Effect of Textures and Motion on 3D Shape Perception in Visualization

Lingji Wang
Gettysburg College
wangli01@gettysburg.edu

Sunghee Kim
Department of Computer Science
Gettysburg College
skim@gettysburg.edu

ABSTRACT

This paper presents an approach on creating moving texture on 3D objects that follows principal directions. In order to get different kinds of drawing look, we design an additional method. Compare to the original approach, it applies non-photorealistic rendering technique and has stroke curves instead of discontinuous straight lines.

ACM Reference format:
Lingji Wang and Sunghee Kim. 2018. Effect of Textures and Motion on 3D Shape Perception in Visualization. In *Proceedings of Conference Name, Conference Location, Conference Date and Year,* 4 pages.
DOI: 10.1145/8888888.7777777

1 INTRODUCTION

To design, create and implement algorithms that can transmit scientific data to be understood accurately and intuitively is a very important goal in visualization research. In order to correctly clarify images, it is significant to accurate percept shape and surface details first. Previous research has shown that humans are able to use the pictorial details in two-dimensional images to recognize three-dimensional shapes. For example, shading is an effective pictorial cue for helping humans perceive three-dimensional shapes. However, when people zoom in on part of a 3D surface or shape, shading alone is not able to provide enough details of such local viewing for humans to correctly perceptive the 3D shapes. On the other hand, previous studies have shown that using an appropriate texture can improve the 3D shape perception, and furthermore, previous studies also found that the first principal direction [1] textures has the most significant effect on 3D shape perception improving[Interrante and Kim 2001].

We consider the impact of motion on the accuracy of shape perception. Structure-from-motion provides a strong shape cue and we hope to evaluate its effect on shape perception by comparing the accuracy obtained through motion of a textured object itself, and through the motion of texture on a stationary object. This paper presents two approaches that both create moving texture in which the texture elements follow principal directions, but one has a non-photorealistic rendering effect while the other one not.

[1]The first principal direction of a vertex on a surface is the greatest curvature at that point.

Conference Name, Conference Location
 978-1-4503-1234-5/17/07…$15.00
DOI: 10.1145/8888888.7777777

2 BACKGROUND AND PREVIOUS WORK

[Lum et al. 2003] presented a visualization technique, named kinetic visualization. With particle system, it is able to create motion which particles flow on the surface of a static 3D object following principal curvature directions. Our method is analogousto to [Lum et al. 2003] when principal curvature is used as moving direction in their case, but we use line rendering instead of point-based rendering. The other difference is that in our case, the new direction is not effected by previous direction so its orientation does not need to be adjusted. Therefore, in our method, the moving directions are definite.

3 ALGORITHMS AND METHODS

In this paper, we introduced two simple approaches on first principal direction texture motion on a stationary object. The difference on these two methods is one (named **principal direction line drawing**) starts at some random vertices, and moves followed by the first principal direction of that vertex, while the other one (named **stroke line drawing**) starts at the centroid of every triangle on the surface, and moves followed by the interpolation of the first principal directions of the three vertices of the triangle that the stroke current is in.

In both cases, we assume that for every 3D object, we have the information about the xyz coordinates and principal directions for every vertex, and the three vertices for every triangle. All triangles connect together will form the surface, as shown in Figure 1.

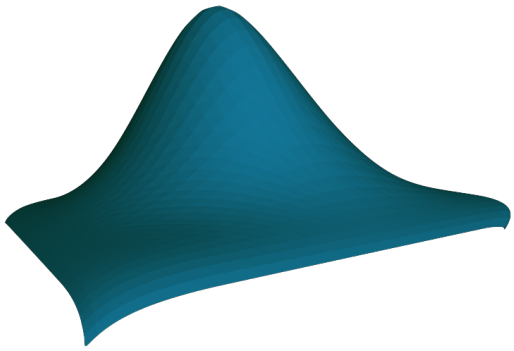

Figure 1: Triangle Surface of a 3D Spline Object

Following are the global data members we used in both methods:

verticesList	contains all vertices of current 3D object
trianglesList	contains all triangles of current 3D object
moveSpeed	the moving speed of each line/stroke
percentLimit	percent of the vertices/triangles choose randomly from verticesList/trianglesList to generate lines/strokes
depth	layers of triangles away from the center point of each cluster
cpList	data member for EACH point (for principal direction line drawing) or stroke (for stroke line drawing), list of points that constitute each point/stroke
maxNumCP	max number of control points in each **cpList**

