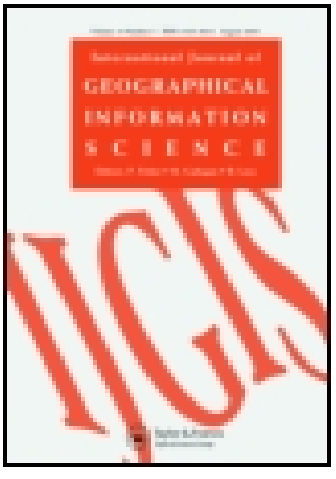

International Journal of Geographical Information Science

Publication details, including instructions for authors and subscription information:
http://www.tandfonline.com/loi/tgis20

A function-based linear map symbol building and rendering method using shader language

Songshan Yue[abc], Jianshun Yang[abc], Min Chen[abc], Guonian Lu[abc], A-xing Zhu[abcd] & Yongning Wen[abc]

[a] State Key Laboratory Cultivation Base of Geographical Environment Evolution (Jiangsu Province), Nanjing, China

[b] Jiangsu Center for Collaborative Innovation in Geographical Information Resource Development and Application, Nanjing, China

[c] Key Laboratory of Virtual Geographic Environment, Nanjing Normal University, Ministry of Education, Nanjing, China

[d] Department of Geography, University of Wisconsin-Madison, Madison, WI, USA

Published online: 20 Aug 2015.

To cite this article: Songshan Yue, Jianshun Yang, Min Chen, Guonian Lu, A-xing Zhu & Yongning Wen (2015): A function-based linear map symbol building and rendering method using shader language, International Journal of Geographical Information Science, DOI: 10.1080/13658816.2015.1077964

To link to this article: http://dx.doi.org/10.1080/13658816.2015.1077964

International Journal of Geographical Information Science, 2015
http://dx.doi.org/10.1080/13658816.2015.1077964

A function-based linear map symbol building and rendering method using shader language

Songshan Yue[a,b,c], Jianshun Yang[a,b,c], Min Chen[a,b,c], Guonian Lu[a,b,c], A-xing Zhu[a,b,c,d] and Yongning Wen[a,b,c]*

[a]*State Key Laboratory Cultivation Base of Geographical Environment Evolution (Jiangsu Province), Nanjing, China;* [b]*Jiangsu Center for Collaborative Innovation in Geographical Information Resource Development and Application, Nanjing, China;* [c]*Key Laboratory of Virtual Geographic Environment, Nanjing Normal University, Ministry of Education, Nanjing, China;* [d]*Department of Geography, University of Wisconsin-Madison, Madison, WI, USA*

(*Received 14 June 2015; accepted 27 July 2015*)

Maps are widely used to visualize geo-information so that map users can develop related understandings about the real world. Such a process for communicating information is largely dependent on the rendering of map elements using different symbols (points and linear and area symbols). To meet the demand of more dynamic and comprehensive visualization in map rendering, it is essential to improve the rendering efficiency. This paper focuses on these research topics, especially the difficulty in constructing and drawing linear map symbols. By employing shader language, a function-based linear symbol building and rendering method is presented in this paper. The basic idea of this function-based method is to build a map-rendering solution that employs graphic processing unit (GPU) acceleration technology to improve the rendering efficiency. A 'function' is used to represent the algorithm that draws certain simple or complex linear map symbols. This function reflects the structure of a linear map symbol (describing the symbol construction information) and also the rendering process of the symbolized linear map elements (handled on a per-pixel basis by the shader program). Based on the Open Geospatial Consortium (OGC), Styled Layer Descriptor (SLD) specifications, four basic line types (i.e., solid lines, dashed lines, gradient color lines, and transition lines) are implemented in the proposed method, and the implementation of line markers, line joins and line caps is also discussed. Three experiments are conducted to demonstrate improvements in map rendering. The results show that a variety of linear map symbols can be constructed in a uniform way, which suggests that the proposed method addresses the difficulty in drawing linear map symbols. With this method, the efficiency of rendering linear map elements is substantially improved compared to using the graphics device interface plus (GDI+) and anti-grain geometry (AGG) methods; it also provides an applicable approach for developing map rendering systems. Using this function-based concept, the complexity of building linear map symbols and drawing linear map elements can be decreased.

Keywords: linear map symbol; map rendering; GPU acceleration; shader language

1. Introduction

Cartography is commonly regarded as an information communication process (Monmonier 1996; MacEachren 2004, Goodchild *et al.* 2007, Kraak and Ormeling 2011) that is used by cartographers to depict information and by map users who 'read' the map to develop related

*Corresponding author. Email: wenyongning@njnu.edu.cn

understandings. Widely approved as a useful tool to present geo-information, cartographic symbols (or map symbols) play an important role in the information communication process and lay the foundation for map rendering (Robinson *et al*. 2012, Trapp *et al*. 2014, Ruas 2015).

With the increasing development of data-processing and information discovery technologies, the ability to present spatial entities and geographic phenomena more dynamically and efficiently is required (Graham and Shelton 2013, Bandrova *et al*. 2014, Chen *et al*. 2015). Rendering map elements more efficiently is an important research target for satisfying both the visualization requirements and the demand for 2D/3D integrated information acquisition (Konecny 2011, Semmo *et al*. 2012, Dübel *et al*. 2014, Roth 2013). In the context of the vast increase in data volume and real-time data (since the Big Data era has arrived), the problem of inefficiency has become more significant.

To improve drawing efficiency and quality, recent research has focused on data visualization in computer graphic devices, proving that using graphic processing unit (GPU) accelerated methods can significantly improve the rendering efficiency (Häberling *et al*. 2008, Buschmann *et al*. 2014), especially in the case of drawing a large amount of data. Based on GPU calculation platforms (such as OpenGL (Open Graphics Library Shader Language, Khronos Group, Beaverton, OR, USA; www.khronos.org) or DirectX) that take full advantage of the ability of hardware acceleration, graphics can be more quickly presented. Shader language is designed and developed in computer visualization studies to help users implement a variety of drawing effects which can even be supported by hardware. Shader language, which provides a programmable interface for users to construct and assemble graphics that can be displayed by a GPU, is widely used in 3D drawing platforms to quickly achieve a variety of presentation effects (Haeberling 2002, Quinn *et al*. 2005, Varcholik 2014).

In cartography and geographic information system (GIS), point symbols, linear symbols, and area symbols are the three basic presentation methods for vector data (Kersting and Döllner 2002, Wen *et al*. 2013, Wu *et al*. 2014). Regarding linear map elements, the existing drawing technologies continue to face difficulties in applying complex line types. Generally, compared to point symbols and area symbols, linear symbols are much more complicated (Turdukulov *et al*. 2014, Zhang and Zhu 2015). Research on the rendering of area symbols can be summarized as the filling of different graphic cells or icons based on the scan line algorithm or its improved algorithms, whereas point symbols can be regarded as the layout of different icons that have their own semantic meanings (Lane *et al*. 1980, Chen *et al*. 2014, Bae *et al*. 2015). However, linear symbols are filled by a variety of graphic cells along a particular direction, which would be inefficient if drawn using polygon rasterization methods (Peterson 2014). The types of linear symbols are diverse; thus, it is difficult to determine a universal processing technique to handle the variety of graphics (Zinsmaier *et al*. 2012). Therefore, the rendering of linear symbols is one of the greatest difficulties that affect the drawing efficiency of maps; the complexity of constructing linear symbols also restricts the map design.

Focused on the difficulties in rendering linear map elements, a few studies have analyzed line drawings using shader language, such as anti-aliased lines (Chan and Durand 2005), Bézier curves (Rueda *et al*. 2008), generic paths (Kilgard and Bolz 2012), and dashed lines (Rougier 2013). These studies can be used to improve the efficiency of drawing specific geometric lines, although problems still exist when constructing a variety of linear map symbols and rendering linear map elements from different cartographic semantics. A more comprehensive method to draw variety map symbols is needed to support the building of thematic maps.

Based on the above analysis regarding the difficulty of rendering and constructing linear map symbols using shader language, this paper proposes an extendable linear symbol drawing method. In contrast to geometric drawing in computer visualization, this paper does not focus on detailed drawing algorithms. Instead, we attempt to build a map rendering solution that employs GPU acceleration technology to improve the rendering efficiency. The construction of a linear map symbol is described using a symbol-related piecewise function, which can be implemented using shader language. The solution primarily uses GPUs to present the linear objects that are tessellated with CPU computation; extendable functions in shader language are used to construct the linear symbols, which is essential for decreasing the complexity of linear map elements and improving the presentation effect.

The remainder of this article is organized as follows. Section 2 discusses the technology of the proposed method, demonstrating that our method is suitable for linear map symbols and can also be useful for presenting point symbols, area symbols, and 3D scenes. Section 3 introduces the basic concept of the function-based method for constructing linear map symbols, which is extendable so that users can implement a variety of linear map symbols. In Section 4, the methodology is presented to explain the steps of the proposed method; four typical types of commonly used linear symbols are explained in detail, and the methods for using these map symbols are discussed. Section 5 introduces and demonstrates the capability of the proposed linear map symbol building and rendering method in practical map rendering applications. Finally, conclusions and a discussion are presented in Section 6.

2. Background on shader language for map rendering

In computer graphics, the 'programmable rendering pipeline' has been developed to provide greater flexibility in graphics rendering, compared with the traditional 'fixed-function rendering pipeline'. The designation of shaders is the core of the programmable rendering pipeline, and shaders are segments of a computer program that are compiled by the GPU to achieve specific rendering results. Shader languages are designed and developed to help users implement a variety of shaders. By programming with a shader language, users can handle most of the drawing effects, such as position, color, texture, and brightness. Because shaders are processed by GPU computation, the efficiency of rendering can be improved at the hardware level.

Generally, there are two basic shader types: vertex shaders and fragment shaders (also called pixel shaders). Figure 1 shows the typical process of the programmable rendering pipeline (other new shader types, such as geometry shaders and tessellation shaders, are not discussed in this paper because they constitute auxiliary steps that are inserted into the main process). To present spatial geometry objects on-screen, the original geometry must be assembled by geometric primitives; vertex shaders allow the programmable vertex processor to handle the traits of every vertex, such as their positions, textures, coordinates, and colors. After the primitive assembly, the screen coordinates of the geometry to be drawn are based on the projection transformation parameters via computations. Through rasterization and interpolation, each pixel in the geometry can be assigned by a fragment shader, which allows the programmable fragment processor to handle the traits of every pixel, including the color, z-depth, and alpha value.

