Real-time visualization of Moebius transformations in space using Quaternionic-Bezier approach

Vytautas Karpavicius
Faculty of Mathematics and Informatics
Vilnius University, Lithuania
vytautas.karpavicius@mif.vu.lt

Rimvydas Krasauskas
Faculty of Mathematics and Informatics
Vilnius University, Lithuania
rimvydas.krasauskas@mif.vu.lt

ABSTRACT

Moebius transformations in space are much more sophisticated than the classical case on the plane, which has been well studied. We present a WebGL approach for visualization of Moebius transformations in 3-space by animating deformations of geometric objects composed of patches parametrized by Quaternionic-Bezier formulas. The idea is to represent Moebius transformations in a quaternionic form as well, and to use GPU shaders for transforming control points, weights, and normals, then seamlessly stitching patches with different levels of detail, and computing points on every patch. Finally, we demonstrate the main classes of Moebius transformations in space on several 3D objects including primitive shapes, Dupin cyclide patchworks, and Utah Teapot.

Keywords

Moebius transformation, Quaternionions, Quaternionic-Bezier, Visualization, GPU, WebGL, Shaders

1. INTRODUCTION

This paper was inspired by a wonderful video clip "Moebius Transformations Revealed" by Arnold and Rognes, which is available in various formats online [Arn09] (the theory behind is described in the paper [Arn08]) and some recent Quaternionic-Bezier surface constructions [Kra11].

Since our goal is to visualize Moebius transformations in 3-space, we cannot apply the aforementioned Arnold-Rognes approach directly. Indeed, for that one needs to show 3-sphere in 4-space. So we switched to the idea of animating deformations of familiar geometric objects in space using real time graphics. The concept of a Quaternionic-Bezier surface is the other important aspect of the proposed visualization method. Moebius transformations of the plane are usually identified with fractional-linear functions of complex variable, when complex numbers are treated as points on that plane (see e.g. [Arn08]). In the 3D case one can change complex numbers by quaternions and derive similar formalism based on 2×2 quaternionic matrices [Bis10].

This approach in combination with ideas of [Gwy12] allows us to derive short original proofs of Theorems 1 and 2, which describe the main properties of Moebius transformations in 3-space.

Finally, we choose WebGL framework for implementation of our visualization method, since this modern technology enables us to deliver interactive real time 3D graphics in web environment.

The paper is organized as follows. We review definitions of quaternions, Quaternionic-Bezier formulas, and we describe the main properties of 3D Moebius transformations in Section 2. Section 3 is devoted to the visualization framework using WebGL. Examples of screenshots, showing the behavior of several 3D objects under Moebius transformations, are presented in Section 4. Finally, the conclusions are derived in Section 5.

2. USING QUATERNIONS

Algebra of Quaternions

We will use the algebra of quaternions **H** with the standard basis $\mathbf{1}, \boldsymbol{i}, \boldsymbol{j}, \boldsymbol{k}$ and product rules:

$$\boldsymbol{i}^2 = \boldsymbol{j}^2 = \boldsymbol{k}^2 = -\mathbf{1}, \boldsymbol{ij} = \boldsymbol{k}, \boldsymbol{jk} = \boldsymbol{i}, \boldsymbol{ki} = \boldsymbol{j}.$$

Any quaternion $q = r\mathbf{1} + x\boldsymbol{i} + y\boldsymbol{j} + z\boldsymbol{k}$ can be decomposed in its real part $\mathrm{Re}(q) = r$ and its imaginary (vector) part $\mathrm{Im}(q) = x\boldsymbol{i} + y\boldsymbol{j} + z\boldsymbol{k} = \boldsymbol{v}$,

$$q = \mathrm{Re}(q) + \mathrm{Im}(q) = r + \boldsymbol{v}.$$

Reals **R** and space $\mathbf{R}^3$ are identified with subsets in **H**: $\{q \in \mathbf{H}: \mathrm{Im}(q) = 0\}$ and $\mathbf{H}_0 = \{q \in \mathbf{H}: \mathrm{Re}(q) = 0\}$.

Other useful notations for quaternions $q = r + \boldsymbol{v}$:

- conjugate $\overline{q} = r - \boldsymbol{v}$
- norm (length) $|q| = \sqrt{r^2 + \boldsymbol{v} \cdot \boldsymbol{v}}$
- inverse (if $q \neq 0$) $q^{-1} = \overline{q}\,/|q|^2$

If $|q| = 1$ then q has a *trigonometric form*

$$q = \cos\alpha + \sin\alpha\,\boldsymbol{u}, \qquad u \in \mathbf{H}_0, \qquad |\boldsymbol{u}| = 1,$$

and a *square root* of q is defined

$$\sqrt{q} = \cos\frac{\alpha}{2} + \sin\frac{\alpha}{2}\boldsymbol{u}. \qquad (1)$$

Moebius Transformations in Space

Moebius (M) transformations in space are defined as compositions of inversions in space with respect to spheres. Alternatively, M-transformations can be generated by 4 *elementary transformations* in $\mathbf{R}^3 = \mathbf{H_0}$:

1. translation $x \mapsto x + a$, $a \in \mathbf{H}_0$,
2. scaling $x \mapsto \lambda x$, $\lambda \in \mathbf{R}$,
3. rotation about axis $\boldsymbol{u}$ by angle 2α: $x \mapsto qxq^{-1}$, $q = \cos\alpha + \sin\alpha\,\boldsymbol{u}$,
4. inversion with a center in the origin and unit radius: $x \mapsto -x^{-1}$.