Table 1: Global Data Members List for Both Methods

Two methods have very similar general structure as specified in Algorithm 1:

Algorithm 1 General Structure

1: Load points, their principal directions, and triangles/triangle surfaces. Generate **verticesList** and **trianglesList**.
2: Compute the **moveSpeed**, which is proportional to the average triangle side length.
3: Generate **pointsList** for principal direction line drawing method (Algorithm 2) or **strokesList** for stroke line drawing method (Algorithm 5).
4: Cluster points/strokes (Algorithm 7).
5: Compute **cpList** for each point (Algorithm 3)/stroke (Algoeithm 6).
6: **while** the program continues **do**
7: Draw current segment for each principal direction line/stroke line and move to next segment for each line (Algorithm 4).
8: Generate dot line effect (3.4).

3.1 Principal Direction Line Drawing

Algorithm 2 and 3 describes how to initialize points and generate their cpList, as well as generate the pointsList. Algorithm 4 describes how to draw principal direction lines for one frame.

Algorithm 2 Initialize Points and Generate pointsList

1: Pick **percentLimit** percent points from **verticesList**.
2: **for** each picked point **p do**
3: Make **p** as a Point object, which contains its position, principal direction (both information are load at step 1 of Algorithm 1) and a **cpList**.
4: Add current position to **cpList**.
5: Initialize a segment index **i** = 0 (in 0-based computer language) for **p**.
6: Add current **p** to **pointsList**.

Algorithm 3 Compute cpList for points

1: **while** there is some unfinished point in **pointsList do**
2: **for** each unfinished point **p do**
3: **if** number of control points in **p**'s **cpList** exceeds **maxNumCP then**
4: Mark **p** finished.
5: **else**
6: Get last point **cp** in **cpList**. (Notice **cp** is a Point object).
7: Move **cp** with **cp's** principal direction and speed **moveSpeed**. Name the new location **newLoc**.
8: Project **newLoc** onto the surface.
9: Find the vertex **v** in **verticesList** that is nearest to **newLoc** and is not **cp**.
10: **if v** already exists in **p**'s **cpList then**
11: Mark **p** finished.
12: **else**
13: Add **v** to **p**'s **cpList**.

Algorithm 4 Draw Principal Direction Line for One Frame

1: **for** each point **p** in **pointsList do**
2: Draw **p**'s current segment from the $\mathbf{i}^{th}$ point to $\mathbf{i+1}^{th}$ point in **p**'s **cpList**.
3: Update **i** by 2.
4: **if i+1**'s point >= **cpList**'s length (in 0-based computer language) **then**
5: Delete current **p** from **pointsList**.

3.2 Stroke Line Drawing

Then we attempted to use a non-photorealistic rendering (NPR) technique to create better images. The stroke line drawing is the approach method. Other than the data members listed in Table 1, each stroke object contains one more data member, named **triList**. Algorithm 5 describes how to initialize strokes and generate the strokesList.

Algorithm 5 Initialize strokes and Generate strokesList

1: Pick **percentLimit** percent triangles from **trianglesList**.
2: **for** each picked triangle **t do**
3: Initialize a stroke object **s**, which contains position as the centroid of **t**, a **cpList**, a **triList** which contains all triangles that **s** passed.
4: Add **t** to **triList**, and add **t**'s centroid to
5: Initialize a segment index **i** = 0 (in 0-based computer language) for **s**.
6: Add **s** to the **strokesList**.

Algorithm 6 describes how to generate the cpList for each stroke. The idea is based upon [Girshick et al. 2000]. The moving direction of each control point is computed by interpolate the principal directions of the three vertices of the triangle that current control point is in. We use barycentric coordinates system to check if a control point moves out of a specified triangle (line 7), and check which neighbor triangle (if exists) contains current control point (line

8). Barycentric coordinates also help to determine the weight on each vertex of the triangle respect to the control point, and we use these weights to interpolate the moving direction (line 13). Notice Barycentric coordinates system works properly only if the control point and the triangle are on the same plane. Therefore, the point needs to be projected on the triangle before using the Barycentric coordinates testing.

