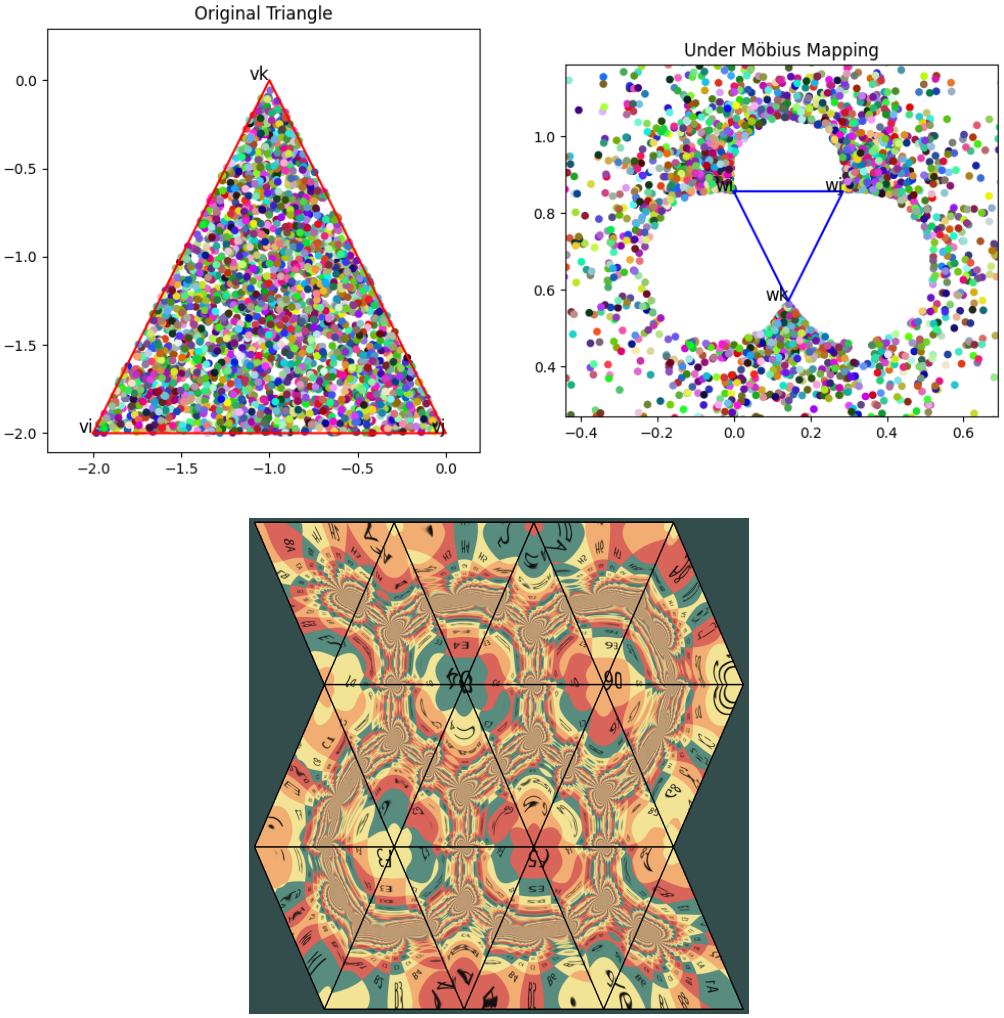
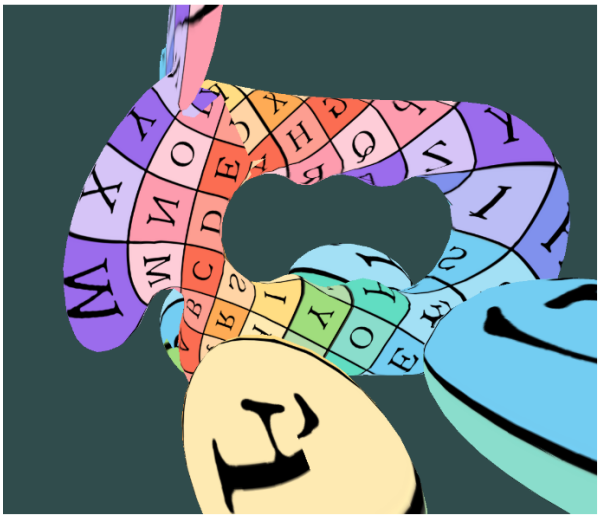
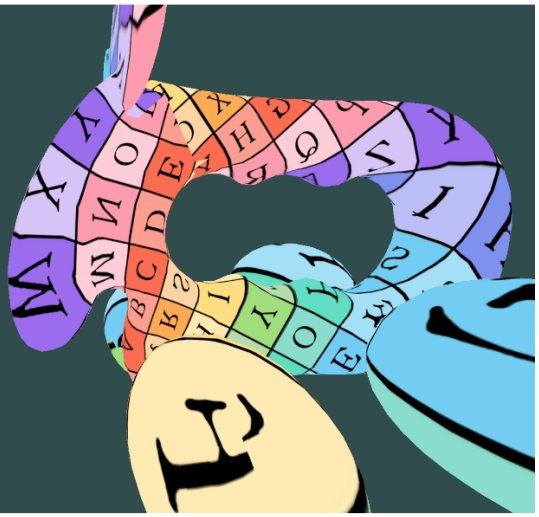
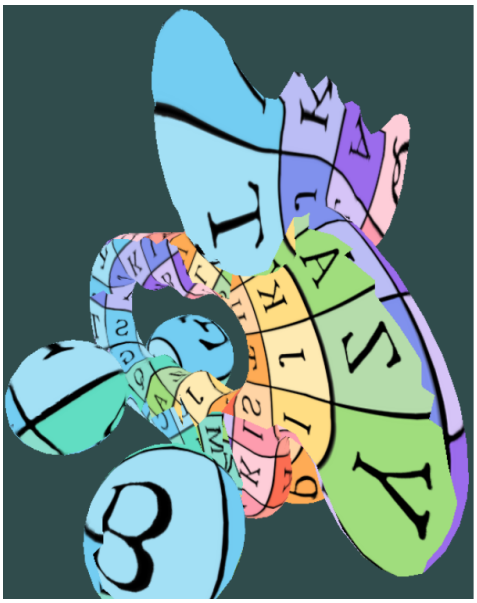