Because triangles are the basic geometric primitives that drive graphics rendering in GPUs, all geometric elements in a map must be tessellated into triangles. Figure 2 shows some simple implementations of the tessellation of different map symbols. The basic drawing method for point symbols is to convert their icons into texture resources and bind the textures with an envelope of the point symbols using the texture coordinates in two triangles. For an

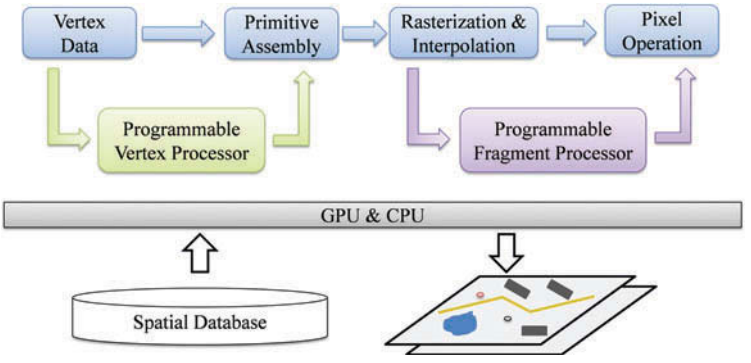

Figure 1. Typical process of a programmable rendering pipeline.

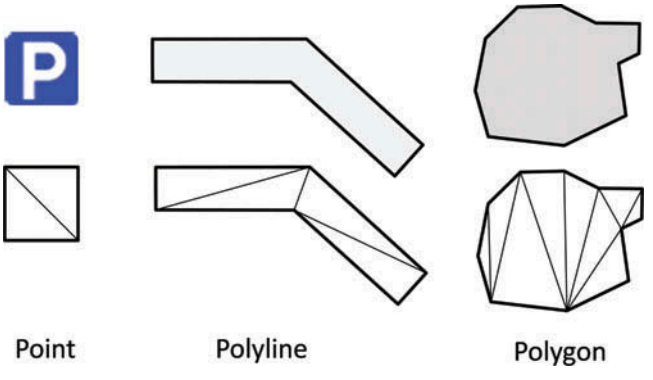

Figure 2. Example of the tessellation of a geometry for GPU rendering.

area symbol, it should first be triangulated. Then, different textures or colors can be used to fill the area. Linear symbols are extended in a perpendicular direction, tessellated into polygons, and subsequently filled with different graphics according to the specific line type.

The map rendering architecture proposed in this study assigns the tessellation work to the CPU, because the fragment processor in the GPU uses a per-pixel computation mode and trigonometric function (sin, cos, and tan) computations in a fragment shader can be extremely time-consuming (Drew 2008). Generally, the triangles are organized using a *Vertex Buffer* and an *Index Buffer*, according to the 3D graphics programming interface (OpenGL and DirectX). As shown in Figure 3(a), the *Vertex Buffer* contains the coordinate information of each vertex, and the *Index Buffer* contains the construction information for building triangles. Once the *Vertex Buffer* and *Index Buffer* are organized, the presentation can be processed via GPU computation, which can be controlled by the shader language programs.

In Figure 3(b), the typical shader programs are introduced (using the OpenGL Shading Language, GLSL). In the vertex shader, the coordinate and projection transform matrix

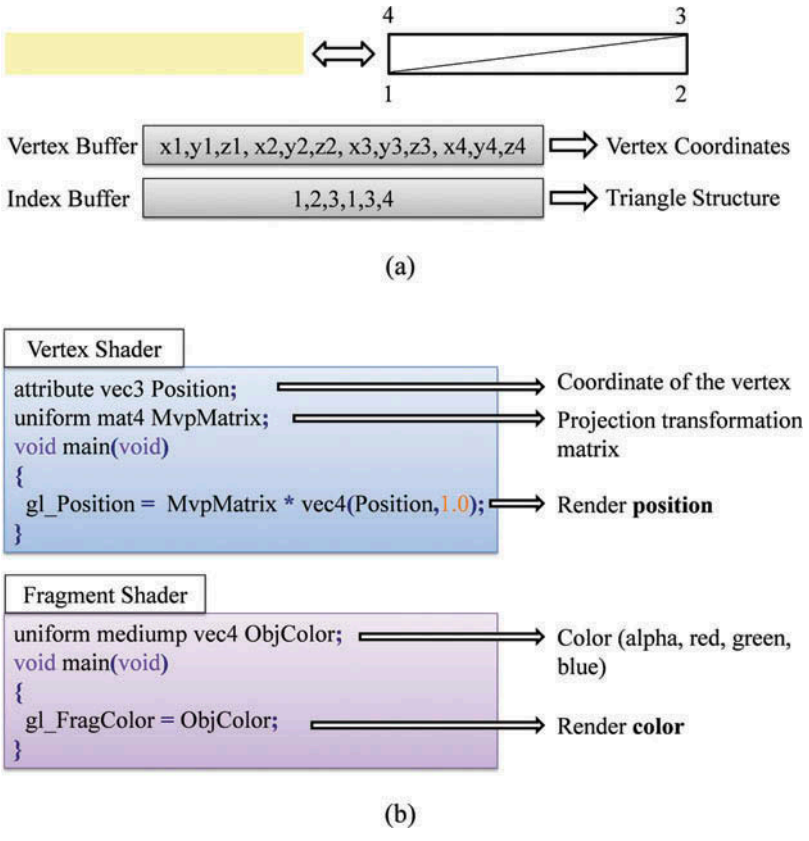

Figure 3. Basic work process of rendering a geometry using shader language.

are the input data, and the resulting output is the render position, which contains the coordinates of the applied transform matrix. The fragment shader assigns the color of each pixel in the presented graphics.

3. Basic concept of the proposed method

3.1. *Rendering linear map elements on a per-pixel basis*

The basic concept of drawing linear map elements is regarded as the filling of repeated graphic cells (i.e., symbols), and each pixel's color in the presented geometry should be computed based on the local coordinates of the single symbol. In terms of a single symbol, the color can be selected based on a function that is dependent on the position. In Figure 4, a simple dash-type linear symbol is employed to help explain the method for constructing a linear symbol. This symbol is constructed in four parts: the top outline part (in gray), the bottom outline part (in gray), the left segment part (in blue), and the right segment part (in yellow).

As shown in Figure 4(a), the variables U and V are employed to help explain the construction information of linear map symbols. U is oriented in the direction of the line, whereas V is perpendicular to the line. The total length of a line is TL, and the line width is W. V is normalized in [0, 1] on the width side, and U is normalized in [0, TL/W] on the length side. Therefore, the Vertex Buffer should be formulated as an

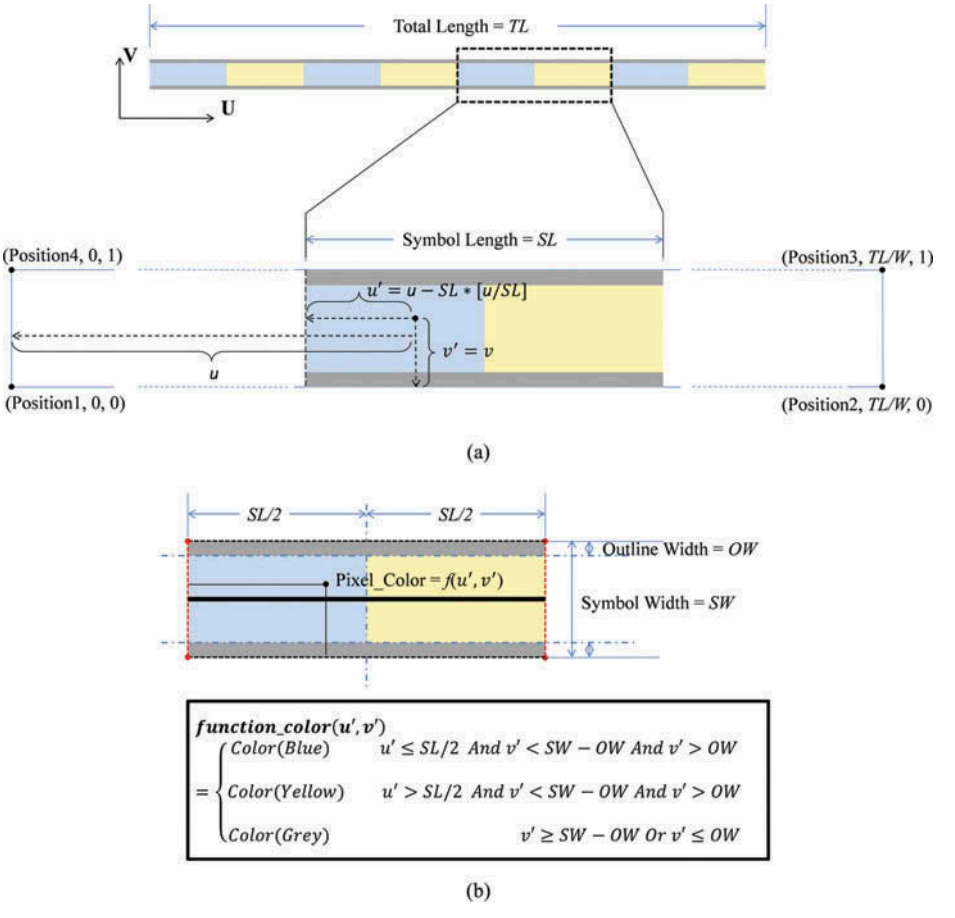

Figure 4. Function-based building method for linear map symbols.

array of (*position, U, V*) so that the vertex's *U* and *V* information for a line segment can be conveyed to the vertex shader program and the fragment shader program. The fragment shader iterates all pixels in (0, V_{max}) and (0, U_{max}). Moreover, to obtain the color of a pixel from the specified symbol, (*U, V*) is converted to the symbol's coordinate. The converted coordinate u' removes the length of the previous segments of the symbol; the resulting algorithm is $u' = u - SL \times [u/SL]$. *SL* stands for the symbol length, and $[u/SL]$ indicates the number of previous segments. Here, $u - SL \times [u/SL]$ can be used to determine the length from the start of the current segment.

