Concurrent Drainage Network Rendering for Automated Pen and Ink Style Landscape Illustration

Author:

James E. Mower

Department of Geography and Planning

University at Albany

Short title for running head:

Concurrent Rendering for Landscape Illustration

Key words:

non-photorealistic rendering, cartography, landscape, crease, drainage

Corresponding author's address:

James E. Mower

Dept. of Geography and Planning, AS 218

University at Albany

Albany, NY 12222

jmower@albany.edu

Abstract

Pen and ink style geomorphological illustrations render landscape elements critical to the understanding of surface processes within a viewshed and, at their highest levels of execution, represent works of art. Although practical and beautiful, the execution of a pen and ink composition requires inordinate amounts of time and skill. This paper will introduce an algorithm for rendering creases—linework representing visually significant morphological features—at animation speeds, made possible with recent advances in graphics processing unit (GPU) architectures and rendering APIs. Beginning with a preprocessed high-resolution drainage network model, creases are rendered from selected stream segments if their weighted criteria (slope, flow accumulation, and surface illumination), attenuated by perspective distance from the viewpoint, exceed a threshold. The algorithm thus provides a methodology for crease representation at continuous levels of detail down to the highest resolution of the preprocessed drainage model over a range of surface orientation and illumination conditions. The paper also presents an implementation of the crease algorithm with frame rates exceeding those necessary to support animation, supporting the proposition that parallel processing techniques exposed through modern GPU programming environments provide cartographers with a new and inexpensive toolkit for constructing alternative and attractive real-time animated landscape visualizations for spatial analysis.

1. Introduction

Since the woodcut landscapes of Murer in the 16th century, linework representations of physical surfaces have been valued for their uncluttered

representations of natural features (Imhof, 2007[1], pp. 3-7). Field geomorphologists still use line drawing today as a simple and effective tool for the analysis of surface features, following Holmes (Fernlund, 2000), Lobeck (1958), Raisz (1938), Imhof (2007), and others who raised the form to an art. This paper introduces an algorithm for the automated real-time rendering of creases—linework that helps an observer understand local surface curvature independently of her point of view—at speeds exceeding those required for animation. It also presents an implementation of the algorithm in the context of a comprehensive pen and ink style rendering system that includes a technique for fast silhouette rendering introduced in Mower (2014). Silhouettes are linework elements that mark the edge of a feature superimposed on a background feature or the sky from the observer's point of view (Figure 1). When presented together with creases, the resulting illustrations imitate much of the composition of a hand drawn image. In that regard, automated pen and ink style landscape illustrations should be able to perform the same roles as their manually constructed counterparts. One goal of this work is to promote their use as alternatives to the shaded relief and draped photographic forms that currently dominate landscape rendering practice. Whether as aids to structural geological interpretation or as works of art in their own right, automated pen and ink style illustrations offer striking and complementary visual elements, expressive of the utility and artistic qualities of traditional hand-drawn pen and ink illustrations, in real-time, interactive environments. When scenes are rendered at animation frame

[1] ESRI Press republished Imhof (1982) *Cartographic Relief Presentation* in 2007. See References for a complete citation. All page references in this text refer to the 2007 edition.

rates, pan, tilt, and zoom operations appear continuous throughout changes in viewing position and orientation, enabling users to gather varied impressions of a landscape from multiple points of view without encountering cumbersome delays. Furthermore, given adequate camera position and attitude data, rendered scenes could function as augmented reality overlays on real-time video capture.

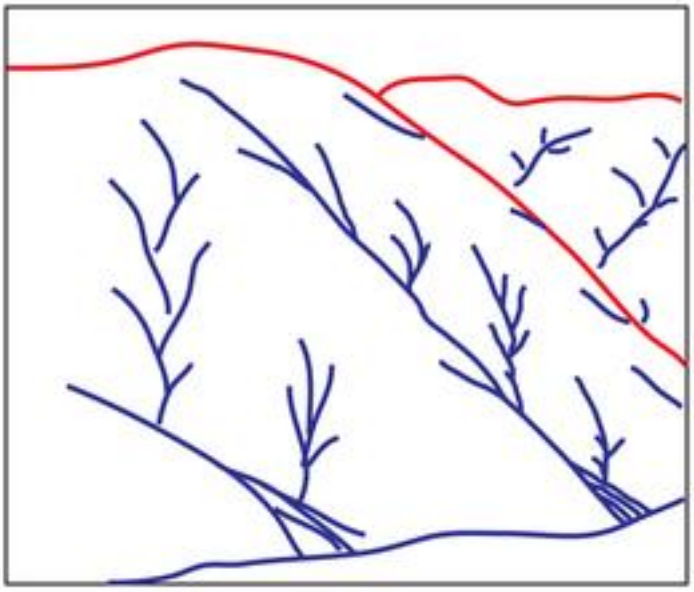

Figure 1. A hand-drawn landscape in the style of W. M. Davis. Silhouettes are highlighted in red; creases in blue.

Current 3D rendering environments are dominated by massively-parallel graphics processing unit (GPU) architectures. In response, graphics programming environments such as the open source OpenGL and proprietary Microsoft Direct3D platforms now all but require that graphics programs of any complexity be written with parallelism in mind. With mid-range GPUs now comprising 2,000 or more computing cores, and each physical core capable of representing many more virtual cores, it is now reasonable (and often compulsory) for graphics programmers to

think about parallel and local, rather than sequential and global solutions to analytical problems. Doing so can effect enormous execution speedup values. In extension, cartographers and other artists can begin to reimagine landscape visualization techniques formerly dismissed as impracticable. When Maxwell and Turpin (1968) first proposed polynomial surface models for landscape rendering, runs were typically generated overnight in batch operations. Current GPUs can render the same surfaces at frame rates well beyond those necessary to support animation. We argue that these environments are now powerful enough to support the real-time construction of much more complicated and arguably more informative analytical surfaces than the dominant shaded relief model in plan or perspective view geometry. Automated pen and ink landscape sketches are one example of a class of non-photorealistic rendering (NPR) techniques that use perspective geometry and 'just enough' linework to unambiguously illustrate surface curvature and discontinuities (Isenberg et al., 2003, Imhof 2007, p. 44). Early NPR work on landscape representations focused on techniques for extracting silhouettes and creases in sequential graphics environments exclusively in object space (Buchin, et al., 2004, Lesage and Visvalingam, 2002, Markosian, et al. 1997, Kennelly and Kimerling, 2006). More recent related areas of research include the enhancement of hillshading technique with the application of 2nd derivative surface models (Kennelly 2008) and the automation of hachured maps through the analytical production of flowlines (Samsonov, 2014).

Current graphics platforms largely permit the implementation of earlier sequential NPR algorithms in 'legacy' or compatibility profile modes, but doing so generally

ignores the advantages of their massively-parallel architectures. Mower (2014) introduces algorithms and an implementation for rendering silhouettes at animation speeds. Designed for modern GPUs, its implementation achieves animation frame rates on contemporary GPUs by limiting data transfer between main system memory and the GPU, by exploiting built-in tessellation stages of the programmable pipeline, and by keeping synchronous inter-core GPU communications to a minimum. This paper presents a complementary algorithm for crease rendering in pen and ink style with a test implementation that renders creases at over 101 frames per second (fps), well above the motion picture industry standard of 24 fps.

2. Methodology

2.1 Modern OpenGL, GPUs, and Concurrency

Contemporary GPUs achieve high speed rendering through the application of massively parallel processing architectures. Programming environments have adapted accordingly to focus on concurrent techniques. To introduce the reader to the opportunities afforded by parallel processing in graphics applications and to make them aware of their constraints, we will trace the information flow in a generic concurrent program before discussing the crease rendering algorithm in section 2.3.

Figure 2 illustrates a simplified version of the mandatory and optional stages of the OpenGL 4.3 programmable pipeline for the test implementation discussed in this paper.