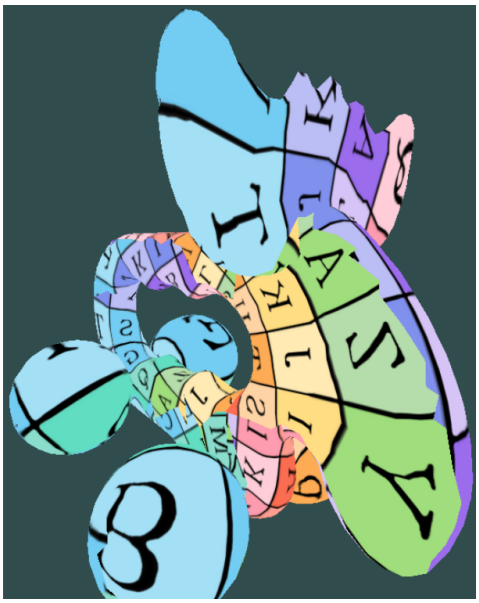
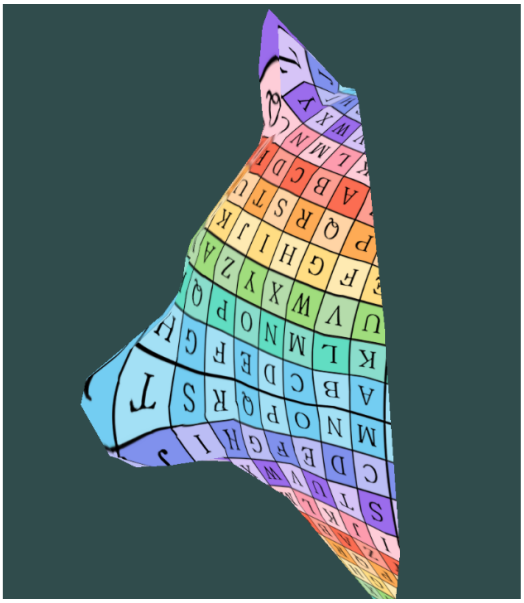
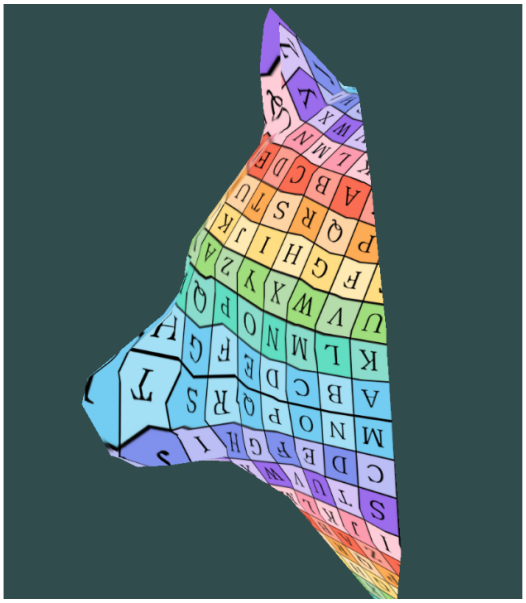
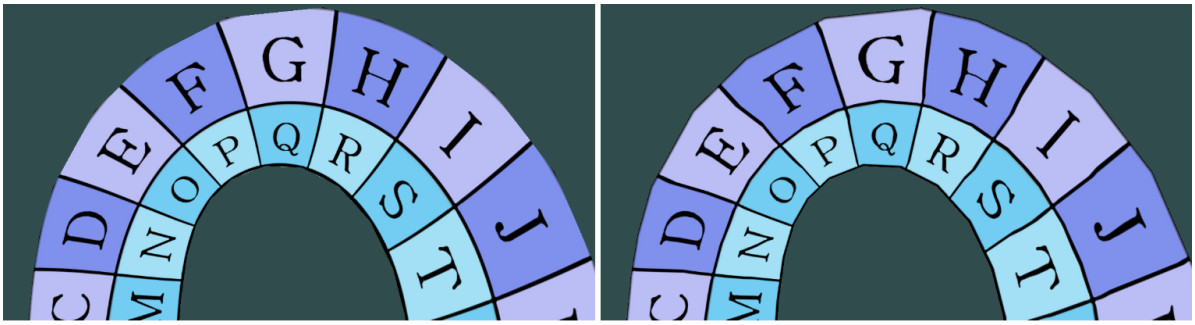