Based on the computed u' and v' values, a function for computing the color can be determined. In Figure 4(b), the symbol length is *SL*, the outline width is *OW*, the symbol total width is *SW*, and both the left and right parts are *SL*/2. To construct this symbol, *function_color*(u', v') needs to be programmed in the fragment shader, and the values of *OW, SW*, and *Color* should be transmitted in the rendering stage. If $v' < SW - OW$ and $v' > OW$, the color should be selected from the left segment or the right segment. For this condition, if $u' \leq SL/2$, the corresponding pixel color is blue; otherwise, the color is yellow. If $v' \geq W - SW$ or $v' \leq SW$, the color is the outline color, i.e., gray. This function can be easily implemented using the fragment shader program.

3.2. Construction of linear map symbols

To implement this function-based concept, a column array, a row array, and a color table were designed to transmit information from the outside environment to the GPU shader program. The column array describes a symbol's vertical structure, and the row array describes its horizontal structure. The color table is implemented as a two-dimensional array which contains the color value of every column–row index.

Using the column and row arrays and the color table, the function programming can be simplified into *If-Else* statements, which are easy to implement. In Figure 5, the basic implementation method is introduced. The sample symbol in Figure 5 is the same as in Figure 4. The width of the sample symbol is normalized to 1, and the length is 2. The outline color is defined as *color1*, the left segment is *color2*, and the right segment is *color3*. The height of the two outlines is 0.1, and the height of the middle filling region is 0.8. The width of both the left and right segments is 1.0.

Accordingly, values can be assigned to the variables that are defined by the shader program: *SymbolLength* (the length of the entire symbol: 2.0), *ColumnCount* (the number of columns in the symbol: 2), *ColumnRowCountArray* (an array indicating the number of rows in every column: [3, 3]), *CoumnWidthArray* (an array indicating the width of every column: [1.0, 1.0]), *RowHeightArray* (an array indicating the height of each row in every column: [0.1, 0.8, 0.1; 0.1, 0.8, 0.1]) and *CellColorTable* (an array indicating the color of every column–row cell: [*color1*, *color2, color1; color1, color3, color1*]).

In Figure 6, the basic method to determine the construction information of the linear map symbols is introduced. The function *getColumnByU()* is used to determine which column the current pixel belongs to. A *For* loop is programmed to iterate over every column to determine the column index of the current pixel. Using the computed column index, the function *getRowByV()* is used to compute the row index of the current pixel, which is also programmed with a *For* loop to iterate over each row in the current column.

The main steps of the proposed method are presented in Figure 7. The vertex shader handles the coordinate transformation, and the fragment shader handles the render color,

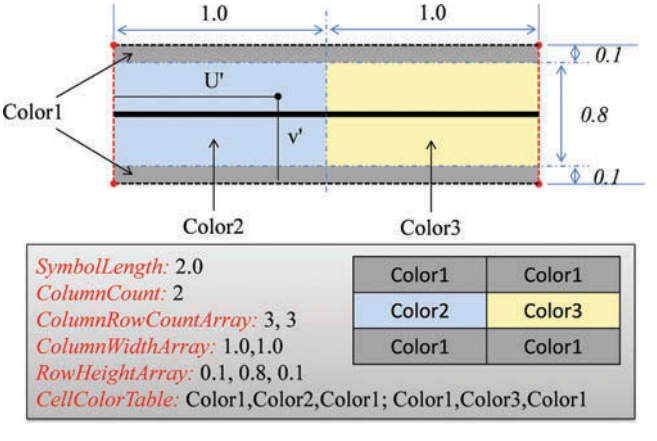

Figure 5. Construction of a sample linear map symbol.

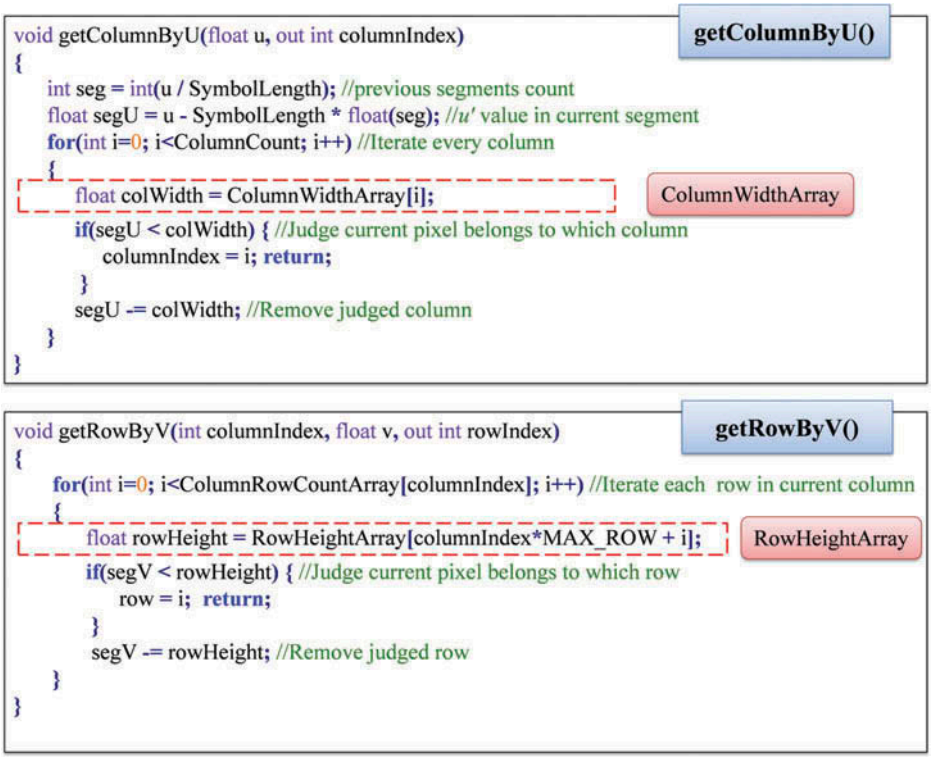

Figure 6. Basic methods to determine the construction information for the symbols.

which can be summarized as follows: *main()* to *getColorByUV()* to *getCellColorByUV()* to *getColumnByU()* and *getRowByV()*. The entrance of the fragment shader program is *main()*, and the color of each pixel is computed using *getColorByUV()*. In *getColorByUV()*, the pixel's color is selected from the color table using *getCellColorByUV(), getColumnByU()*, and *getRowByV()*, which computes the column-index and row-index in the color table.

4. Method of rendering linear map elements based on shader language

To construct different symbols, the Open Geospatial Consortium (OGC) has published the Styled Layer Descriptor (SLD) specification, which defines rules and symbols that control the appearance of maps (OGC 2012). Regarding linear map symbols, the SLD specification proposes the use of *Color, Width, Dash offset, Dash array, Line cap*, and *Line join* attributes for controlling the drawing of a linear map element. In addition, there are three basic stroke types: *solid-color, GraphicFill* (stipple), and repeated linear *GraphicStroke*; both the *GraphicFill* and *GraphicStroke* types can be regarded as the filling of an array of dashed graphics. Based on these attributes, simple or complex linear map symbols can be dynamically constructed. This specification has been widely approved as the standard for building symbol libraries.

Based on the SLD specifications, the method presented in this paper is organized into three stages: (1) rendering the basic line type (including the *Color, Width, Dash offset*, and

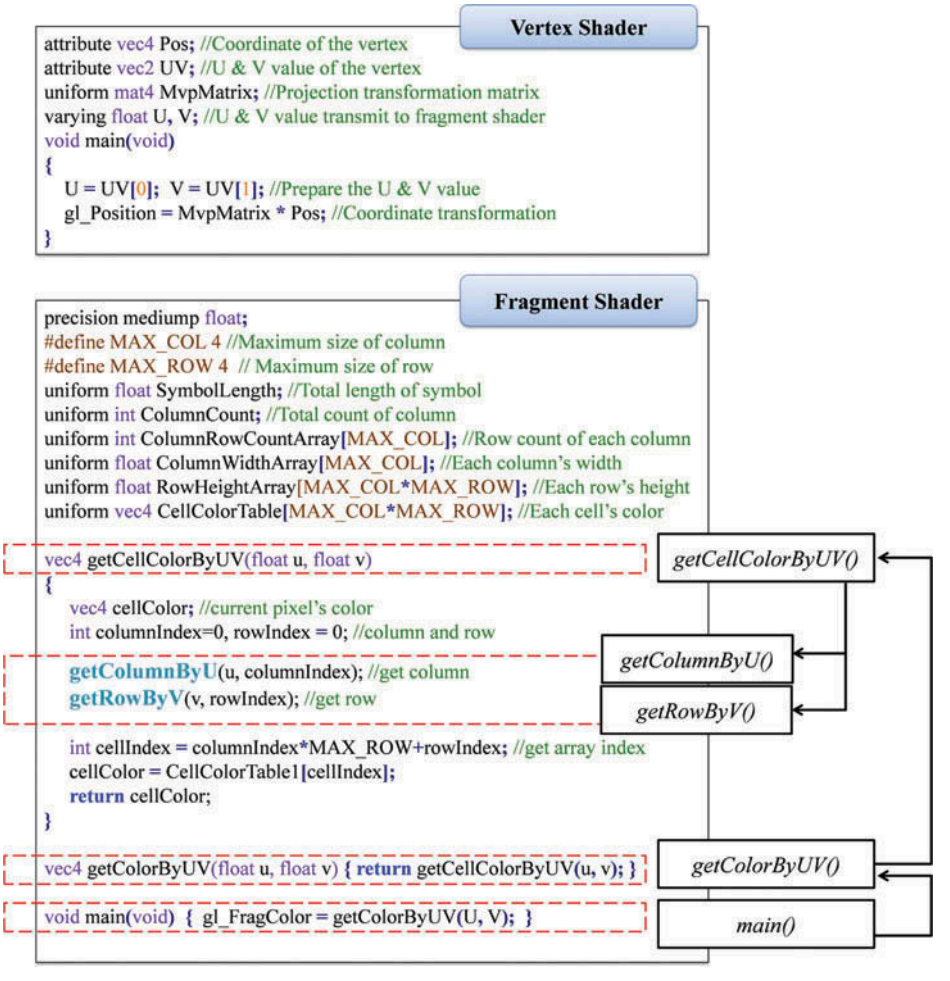

Figure 7. Main processing steps for the function-based method.