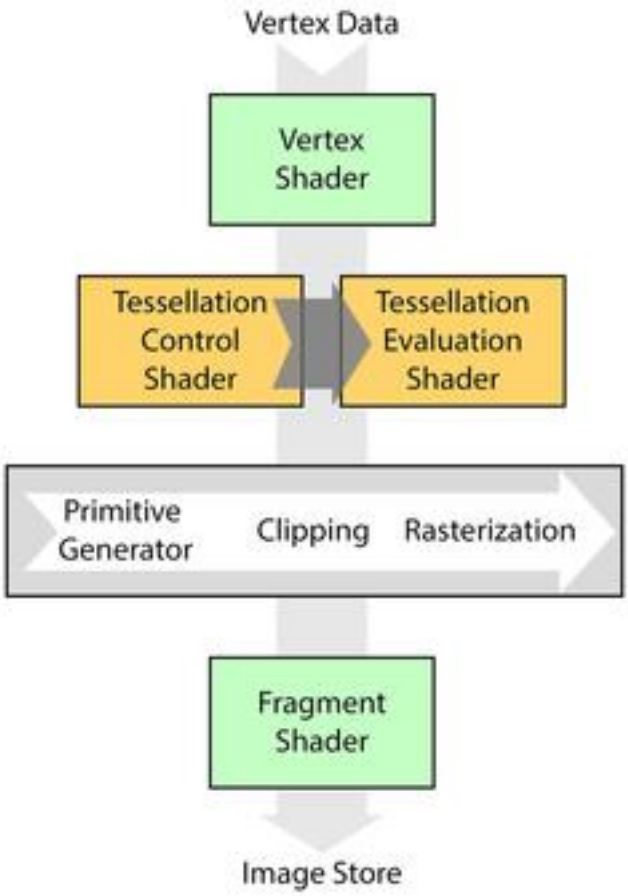

Figure 2. The OpenGL 4.3 rendering pipeline for the implementations discussed in this paper. Mandatory programmable stages are highlighted in green, optional stages in orange, and mandatory, non-programmable stages in gray. The optional geometry and compute stages (both unused in the implementation for this paper) are not shown.

A program written for core profile versions of OpenGL 3.0 and later is actually composed of 2 independent programs, one that runs on the CPU (the client) and another, the server, which runs on the GPU. The server program, written in GLSL (OpenGL Shading Language), is compiled and linked by the client at run time. Once loaded on the GPU, the client can initiate server program execution and can communicate with it through memory buffers. Generally, the client will make

buffered data available to the server (through a binding or copying process) at the start of a rendering pass but will not interact with the buffer again until the pass has completed.

A server program must be composed of at least 2 mandatory 'stages' dedicated to vertex and fragment processing (Sellers, et al., 2014, pp. 8-10, Shreiner, et al., 2013, p. 11). In this simplest programmable pipeline model, a vertex shader, the first stage of the graphics pipeline, reads buffered vertex data on the GPU and can (but need not) perform coordinate transformations, vertex coloring, and other vertex-specific operations. The vertex shader stage runs in independent, parallel instances on separate virtual cores, one for each vertex in the input stream with no implicit communication between vertices on other virtual cores and no guaranteed ordering of vertex processing. Following this stage, the non-programmable primitive generator assembles the output of the vertex shader instances into point, line, triangle, or patch primitives depending on the user's specification. After clipping, the rasterizer partitions the output of the primitive generator into fragments (image units roughly equivalent to pixels) and feeds them to the fragment shader, the next (and final) mandatory programmable stage. Each instance of the fragment shader determines the color of its portion of an associated framebuffer. A framebuffer may be associated directly with the output device (the on-screen framebuffer) or to off-screen graphics memory. The second option is frequently useful for communicating partial rendering information between rendering passes, where each pass is a complete traversal of the pipeline. Like the vertex shader, the fragment shader also

runs as independent instances on separate cores, but in this case each instance represents an individual image fragment in isolation from its neighbors. Sophisticated graphics programs can function with only vertex and fragment shading stages but OpenGL versions 4.0 and later give the programmer the means for finer control of the graphics pipeline (Wikipedia 2014). Mower (2014) and this paper both use tessellation control and evaluation shader stages to create Bezier surfaces for evaluation at finely tessellated mesh vertices. Doing so produces analytical surfaces with less surface variability for subsequent feature extraction than do non-smoothed surfaces. Figure 3 illustrates how silhouette rendering on a non-smoothed surface produces unwanted features.

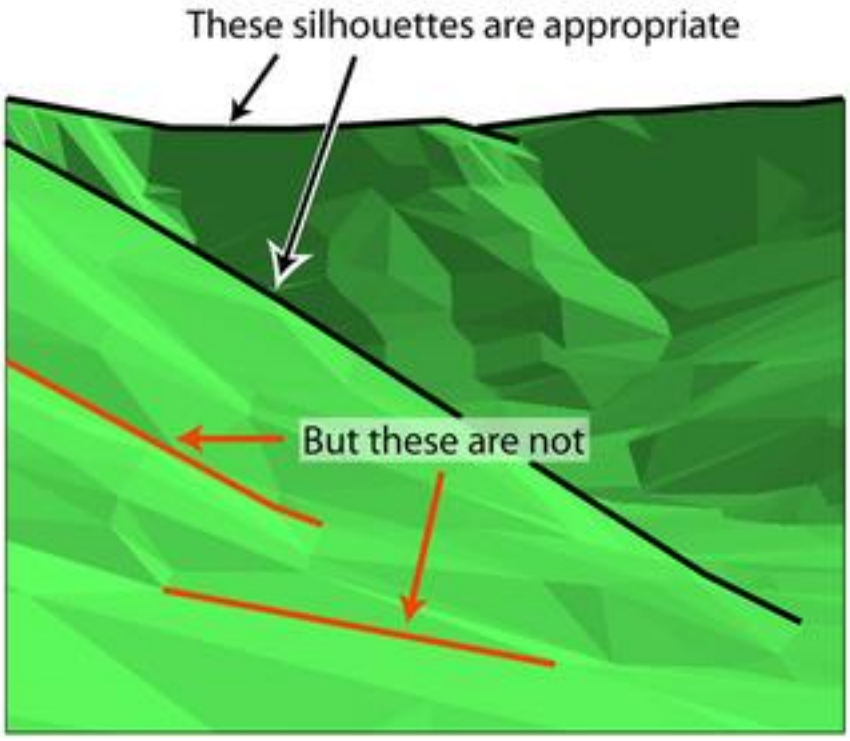

Figure 3. On an unsmoothed rendering of a TIN DEM, low order surface variability generates unwanted silhouettes (in red).

In a modern concurrent OpenGL program, rendering speed is greatest when inter-processor communication is kept to an absolute minimum. However, some graphics problems require that information be shared among a neighborhood of processors representing a set of spatially adjacent features. To do so, such programs make use of multiple rendering passes to perform more sophisticated analyses than are possible with one. Generally, this strategy involves the creation of temporary data products in gridded pixel formats ('textures') or vector data structures ('shader storage buffer objects') that represent intermediate processing results analogous to 'flaps' in manual graphic arts production. Textures are relatively simple 2D data structures that hold color data. Introduced as static 'wallpaper' patterns in OpenGL 1.1, textures have since evolved into dynamic 2D storage mechanisms for color and other data rendered to off-screen framebuffers. Shader storage buffer objects (SSOs), on the other hand, are often used for reading and writing arbitrarily complex structured data, especially when their referent objects do not map intuitively to 2-dimensional grids. Both textures and SSOs can be shared among processors at any stage of the programmable pipeline. Fortunately, writing and reading operations on SSOs and textures can be performed asynchronously and thus do not incur the large time penalties often associated with synchronous memory access operations (Shreiner et al, 2013, pp. 576-608).

In addition to the rendering stages in the programmable pipeline, OpenGL provides a generic compute shader that operates outside of the rendering pipeline. Although compute shaders typically read and write data from and to textures and SSOs, they do not consume vertex data nor do they write color and depth data directly to the

on-screen framebuffer. They are often employed as a mechanism for performing large-scale numeric processing in parallel across the GPU cores and operate as the only shader stage in a single pass. The implementation described in section 3 does not require a compute shader but programs requiring large scale numeric analysis within visualization contexts often do.

The crease algorithm introduced in section 2.3 makes use of a multipass rendering strategy where an individual pass creates a 'layer' of cartographic information. Each pass is a complete traversal of the rendering pipeline but only the final pass writes to the on-screen framebuffer. All other passes write color and perspective depth data to textures attached to an off-screen framebuffer object (FBO) or write non-image data to a SSO. The fragment shader of the final pass samples from the intermediate data products to create an image for the on-screen framebuffer. The algorithm also uses the intermediate data products to provide fast, asynchronous cross-core data sharing and to facilitate the creation of selectable image layers. The entire process then repeats for each subsequently rendered frame.