Note that the first 3 types of elementary transformations generate *Euclidean similarities*.

Arbitrary M-transformations can be represented by fractional-linear functions

$$F(x) = (ax + b)(cx + d)^{-1} \qquad (2)$$

of quaternion variable x or by 2×2 matrices

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}, \qquad a, b, c, d \in \mathbf{H},$$

using notations $F(x) = A * x$.

It is easy to check that usual multiplication of matrices corresponds to the composition of fractional-linear functions. The formula (2) defines transformation of all quaternions **H**. The subgroup of all 2×2 matrices that define M-transformations of $\mathbf{H}_0$ is characterized in [Bis10], Theorem 11.1.

Elementary M-transformations correspond to the following matrices (here $\in \mathbf{H}_0$, $\lambda \in \mathbf{R}$, $|q| = 1$):

$$\mathrm{Tr}(a) = \begin{pmatrix} 1 & a \\ 0 & 1 \end{pmatrix}, \qquad \mathrm{Rot(q)} = \begin{pmatrix} q & 0 \\ 0 & q \end{pmatrix},$$

$$\mathrm{Sc}(\lambda) = \begin{pmatrix} \lambda & 0 \\ 0 & 1 \end{pmatrix}, \qquad \mathrm{Inv} = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}.$$

Two theorems below are formulated following similar results in [Gwy12] about M-transformations in 4-space, but our proofs are different.

Theorem 1. For any given three distinct points $a_0, a_1, a_2 \in \mathbf{H}_0$ and another triple of distinct points $b_0, b_1, b_2 \in \mathbf{H}_0$ there exists an M-transformation F, such that $F(a_0) = b_0$, $F(a_1) = b_1$, $F(a_2) = b_2$.

If both triples of points are not collinear, F maps a plane going through points a_0, a_1, a_2 to another plane.

Sketch of the proof. Similar to the 2D case it will be convenient to extend our 3-space by adding the infinite point ∞. Then we observe that M-transformations that preserve ∞ are Euclidean similarities. Define transformation

$$H(a_0, a_1) = \mathrm{Tr}(-a_1') * \mathrm{Inv} * \mathrm{Tr}(-a_0),$$
$$a_1' = \mathrm{Inv} * \mathrm{Tr}(-a_0) * a_1,$$

that maps the triple (a_0, a_1, a_2) to $(\infty, 0, a_2')$. Similarly, $H(b_0, b_1)$ maps (b_0, b_1, b_2) to $(\infty, 0, b_2')$, and it remains to find an appropriate transformation $R(a_2', b_2')$ from $(\infty, 0, a_2')$ to $(\infty, 0, b_2')$. This can be a composition of rotation and scaling (see (1))

$$R(\mathrm{a}, b) = \mathrm{Sc}(|q|) * \mathrm{Rot}\left(\sqrt{q\,|q|^{-1}}\right), \qquad q = ba^{-1}.$$

Finally, the composition

$$F = H(b_0, b_1)^{-1} * R(a_2', b_2') * H(a_0, a_1)$$

will map (a_0, a_1, a_2) to (b_0, b_1, b_2).

Theorem 2. Let $a_0, a_1, a_2 \in \mathbf{H}_0$ be four distinct points, and let $b_0, b_1, b_2 \in \mathbf{H}_0$, be three distinct points. Then the set of the images $F(a_3)$ for all M-transformations F, such that $F(a_0) = b_0$, $F(a_1) = b_1$, $F(a_2) = b_2$ is either a single point (if points a_0, a_1, a_2, a_3 are co-circular), a line or a circle.

Sketch of the proof. According to the Theorem 1 one can suppose that $(a_0, a_1, a_2) = (b_0, b_1, b_2) = (\infty, 0, \boldsymbol{i})$. Then 3 points $\infty, 0, \boldsymbol{i}$ are fixed and we are looking for Euclidean similarity with 2 fixed points $0, \boldsymbol{i}$. There are two cases: $0, \boldsymbol{i}, a_3$ are collinear or not. In the first case all 4 points $\infty, 0, \boldsymbol{i}, a_3$ are on a line (i.e. points a_0, a_1, a_2, a_3 are co-circular) and we cannot move a_3 to any other position. In the second case the only possibility for a_3 is to rotate around the axis going through the both $0, \boldsymbol{i}$. Therefore, $F(a_3)$ can be any point of the particular circle (or line in general).

Quaternionic-Bezier (QB) Formulas

Here we remind some results publicated in [Kra11].