Dash array attributes); (2) rendering line markers (including the *GraphicFill* and *GraphicStroke* types); and (3) handling *Line Caps* and *Line Joins*.

4.1. Linear map symbols of typical line types

Based on the commonly used map representation platforms (e.g., Google maps, ESRI ArcGIS, and MapServer), four types of linear map symbols are implemented in this paper: solid lines, dashed lines, gradient color lines, and transition lines.

4.1.1. Solid lines

For linear map elements, 'solid lines' and 'dashed lines' are two frequently used ways to distinguish different lines. Solid lines mainly repeat constantly in the along-path (horizontal) direction and usually have diverse characteristics in the perpendicular (vertical)

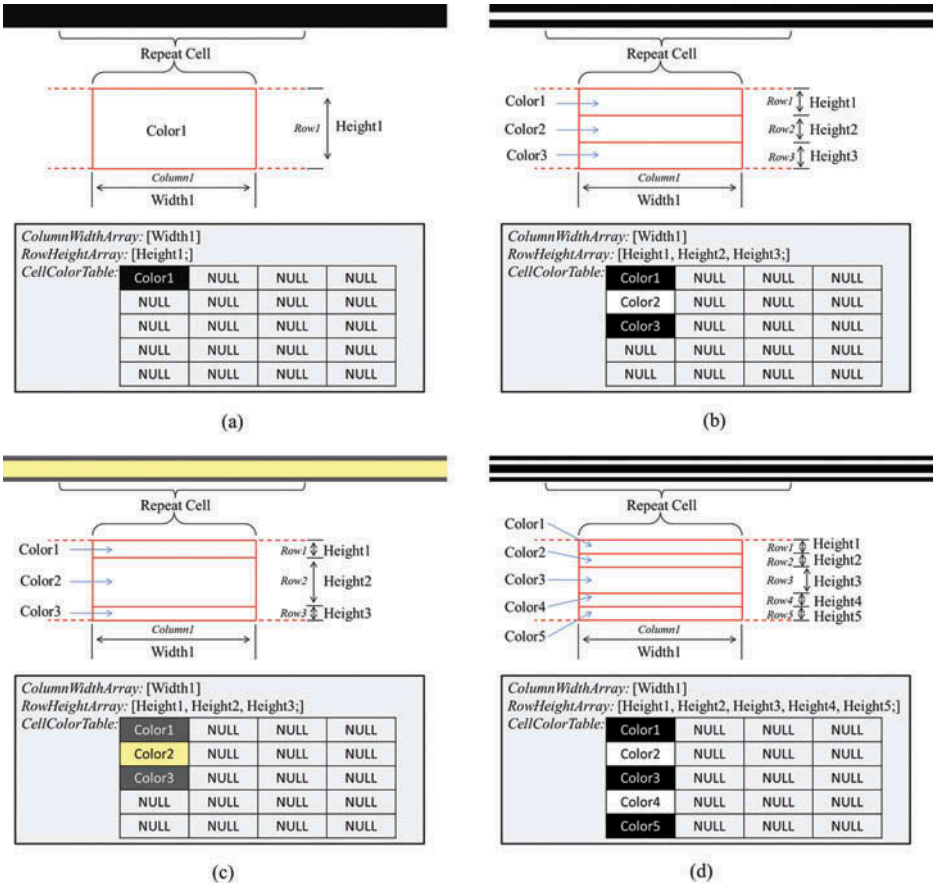

Figure 8. Example of solid lines: (a) single solid line, (b) double solid line, (c) single solid line with outline, and (d) thin-thick-thin triple solid line.

direction. Thus, the size of the column array is 1, the size of the row array may be diverse (1 to n) according to the specific line type, and the color table's size is $1 \times n$.

To ensure that the designed program can be reused for multiple symbols, the column width array, row height array, and cell color table are defined with a maximum size. In the example shown in Figure 8, the maximum size of the column width array is 4, the row height array maximum is 5, and the maximum of the color table is 4×5. Because the horizontal filling of solid lines keeps the same, the single width can be assigned as 1 unit. For a single solid line, which is shown in Figure 8(a), *ColumnWidthArray* has only one value (Width1), *RowHeightArray* has only one value (Height1), and *CellColorTable* also has only one value (*color1*). A double solid line is shown in Figure 8(b), a single solid line with an outline is shown in Figure 8(c), and a thin-thick-thin triple solid line is shown in Figure 8(d).

4.1.2. *Dashed lines*

Generally, dashed lines are rendered by repeated geometrical patterns, typically using spaces to separate the connected patterns. In this paper, lines filled by some type of pattern that is not constant in the horizontal direction (such as the solid lines discussed in Section 4.1.1)

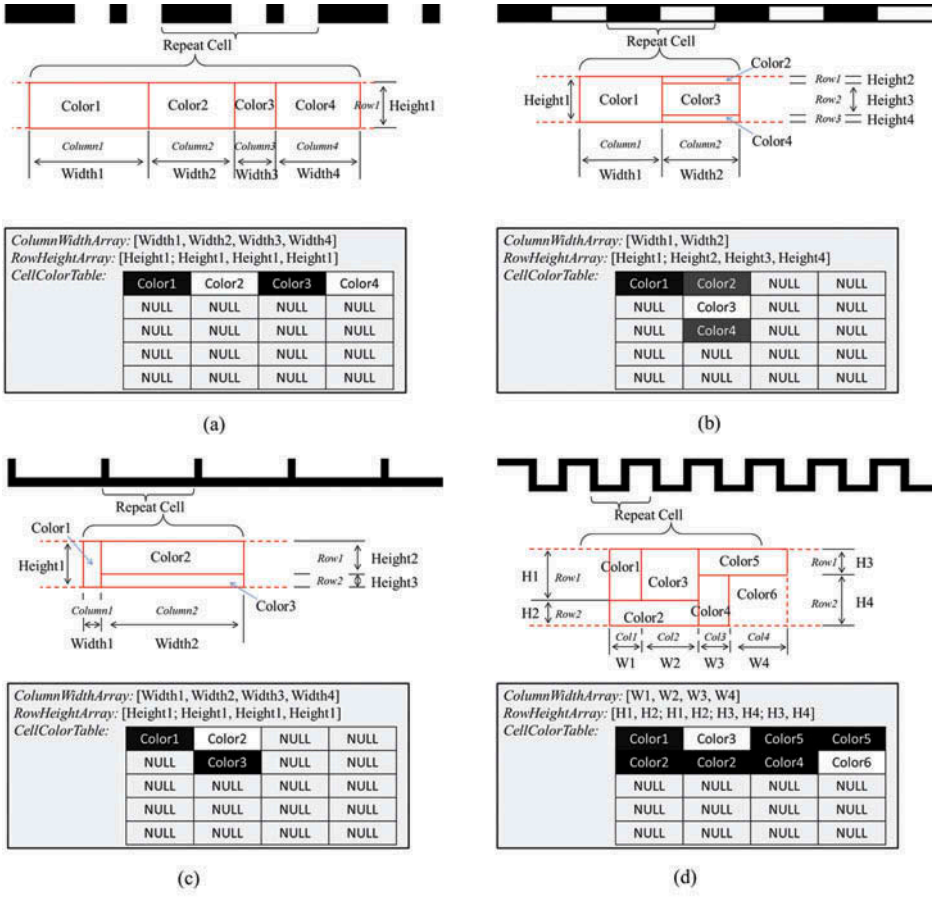

Figure 9. Example of dashed lines: (a) dash-dot line, (b) dashed line with an outline, (c) line with a vertical segment, and (d) right angle curve line.

are all treated as dashed lines. As shown in Figure 9(a), dash-dot lines are normal dashed lines. Meanwhile, dashed lines with an outline (which is commonly used to represent a railway), lines with vertical segments (which are commonly used to represent a boundary), and right-angle curve lines (which are commonly used to represent city walls) are also considered to be general dashed lines. Each of these types of lines is constructed with repeated patterns, which can be described using the designed column-row-color methods. For dash-dot line, *ColumnWidthArray* consists of the width of four segments, corresponding to the long line, space, dot, and space. Moreover, *RowHeightArray* is constructed by four one-dimensional arrays, which represent the height of each column. *CellColorTable* is built as a 1 × 4 array to record each cell's color.

4.1.3. *Gradient color lines*

Gradient color lines are commonly used to represent linear elements that are characterized by a gradually changing attribute, such as the country coastline and traffic lines symbolizing speed. To implement the rendering of gradient color lines, the *CellFillType* variable is

employed, which is an $n \times n$ array that stores flags for the fill type in each cell. Flag 1 indicates that the fill type is solid, and flags 2–5 indicate vertical, horizontal, rectangular, and custom gradients. *CellFillType* can be extended to represent different fill types, and the shader program is implemented to correspond with the definition of the fill types. In Figure 10, typical gradient color lines are presented to demonstrate the building method. There are four *CellColorTable* variables that are extended to describe the four vertexes of a line segment. For vertical gradient color lines, horizontal gradient color lines, and rectangle gradient color lines, only the first two tables (*CellColorTable1* and *CellColorTable2*) are used to construct the symbols. For vertical gradient color lines, the *V*-direction percentage can be computed using the *getRowByV()* function, and the color of the current pixel can be computed using a linear function: *Color* = *CellColorTable1*[*index*] + (1 − *v_Percent*) × *CellColorTable2*[*index*]. For horizontal gradient color lines, the *U*-direction percentage can be computed using the *getColumnByU()* function. The linear function is *Color* = *CellColorTable1*[*index*] + (1 − *u_Percent*) × *CellColorTable2*[*index*]. The color function can be customized, as in Figure 10(c): *Color12* = *CellColorTable1*[*index*] + (1 − *u_Percent*) × *CellColorTable2*[*index*];

