Real-time Contour Map Reconstruction with 3D Terrain on Modern Graphics Hardware

Chen Zhuo*, Zhao Yanqing, Yang Chongjun

State Key Laboratory of Remote Sensing Science, Jointly Sponsored by the Institute of Remote Sensing Applications of Chinese Academy of Sciences and Beijing Normal University, 100101, China

KEY WORDS: Contour Map, Contour Lines, 3D terrain, GPU, Digital earth

ABSTRACT:

A new algorithm for reconstructing contour map from raster DEM data is presented. It runs completely within the programmable 3D visualization pipeline in real-time. During the rendering, we first make an elevation gradient map out of original terrain vertex data. Then figure out the final contour lines with image-space processing, and blend the values on the original scene to obtain the final scene with contour map. We implemented this method in our global 3D-digitalearth platform, with interactive frame-rates and high image quality.

1. INTRODUCTION

Contour maps are important in many fields of application. Today's 3D virtual-earth systems use raster DEM data as a main terrain data source. Superimposing massive related data such as contour maps as vector or image layers are space and time consuming, requiring much storage, memory and band-width resources which are crucial to other system functionalities. It also becomes incapable when dynamic DEM data is introduced. So figuring out contour maps from the basic DEM data is considerable. Classic contour line algorithms are mostly based on CPU, taking too long time to run in real-time. In this paper we demonstrate a new algorithm completely based on modern graphics processing unit (GPU) and programmable pipelines. It generates and displays contour map at the same time on the fly.

First we prepare required data for the graphics card which will do all the computing (in fact you will find that no more work than just handle in a few parameters essential to contour maps will be done).Then the details of two main rendering passes: a intermediate pass to produce a gradient map, and the final full-screen edge-detecting pass to get the contour map and blend it onto the original terrain scene, including the GPU-program pseudo-code. Finally the performance and quality issues are discussed.

2. PRE-REQUESTED DATA AND PARAMETERS

DEM data is required. Both regular and irregular height map data can be used. In our case, the NASA *.bil* format height data are used as the source. According to projection and other translations, a regular triangle vertex data for each terrain tile with a spherical reference surface was produced. Each vertex contains a 3D-vector as its world position. Of course this leads to a floating point artifacts and some tricking correction was made. It is a routine process for terrain rendering and more details can be found on the the virtual terrain project (http://www.vterrain.org). Many terrain LOD techniques can be used, such as ROAM (Duchaineau M. et all, 1997), geo-Mipmapping (Willem H. 2000), and Geo-Clipmaps (Arul and Hoppe, 2005), which are all compatible with our contour map algorithm.

For contour lines, we have a few parameters. The Contour interval stands for the elevation interval between adjacent contour lines; Base elevation stands for the basic elevation for contour lines, usually be zero; And the Contour Line Color, which is set by users. These are set to the pipeline through constant registers.

3. FIRST RENDERING PASS: THE GRADIENT MAP

In this pass, we take the vertex position as main input as well as part of the contour line parameters. The GPU-program pseudo-code is listed in Appendix A.

We first transform the vertex from local space to world space with the local-to-world matrix, and then transform the world coordinate into post-perspective space with the view-projection matrix. The length of its absolute world-position was calculated and subtracted by the WORLD_RADIUS constant, which was introduced when setting up the terrain vertices, so we get the elevation of the vertex. The process is much alike an inverse process of terrain vertex data construction, and can be described as follows:

$$\mathbf{E} = \mathit{length}(\boldsymbol{V}) - \mathrm{R} \qquad (1)$$

where E = Elevation of vertex.
$\boldsymbol{V}$ = Vertex coord in world space, a 3D Vector.
R = Equatorial World Radius.
length() is a intrinsic function of gpu programs that returns the length of a vector.

We store this value into a TEXCOORD component of the vertex-program output, so that it will be interpolated linearly by the rasterizer automatically, and we'll get the correct elevation for each pixel in the pixel program.

In the pixel program, we divide the elevation value of each pixel by the contour interval, so that we get a gradient value which increases by one linearly for each interval. Then we get the floor of the value, so that it increases by one at the very point where the contour lines exist (Equation 2 and Figure 1).

$$\mathrm{grad} = \mathrm{floor}(\mathbf{E} / \mathbf{I}) \qquad (2)$$

where grad = gradient value.
I = Contour Interval.