Many excellent tutorials on modern OpenGL programming can be found on the web. The author has found those provided by Meiri (2014) to be particularly good demonstrations of processing flow in multistage and multipass rendering applications.

2.2. PN triangles

The choice of an underlying surface model for the Creases algorithm largely depends upon the model's ability to render smooth analytical surfaces quickly without producing apparent discontinuities. An explanation of the selected model,

point normal or PN triangles, will help the reader to understand its advantages for the discussion of Creases in the following section.

Mower (2011) introduced an algorithm for rendering creases from drainage network segments generated from evaluated B-spline surfaces. Drainage segments are good for crease production because they align with local surface dip direction and, when evaluated in a drainage accumulation model, their flow values serve as a useful selection criteria.

B-splines provide a smooth, low-noise surface for extracting surface morphology but their generation is costly in the fixed pipeline version of OpenGL (v. 2.1) employed in that project. Although version 2.1 includes the NURBS (non-uniform, rational B-spline) facility, its associated methods, hosted on the CPU rather than on the GPU, are hampered by poor performance. OpenGL versions 4.0 and later provide a much better solution in the form of the tessellation control and evaluation stages. The tessellation shading stages allow the programmer to define a polynomial surface as a series of implicitly connected patches and to evaluate points within the patches to any level of detail that may be required. All patch definitions and subsequent surface evaluation are conducted on the GPU.

Since patches are processed independently of one another, an algorithm must employ a mechanism to ensure visual continuity between rendered patches. Mower (2014) and the algorithm described in section 2.3 use point–normal (PN) triangle patches to create a smooth analytical surface from a triangulated regular grid elevation mesh (Vlachos et al., 2001). A point or vertex normal is a vector associated with a surface mesh vertex that represents the normalized sum of the vertex-

adjacent face normals (Figure 4). Within a mesh, a triangle defined by 3 such vertices obtains enough information about its neighboring facets to render them with the appearance of tangent continuity. Since the point normal (PN) triangle vertices and point normals are constructed in a one-time preprocessing operation, there is no need for patches to communicate with one another during a rendering pass for the sake of maintaining continuity.

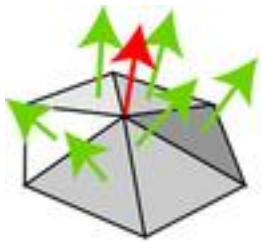

Figure 4. A point or vertex normal (red) is a vector that represents the normalized sum of the connected face normals (green).

A PN triangle defines a local Bezier surface patch as a set of 10 control points, 3 of which coincide with vertices sampled from the elevation mesh and their associated point normals. The remainder of the control points are interpolated at fixed barycentric positions along the edges and at the center of the PN triangle (Figure 5). As a result, the PN triangle control points define coefficients of a bicubic polynomial surface. The GLSL server code for this project defines the position of the control points in the tessellation control shader (TCS). The TCS is also responsible for tessellating the PN triangle to a finer triangular mesh of a desired resolution. Each TCS instance passes a single vertex of the tessellated mesh to an instance of the tessellation evaluation shader (TES). The TES takes the U,V coordinates of the tessellated vertex from the TCS and evaluates it to a Bezier surface that uses the values of the TCS control points as coefficients. Subsequently, the non-programmable primitive generator stage gathers the output of the TES instances into triangles and passes them along to the clipping, rasterization, and fragment stages for on or off-screen rendering.

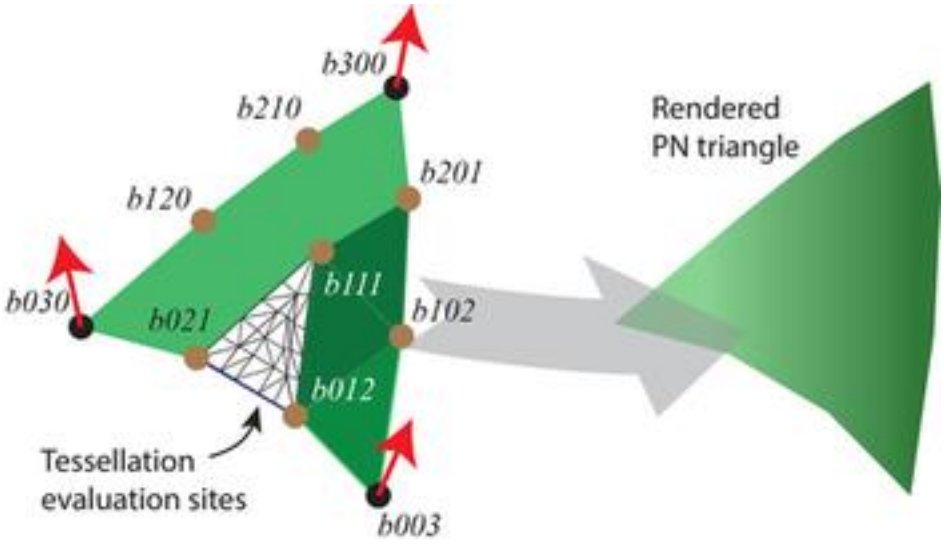

Figure 5. A point normal (PN) triangle. Vertices b003, b030, and b300 (black) are Bezier control points corresponding to mesh elevation samples with pre-calculated point normals (red). The remaining control points (brown) are interpolated from the vertices' coordinate values and normals at fixed barycentric coordinate positions. A cutaway shows tessellation evaluation sites for a portion of the PN triangle. The resulting rendered image (right) appears smooth.

Since connected PN triangles share point normals at their vertices, their apparent continuity is good enough for generating an analytical surface for both silhouettes and creases without visible breaks. The image in Figure 6 demonstrates that PN triangle surface renderings lack visible patch boundary discontinuity artifacts.

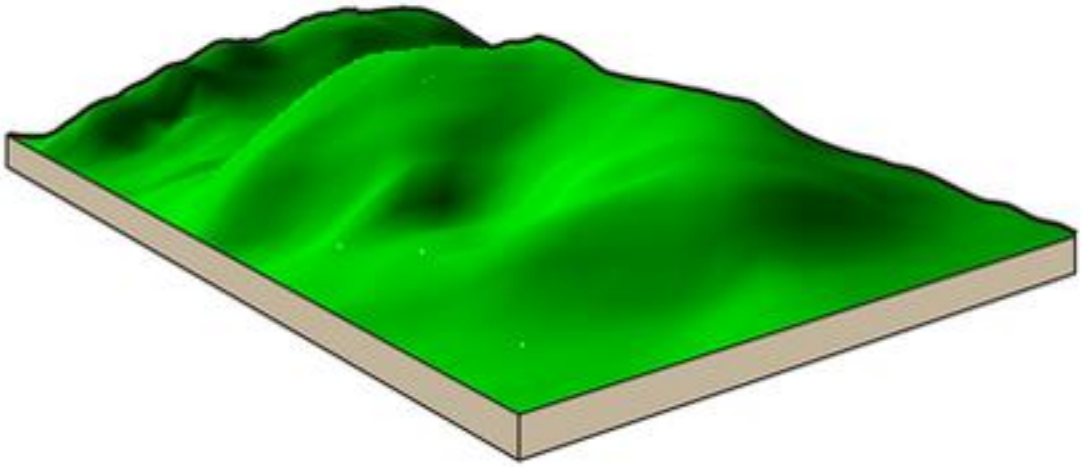

Figure 6. A portion of the mid-Hudson valley rendered as a surface of connected point normal (PN) triangles with square grid vertices at 50 meter horizontal intervals. Apparent continuity is maintained along triangle patch edges. The original image has been cropped and mounted on a base in Adobe Illustrator for this figure.

The efficiency of the tessellation stages allows the server program to create new polynomial surfaces for each rendered frame at animation rates well above the standard 24 fps. Of course, the point of this project is to use the evaluated surfaces analytically to extract and render linework at animation speeds. To do so, the crease algorithm must limit data transfer across the system, keep inter-processor synchronization to an absolute minimum, and avoid the creation of global topological models in object space.

2.3. `Creases`—an algorithm and methods for rendering creases on programmable pipeline GPUs

`Creases` is designed with these performance constraints in mind and serves as the basis for a tested implementation that renders creases above 101 fps. To further

help the reader understand the nature of parallel data processing in modern GPUs, we expand the listing of the Creases algorithm (Appendix A) slightly to include the methods and techniques necessary to implement it. We also present steps that refer exclusively to silhouette production (italic font) to provide the reader with an overview of the entire landscape rendering process.