Circular Arc

Let p_0 and p_1 be two endpoints of a circular arc C in $\mathbf{R}^3 = \mathbf{H_0}$, and let q be some interior point on C. *QB-curve* of degree 1 defined by the formula $C(t)$

$$= (p_0 w_0(1-t) + p_1 w_1 t)(w_0(1-t) + w_1 t)^{-1}$$

with quaternionic control points p_0, p_1 and weights

$$w_0 = (q - p_0)^{-1}, \qquad w_1 = (p_1 - q)^{-1},$$

defines a rational parametrization $[0,1] \to \boldsymbol{H_0}$ of C. A tangent vector v_0 at the endpoint p_0 is

$$v_0 = (p_1 - p_0) w_1 w_0^{-1}. \qquad (3)$$

Bilinear QB-surface patch

Let us define a *bilinear QB-surface patch* by the usual rational Bezier formula but with quaternionic control points p_{ij} and weights w_{ij} $(0 \le s, t \le 1)$:

$$P(s,t) = \left(\sum_{i=0,1}\sum_{j=0,1} p_{ij}\, w_{ij} s_i t_j\right)\left(\sum_{i=0,1}\sum_{j=0,1} w_{ij} s_i t_j\right)^{-1},$$

where $s_0 = 1 - s$, $s_1 = s$, $t_0 = 1 - t$, $t_1 = t$.

Here we consider just two important cases: Dupin cyclide and Darboux cyclide (cf. [Kra11], [Pot12]).

Let p_0, p_1, p_2, p_3 be any 4 points on a circle in $\mathbf{R}^3 = \mathbf{H_0}$, and let $\boldsymbol{v}_1, \boldsymbol{v}_2$ be two orthonormal vectors, i.e. $|\boldsymbol{v}_1| = |\boldsymbol{v}_2| = 1$ and $\boldsymbol{v}_1 \perp \boldsymbol{v}_2$. Then there is a unique *principal Dupin cyclide patch* with corners in these points, and bounded by circular arcs with tangent vectors $\boldsymbol{v}_1, \boldsymbol{v}_2$ at the corner p_0, which can be rationally parametrized by the formula $P(s,t)$, where double index notations are changed to single ones

$$\{00, 01, 10, 11\} \longrightarrow \{0, 1, 2, 3\}.$$

Here weights w_i are computed by formulas:

$w_0 = 1, \quad w_1 = q_{10}v_1, \quad w_2 = q_{10}v_1,$

$w_3 = \delta_{12}\delta_{03}^{-1} q_{31} w_1 q_{20} w_2,$

where

$$\delta_{ij} = |p_i - p_j| \in \mathbf{R}, \qquad q_{ij} = (p_i - p_j)\delta_{ij}^{-1} \in \mathbf{H}_0.$$

A *Darboux cyclide patch* is the most general case of a bilinear QB-surface [Kra11]. For our purposes it will be enough to consider cases with unit weights and their images under M-transformations, when the correct weights will be automatically computed as explained below.

Moebius invariance

QB-curves and QB-surfaces are Moebius invariant: their image under a certain M-transformation has the same formula, where only control points and weights need to be changed $p \mapsto p', w \mapsto w'$:

$$p' = F(p) = (ax + b)(cx + d)^{-1}, \tag{4}$$

$$w' = FW(p, w) = (cp + d)\mathrm{w}. \tag{5}$$

3. WEBGL FRAMEWORK

Here we describe a framework which enables us to visualize Moebius transformations in the web environment. Transforming, computing and rendering QB surfaces are very computionally intensive tasks. While it is possible to do it all in javascript, it would be terribly slow and would not allow us to represent real-time animations of such transformations. This framework offloads most of the computation tasks to the GPU which is well suited for such massively parallelizable computations [Bro13].

We have chosen to deliver visualization content in web environment so that it would be available to wider audience with ease of access.

GPU is reached through an API exposed by WebGL technology [Khr12], which is based on OpenGL ES 2.0 [Khr10]. Even if it does not allow to use the most modern GPU features, it is possible to achieve desired result solely through WebGL API.

An abstract outline of the framework algorithm:

1. Initialization
   1.1. Loading a model
   1.2. Preparing buffers
2. Computing surface points
   2.1. Transforming position, weights, normals
   2.2. Estimating the level of details for patches
   2.3. Mapping patches to batches
   2.4. Computing batches
3. Rendering surface points

Surface points are recomputed every time the surface is transformed. Surface gets rendered when the camera has moved or user changes any of the display options (e.g. switches to wireframe mode).

Loading a Model

We start by loading a model from .obj file. The file format is adjusted to accommodate weights which are required by QB patches. Each weight is defined by a line starting with w and following 4 real numbers separated by spaces – representing 4 components of the quaternion in the order: x, y, z, r. Then each face references their weights by the index in adjusted face vertex definition format: position/uv/normal/weight.

Three textures are created to fit positions, normals and weights. Each face stores their vertex attributes separately in these textures (vertices are no longer shared among faces). Face vertices correspond to 4 subsequent pixels in textures. After this data has been uploaded to GPU, each face vertex object in javascript gets a reference to their respective data in textures, that is, a fetch coordinate is stored.

After loading the model, we search for adjacent faces. This is required later when patches of different LOD are stitched together. Each face gets references to their adjacent left, right, top and bottom faces.

