International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:7, No:2, 2013

An Implementation of Stipple Operations

Nakhoon Baek

***Abstract*—**Stipples are desired for pattern fillings and transparency effects. In contrast, some graphics standards, including OpenGL ES 1.1 and 2.0, omitted this feature. We represent details of providing line stipples and polygon stipples, through combining texture mapping and alpha blending functions. We start from the OpenGL-specified stipple-related API functions. The details of mathematical transformations are explained to get the correct texture coordinates. Then, the overall algorithm is represented, and its implementation results are followed. We accomplished both of line and polygon stipples, and verified its result with conformance test routines.

***Keywords*—**Stipple operation, OpenGL ES, Implementation.

I. Introduction

IN the field of computer graphics, the line stippling means applying the given patterns while drawing a line segment, as shown in Fig. 1. Most of line stipples are used for drawing dotted or dashed lines [1]. Without this feature, we need a huge set of line segments to represent dotted or dashed lines, even with remarkable efficiency degradation.

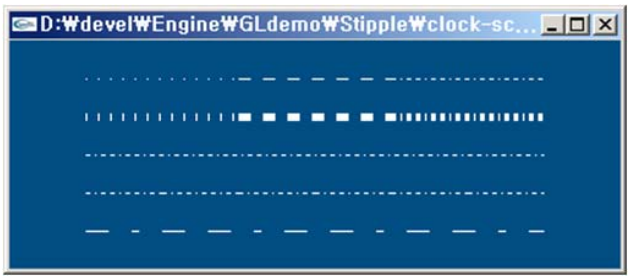

Fig. 1 Stippled line segments from our implementation

For the interior pixels of a polygon, the *polygon stippling* controls the final decision of pixel output, at the final output stage [1]. The polygon stipple patterns are used as a final stage masking patterns. The user provides a black-white image as the *mask*, and this small pattern is regularly repeated in the polygon interior. This feature is imperatively necessary for the implementation of the *screen door transparency* [2]. Fig. 2(c) shows an example of screen door transparency effect, with the mask of Fig. 2(b).

In the case of desktop-based original OpenGL, they provide all the features of such line stippling and polygon stippling. However, in the area of mobile graphics applications, the target systems of mobile phones and tablet PC's have limited resources, and they intensively removed these not yet widely used features.

As the result, the typical mobile graphics standards, including OpenGL ES 1.1 [3] and OpenGL ES 2.0 [4], do not have any stippling support. In contrast, a considerable number of applications still need the line stipple features and/or the polygon stipple features.

In this paper, we represent a systematic way of realizing the line stipple and the polygon stipple features, based on the fixed graphics pipeline of OpenGL ES 1.1. At least to our best knowledge, there has been no literature on the implementation details of stipples.

In Section II, we start from our basic strategy, we show internal transformation sequences and details of our implemented algorithm. Our implementation results are presented in Section III. Conclusions and future works are followed in Section IV.

II. Our Strategy

We used the texture mapping technique with alpha channels, as the fundamental strategy. From the API functions for stipple operations, we will explain the texture coordinate calculation process in Section II B, and the overall algorithm is followed in Section II C.

A. API Functions for Stipple Operations

To specify the stipple operations, we need some API functions. As an example, we show the stipple-related API functions from the OpenGL specification, as follows:

Line Stipple (int factor, ushort pattern);

The parameter factor specifies a multiplier for each bit in the line stipple pattern. If factor is 3, for example, each bit in the pattern is used three times before the next bit in the pattern is used.

The parameter pattern specifies a 16-bit integer whose bit pattern determines which pixels of a line will be drawn when the line is rasterized.

Line stippling masks out certain fragments produced by rasterization; those fragments will not be drawn.

Enable(LINE_STIPPLE);
Disable(LINE_STIPPLE);

Enables and disables the line stipple features, resepectively.

PolygonStipple (unsigned byte patter[4]);

The parameter pattern specifies a pointer to 32×32 stipple

Prof. Nakhoon Baek is with the School of Computer Science and Technology, Kyungpook National University, Daegu 702-701, Korea (e-mail: oceancru@gmail.com).

This research is supported by Ministry of Culture, Sports and Tourism(MCST) and Korea Creative Content Agency(KOCCA) in the Culture Technology(CT) Research & Development Program(Immersive Game Contents CT Co-Research Center).

pattern, as shown in Fig. 2(b). This pattern is used to masks out certain pixels produced by rasterization, as shown in Fig. 2(c). Notice that the original image shown in Fig. 2(a) is stippled with the pattern in Fig. 2(b), to produce the masked pixels.