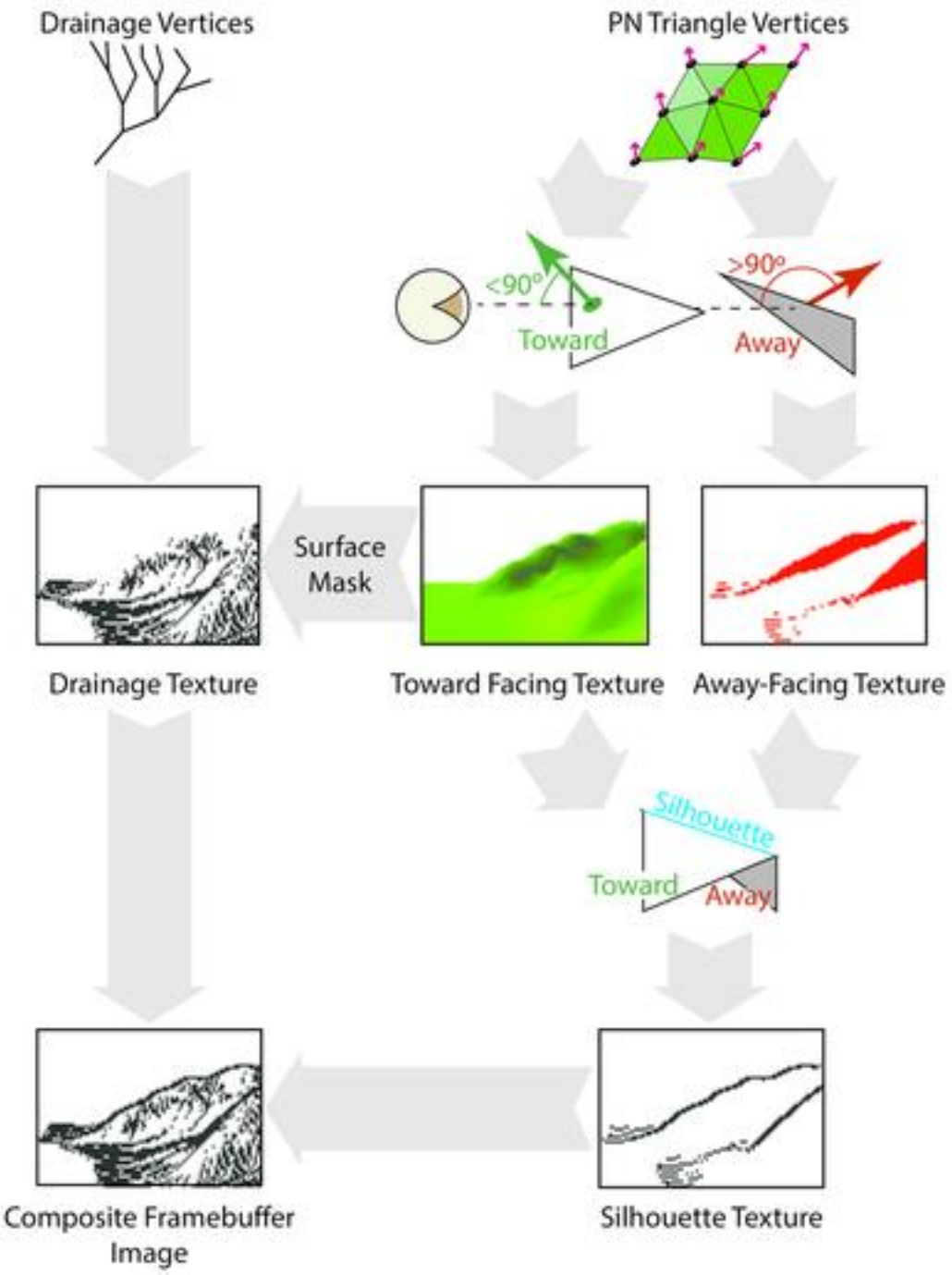

Figure 7. Intermediate rendering products and the composite framebuffer image.

Figure 7 outlines the flow of intermediate data products for `Creases`. Referring to its listing in Appendix A, step 1 of `Creases` describes data preprocessing

operations conducted on the CPU. For this project, the elevation data referred to in 1.1 were extracted from the National Elevation Dataset (USGS 2014) and formatted into a square grid suitable for the `Creases` implementation. At run time, the implementation reads the gridded data set, assembles the samples into a triangular mesh, subdivides each 4-sample grid cell into 2 triangular regions (each becoming a PN triangle patch on the GPU), computes face normals for each triangular facet, and computes point (vertex) normals for all non-edge samples (Figure 8).

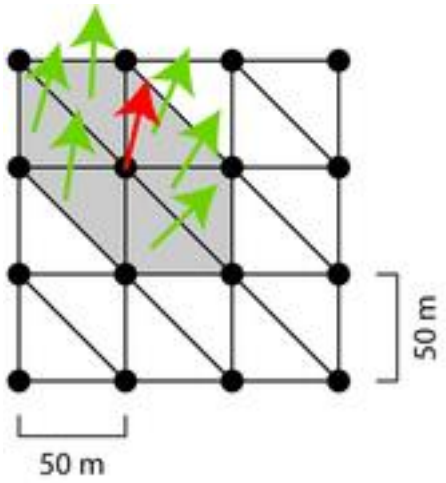

Figure 8. Each cell in the regular grid DEM (defined by 4 elevation samples) is partitioned into 2 triangles that each define the region of a PN triangle patch. The green face normals for the shaded triangles contribute to the red point normal at the common vertex. Horizontal sampling resolution is 50 meters in easting and northing for this project.

`Creases` draws lines following surface dip direction by sampling line segments from a preprocessed, high-resolution drainage network model. To build the model, the author first experimented with an image space, compute shading stage to implement real-time drainage accumulation modeling from a rendered elevation map but the results were unsatisfactory—rendering speed fell well below animation

rates and the quality of the rendered drawing was poor. The author abandoned this approach for a preprocessed, high-resolution, drainage vertex database that could be resampled at any desired level of detail less than or equal to its highest resolution (1.2). The drainage vertex database (DV), created in a separate implementation from `Creases`, is a 3D, object space drainage accumulation model for the target region derived from evaluated points on a bicubic polynomial surface. Vertices along drainage branches are selected for rendering with respect to local slope, illumination, accumulated flow and world distance of the segment to the viewpoint. Loosely based on the method proposed by O'Callaghan and Mark (1984), the drainage model is written out as a list of vertices organized into branches. Every vertex record contains the amount of accumulated flow through its position on its stream segment. A branch is composed of connected vertices sharing a common stream order.

2.4. Building a drainage vertex database

The construction of the drainage vertex data is summarized briefly as follows:

1. Construct a polynomial reference surface;
2. Sample the surface in a regular grid format;
3. Find the drainage direction at each grid point;
4. Starting at each pit, discover all grid points in the pit's basin and find the accumulated flow passing through each point.
5. Normalize the flow at each grid point to the total flow through the basin.

Figure 9 illustrates the drainage vertex data assembly process. For the regular grid used in this project, drainage direction is recorded as the steepest of the 8-case

downhill paths surrounding a grid point. Points may be classified as pits, ridges, junctions, or channels. Ridge points have 0 adjacent uphill neighbors, channel points have 1, junction points have more than 1, and although pits may have up to 8 uphill neighbors, they must have exactly 0 downhill neighbors. Because drainage direction is conducted on samples taken from a smoothed polynomial surface, the discovery of spurious or false pits is largely avoided. Drainage basin assignment and flow accumulation begins at a pit and recurses uphill, stopping at ridge points.