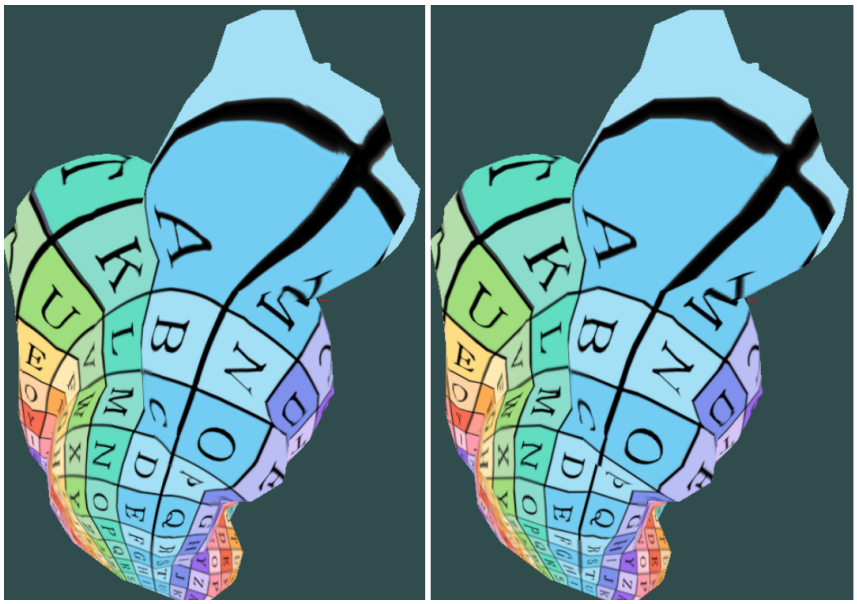
Figure 4: (Top) Möbius map when source and target triangle have reverse orientation. (Bottom) Result of applying BPM in such cases. Each triangle is mapped to an infinite amount of texture.