Enable(POLYGON_STIPPLE);
Disable(POLYGON_STIPPLE);

Enables and disables the polygon stipple features, resepectively.

Now, we aim to implement all these API features independently on OpenGL ES or other stipple-not-supported graphics libraries.

B. Calculation of Texture Coordinates

To implement the stipple operations through texture mapping process, we naturally need to calculate the suitable texture coordinates for each vertex of geometric entities. Typical 3D graphics libraries, such as OpenGL and DirectX, manages their own full 3D geometry transformation pipelines, and user-provided object coordinates (x_{obj}, y_{obj}, z_{obj}, w_{obj}) are transformed to the final screen coordinates (x_{scr}, y_{scr}, z_{scr}, 1), as follows:

$$\begin{bmatrix} x_{scr} \\ y_{scr} \\ z_{scr} \\ 1 \end{bmatrix} = \mathbf{M} \cdot \begin{bmatrix} x_{obj} \\ y_{obj} \\ z_{obj} \\ w_{obj} \end{bmatrix} \quad (1)$$

where *M* is the internal transformation matrix. Then, we assign the 2D texture coordinates (*s*, *t*) to the exactly same value of the screen coordinate (x_{scr}, y_{scr}) as follows:

$$\begin{bmatrix} s \\ t \end{bmatrix} = \begin{bmatrix} x_{scr} \\ y_{scr} \end{bmatrix}, \quad (2)$$

and we can accomplish the correct stipple results.

Only the technical problem is that the typical 3D graphics libraries do not provide any method to get the finally calculated screen coordinates, (x_{scr}, y_{scr}, z_{scr}, 1). To overcome this technical difficulty, we have implemented our own emulated 3D transformation process, which works just as the system 3D geometry transformation pipeline. For OpenGL applications, as an example, we queried the contents of its internal *modelview* matrix and the *projection* matrix, to calculate the overall transformation matrix *M*. Then, we perform the geometric transformation of Equation (1) for each vertex, and assign their corresponding texture coordinates.

C. Overall Algorithm

The details of operation steps can be summarized as follows:

Step 0. Generate texture images from the line/polygon stipple patterns, as the pre-processing step. They specify a 1D pattern for the line stipples, while a 2D image pattern for polygon stipples. We first reconstruct a 3D texture image from these patterns, and store them into the graphics system. The stipple pattern should be converted to an alpha-channel image, where the alpha values are 1.0 for opaque pixels and 0.0 for transparent pixels.

Step 1. From user-defined vertices, get the window coordinates on the screen, based on the geometry pipeline settings. First of all, the line stipples and polygon stipples are specified in the final window coordinates. Thus, we need to calculate the window coordinates for each of target vertices. We simulate the geometric transformation pipeline, to finally get the window coordinates from the original object coordinates of target vertices.

Step 2. We set the corresponding texture coordinates, based on the window coordinates. For each vertex of the given lines and polygons, set its texture coordinate according to its window coordinate. These texture coordinates are used to simulate the stipple patterns with the texture mapping process.

Step 3. Draw the line or the polygon primitive, with alpha-channeled texture. As the final step, the output primitives are processed with the alpha-blending feature enabled. Thus, only the pixels with alpha values of 1.0 will be rendered, which performs the actual stipple processing.

This algorithm steps are implemented to accomplish both of line and polygon stipples. The implementation results are followed in the next section.

III. Implementation Results

We have implemented the line stipple and polygon stipple API (application program interface) functions, according to the OpenGL 2.1 specification [1], over the OpenGL ES 1.1 hardware, which lacks all the stippling operations. Our full software implementation requires one texture unit wholly for the stippling operations. Since typical OpenGL ES 1.1 hardware systems provide more than or equal to 4 texture units, our requirement is not critical for most applications.

Fig. 1 shows the examples of the line stipple patterns rendered by our implementation. As shown here, various kinds of dotted and dashed lines are easily accomplished. Fig. 2, 3, and 4 show the original image, the polygon stipple pattern, and the final polygon-stippled image, respectively, as a good example of the *screen door transparency* effect.

To approve the correctness of our implementation, we used some selected tests from the official OpenGL-related conformance test suites [5]. Our implementation finally passed all these test routines.