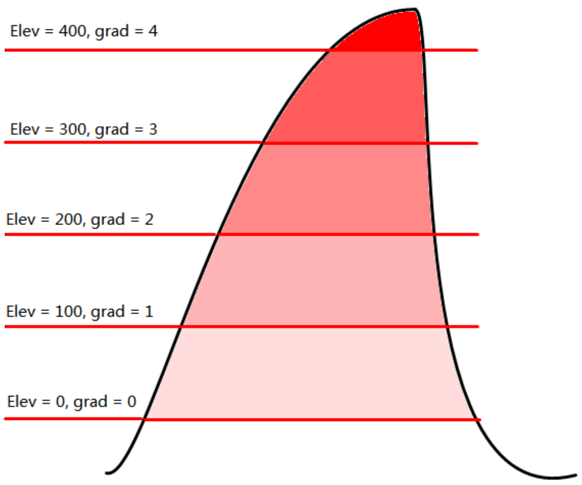


Figure 1. Assuming contour interval as 100, the floored gradient will be like this.

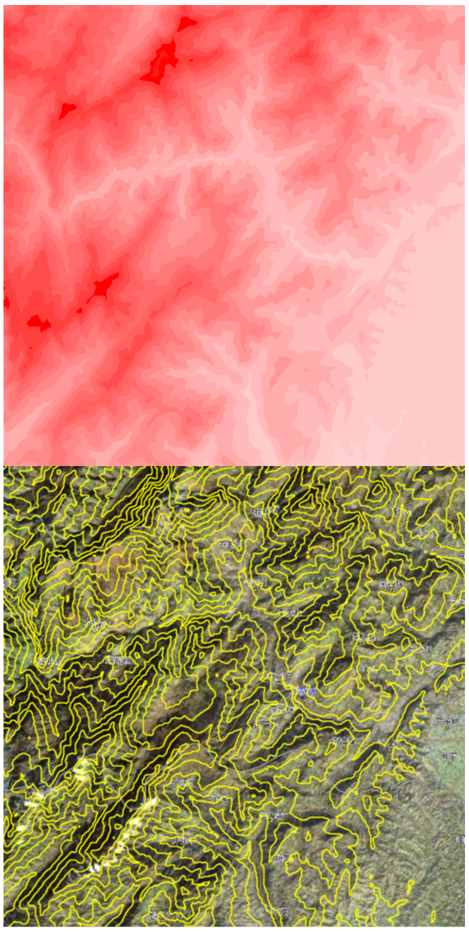


Figure 2. The visualized gradient buffer and its corresponding final contour map

This value is output as one component of the pixel-program's output color, into a floating point buffer (render-target texture in Direct3D or FBO in OpenGL) (Figure 2). The first pass is then completed.

4. SECOND PASS: EDGE-DETECTING AND BLENDING

The second pass is an image space procedure. It uses the floating point render-target buffer containing the gradient values as texture, to render a full-screen quad. The idea of full-screen pass is very common in today's 3D programs for special effects such as blooming (Greg James et all. 2004) and HDR lighting, and is relatively simple. For each pixel to draw, we sample the gradient-map texture with coordinate corresponding to the pixel itself and the pixels around it (Figure 3). Then compare those gradient values, if the current gradient differs from a neighbouring one, it indicates that there's contour line across between the two pixels, and we add a weight value for the current pixel according to the distance between the two pixels (Oles. 2005). At last, the sum weight is used as the alpha value of the output color, whose RGB channels are written with the line color which was set as contour line parameters. The pixel is written onto the original scene with alpha-blending, so that pixels without contour lines become transparent. The pseudo-code of the pixel program is shown in Appendix B.

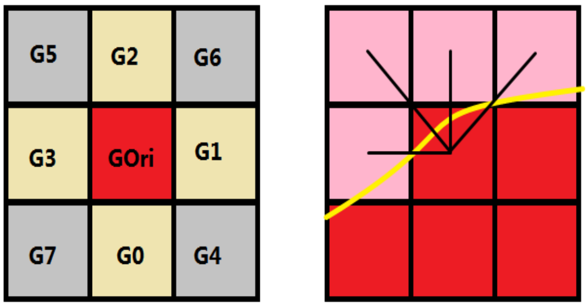


Figure 3. (Left) The sampled pixels, see Appendix B for calculation of G(n) values. Notice that G0~G3 are with the same weight value and G4~G7 are with another one. (Right) the comparing of pixel's gradient values. The four black lines show difference in gradient value of sampled pixels.