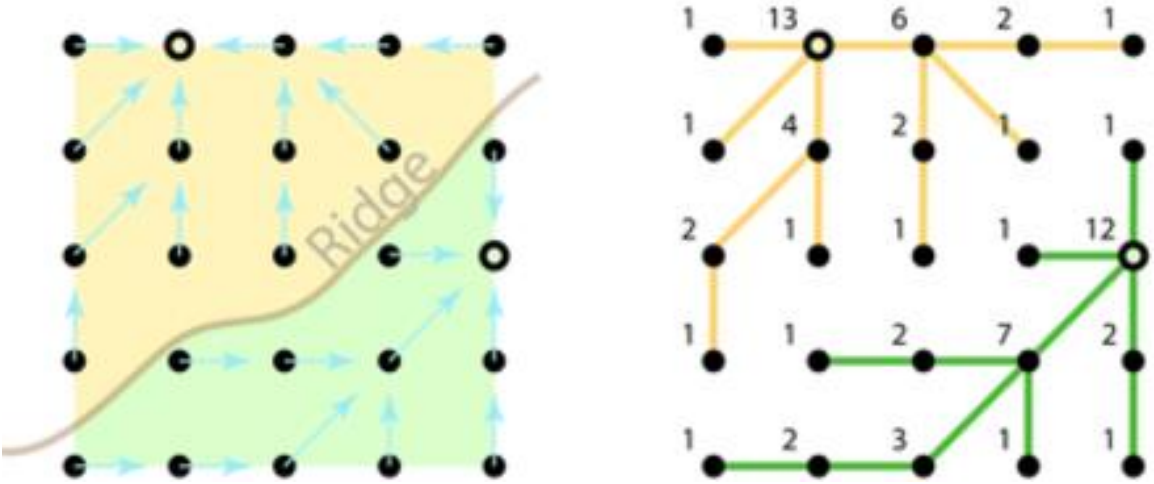

Figure 9. The drainage vertex assembly process. On the left, drainage directions define 2 basins, one draining to the upper pit (outlined circle) and the other draining to the lower. On the right, drainage accumulation values show the accumulated flow through a cell, equaling the input flow from uphill neighbors plus 1. The total flow through the system is the sum of the accumulated flow at the pits. The total flow is also equal to the number of grid points.

The accumulated value at each grid point is the flow received from its uphill neighbors plus 1, setting the total flow through the system equal to the number of grid points. The flow through a vertex is normalized to the total flow of its basin and

this normalized value is stored in the DV list for use as one of the rendering criteria. By using normalized flow values, the `Creases` implementation can apply constant minimum rendering threshold values across all basins without underrepresenting smaller, but potentially important, drainage systems. Segment slope, the other surface criterion, is simple enough to be calculated during each rendering pass without a noticeable speed penalty. In 1.3, the vertex and normal data is 'bound' or copied from system memory to graphics memory—all subsequent rendering operations refer to data in the graphics memory store.

2.5. Rendering passes and intermediate data products

`Creases` requires 3 GPU pipeline passes to render drainage segments as creases, using both SSOs and textures to share intermediate data and rendering products between passes (silhouette rendering requires 2 additional passes). The first pass (2.1), creates a PN triangle surface from the indexed vertex data bound to graphics memory and supplied to the vertex shader. In 2.1.1, each instance of the vertex shader simply passes a single mesh vertex and its accompanying vertex or point normal through to the tessellation control stage (TCS). In 2.1.2, each instance of the TCS assembles 3 mesh-adjacent vertices and their normals into a PN triangle patch, defines the positions of the intermediate Bezier control points, and performs a fine tessellation of the PN triangle interior. The patch coefficients are written to a SSO in 2.1.3 for consumption in the drainage rendering pass (2.3.1). Each vertex output from the tessellation is operated upon by an independent instance of the tessellation evaluation shader (TES—2.1.4). Each TES instance takes the world coordinates of its tessellated vertex and evaluates it with respect to a Bezier equation weighted with

the patch control points (coefficients) generated in 2.1.3. The primitive generator assembles the evaluated vertices from the TES instances into triangles and passes them off to the non-programmable clipping and rasterization shader stages. In the last programmable stage for the rendering pass, the fragment shader (2.1.5) takes the output from the rasterizer and renders color data (typically the image background color) to a texture attached to an off-screen framebuffer object (FBO) only if

1) its associated surface normal points toward the viewpoint and
2) the generating world surface is closer to the viewpoint than that for any other surface fragment at the same framebuffer coordinates.

Fragments satisfying both these criteria write their depth from the viewpoint to an associated depth texture so that subsequent masking operations can define obscuring surfaces.

Since we are not concerned directly with silhouettes in this paper, the reader may refer to Mower (2014) for a complete description of that process.

In step 2.3, vertices on the preprocessed DV list are evaluated to the containing Bezier surface patches generated in 2.1. First (2.3.1), the vertex shader multiplies a drainage vertex by the surface normal (from 2.1.3) to extend the vertex far enough beyond the front face of the surface to avoid depth fighting artifacts. This step may be implemented in one of 2 ways. Vlachos, et al. (2001, p. 163) provide equations for generating a surface normal relative to any point on the curved PN triangle surface. However, this procedure requires a relatively large number of vector operations per frame. For now, the `Creases` implementation constructs a normal to the plane

passing through the 3 PN triangle vertices and stores it in the SSO along with the Bezier coefficients in 2.1.3. Although this solution greatly reduces depth fighting artifacts while maintaining near animation frame rates, it does so at the expense of inappropriate crease clipping in some surface orientations. Future work will investigate more efficient implementations of PN triangle normals to provide more precise control of drainage vertex offsets from the underlying surface.

2.6. Sampling techniques and rendering decisions

The third pass (2.5) renders the intermediate data products from steps 2.1.5 and 2.3.2 (and possibly silhouette data from 2.4.2) to the on-screen framebuffer through a sampling process (Figure 10).

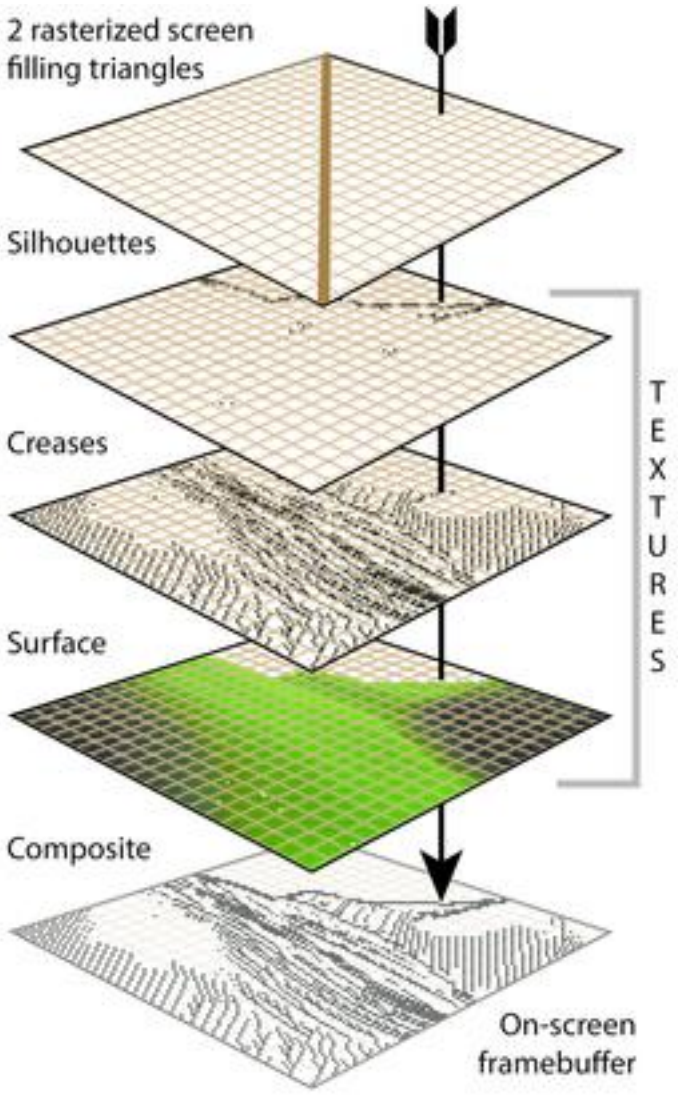

Figure 10. The image created for display in the on-screen framebuffer is composed of sampled color data from the intermediate data products on a per-fragment basis. For each fragment, data from the creases and silhouette layers are only copied to the composite layer if their perspective distance to the viewpoint is less than the corresponding surface fragment distance.

Following common practice, the vertex shader (2.5. 1) generates 2 screen-filling triangles from a six-vertex buffer. After the triangles are rasterized, the fragment

shader (2.5.2) samples color and depth data for each of the user-selected intermediate data products at each fragment instance. First, each instance samples the surface texture color data and copies it to the on-screen framebuffer regardless of its depth from the viewpoint (each surface color fragment already represents the closest toward-facing surface to the viewpoint). Next, it samples the drainage texture but only copies its color data to the on-screen framebuffer if its depth is less than or equal to that of the previously sampled surface color data (Figure 11). Finally, it samples the silhouette data in the same manner as the drainage depth data.