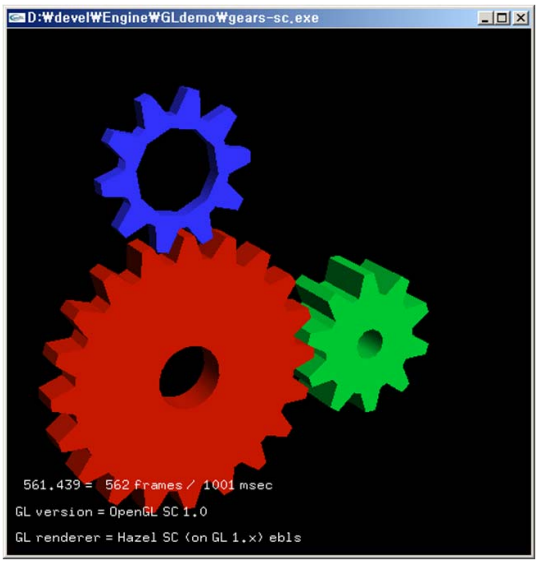

Fig. 2 Original Image

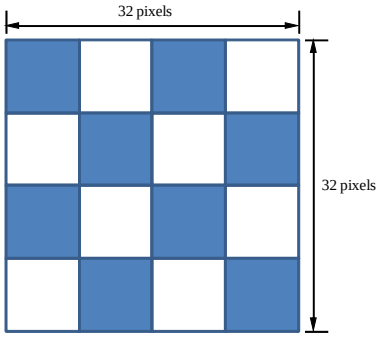

Fig. 3 A polygon stipple pattern

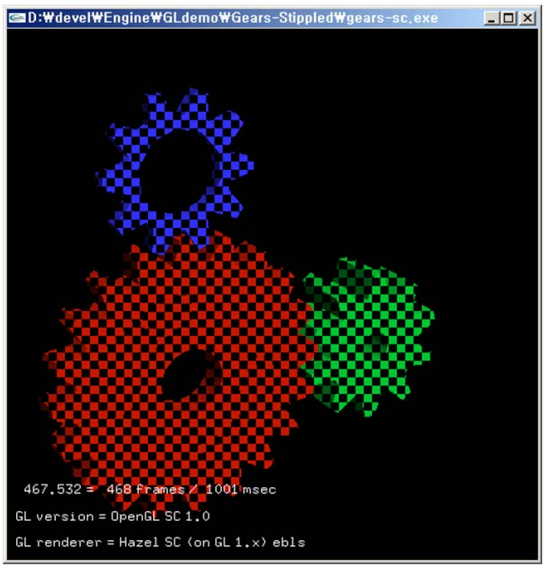

Fig. 4 An example polygon stipple image from our implementation

IV. Conclusion

Stipple patterns are still useful for the lines and polygons drawing, in computer graphics area. In contrast, current OpenGL ES graphics standards do not support these features, mainly due to their limited hardware resources. In this paper, we propose a systematic way of supporting these features. Our implementation supports stippling operations specified in the official OpenGL 2.1 standard, and its correctness is proven with official conformance test routines.

Acknowledgment

This research is supported by Ministry of Culture, Sports and Tourism (MCST) and Korea Creative Content Agency (KOCCA) in the Culture Technology (CT) Research & Development Program (Immersive Game Contents CT Co-Research Center).

References

[1] M. Segal and K. Akeley, *The OpenGL Graphics System: A Specification, Version 2.1*, Dec 2006.
[2] J. D. Mulder, F. C. A. Groen, and J. J. van Wijk, "Pixel Masks for Screen-Door Transparency", *Proc. of VIS'98*, 1998.
[3] D. Blythe, *OpenGL ES Common/Common-Lite Profile Specification*, Khronos Group, 2007.
[4] A. Munshi and J. Leech, *OpenGL ES Common Profile Specification, Version 2.0.24 (Full Specification)*, Apr 2009.
[5] OpenGL SC Conformance Test Suites, http://www.khronos.org/openglsc.

Nakhoon Baek is currently an associate professor in the School of Computer Science and Engineering at Kyungpook National University, Korea. He received his B.A., M.S., and Ph.D. degrees in Computer Science from Korea Advanced Institute of Science and Technology (KAIST) in 1990, 1992, and 1997, respectively. His research interests include graphics standards, graphics algorithms and real-time rendering. He is now also the Chief Engineer of Mobile Graphics Inc., Korea.