5. RESULTS AND DISCUSSION

We ran our 3D digital-earth system on some PCs with DEM and satellite image data downloaded from a remote map server dynamically, and got the results (Appendix C) in interactive frame-rates, which are measured with Fraps 2.9.6. The results on different hardware platforms are listed in Table 1.

Platform	**No.1**	**No.2**	**No.3**
Contour map off	1071 fps	341 fps	189 fps
Contour map on	542 fps	190 fps	88 fps
Detailed configuration	Intel Core2 Q6600 2.4G NVIDIA GTX280 1GB 3.25GB RAM	Intel Core2 E4600 2.4G NVIDIA GeForce 8600 GT 256 MB 2GB RAM	Intel Core2 E8200 2.66G ATI HD2400 XT 512MB 2GB RAM

Table 1. Frame-rates on different platforms.

From the table we can conclude that the most important issue that influences the frame-rates is the GPU capability, since the algorithm is CPU and RAM free. The frame-rates are stable, as the calculation procedure does not change with contour line parameters, so we get almost the same frame rate for a 200-interval contour map and a dense 50-interval contour map.

Since the terrain was cut by the virtual plane of each elevation grade, the resulting contour lines are guaranteed to be topologically correct, no manual corrections are needed.

There are some improvements that can be taken. First one is the geometry-edge artifacts, which occurs when geometry edges are present in the scene. Since the gradient value between two fragments of geometry can be different, it will be considered as neighbouring gradient edge by the edge-detector and a contour line weight will be added unexpectedly (Figure 4). To correct this issue, we can render in the first pass a second value besides gradient value, the distance between the viewer to the pixel, or the Depth for short, into another color channel of the buffer. Then in the second pass, we sample the Depth value as well as the gradient value. If the differences of the Depth value between two neighbouring pixels are larger than a barrier, it is considered as geometry-edge rather than gradient-edge, so the weight is ignored.

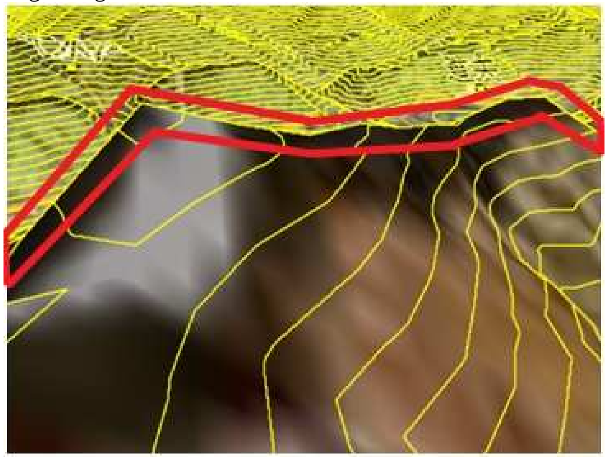


Figure 4. The geometry-edge artifacts. Notice the edge of near mountains.

Second, the first pass can be integrated into the normal rendering pass, with the help of MRTs (Multi-Render Targets). So that we can save half of the draw calls, which will obviously increase the frame-rate.

At last, contour line tags and labels can be added by reading the gradient data back from the buffer. By API features such as IDirect3DSurface9::LockRectangle() in Direct3D, we can easily get the gradient data, then calculate the contour value with the inverse process of the first pass. And show them on the map.

This method is easily implemented on most of consumers' graphics hardware, which support DirectX 9 or OpenGL 2.0 with shader model 2.0 or higher. It can be applied on to various terrain platforms, including both spherical digital-earth and localized non-spherical scenes, even the 2D maps (for which a single second pass is enough). Also, the generated contour map can be saved to hard disk for future usage, with desired plotting scale.
This method also enables a full dynamic ability to the terrain system. Once the DEM data is changed, the new contour maps will be displayed on the final image immediately, without any kind of data pre-processing.

6. CONCLUSION

In this paper we presented an algorithm for reconstructing contour map from raster DEM data for digital-earth and other terrain platforms. With the power of programmable pipelines, we make the elevation gradient map out of original terrain vertex data, and the final contour lines with image-space processing. The method was implemented and tested on different PC platforms and proved to be efficient with real-time frame-rates and high image quality. It can be easily modified and extended for better image quality and broader applications.