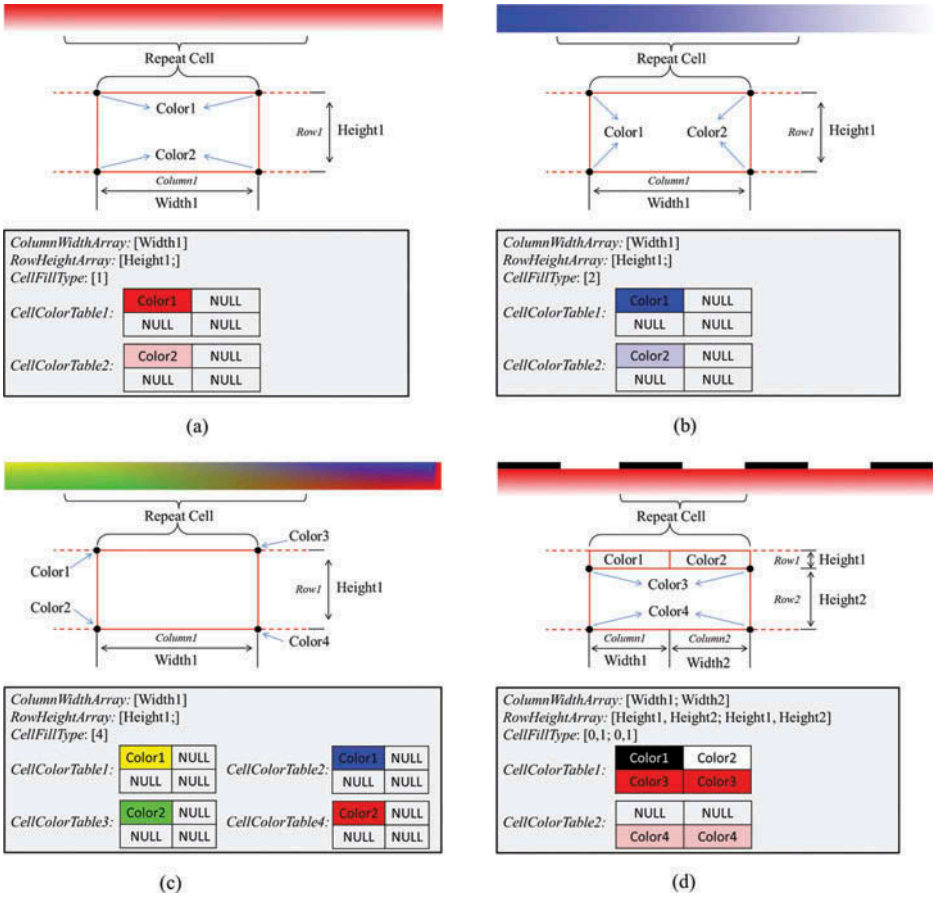

Figure 10. Example of gradient color lines: (a) vertical gradient color line, (b) horizontal gradient color line, (c) custom gradient color line, and (d) gradient color line with dashed side line.

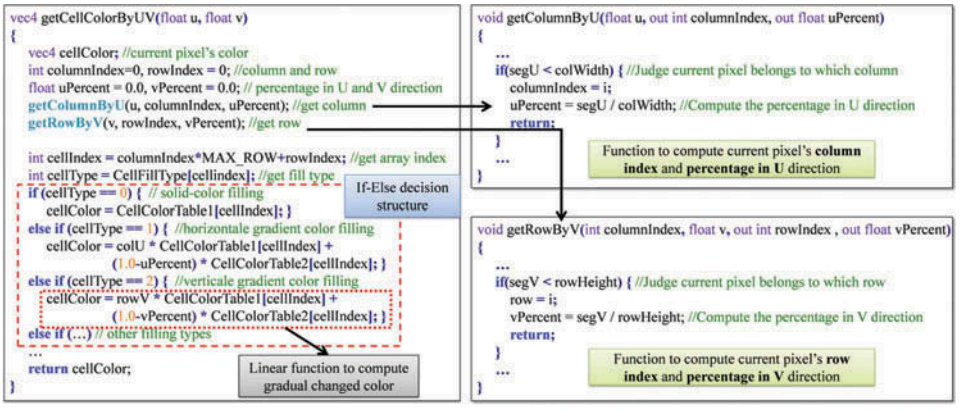

Figure 11. Sample shader program used to handle different fill types.

Color34 = *CellColorTable3*[*index*] + (1 − *u_Percent*) × *CellColorTable4*[*index*]; and *Color* = *Color12* + (1 − *v_Percent*) × *Color34*.

Figure 10(d) shows a gradient color line with a dashed side line. In this case, the symbol is constructed with two columns and two rows: *ColumnWidthArray* is [Width1, Width2], *RowHeightArray* is [Height1, Height2; Height1, Height2], *CellFillType* is [0, 1; 0, 1] (0 indicates solid filling, whereas and 1 indicates gradient filling), *CellColorTable1* is [*color1, color3; color2, color3*], and *CellColorTable2* is [NULL, Color4; NULL, *color4*].

The shader program is implemented using an *If-Else* decision structure (see the right portion of Figure 11). The functions *getColumnByU()* and *getRowByV()* are extended to obtain the *U*- and *V*-direction percentages (left portion of Figure 11). The program segment (*getCellColorByUV*) shown in Figure 11 is supplemented with the integral shader program shown in Figure 7.

4.1.4. *Transition lines*

Transition lines are commonly used to represent linear map elements (the line width changes gradually). Generally, the width of transition lines changes along the center-line, and the line width can be computed using a *U* length-based function. In Figure 12, a simple transition line is employed to discuss the method of building such lines. Along with the line direction, the line width gradually decreases following a linear function. As shown in Figure 12(a), the width can be computed as an isosceles trapezoid: *W* = *Width2* + *(Width1* − *Width2)* × *(MaxULength* − *u)*/*MaxULength*. In this function, *Width1* and *Width2* are the top and end widths of the line, *W* is the width at any position, *MaxULength* is the maximum *U* value, and *u* is the *U* value of the current pixel.

To implement the rendering of such a line, a flag, i.e., *IsWidthScale*, is added to the shader program to indicate whether a line is a transition line. *Scale1* and *Scale2* are added to describe the scale of the line top and line end, and *MaxULength* is added to convey the maximum *U* value to the shader program. As shown in Figure 12(b), the width scale can be computed using the same function introduced in Figure 12(a). For a position in such a line, the width is represented by the *cb* segment. The original *V* value is in [0, 1]; thus, the *cb* segment should be expanded into [0, 1]. For any pixel that the

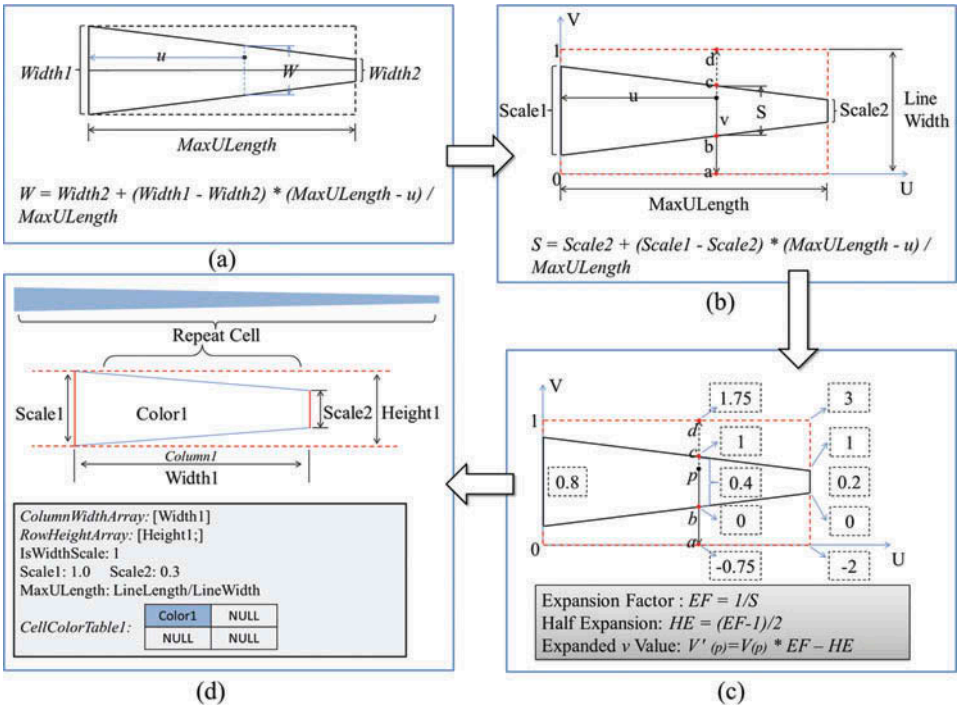

Figure 12. Basic method for constructing a transition line.

shader program iterates on, its V value must be expanded according to the computed *Scale* value. Therefore, the area outside the simplified isosceles trapezoid can be drawn as transparent.

In Figure 12(c), the expansion factor (*EF*) is introduced to explain the computation method. The expansion factor is the reciprocal of the current position's width scale, and it indicates how much the original V value should be expanded. As the line width decreases based on the centerline, there are two transparent areas outside the line, which are presented as segments *ab* and *cd*. After expanding the original V value, the half expansion (HE) should be removed. The final function is demonstrated in Figure 12(c): $V'_{(p)} = V_{(p)} \times EF - HE$. Based on this function, the V value of each pixel inside the line is in [0, 1] and is <0 or >1 if it is outside the line. The final result is presented in Figure 12(d).

Based on the above analysis, the shader program can be designed using *If-Else* statements based on the *IsWidthScale* flag value (the sample shader program is shown in Figure 13). The original function introduced in Figure 7, i.e., *getColorByUV()*, is extended. First, the cell color is computed according to the function *getCellColorByUV()* (see Figure 11); then, if the line symbol type is a transition line (which means that *IsWidthScale* is equal to true), the expansion factor is computed according to the function introduced in Figure 12(c) and an *If-Else* judgment is executed to determine whether the current pixel belongs inside or outside the line. Finally, the correct color is returned.

```
uniform float MaxULength; //The maximum value of U
uniform int IsWidthScale; //A flag indicates if the width is gradually changed
uniform float Scale1; //The scale of line top
uniform float Scale2; //The scale of line end
vec4 nullColor = vec4(0.0, 0.0, 0.0, 0.0); //Transparent color

vec4 getColorByUV(float u,float v)
{
   vec4 resultColor; //Current pixel's color
   resultColor =getCellColorByUV(u,v);
   if (IsWidthScale == 1) { // When current line's width changed
      float ef = 1.0 / (Scale2 + (Scale1-Scale2)*(MaxULength-u)/MaxULength);
      float expandedV = v * ef  - ef / 2.0; //Get the expanded V value
      if (expandedV<0.0 || expandedV > 0.0) //Outside the line
         return nullColor;
      else //Inside the line
         return resultColor ;
   }
   else {//When current line is normal, the width is not changed
      return resultColor ;
   }
}
```

Referenced in figure 11

Line width changed

If-Else judgment:
Inside Or Outside

Figure 13. Sample shader program used to handle transition lines.