Preparing Buffers

Buffers which do not depend on transformation state get precomputed and uploaded to GPU in advance. For every LOD level 7 buffers need to be precomputed. 4 of those will be used during the computation stage, other 3 – during the rendering stage. Those buffers will be described in surface point computation and rendering sections respectively.

Computing Surface Points

3.1.1 Render to Texture

To use GPU for arbitrary computation instead of just rendering images, we take advantage of the technique called render-to-texture. Instead of drawing to the screen, custom texture is attached to the framebuffer and rendered image gets stored in it. The data from the framebuffer then can be read back to the main memory and processed with the CPU. Alternatively the texture containing rendering results can be used in subsequent rendering passes as a data source.

A quad is drawn to cover entire texture. Quad vertices have attributes (0;0), (0;1), (1;0) and (1;1) which get interpolated and passed to the fragment shader. This interpolated parameter lets the fragment shader know which fragment it is working with and proceed with computation accordingly.

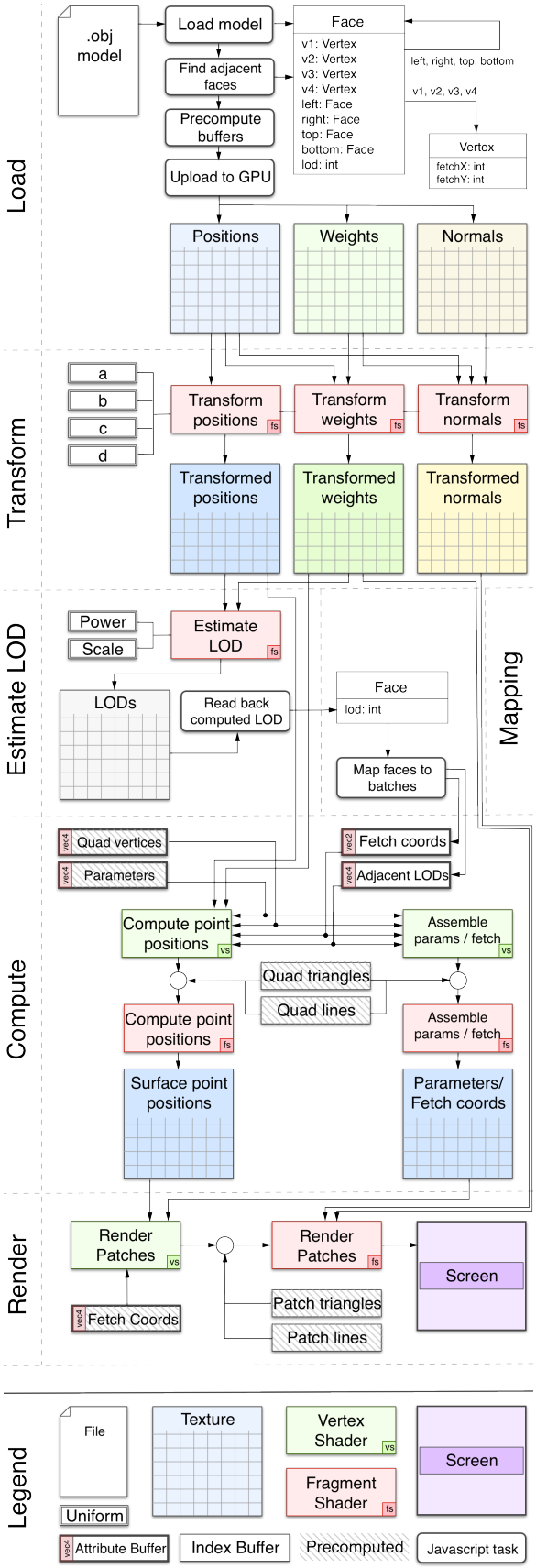

Figure 1. Diagram showing data flow across all stages in the framework.

3.1.2 Applying transformations

All Mobius transformations can be reduced to 4 quaternion coefficients a, b, c, d (see (2)). Those coefficients are then set as vec4 uniform parameters in 3 fragment shaders which transform positions, weights and normals of control points.

Formulas (4), (5) are used for transforming control points and weights. The transformation of normal n at point p_0 can be achieved by transforming to two points: p_0 and $p_1 = p_0 + n$. In general Moebius transformation of a line segment $[p_0, p_1]$ will be a circular arc. Therefore, to get the transformed normal we compute a tangent to that arc according to formulas (3)-(5):

$$n' = \big(F(p_1) - F(p_0)\big) FW(p_1, 1) FW(p_0, 1)^{-1}.$$

Estimating Level of Detail

Moebius transformation can significantly deform the surface of the model. Our framework implements a feature to estimate dynamically the level of detail (LOD) for each patch. The more curved patches will get higher level of detail, the more surface points will be evaluated.

The level of detail for each patch is computed in a separate shader by taking transformed positions and weights of the patch. The results are then read back to the main memory so that javascript code could map patches to appropriate buffers.

One can choose different ways to estimate LOD. Note that it is not crucial to have mathematically exact formula to find the curvature of the patch. Our implementation takes into account two measures: the size of the patch L and the distance H between middle points of transformed patch f and flat plane on patch control points c.