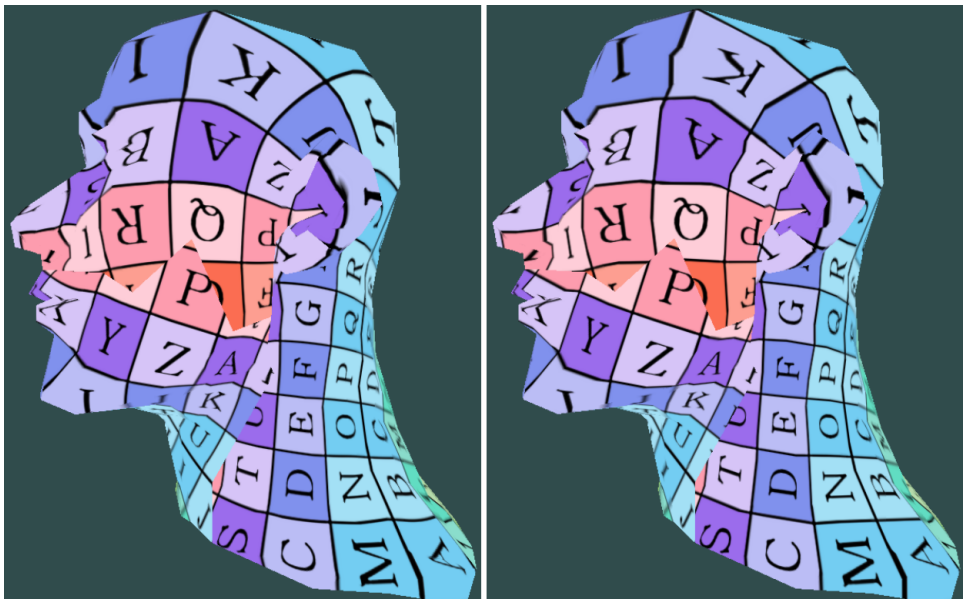
2 Results

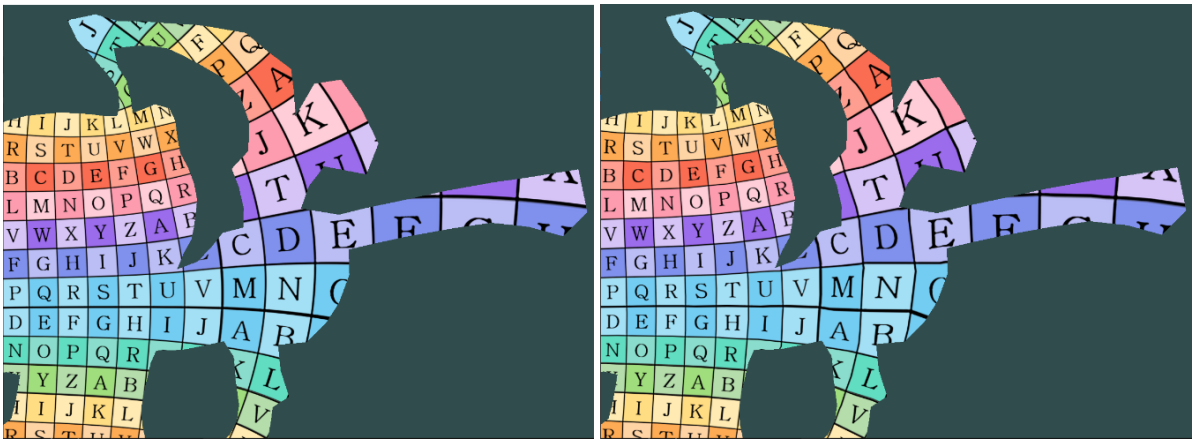
The left column displays the BPM texture interpolation, and the right column the Linear Piecewise interpolation.









2.1 Challenges

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

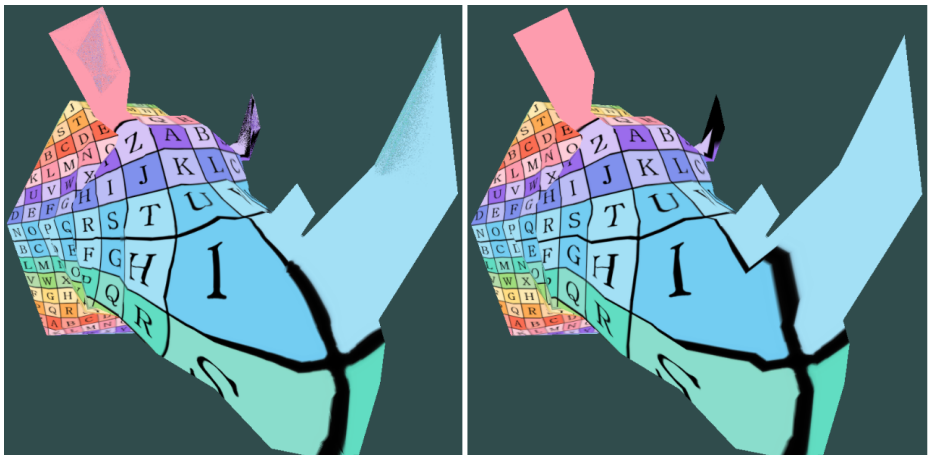


Figure 5: Speckling Issue

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.