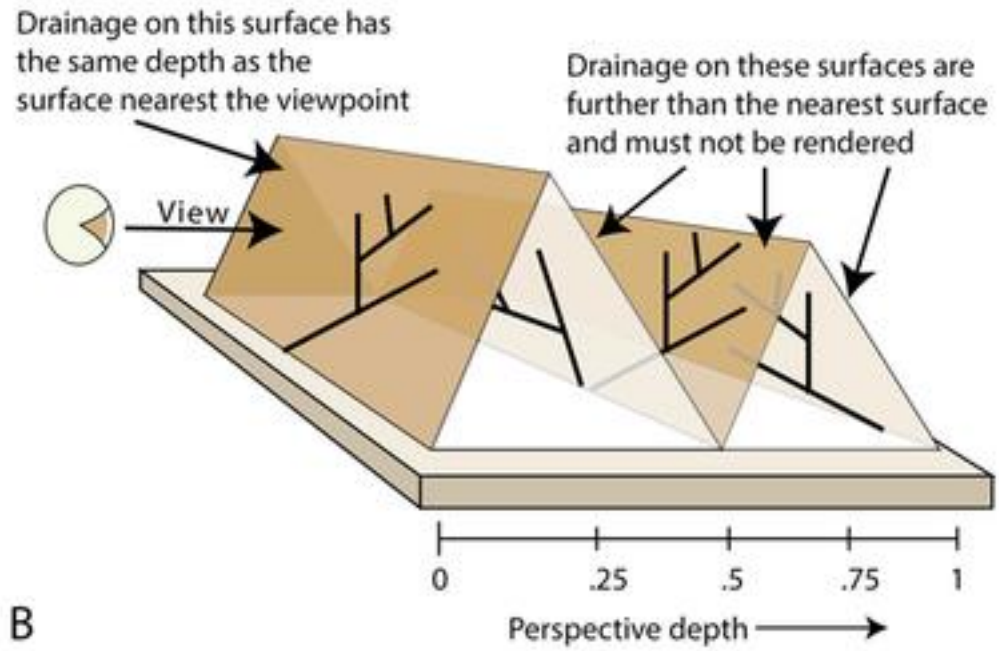

Figure 11. In A, surfaces in brown tint are invisible from the viewpoint. Their surfaces and drainage must be left unrendered. In B, the hills in A have been represented in schematic form. Drainage segments with greater perspective depth than the rendered surface (the surface closest to the viewpoint) are not rendered.

2.7. Drainage rendering criteria

Returning to 2.3.2, crease rendering depends upon 3 weighted criteria:

1) the 'slope' of the underlying surface,
2) the direction of the illumination vector, and
3) the accumulated flow through a drainage vertex.

The implementation defines slope as the angle of the local surface normal to the normal of the horizontal base plane (Figure 12). Currently, the local surface refers to the plane passing through the 3 patch vertices although future implementations may use PN triangle surface normals. The dot product of the normalized surface

normal and the Y axis gives the cosine of the angle between them, ranging from 0 (90° apart) to 1 (0°) (equation 1). A cosine of 0 indicates the steepest possible surface. However, equation (6) multiples the slope weight by the slope component as a partial determination of an overall rendering score. To increase slope importance as α increases, we use its cosine subtracted from 1 (equation 2) in the overall rendering score in equation (6).

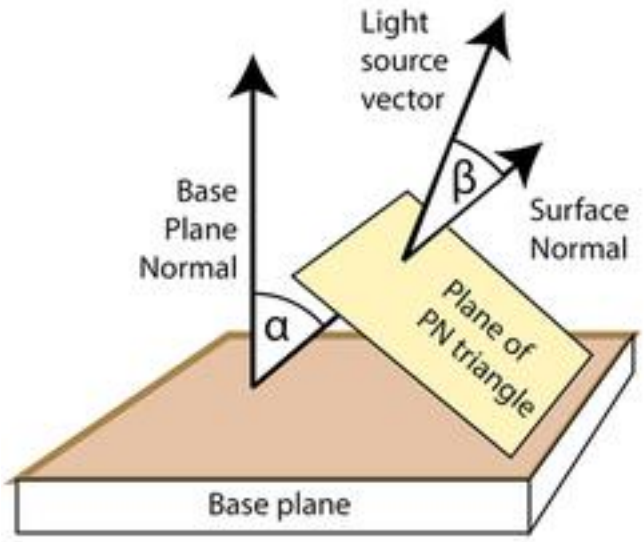

Figure 12. Angle serves as the slope angle for the plane of the PN triangle and is its illumination angle. As approaches 0°, the plane of the PN triangle becomes less steep. As approaches 0°, surface illumination increases.

$$\cos\alpha = ||\boldsymbol{sn}|| \cdot ||\boldsymbol{bn}|| \quad (1)$$

where $||\boldsymbol{sn}||$ is the normalized surface normal and $||\boldsymbol{bn}||$ is the normalized base plane normal (Y axis).

$$sc = 1 - \cos\alpha \quad (2)$$

where *sc* is the slope component.

Surface illumination is dependent upon the angle between a given light source vector and the surface normal (Brassel 1974). If both vectors are normalized, their dot product provides the cosine of the angle between them (equation 3). A cosine of 1 indicates coincidence of the 2 vectors and the maximum surface illumination (equation 4).

$$\cos\beta = ||\boldsymbol{sn}|| \cdot ||\boldsymbol{ln}|| \quad (3)$$

where $||\boldsymbol{ln}||$ is the normalized light source vector.

$$ic = 1 - cos\beta \quad (4)$$

where *ic* is the illumination component. If the light source and surface normal vectors are parallel, β is 0°, $cos\beta$ is 1, and the surface is fully illuminated. To reduce the number of selected lines with increasing surface brightness, the illumination component passed to equation (6) is also inverted.

The third criterion, flow accumulation, is simply read from the DV list for each drainage vertex (equation 5)

$$fc = flow_v \quad (5)$$

where *fc* is the normalized flow accumulation component and $flow_v$ is the normalized flow value at a vertex. Larger flow values indicate greater rendering importance so this component can be used directly in equation (6).

The implementation of `Creases` allows the user to select the weight that each criterion contributes to the rendering score (*rs*) evaluated for the binary rendering decision (equation 6). If $rs \geq 0$, the vertex is rendered. If not, it is rejected. Drainage vertices are rendered in line strip primitives—a decision to reject a vertex results in an unrendered line segment.

$$rs = (sw \times sc + iw \times ic + fw \times fc)/3 - rs_{min} \quad (6)$$

where *rs* is the rendering score, rs_{min} is the minimum acceptable rendering score, *sw*, *iw* and *fw* are the slope, illumination and flow accumulation weights, and *sc*, *ic*, and *fc* are the slope, illumination, and flow accumulation component values for the vertex. Given that each weight ranges between 0 and 1 as does the range of each component value, the total of the weight and value components can be normalized by dividing by 3. The base minimum acceptable rendering score is set by the user (ranging between 0 and 1 inclusive); it increases linearly, attenuated by distance from the viewpoint, to reduce linework density in the perspective background.

As implemented for this project, the rendering score and rendering decision computed in equation (6) does not guarantee or attempt to define an 'optimal' rendered image. Rather, they serve as a framework for future research in the automated selection of weights and minimum rendering scores. This paper is primarily concerned with keeping the number of rendered frames per second near acceptable animation frame rates.

3. Results

Like all modern OpenGL programs, the `Creases` implementation, combined with the silhouette rendering algorithm introduced in Mower (2014) and referred to below as `SPI` (shader pen and ink), is composed of a C++ client program that executes on the CPU and a host program written in GLSL version 4.3 for the GPU. The performance statistics for `SPI` were generated from runs on a Dell Precision T3600 desktop computer with a 4-core, 3.6 GHz Intel Xeon processor, 16 GB system

memory, and an NVIDIA GeForce GTX 780 GPU containing 2034 physical cores, executing at a clock speed of 976 MHz, and referencing 3 GB video memory.