4.2. *Line markers*

Line markers are commonly used to enhance visualization and represent more useful information in linear map elements, such as the driving direction of city roads. To implement the rendering of line markers, *MarkerNum* is introduced to indicate how many types of markers are in the line (in the example given in Figure 14, the maximum value is defined as *MAX_MARKER_NUM*, 4). *MarkerInterval* (the spacing distance between two markers), *MarkerOffset* (the distance between the line top and the first

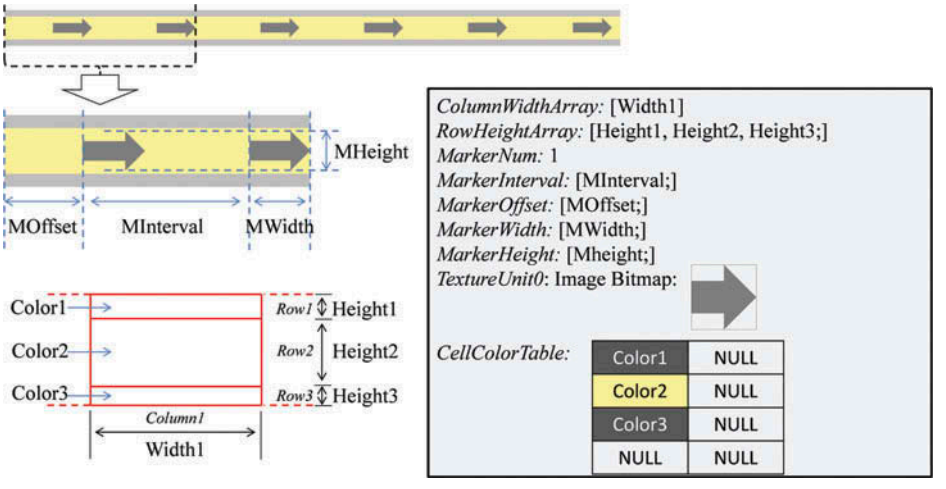

Figure 14. Implementation method of line markers.

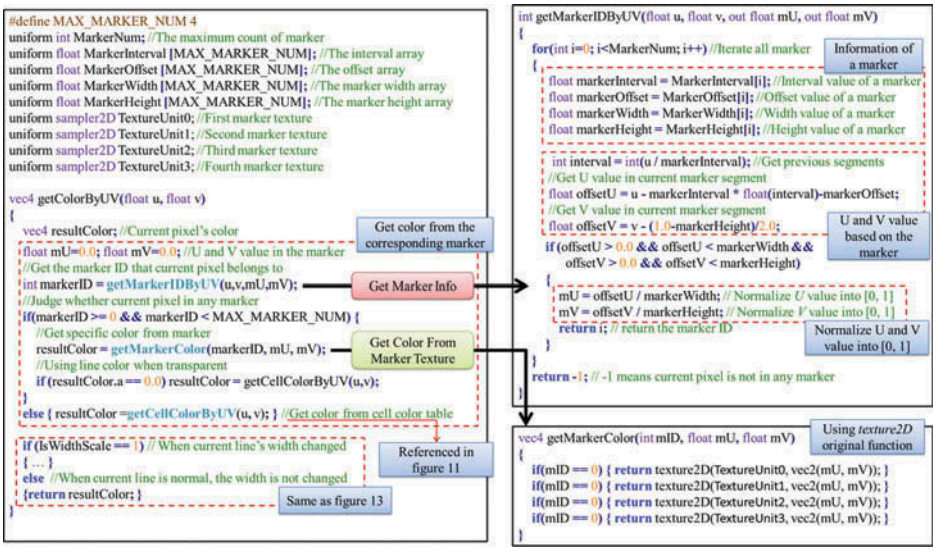

Figure 15. Extended *getColorByUV()* function to handle line markers.

marker), *MarkerWidth* (the width of a marker), *MarkerHeight* (the height of a marker) are four arrays that store the information of each marker type (as shown in Figure 14). The marker's graphics are described by texture, which can be loaded into the GPU and drawn quickly. The other parameters of the sample line in Figure 14 are the same as in the example shown in Figure 8(c) in Section 4.1.1.

In Figure 15, the extended *getColorByUV()* function is presented. Four textures are defined to convey the texture of all markers to the shader program. The function *getMarkerIDByUV()* is designed to obtain the index of the marker that the current pixel belongs to. If the current pixel does not belong to any marker, −1 is returned to indicate that the color of the current pixel should be selected from *CellColorTable*. In the top-right portion of Figure 15, the *getMarkerIDByUV()* function is shown. The new U and V values of the current pixel located in the marker are computed to assist in determining the color from the marker texture. As presented in the bottom-right portion of Figure 15, using the original *texture2D* function, the current pixel's color can be computed. In the left part of Figure 15, the *getColorByUV()* function is extended by adding the judgment of the marker color before obtaining the color from *CellColorTable*.

4.3. *Line cap and line join*

To use various linear map symbols in drawing a line string, it is essential to handle line join and line cap types, which can significantly change the shape of a line string. According to the SLD specifications and other graphics rendering libraries (such as Cairo, anti-grain geometry (AGG), and graphics device interface plus (GDI+)), there are four basic line join types (i.e. none join, miter join, butt join, and round join) and four basic line cap types (i.e. flat cap, square cap, triangle cap, and round cap). The geometric building process of these line join types and line cap types can be implemented in the CPU. Once the construction is completed, the results can be 'passed' to the GPU for rendering. For an always-visible line string, the map actions (such as move, zoom in, and zoom out) should not require rebuilding.

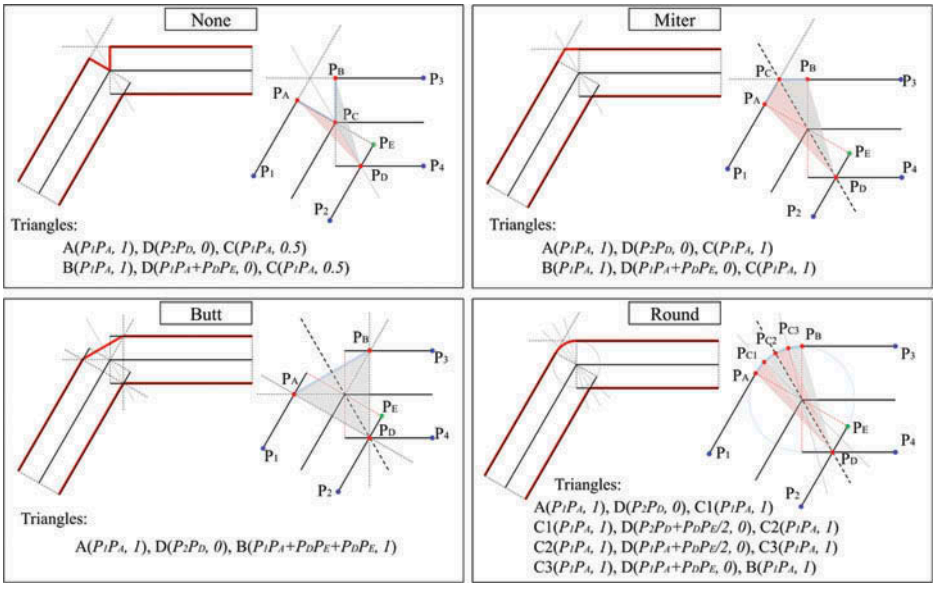

Figure 16. Typical line join types and their simple implementations.

For a line segment, the *V* parameter is defined between [0, 1], and the *U* parameter is normalized into [0, *LineLength/LineWidth*]. However, for the corner of a line string, the *U* and *V* values cannot be simply assigned as 0 or 1 or *LineLength/LineWidth*, because the points of the line join are not always parallel to the original centerline. In Figure 16, four line join types are presented to explain how to handle the *U* and *V* values in different situations.

As with the tessellation of a line segment, line joins are converted to triangles before 'passing' to the GPU for rendering. Each point in the line joins is assigned *U* and *V* values according to the coordinates. The example lines in Figure 16 are based on the angular bisector algorithm for building line joins, and every point that is used to build triangles is designated with a red circle.

For the none join type, there are four points: P_A, P_B, P_C, and P_D, and the final line is constructed as P_A-P_D-P_C and P_B-P_D-P_C. In the P_A-P_D-P_C case, *V* is equal to 1.0 and *U* is equal to P_1P_A/P_AP_E for P_A; *V* is equal to 0.0 and *U* is equal to P_2P_D/P_AP_E for P_D; and *V* is equal to 0.5 and *U* is equal to P_1P_A/P_AP_E for P_C. In the P_B-P_D-P_C case, P_B and P_C are the same as P_A-P_D-P_C; however, for P_D, *V* is equal to 0.0 and *U* is equal to $(P_1P_D + P_DP_E)/P_AP_E$.

For the miter join type, the construction area is P_A-P_D-P_C and P_B-P_D-P_C. In this case, the *U* and *V* values are the same as the none join type, although the *V* value of P_C is 1.0. Because the *U* and *V* values of P_A, P_B, and P_C are all the same, the area P_A-P_B-P_C-P_D and the line direction can be drawn continuously.

The round join type is similar to the miter join case, although there are middle points in the join area: P_{C1}, P_{C2}, and P_{C3}. All the middle points share the same *U* and *V* values with *PA* and *PB* (P_1 P_A/P_AP_E and 1.0). In this example, only four fans are used to divide the round join area; more points can be used to obtain a better representation using the same *U*-*V* assignment method.

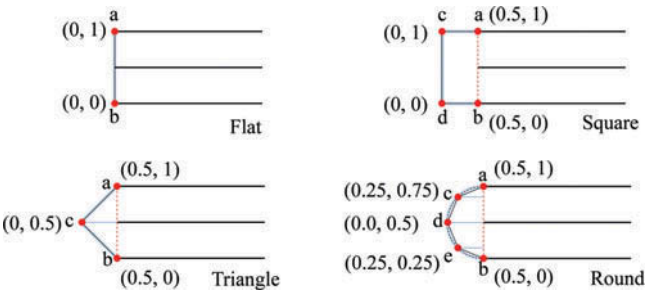

Figure 17. Typical line cap types and their simple implementations.