Algorithm 6 Compute cpList for strokes and Adjust strokesList

1: **while** there is some unfinished stroke in the **strokesList**, **do**
2: **for** each unfinished stroke **s**, **do**
3: **if s** has exceeds **maxNumCP then**
4: mark **s** finished.
5: **else**
6: Get last triangle **t** in **triList** and **s**'s current location **p** (the last point in the **cpList**). Get the barycentric coordinates (u, v, w) for **p** with respect to **t**.
7: **if** at least one of u, v, w $<$ 0 (which means **p** is outside **t**) **then**
8: Check through all neighbor triangles surround using barycentric coordinates system for **p** until either find the triangle that has u, v, w all $>$ 0 or all neighbor triangles has been checked.
9: **if p** is not in any of the neighbor triangles **then**
10: mark **s** finished.
11: continue
12: Add current triangle **ct** into **s**'s **triList**.
13: Use the last barycentric coordinate (u, v, w) (which all u, v, w $>$ 0) and the principal directions of the three vertices of current triangle (either **t** or **ct**) to interpolate the principal direction **pd** at **p**.
14: Move **p** to new location **newLoc** with direction **pd** and speed **moveSpeed**.
15: Add **newLoc** to **s**'s **cpList**.
16: Get rid of strokes in the **strokesList** that either circle back or too short (i.e. number of control points in **cpList** $<$ 2).

The way to draw stroke lines is very similar as to draw principal direction lines. We just use Algorithm 4 and simply replace point p as stroke s and pointsList as strokesList.

3.3 Cluster Method

In **principal direction line drawing**, one point is created for each vertex and in **stroke line drawing**, one stroke is created for each triangle. For a large model we may end up with too many points or strokes that are too close to each other. One way to avoid such dense problem is to adjust parameter **percentLimit**, but since selected points/strokes are randomly picked, there is chance two neighbor points/strokes being selected. In order to more evenly spaced points/strokes out, we apply a clustering algorithm (step 4 of Algorithm 1) which groups triangles on the surface based on the nearness.

Recall the data member **depth** controls the size of each cluster. So both **percentLimit** and/or **depth** effect the density of texture lines on a 3D object. The benefit of cluster is evenly spaced while the benefit of **percentLimit** is randomness. Users can choose the one they prefer and set the parameter based on their requirements (percentLimit = 100% or depth = 0 means no density reduced). Notice for **principal direction line drawing**, reduce density of texture lines is necessary otherwise users are not able to see any moving motion.

Figure 2 gives a cluster example with different depth value. Assume the hexagon in the figure is part of the triangle surface of a 3D object, and the largest blue dot vertex represents the center of a cluster. Then the dark blue part on each hexagon represents the area belongs to the cluster under given depth. Algorithm 7 described in

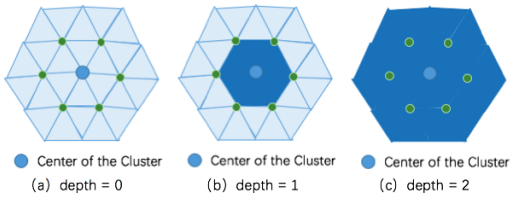

Figure 2: Cluster with different depth

detail how we do the clustering on a 3D model.

Algorithm 7 Cluster Algorithm

1: **while** there is some unmarked vertex in **verticesList do**
2: Random choose one unmarked vertex **v** from .
3: A cluster is created with center at **v**.
4: Mark **v**.
5: Get all triangles that connected to **v**.
6: **while** depth $>$ 0 **do**
7: **for** each triangle **t** that has some unmarked vertices **do**
8: Add **t** to current cluster.
9: **for** each unmarked vertex **uv** in **t do**
10: Mark **uv**.
11: Recursively get all triangles that connected to **uv** and have some unmarked vertices, and add them to current cluster.
12:
13: Decrease depth by 1.
14: Find the farthest unmarked vertex from **v** and assign **v** to it.

Figure 3 show color coded clusters on a half cylinder model with depth 3. In this figure, each cluster has a unique color and all triangles belong to that cluster use its color.

3.4 Dot Line Effect

Recall that we get rid of points/strokes that reach the end of their cpLists after every frame drawing (step 5 of Algorithm 4). Without applying this effect, after certain number of frames(at most half of maxNumCP), there will be no textures moving or appearing on the surface because all points/strokes are deleted from the pointsList/strokesList.