REFERENCE

Arul and Hoppe., *Terrain rendering using gpu-based geometry clipmaps*. GPU Gems 2. 2005, http://developer.nvidia.com/object/gpu_gems_2_home.html

Duchaineau, M., Wolinski, M., Sigeti, D., Miller, M., Aldrich, C., and Mineev-Weinstein, M. ROAMing Terrain: Real-time Optimally Adapting Meshes. http://www.llnl.gov/graphics/ROAM

Greg James. and John O'Rorke., *Real-Time Glow*. GPU Gems. 2004.

Oles Shishkovtsov., *Deferred shading in S.T.A.L.K.E.R.* GPU Gems2. 2005.

Willem H. de Boer, Fast Terrain Rendering Using Geometrical MipMapping, E-mersion Project, October 2000, http://www.connectii.net/emersion.

ACKNOWLEDGEMENTS

Thanks to xtaci for the discussion on GPU programming and a lot of fundamental work on the terrain engine and map server, and HuangShan (Kingsoft co. Ltd) for providing us many useful books and links. Thanks to Wang Gang (GeoBeans co. Ltd) for providing map services. This work is also partly supported by the MOST Program Grant 2009AA12Z215 and Grant 2008BAH23B04..

APPENDIX A. FIRST PASS GPU PROGRAMS

Vertex Program:

Constants:

```
   float WORLD_RADIUS;
   matrix matrixWorld;
   matrix matrixViewProjection;

foreach Vertex do
{
   float4 vertexWorldPosition = multiply( Vertex.Position, matrixWorld )
   Output.Position as POSITION = multiply( vertexWorldPosition, matrixViewProjection );
   Output.Elevation as TEXCOORD = length( vertexWorldPosition ) – WORLD_RADIUS;
   return Output;
}
```

Pixel Program:

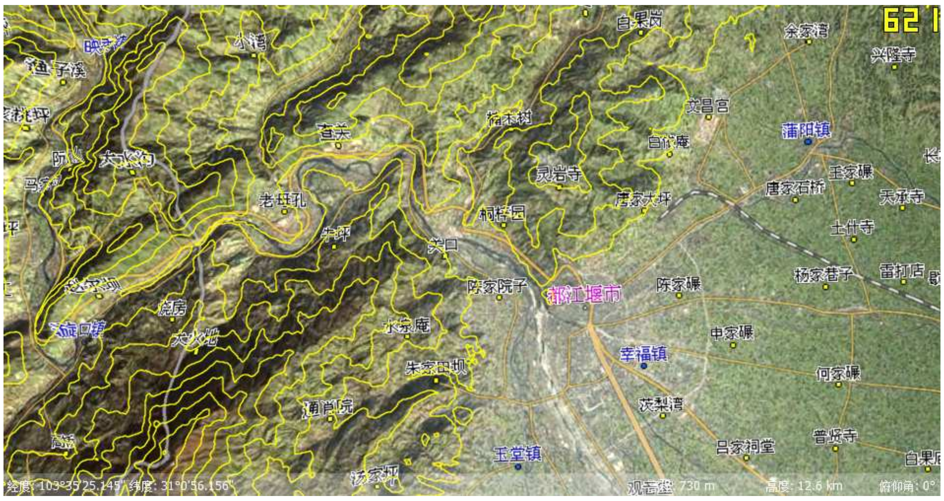
Constants:

```
   float ContourInterval;
   float BaseElevation;

foreach Pixel do
{
   float grad = (Pixel.Elevation – BaseElevation) / ContourInterval;
   float gradAsInteger = floor(grad);
   return Color, gradAsInteger as the Red component;
}
```