The elevation data for the test runs were extracted from the National Elevation Dataset for a 6 km by 6 km region of the mid-Hudson River valley of New York State in the vicinity of West Point. The author chose this area for its combination of rolling hills and cliff faces, providing a broad range of landscape features for testing the implementation. Elevations were resampled at 50 m horizontal intervals in easting and northing to form a vertex grid of 120 rows and 120 columns, producing 28,322 PN triangle patches ($patches = (rows - 1) \times (columns - 1) \times 2$). This dataset was used for 2 purposes:

1) to create a preprocessed drainage accumulation model, and
2) to serve as the PN triangle patch vertices for rendering steps 2.1 and 2.2.

The higher resolution elevation grid for the drainage model was created by evaluating points at 10 meter horizontal and vertical intervals with respect to the Bezier surfaces of the PN triangle patches. This was done once in a separate implementation; `SPI` reads the drainage data created from this dataset at runtime.

For context, the non-shader, legacy OpenGL crease rendering implementation in Mower (2011) took approximately 3 minutes and 6 seconds to render creases in a single frame on the same hardware testing platform. Since the overall rendering times for the fixed and programmable pipeline versions are so dissimilar, no finer resolution comparisons will be made between them.

Performance data were collected for `SPI` using the Microsoft QueryPerformanceCounter high-resolution time stamp API. `SPI` initiates

programmable pipeline passes through calls to the OpenGL function `glDrawElements()` in the client section. Time stamp requests were placed before and after each call to `glDrawElements()` to isolate the contributions of each pass to the overall execution time for each frame. The time stamp differences (in milliseconds) are interpreted as execution times for the passes. The raw timing statistics are explained in Table 1 and their summary statistics are defined in Table 2. The data in Tables 3 and 4 were collected from runs conducted in batch mode. In this mode, `SPI` reads records from a batch file where each record represents a frame with independent viewing parameters. After the frame is rendered, its image is captured to a file and frame performance statistics are collected and written to a log file. Since the performance collection operators bracket the rendering operations exclusively, the statistics are independent of the image archiving and logging operations.

To keep the timing environment consistent for all runs, the statistics quoted below were collected immediately after a system reboot had been performed. Other than necessary operating system processes, no other application-level processes were active on the testing platform during timing runs.

`Creases` Step (pass)	**Step function**
2.1 (*TFT*)	Render tessellated toward-facing triangles to texture
2.2 (*AFT*)	Render tessellated away-facing triangles to texture
2.3 (*DM*)	Render drainage to texture
2.4 (*SIL*)	Render silhouettes to texture
2.5 (*LAY*)	Render textures to on-screen framebuffer

Table 1. Raw timing statistic definitions for steps in the Creases implementation SPI.

To provide the reader with an overall sense of the separate and combined program speeds for crease and silhouette production, 3 summary statistics were generated that combine the raw statistics in Table 1.

Combined statistics	**Meaning**	**Summary statistic**
TFT, DM, LAY	Operations necessary to render creases	*CREASEfps*
TFT, AFT, SIL, LAY	Operations necessary for rendering silhouettes	*SILHfps*
TFT, AFT, DM, SIL, LAY	Operations necessary for rendering creases and silhouettes together	*TOTALfps*

Table 2. Summary statistics for SPI.

CREASEfps, *SILHfps*, and *TOTALfps* measure performance in frames per second (fps) (equation 7). Although silhouette production is not the main topic of this paper, its timing statistics were provided for completeness.

$$CREASEfps = \frac{1000}{(TFT + DM + LAY)} \tag{7a}$$

$$SILHfps = \frac{1000}{(TFT + AFT + SIL + LAY)} \tag{7b}$$

$$TOTALfps = \frac{1000}{(TFT + AFT + DM + SIL + LAY)} \tag{7c}$$

where *TFT*, *AFT*, *DM*, *SIL*, and *LAY* are given in milliseconds and *CREASEfps*, *SILHfps*, and *TOTALfps* are calculated in frames per second.

Table 3 lists the average `SPI` execution times by `Creases` step (each implemented as a separate `SPI` rendering pass) for viewpoints spaced along the test path. Starting at UTM coordinates 588497mE 4589846mN 18N at 548 m elevation (frame 0), and proceeding at a viewing altitude angle of 0° along an azimuth of 225° to 585008mE 4586381mN 18N (frame 98), each frame viewpoint is spaced approximately 49.7 meters apart from its neighbors on the path, The first row of Table 3 presents averaged statistics over all 99 frames. The averages on the second row include only the first 3 rendered frames. The author has found that the first 3 frames of every run execute much more slowly than the remaining frames. Of all of the rendering passes, *TFT* (the rendering pass that builds PN triangle patches and renders their tessellated, toward-facing triangles) shows the greatest difference of any pass between the initial 3 frames and the remaining frames. The cause of the slow execution speeds on the first 3 runs is unclear but may be due to data caching issues. The third row of Table 3 shows averages for all but the first 3 frames.

	TFT	***AFT***	***DM***	***SIL***	***LAY***	**TOTAL**
Average, all 99 frames	5.69	6.42	6.13	27.17	0.04	45.46
Average, first 3 frames	67.38	17.78	8.83	61.77	0.05	155.81
Average, all but first 3 frames	3.76	6.07	6.05	26.09	0.04	42.01

Table 3. Averaged execution times in milliseconds by rendering pass (see Table 1).

Table 4 lists the average frame rates for crease, silhouette, and combined rendering operations. It is important to note that each of the values presented for *CREASEfps*, *SILHfps*, and *TOTALfps* were found by summing their respective calculated values for each record and dividing by the number of records, not by operating on the average statistic values presented in Table 3. The frame rates for *CREASEfps* in Table 4 suggest that a drainage accumulation database resolution greater than the 10 meter sampling tested here could still support animation speed rendering.

	CREASEfps	***SILHfps***	***TOTALfps***
Average, all frames (fps)	101.18	27.48	23.54
Average, first 3 frames (fps)	53.43	11.73	10.42
Average, all but first 3 frames (fps)	102.67	27.98	23.95

Table 4. Frame rates (in frames per second or fps) for the combined rendering operations listed in Table 2.

Table 5 lists the viewing parameters common to each frame along the test path sequence for the runs summarized in Tables 3 and 4. Field of view (FOV) was held constant at 22°.

View Az°	View Alt°	FOV°	Light Az°	Light Alt°	Vert exag	Norm slope weight	Norm illum weight	Norm flow weight	Min rend score
225	5	22	180	45	1	.1	.3	.6	.8

Table 5. The viewing parameters for the data presented in Tables 3 and 4 and for Figure 13b. View Az and Alt are viewing azimuth and altitude angles, FOV is the image field of view, and Light Az and Alt are illumination angles. All angles are expressed in degrees. Norm slope, Norm illum, and Norm flow all represent normalized weights. Vert exag is the elevation scale factor relative to the horizontal scale. Min rend score is the minimum render score cutoff value.

Table 4 shows that the average crease rendering frame rate for all 99 frames exceeded 101 fps, over 4 times the 24 fps lower limit for animation frame rate rendering. Although silhouette rendering by itself executed at over 27 fps for all frames, crease and silhouette rendering together (*TOTALfps*) averaged 23.54 frames per second, approximately 2% short of the minimum animation frame rate. Excluding the data from the first 3 of the 99 frames in the *TOTALfps* calculation raises the combined frame rate to 23.95 fps, virtually equal to the lower limit.

The reader is invited to inspect an animation of the entire 99 frame sequence at the Transactions in GIS website (WPCreasesOverall.mpg). A KML file of the approximate view for frame 0 is also provided as PenAndInkCreasesFlightPathOrigin.kml. Figure

13 depicts frame 0 in the sequence along with a shaded surface of the same scene for comparison. Note that the minimum acceptable score for crease rendering increases linearly with perspective depth, causing the rendering of more detail in the perspective foreground than in the background. The author selected rendering weights for the images in this sequence (Table 5) to highlight surface shape with a relatively sparse set of linework in the foreground and background.