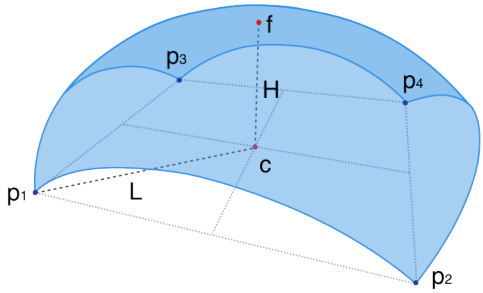

Figure 2. Measures used in estimating LOD.

The level of detail is computed by taking ratio H/L, raising it to the power p and then scaling by s. Constants p and s are passed to the shader as uniform parameters. They allow a user to fine tune surface LOD.

$$\mathrm{LOD} = s\left(\frac{H}{L}\right)^p \qquad (4)$$

Mapping Patches to Batches

The patches of the surface are computed and rendered in groups. A bunch of patches having the same level of detail gets layed out in the texture next to each other. Such texture is called a batch. In other words, a buffering mechanism for data sent to the GPU is created in order to achieve better performance.

All batches are of the size 256×256. It gives us a total of 65536 (or 2^{16}) pixels in the texture. Later, when the batch is rendered, an index buffer of triangles (or lines) is used, where each index is of type gl.UNSIGNED_SHORT. Therefore a whole batch gets rendered in one draw call. There is no point to make bigger batches since we will not be able to render them at once anyway.

Minimum level of details is 2×2. It takes only original control points without interpolating any additional internal points. One such batch can hold up to 16384 patches. Maximum available LOD – 256×256. A patch with this LOD takes up the whole batch. So there are 8 different levels of details: 2, 4, 8, 16, 32, 64, 128 and 256.

Every time the model is transformed, we need to remap patches since their LOD may have changed. It is done by storing fetch coordinates of face vertices to the attribute buffer. The shader that is used to compute the batch will use these coordinates to fetch transformed positions and weights from textures that had got computed in the transform stage. At this point we also prepare LODs buffer of adjacent patches. Each vertex gets vec4 attribute where components x, y, z, w store LODs of bottom, right, top and left adjacent patches. Finally, we increment the counter that stores the amount of patches this batch holds. If all patches of the same LOD do not fit in one batch, additional batches are created.

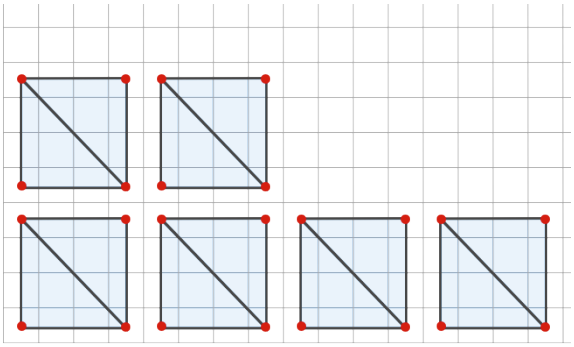

Figure 3. Example of 4×4 patches (quads) mapped to the batch texture.

Computing Batches

3.1.3 Batch buffers

A batch gets computed by drawing a bunch of quads to the texture. Each quad represents a patch. In order to draw those quads we need 6 buffers. 2 of them are prepared during the mapping stage – a buffer of fetch coordinates and a buffer of adjacent LODs. The other 4 are precomputed in advance. Those include: quad vertex buffer, parameter buffer, triangle index buffer and lines index buffer.

The quad vertex buffer contains coordinates(x,y) in batch space. That is, they are integers from zero which refer to the pixels of the texture. For LOD=4, this buffer would look like: (0,0), (3,0), (0,3), (3,3), (4,0), (7,0), (4,3), (4,7)…

In the vertex shader these coordinates are transformed to homogeneous space using this function:

$$f(\vec{x}) = \begin{pmatrix}-1\\-1\end{pmatrix} + \frac{1}{256}\left(2\vec{x} + \begin{pmatrix}1\\1\end{pmatrix}\right) \qquad (5)$$

The scaling factor 1/256 comes from the fact that we use 256 sized batches. As fragments are evaluated at their centers, we also need add vector (1,1). This shifts quads so that their cornerswill be at the center of fragments.

The previous function can also be written as matrix, transforming quad vertices to homogeneous space. The quad vertex vector also needs to be expanded to 4 components vector: (x,y) → (x,y,0,1).

$$\begin{bmatrix} 1/128 & 0 & 0 & -255/256 \\ 0 & 1/128 & 0 & -255/256 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The parameter buffer holds a repeating sequence of (0,0), (1,0), (0,1), (1,1) vectors that will act as s and t parameters in the equation of the bilinear QB-surface $P(s,t)$. These vectors are passed to fragment shader, where they will get interpolated for every fragment.

Finally, we have index buffers for drawing triangles and lines. The usage of triangles index buffer is straightforward – to connect vertices into primitives (quads). However, because of rasterization rules not all fragments get covered by triangles. A fragment is covered by a triangle if the center of that fragment is inside the triangle. If a fragment center happens to be exactly on triangle edge – the top-left rule is applied [Mic12]. It states that the fragment is rasterized if it is on the left edge (or the top, in case the edge is horizontal).