The butt join type has only three points in the join area. P_A and P_B have different U values. P_A is $P_1\ P_A/P_AP_D$, whereas P_B is $(P_1\ P_A+P_DP_E+P_DP_E)/P_AP_D$. For P_D, the value is $(P_1\ P_A/P_AP_D,\ 0.0)$.

Regarding the handling of line caps, the method involves setting offsets into the original U values for the start points and end points of a line, and the points in the line cap area are U and V values appended according to the specific shape of the line cap. In Figure 17, typical examples for the top of a line are presented. For a square line cap, the extended points c and d are assigned U and V values of (0, 1) and (0, 0), and the original start points a and b are assigned values of (0.5, 1) and (0.5, 0). For a triangle line cap, the extended point c is assigned a value of (0, 0.5), and for a round line cap, three points are added to simulate a half circle with c, d, and e having values of (0.25, 0.75), (0, 0.5), and (0.25, 0.25) (this paper uses three points to represent round caps, although more points can be used in practical implementation). Similarly, the end of a line can be handled in the same way, which should assign the U value by appending the U value of the line length and the U value of the line top.

According to the above analysis, by computing U and V values for every added point, the line join areas and line cap areas can be filled using the same function in the shader program.

5. Experiment and evaluation

To validate the capabilities and practicability of the proposed method, an experimental map rendering system was designed. The system was realized using the C++ programming language and implementing the shader program for this function-based concept in OpenGL ES 2.0. The test data were downloaded from the OpenStreetMap website.

5.1. Experiment on the capability of drawing various linear map symbols

Based on the presented function-based method, this study implemented a linear map symbol library to support the map rendering systems. Figure 18 introduces several commonly used linear map symbols that are stored in the developed symbol library. Typical road symbols are presented in Figure 18(a); boundary symbols (dashed lines) are presented in Figure 18(b); and railway symbols are presented in Figure 18(c). In Figure 18 (d), gradient-color linear symbols and variable-width linear map symbols are introduced.

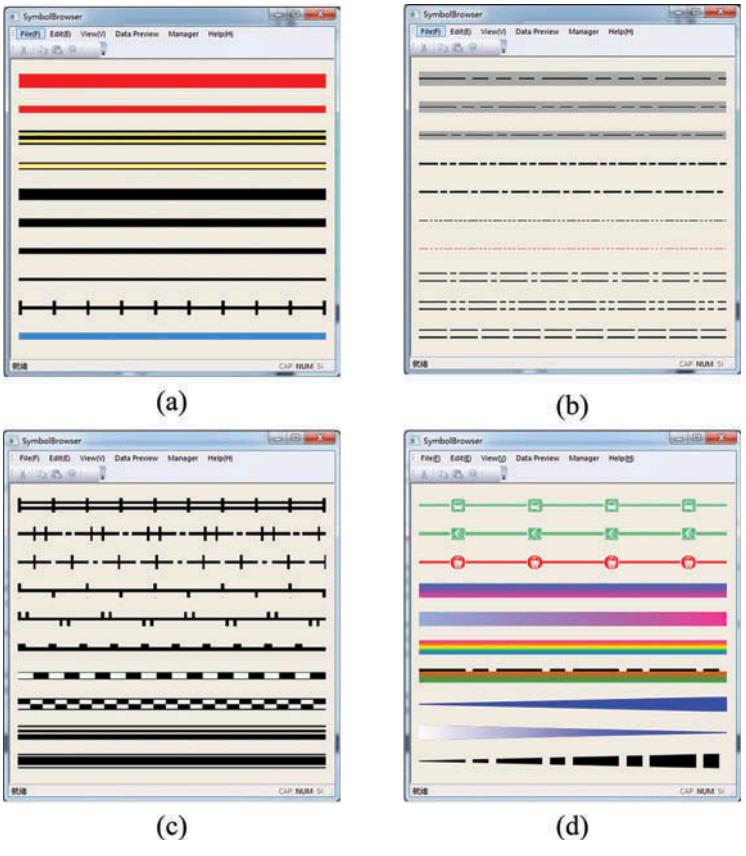

Figure 18. Sample linear map symbols generated using the function-based method.

In Figure 18(a), the symbols (numbers 5–7) represent line width difference; the second symbol and the last symbol show the line color differences. In Figure 18(b) and (c), various line types are shown to represent different linear objects. In Figure 18(d), the symbols (numbers 1–3) represent different line markers.

In addition, using the proposed method, complex linear map symbols can be drawn one at a time. As shown in Figure 19, the gradient color line as well as the dashed line

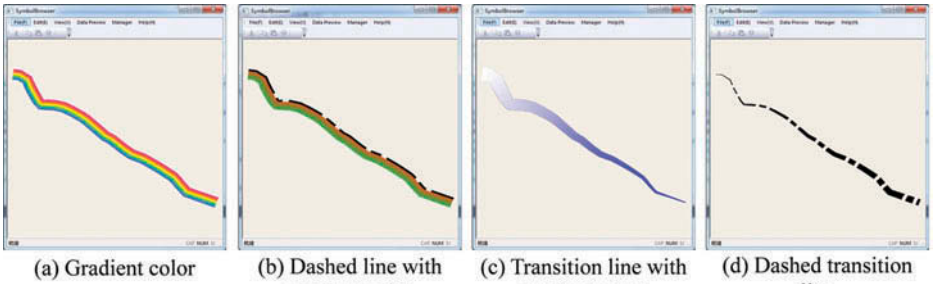

Figure 19. Rendering results of several linear symbols.

with gradient color can both be drawn in one operation instead of using two or more parallel lines to simulate the gradient effects. In commonly used cartographic tools or platforms (such as ESRI ArcGIS, MapInfo, and DotSpatial), transition lines (width changing along the line direction) must be processed by linking segments of different widths. With the presented method, transition lines can be drawn more smoothly. As shown in Figures 18(d), 19(c) and 19(d), all gradient color lines and dashed lines can be controlled by gradually changing the width.

5.2. *Experiment on the rendering efficiency*

Four frequently used linear symbol types (solid lines with an outline to represent roads, dashed lines to represent railways, transition lines to represent natural rivers, and gradient color lines to represent boundaries) are used to compare the time costs for each rendering method. Polyline data were used in this experiment to compare the rendering time of three graphics drawing methods: GDI+, AGG and the method proposed herein. Although using the GPU to draw graphics has already been recognized as being significantly faster than both GDI+ and AGG, the comparison can also explain the detailed drawing results and effects of the proposed method.

In Figure 20, the rendering results using the function-based method with these four different line symbols are presented. The *Huanghe* River (top) and the *Changjiang* River (bottom) are used as the sample data in this experiment. There are 994 (*Huanghe*) and 989 (*Changjiang*) vertexes in the two sample lines.

Because the rendering efficiency of the GPU is typically determined by the frames per second (fps) value, an experiment to test the time cost of the proposed method was conducted in the drawing process for every frame. To simulate real map operations (such as moving, zooming in, and zooming out), every line was rendered 1000 times. The test environment was Windows7, Intel i5, and an Intel HD Graphics 3000 GPU.

Table 1 shows the time cost for rendering four sample line types. The experiment was conducted 10 times to obtain the average time cost results. As shown in Table 1, the method proposed in this paper can significantly improve the rendering efficiency. For the rendering of simple roads, the method proposed in this paper is much faster than using the

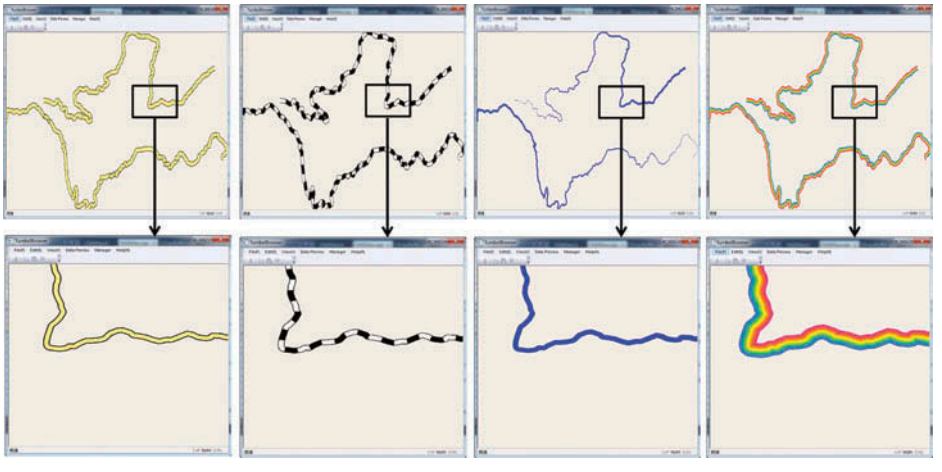

Figure 20. Symbolization result of the sample polylines.

Table 1. Time cost comparison for rendering 1000 times based on different methods (in milliseconds).

Line type	GDI+ method	AGG method	Proposed method
Road	10838	801	499
Railway	10816	741	511
River	6543	661	531
Boundary	22141	756	546

GDI+ or AGG methods, reducing the time cost by approximately 95% and 37% compared with the GDI+ and AGG methods, respectively. The comparison results are very nearly the same for the other line types. In addition, the results show that the time costs of rendering different line types using the proposed method are relatively stable. GDI+ method and AGG method would consume more time if a symbol needs to be drawn using two or more lines with different colors (e.g. the road symbol and boundary symbol). And the efficiency improvement of different line types is variable because of the detailed program implementation and the complexity of specific linear symbols.

The improvement in drawing efficiency is primarily due to hardware acceleration, and the difference in the drawing process also leads to improved efficiency. The implementation of the GDI+ and AGG methods is based on the combined drawing method, which is widely studied in map rendering (such as open source projects: SAGA, QGIS, and MapWindow). For river symbols, multiple segments with various line widths should be used to simulate the gradient effects. Unlike the GDI+ and AGG methods, the presented function-based method conducts its drawing tasks using a single process. The color of a pixel (inside of the polyline) is computed by a symbol-related function. River symbols can only be drawn one at a time, and the gradient result is smoother than using multiple segments to simulate varying widths (see in Figure 19).