Initially, when some points/strokes reach the end of their cpLists, instead of delete them from pointsList/strokesList, we reset their

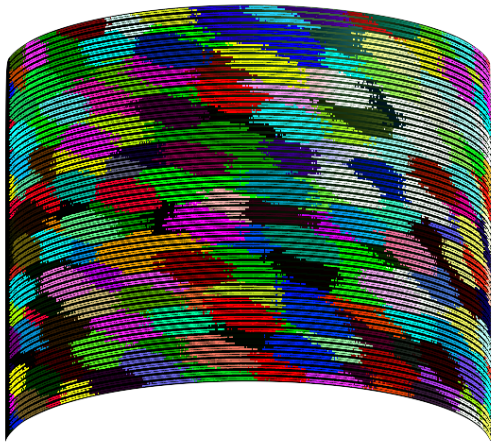

Figure 3: Color coded clusters on half cylinder model

segment index **i** back to 0 and continue the program. Since we only draw one segment of each point/stroke each frame, and there may be lots of segments for one point/stroke based on **maxNumCP**, the problem for this approach is textures become too sparse either in the region points/strokes passed, or in the region they are going to reach.

To avoid this problem, we decide to copy all points/strokes in the pointsList/strokesList every a specified amount of frames, add them into pointsList/strokesList. The copied points/strokes have all features as the original once but they all had segment index **i=0**. Under this approach, we are able to see multiple segments of a point/stroke appear and moving on the frame. All visual segments of one point/stroke form a dot principal direction line/stroke line. Since some points/strokes would be deleted after each frame drawing, we do not need to worry about the size of pointsList/strokesList continues growing.

4 RESULTS AND FUTURE WORK

We use **pyglet** in **python** for drawing. The following four figures show our line drawing algorithms on a complex 3D spline model. Yellow line texture on figure 4 and 5 represents Principal Direction Line Drawing and purple line texture on figure 6 and 7 represents Stroke Line Drawing. Table 2 are the parameter values we used to produce the results.

moveSpeed	average triangle side length / 10
depth	1
maxNumCP	10 (for points), 20 (for strokes)

Table 2: Parameter values used to produce

The first frame from both line drawing animation represents the initial location of each line before motion starts.

Future work of this research includes improving the moving textures on complex models that have many changing directions as well as designing and running a user study to compare the performance of human observers on shape perception tasks under two different motion conditions: the motion of a textured object itself, and the motion of texture on a stationary object.

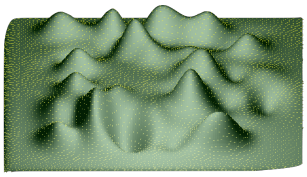

Figure 4: First frame from Principal Direction Line Drawing animation

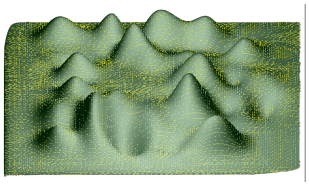

Figure 5: A single frame from Principal Direction Line Drawing animation

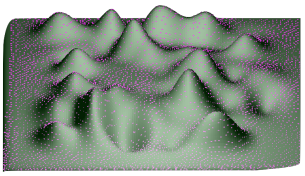

Figure 6: First frame from Stroke Line Drawing animation

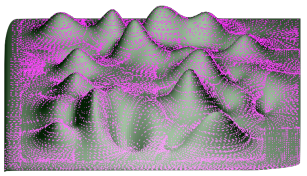

Figure 7: A single frame from Stroke Line Drawing animation

REFERENCES

Ahna Girshick, Victoria Interrante, Steven Haker, and Todd Lemoine. 2000. Line Direction Matters: An Argument For The Use Of Principal Directions In 3D Line Drawings. http://www.afhalifax.ca/magazine/wp-content/sciences/EmpilementDeDisques/delaunaycourbe/MagnifiqueLineDrawjngOnSurfaces.pdf. (2000).

Victoria Interrante and Sunghee Kim. 2001. Investigating the Effect of Texture Orientation on the Perception of 3D Shape. https://www-users.cs.umn.edu/~interran/papers/spie01.pdf. (2001).

Eric B. Lum, Aleksander Stompel, and Kwan-Liu Ma. 2003. Using Motion to Illustrate Static 3D Shape-Kinetic Visualization. http://web.cs.ucdavis.edu/~ma/kinvis/kinvis2.pdf. (2003).