The bottom and right edges of the quad do not get rasterized. For this reason we also need lines buffer which is used for drawing edge lines of the patch.

3.1.4 Evaluating surface points

The evaluation of surface points is performed in the fragment shader. Bilinear interpolation of 4 control points is achieved using 3 `mix` instructions. Two of them interpolates in x axis (between 0-1 and 2-3 points). The third then interpolates in y axis between the results of the previous. This interpolation needs to

be performed for $p_i w_i$ and for w_i. These will give us the numerator and denominator of the bilinear QB-surface $P(s,t)$. The excerpt of fragment shader is given below:

```
vec4 a = mix(vPW[0], vPW[1], s);
vec4 b = mix(vPW[2], vPW[3], s);
vec4 nomin = mix(a, b, t);
a = mix(vW[0], vW[1], s);
b = mix(vW[2], vW[3], s);
vec4 denomin = mix(a, b, t);
gl_FragColor = qMult(nomin, qInv(denomin));
```

Note that premultiplied $p_i w_i$ and w_i of each control point are passed from vertex shader as varying parameters. Even though varying parameters are interpolated, we do not need that. They are used just as a way to pass data from the vertex shader. To cancel interpolation, each vertex in the quad has to compute the same values for the varyings.

The vertex shader receives fetch coordinates in an attribute buffer that was assembled during patch mapping. Using those coordinates, it can fetch transformed positions and weights of control points. It is not enough for a vertex to fetch its own position and weight. All 4 control points need to be fetched so that they could be passed to the fragment shader canceling unwanted interpolation.

Consider that current quad vertex has fetch coordinate (x,y). Then all 4 fetch coordinates $f_i, i = 0,1,2,3$, can be found using:

$$f_i = \begin{pmatrix} x - mod(x,4) + i \\ y \end{pmatrix} \qquad (6)$$

Note that if we were working with triangle patches (instead of quad patches), this varying-cancelation mechanism would not be required. Every vertex would simply fetch its own attributes and pass them through varyings to the fragment shader where they would be already interpolated. However, the interpolation that occurs during rasterization, works only for triangles, that is why we need our own interpolation system for quads.

3.1.5 Stitching patches of different LODs

Dynamically computing patches of different level of detail allows us to render them with enough accuracy while keeping performance required for real-time animations. Due to this, patches with different LOD do not align perfectly and create gaps.

We have established a patch stitching mechanism which alters how border points of the patch surface are evaluated. The idea is to supply every patch with information about LOD of neighbouring patches. In this way the patch checks if neighbouring patch has lower LOD, and if this is the case, that means the points of the edge are snapped to the nearest points of the neighbour patch.

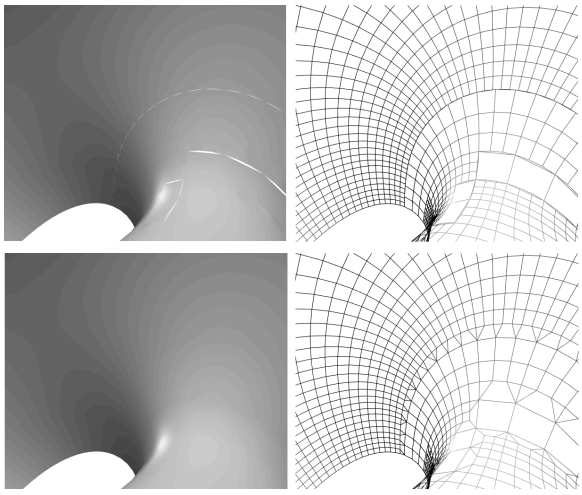

Figure 4. Surface without patch stiching (upper) and surface with patch sitching (lower).

Given that surface point evaluation coordinate (across X axis) is x and n is required LOD for the edge of this patch, evaluation coordinate can be adjusted using the following formula

$$x' = \text{round}\big(x(n-1)\big)/(n-1). \qquad (7)$$

This correction has to be applied to the bottom $(y = 0)$ and top $(y = 1)$ edges. Similarly for the left $(x = 0)$ and right $(x = 1)$ edges the same correction is applied, except for different axis – Y.

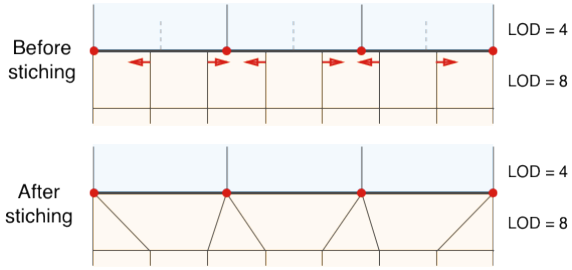

Figure 5. Patch edge vertices snapped to lower level of detail neighbouring patch.

Shader program does not need to know what the LOD of current patch is. It needs only 4 LODs for patch edges. This information is collected in javascript and passed to the shader as an attribute buffer.