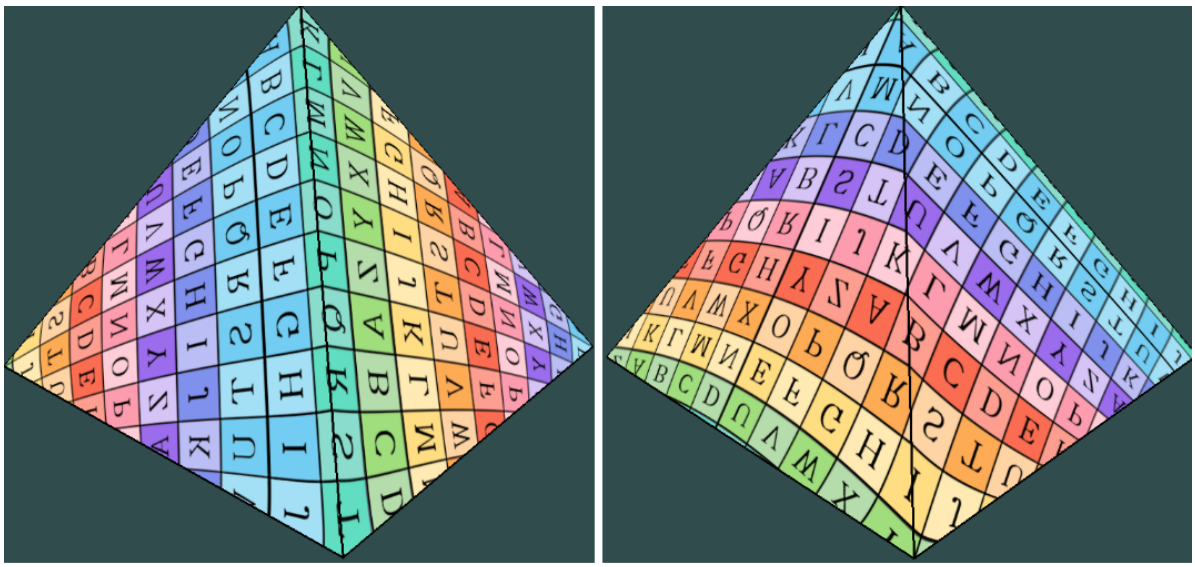


Figure 6: Two views of the hexagonal pyramid under BPM, with wireframe overlayed for visualization. No speckling occurs.

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 propoprtion 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 triagnles.

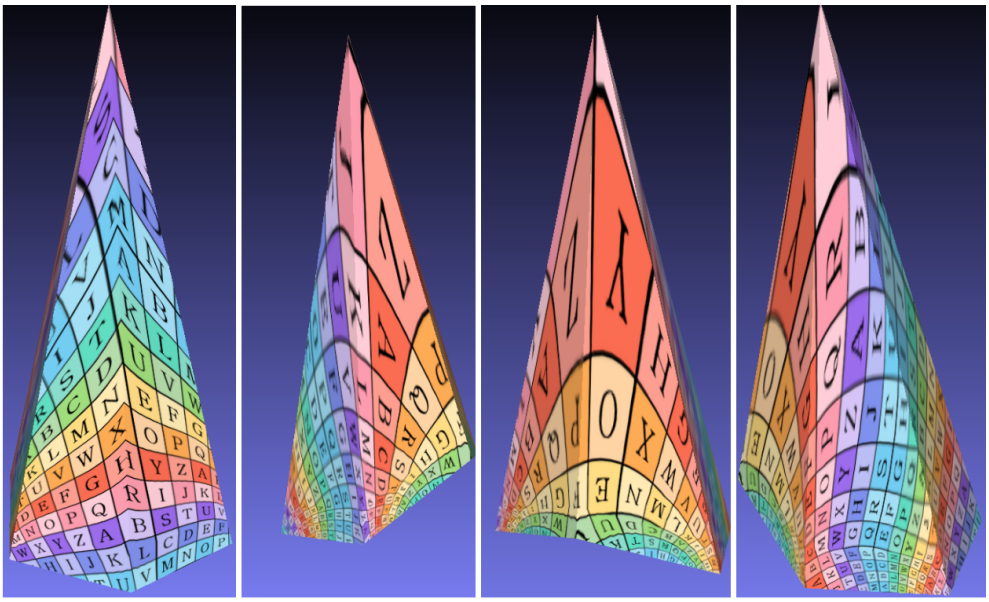


Figure 7: Rendering of the Rhino Horn. (left) the two triangles where no speckling occurs. (mid-right) triangles where rendering occurs.

The following attempts had no effect on speckling:

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

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. Indeed, when using a monochromatic texture, no speckling occurs, as shown in Fig. 8.

Figure 8: BPM interpolation with a monochromatic texture.

An interesting attempt would be to try to render this model using the CPU, as it implements different floating-point behavior than the GPU, in order to pin-point the cause of the effect.