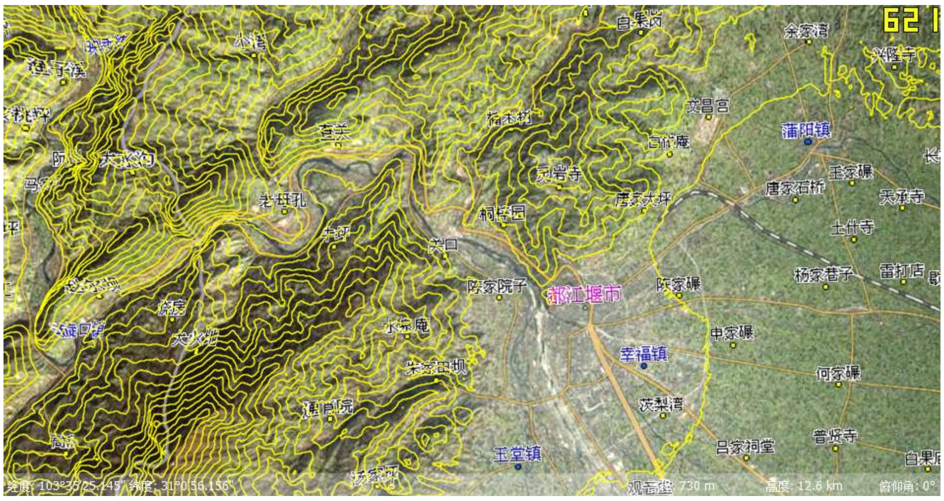
APPENDIX B. SECOND PASS PIXEL PROGRAM.

```
Pixel Program:

Constants:
   float2 DeltaUV;
   float3 LineColor;
   float ColorBarrier = 0.5f;
   float weight1 = some weight;
   float weight2 = weight1 / 2;
   Texture gradientMap;

foreach Pixel do
{
   float GOrigin = sample( gradientMap, Pixel.Texcoord);
   float G[8];

   G[0] = sample( gradientMap, Pixel.Texcoord + float2(0, DeltaUV.y) );
   G[1] = sample( gradientMap, Pixel.Texcoord + float2(DeltaUV.x, 0) );
   G[2] = sample( gradientMap, Pixel.Texcoord + float2(0, -DeltaUV.y) );
   G[3] = sample( gradientMap, Pixel.Texcoord + float2(-DeltaUV.x, 0) );

   G[4] = sample( gradientMap, Pixel.Texcoord + DeltaUV );
   G[5] = sample( gradientMap, Pixel.Texcoord – DeltaUV );
   G[6] = sample( gradientMap, Pixel.Texcoord + float2(DeltaUV.x, -DeltaUV.y) );
   G[7] = sample( gradientMap, Pixel.Texcoord - float2(DeltaUV.x, -DeltaUV.y) );

   float alpha = 0;
   foreach g in G[0] to G[3] do
   {
      if( abs(GOrigin – g) > ColorBarrier)
      {
         alpha += weight1;
      }
   }
   foreach g in G[4] to G[7] do
   {
      if( abs(GOrigin – g) > ColorBarrier)
      {
         alpha += weight2;
      }
   }
   return Color, LineColor as RGB, alpha as A;
}
```

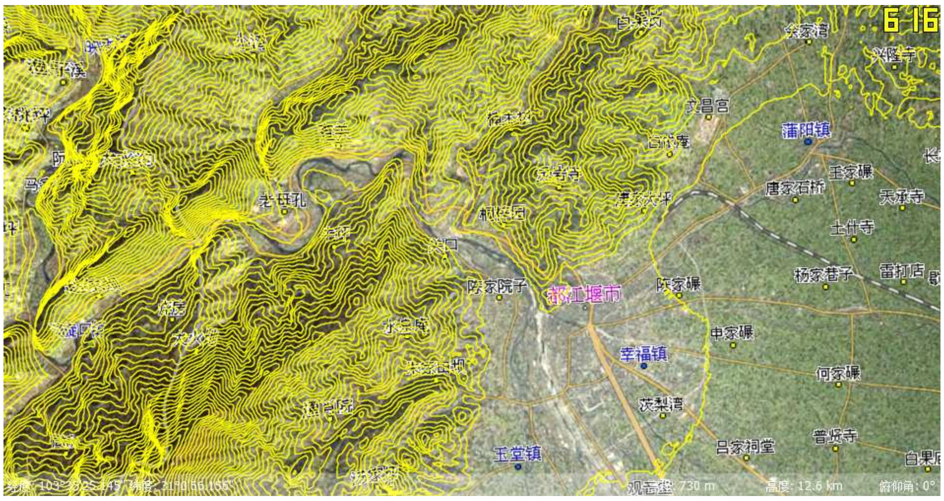
APPENDIX C. IMAGE RESULTS.



(1) contour interval = 200



(2) contour interval = 100



(3) Contour interval = 50