Rendering Batches

Batches are rendered by fetching vertex positions from textures computed in the last stage. Fetch buffers for every different LOD are precomputed in advance. They contain (x,y) coordinates of vertices that make up patches. Two index buffers are also prepared for every different LOD batch. The lines index buffer is used for displaying a model in wireframe mode. The triangle index buffer is used for a solid mode. Alternatively, a user can choose to display only surface points. No index buffer is used then.

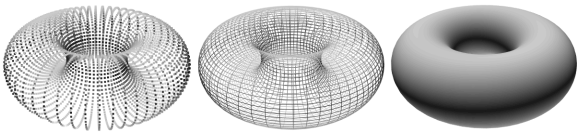

Figure 6. Different display modes: points, wireframe and solid.

While geometry of the surface is approximated to a certain level of detail, normals are computed exactly. They get evaluated per-fragment when rendering the final image which results in more accurate and better looking lighting.

For each batch auxiliary texture, containing fetch coordinates and parameters, is prepared. The texture components *xy* are used to store QB-patch parameter. The components *zw* store fetch coordinates. The sole purpose of this texture is to carry information that is available during computation phase to the rendering phase. This texture would not be required if we have chosen to compute normals together with positions in the batch computation stage. Positions/Normals textures would be passed from the compute stage to the rendering stage. However, this approach would only give us normals per-vertex.

Performance

The prototype of the framework was implemented and its performance was evaluated. Below there are the results of the visualization of various models. The evaluation was performed on the Nvidia 9400M GPU, Safari 6 browser. The results were collected while the 5 seconds animation of Clifford transformation was rotating model from 0 to 2π. During transformation number of computed points varied because LOD of patches was changing.

Model	Patch count	Min points	Max points	Min FPS	Max FPS
Square	1	4	65536	39	74
Cube	6	24	82944	31	52
Sphere	4	4096	4096	45	76
Cone	2	2048	2048	47	84
Cylinder	2	2048	8192	49	77
Torus	4	1024	40960	36	55
Teapot	3305	13220	467648	12	16
Model1	150	21504	40512	35	56
Model2	96	22800	34176	38	58
Model3	56	37376	57344	28	65

Table 1. Performance of the framework.

The results show that the performance of all models except for the teapot was high enough to visualize transformations in real-time.

An alternative for this framework could be brute force approach: precompute points of all patches with high enough fixed LOD. Transformation would then be applied for all final points directly. Even though this might work well on modern GPUs, we would lose the structure of the surface. That is, it will no longer be possible to modify control points as well transformation parameters without recomputing data buffers.

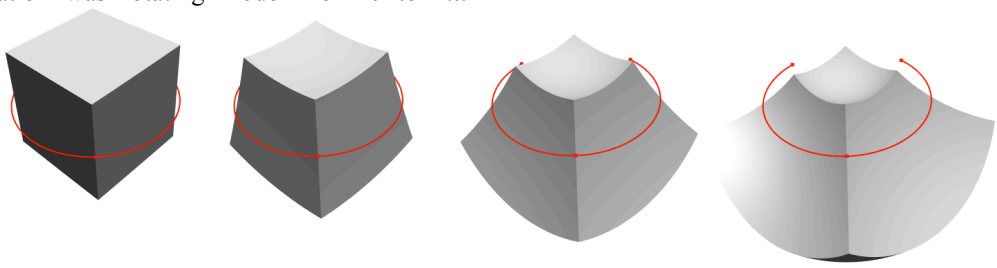

Figure 7. "Rotating" a cube about the fixed circle.

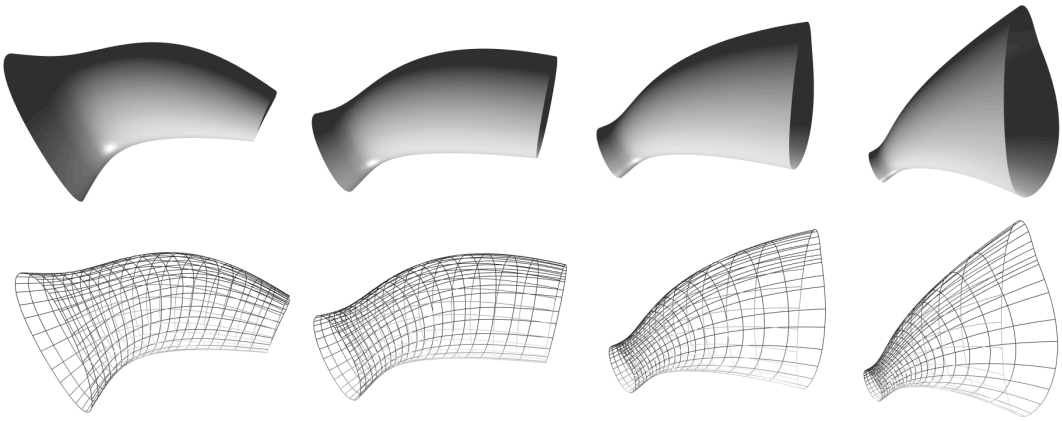

Figure 8. Deformation of a smooth Dupin cyclide patchwork (surface model by [Bo10]).