5.3. Experiment on the application for map rendering

Based on the rendering architecture introduced in Section 2, a prototype map rendering system was developed to verify the practical use of the proposed method. Figure 21 shows several snapshots of the map rendering system.

In this map rendering system, several linear map symbols are used to represent a variety of spatial objects: administration outlines, city roads with different ranks, and railway paths and tunnels. The drawing of linear map elements can be controlled via the map scale. The width of railway symbols is relatively fixed and does not change with the map scale. However, the width of city road can change. This experiment indicates that the proposed method can render a variety of linear map symbols and can be used to present different map scenes.

6. Conclusion and future work

The purpose of this study is to improve the efficiency of drawing various linear map symbols and to further support the development of 2D/3D integrated map applications. By taking advantage of the proposed function-based linear map symbol building and rendering method, linear map symbols can be flexibly constructed and drawn on a per-pixel basis. With this method, different line types can be implemented, and line joins, line caps, and line markers can also be easily handled. The shader programs in this paper are

Figure 21. Prototype map rendering system.

primarily focused on explanation, and this function-based method can be extended to a variety of drawing effects.

However, because map rendering and visualization research are synthetic work, future research is needed, especially regarding the following aspects:

(1) Anti-alias processing of lines. To achieve smooth results, anti-alias processing is necessary. In this paper, main attention is placed on constructing and rendering different types of linear map symbols; the implementation of anti-aliasing is not discussed. Based on the proposed function-based method for drawing lines, the basic concept of anti-aliasing involves using the color of a specific transparency to fill pixels in the buffer of the line edge. This idea could be implemented in the fragment shader program.
(2) Tessellation of line joins. Because line joins are important for the shape of a line with a certain width, the line joins should be tessellated and constructed to form an entire line. This paper introduces the U and V attributes for every key vertex in a line join, although the tessellation work must be explored in depth, and a proper tessellation method should be developed to meet the demands of drawing quality and efficiency.
(3) User interface for building linear map symbols. Although the proposed linear map symbol building and rendering method can be implemented to draw lines of

different line types, significant programming work must be performed to formulate the shader program. To reduce the effort required for the programming work and to make it more convenient for users to design maps, a strategy that can support the reuse of different drawing methods that have been implemented with the shader language should be designed and developed.

(4) Applying for building different map rendering platforms. The proposed method can be implemented with OpenGL ES, and using this method to develop mobile map rendering applications needs to be evaluated (e.g., the effects on battery life/power consumption of using GPU computation versus CPU computation). The detailed implementation of shader program can be optimized and extended to adapter to more complex software/hardware conditions (e.g., the generalization methods should be considered to help improve the drawing effects).

Acknowledgments

We appreciate the detailed suggestions and comments from the editor and the anonymous reviewers.

Disclosure statement

No potential conflict of interest was reported by the authors.

Funding

The work described in this article involves many geo-analysis models and was supported by the following research programs: the National Basic Research Program of China (973 Program) [grant number 2015CB954102]; the National Natural Science Foundation of China [grant number 41471317], [grant number 41371424]; the Priority Academic Program Development of Jiangsu Higher Education Institutions.

References

Bae, W.D., *et al*., 2015. Optimizing map labeling of point features based on an onion peeling approach. *Journal of Spatial Information Science*, 2015 (2), 3–28

Bandrova, T.L., Konecny, M., and Yotova, A., 2014. Cartography development and challenges on the basis of big data. *In*: *5th international conference on cartography and GIS*. Sofia: Bulgarian Cartographic Association, 164–173.

Buschmann, S., *et al*., 2014. Hardware-accelerated attribute mapping for interactive visualization of complex 3D trajectories. *In*: *Proceedings of the 5th international conference on information visualization theory and applications*, Lisbon, Portugal, 355–363.

Chan, E. and Durand, F., 2005. *Fast prefiltered lines* [online]. GPU Gems 2. Indianapolis, IN: Addison-Wesley. Available from: http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter22.html [Accessed 13 June 2015].

Chen, M., *et al*., 2015. An object-oriented data model built for blind navigation in outdoor space. *Applied Geography*, 60, 84–94. doi:10.1016/j.apgeog.2015.03.004

Chen, M., Wen, Y., and Yue, S., 2014. A progressive transmission strategy for GIS vector data under the precondition of pixel losslessness. *Arabian Journal of Geosciences*, 8 (6), 3461–3475. doi:10.1007/s12517-014-1467-y

Drew, Y., 2008. A closer look at GPUs. *Communications of the ACM*, 51 (10). doi:10.1145/1400181.1400197

Dübel, S., *et al*., 2014. 2D and 3D presentation of spatial data: a systematic review [online]. *In: IEEE VIS international workshop on 3DVis (3DVis)*, 9 November 2014, Paris. IEEE, 11–18.

Available from: http://blogs.evergreen.edu/vistas/files/2015/02/dubel-topost-2dvs3d-visieeevis2014_submission_3.pdf [Accessed 13 June 2015].
Goodchild, M.F., Yuan, M., and Cova, T.J., 2007. Towards a general theory of geographic representation in GIS. *International Journal of Geographical Information Science*, 21 (3), 239–260. doi:10.1080/13658810600965271
Graham, M. and Shelton, T., 2013. Geography and the future of big data, big data and the future of geography. *Dialogues in Human Geography*, 3 (3), 255–261. doi:10.1177/2043820613513121
Häberling, C., Bär, H., and Hurni, L., 2008. Proposed cartographic design principles for 3D maps: a contribution to an extended cartographic theory. *Cartographica: The International Journal for Geographic Information and Geovisualization*, 43 (3), 175–188. doi:10.3138/carto.43.3.175
Haeberling, C., 2002. 3D map presentation – a systematic evaluation of important graphic aspects. *In*: *Proceedings of ICA mountain cartography workshop "Mount Hood"*, Timberline Lodge, 1–11.
Kersting, O. and Döllner, J., 2002. Interactive 3D visualization of vector data in GIS. *In*: *Proceedings of the 10th ACM international symposium on advances in geographic information systems*. New York: ACM, 107–112.
Kilgard, M. and Bolz, J., 2012. GPU-accelerated path rendering. *ACM Transactions on Graphics (TOG)*, 31 (6), 172. doi:10.1145/2366145.2366191
Konecny, M., 2011. Review: cartography: challenges and potential in the virtual geographic environments era. *Annals of GIS*, 17 (3), 135–146. doi:10.1080/19475683.2011.602027
Kraak, M.J. and Ormeling, F., 2011. *Cartography: visualization of spatial data*. New York: Guilford Press.
Lane, J.M., *et al*., 1980. Scan line methods for displaying parametrically defined surfaces. *Communications of the ACM*, 23 (1), 23–34. doi:10.1145/358808.358815
MacEachren, A.M., 2004. *How maps work: representation, visualization, and design*. New York: Guilford Press.
Monmonier, M., 1996. *How to lie with maps*. Chicago: University of Chicago Press.
OGC, 2012. *Styled layer descriptor* [online]. Open Geospatial Consortium Inc. Available from: http://www.opengeospatial.org/standards/sld [Accessed 13 June 2015].
Peterson, G.N., 2014. *GIS cartography: a guide to effective map design*. Boca Raton: CRC Press.
Quinn, M., *et al*., 2005. SUMIT: A virtual reality embedded user interface prototyping toolkit [online]. *In*: *Proceedings of Virtual Concept 2005*. Available from: http://members.bitstream.net/~mquinn/bae_sumit.pdf [Accessed 13 June 2015].
Robinson, A.C., *et al*., 2012. Developing map symbol standards through an iterative collaboration process. *Environment and Planning: Part B*, 39 (6), 1034. doi:10.1068/b38026
Roth, R.E., 2013. Interactive maps: what we know and what we need to know. *Journal of Spatial Information Science*, 2013 (6), 59–115.
Rougier, N., 2013. Shader-based antialiased dashed stroked polylines. *Journal of Computer Graphics Techniques*, 2 (2), 91–107.
Ruas, A., 2015. Models and methods to represent and explore phenomena on GIS [online]. *In*: *Modern trends in cartography*. Springer International Publishing, 259–267. Available from: http://link.springer.com/chapter/10.1007/978-3-319-07926-4_20 [Accessed 13 June 2015].
Rueda, A.J., De Miras, J.R., and Feito, F.R., 2008. GPU-based rendering of curved polygons using simplicial coverings. *Computers & Graphics*, 32 (5), 581–588. doi:10.1016/j.cag.2008.07.005
Semmo, A., *et al*., 2012. Concepts for cartography-oriented visualization of virtual 3D city models. *Photogrammetrie-Fernerkundung-Geoinformation*, 2012 (4), 455–465. doi:10.1127/1432-8364/2012/0131
Trapp, M., *et al*., 2014. Interactive rendering and stylization of transportation networks using distance fields. *International Journal of Geographical Information Science*, 28 (10), 2030–2051.
Turdukulov, U., *et al*., 2014. Visual mining of moving flock patterns in large spatio-temporal data sets using a frequent pattern approach. *International Journal of Geographical Information Science*, 28 (10), 2013–2029. doi:10.1080/13658816.2014.889834
Varcholik, P., 2014. *Real-time 3D rendering with DirectX and HLSL: A practical guide to graphics programming*. Indianapolis: Addison-Wesley Professional.
Wen, Y., *et al*., 2013. A characteristic bitmap coding method for vector elements based on self-adaptive gridding. *International Journal of Geographical Information Science*, 27 (10), 1939–1959. doi:10.1080/13658816.2013.774006

Wu, C., *et al*., 2014. Research on national 1:50000 topographic cartography data organization. *ISPRS Annals of Photogrammetry, Remote Sensing and Spatial Information Sciences*, 1, 83–89. doi:10.5194/isprsannals-II-4-83-2014

Zhang, J. and Zhu, Y., 2015. A method based on graphic entity for visualizing complex map symbols on the web. *Cartography and Geographic Information Science*, 42 (1), 44–53. doi:10.1080/15230406.2014.981586

Zinsmaier, M., *et al*., 2012. Interactive level-of-detail rendering of large graphs. *IEEE Transactions on Visualization and Computer Graphics*, 18 (12), 2486–2495. doi:10.1109/TVCG.2012.238