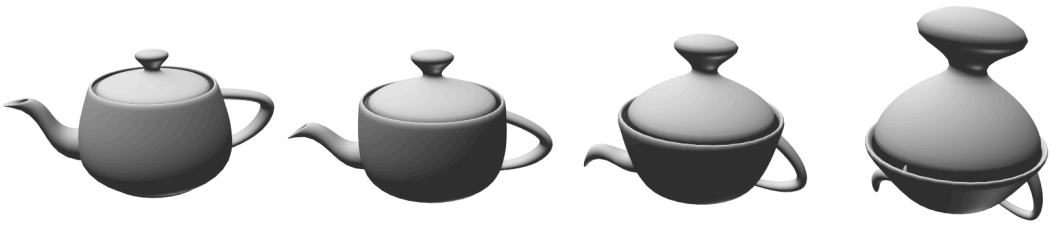

Figure 9. Clifford translation of Utah Teapot.

4. EXAMPLES

Using Theorems 1 and 2 one can control M-transformations by 3 or 4 points. Several examples are illustrated by figures:

Example 1. *Rotation about a circle* (Fig. 7): 3 points are fixed on the given circle and the 4-th point is a vertex of the cube and moves in circular orbit (cf. Theorem 2). Faces of the cube are deformed to spherical patches.

Example 2. *Hyperbolic transformation* (Fig. 8): 2 endpoints of the circular arc are fixed and the 3-rd point moves along that arc (see Theorem 1). The model is a smooth patchwork of Dupin cyclides, borrowed from [Bo10] (look for details in [Bo11]).

Example 3. *Clifford translation* (Fig. 9): M-transformation of Example 1 composed with Euclidean rotation along the circle going through the first 3 points. The model is the quad mesh of Utah Teapot.

5. CONCLUSIONS

A web-based approach for real time interactive visualization of Moebius transformations in 3-space was introduced. Both surface constructions and space transformations were presented in a uniform quaternionic setting based on Quaternionic-Bezier formulas. Original proofs of the main properties of Moebius transformations in 3-space were derived. The proposed WebGL framework was evaluated in the prototype implementation.

One possible extension of the proposed visualization method is related to increasing degrees of surface patches. Indeed, the formulas (4) and (5) used for the transformed control points and weights can be generalized for arbitrary degrees. Then, for example, one can apply Moebius transformations to arbitrary NURBS surfaces.

We hope this paper will not only be useful for better understanding of Moebius transformations in space but also will attract attention to new opportunities for shape modeling and animations.

6. ACKNOWLEDGMENTS

Our thanks to Pengbo Bo for allowing us to use models of smooth Dupin cyclide patchworks from his PhD thesis [Bo10].

7. REFERENCES

[Arn08] Arnold D.N., Rognes J., Moebius Transformations Revealed, *Notices of the AMS* 55 (2008), 1226-1231.

[Arn09] Arnold D.N., Moebius Transformations Revealed, *Webpage,* updated February 14, 2009, http://www.ima.umn.edu/~arnold//moebius/

[Bo10] Bo P., Surface Fitting and Developable Surface Modeling, *PhD Thesis, The University of Hong Kong*, 2010.

[Bo11] Bo P., Pottmann H.,Kilian M., Wang W., Wallmer J., Circular arc structures, *ACM Transactions on Graphics* 30 (2011), #101,1-11. http://www.geometrie.tugraz.at/wallner/cas.pdf

[Gwy12] Gwynne E., Libine M., On a Quaternionic Analogue of the Cross-Ratio*, Advances in Applied Clifford Algebras* 22 (2012), 1041-1053.

[Bis10] Bisi C., Gentili G., Moebius Transformations and the Poincare Distance in the Quaternionic Setting*, Indiana University Mathematics Journal* 58 (2010), 2729-2764.

[Kra11] Krasauskas R., Zube S., Bezier-like parametrizations of spheres and cyclides using geometric algebra, in: Guerlebeck, K. (Ed.), *Proceedings of 9-th International Conference on Clifford Algebras and their Applications in Mathematical Physics*, 2011, Weimar, Germany. http://www.mif.vu.lt/~rimask/old/pdf/Bezier-like.pdf

[Pot12] Pottmann H., Shi L. and Skopenkov M., Darboux cyclides and webs from circles, *Computer Aided Geometric Design* 29 (2012), 77–97.

[Khr10] The Khronos Group. OpenGL ES Common Profile Specification. Version 2.0.25, November 2, 2010. http://www.khronos.org/registry/gles/specs/2.0/es_full_spec_2.0.25.pdf

[Khr12] The Khronos Group. WebGL Specification, Version 1.0.1, 2012. https://www.khronos.org/registry/webgl/specs/1.0.1/

[Bro13] Brodtkorb A.R., Hagen T.R, Sætra M.L., GPU Programming Strategies and Trends in GPU Computing, *Journal of Parallel and Distributed Computing* 73 (2013), 4-13.

[Mic12] Microsoft Developer Network, Rasterization Rules, 2012. http://msdn.microsoft.com/en-us/library/windows/desktop/cc627092(v=vs.85).aspx