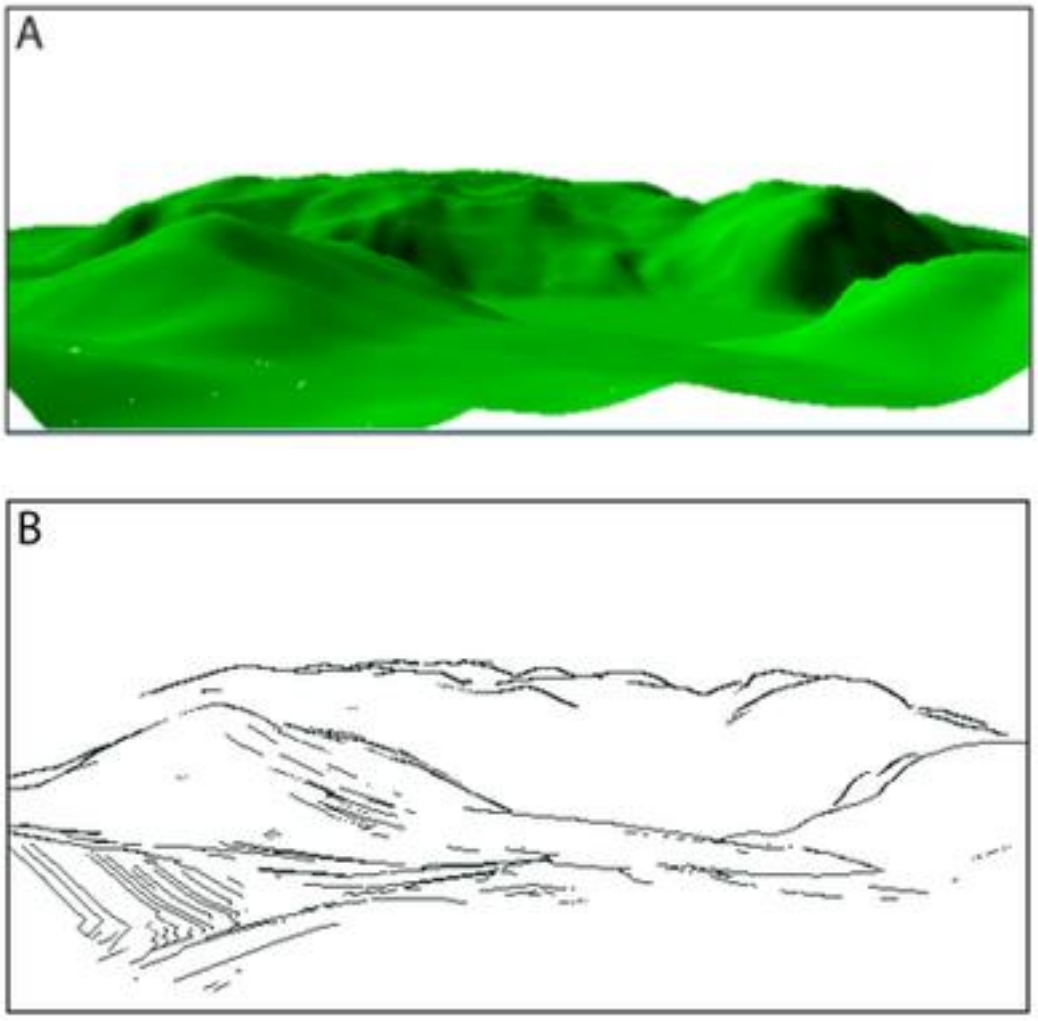

Figure 13. To provide context for the reader, a shaded perspective image of the scene covered by the viewing geometry of the initial frame is rendered by SPI in A. In B, creases

and silhouettes are rendered for the scene over the surface now rendered in white as a mask.

It is important to note that crease rendering is relatively insensitive to the total number of lines rendered; the *CREASEfps* scores for the 3 images in Figure 15 using minimum rendering score values of .10, .19, and .25 were 109, 101, and 107 fps respectively. Figure 14 shows the 56th rendered frame produced along the test path in which the viewpoint has moved appreciably closer to the Hudson River than in the previous view. As a result, linework representing virtual drainage channels on the opposite valley wall appear where none did in Figure 13.

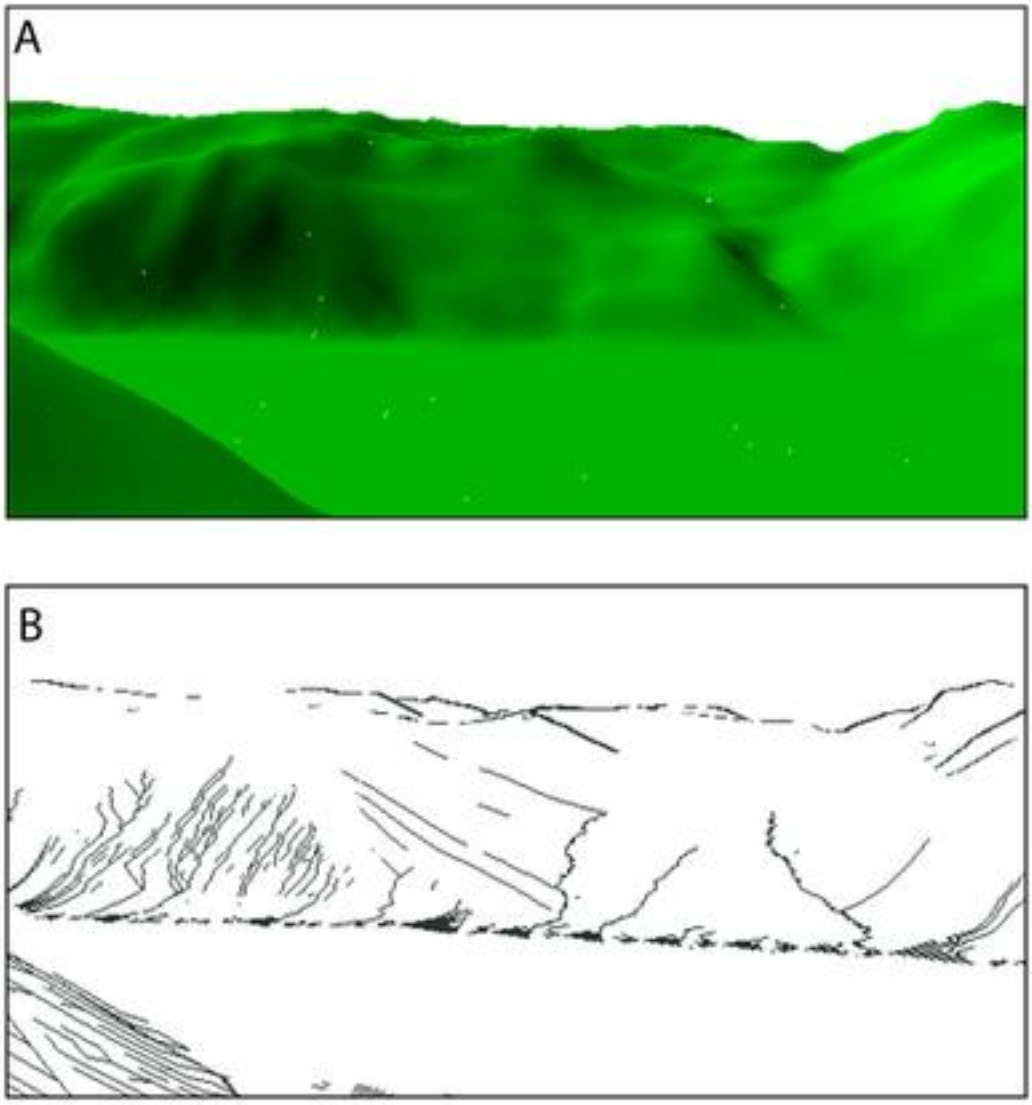

Figure 14. The scene for the 56th rendered image in the sequence with a viewpoint approximately 2.6 km to the southwest of that for Figure 13. The Hudson River is rendered as the flat region in the foreground. As the viewpoint approaches the far shore, more drainage segments are selected than in the previous figure.

The current parameter weighting strategy in `SPI` assumes that each (slope, flow, and illumination) carries equal importance in image composition. This is probably not the case if, as expected, slope and flow for a given surface patch are highly correlated. Therefore, the selected weights for the images in the sequences below

were chosen empirically. The precise contributions of the parameter weights to drainage segment selection will be investigated in future research. Incrementing the base minimum rendering score for an image sequence created from a fixed viewpoint has the effect of rendering fewer numbers of creases per frame. Figure 15 represents 3 snapshots in a sequence of 101 frames taken from viewpoint 586523mE 4587886mN 18N, varying the minimum rendering score from 0 to 1 in .01 steps and holding the viewpoint and the remaining view parameters listed in Table 6 constant (The animated image sequence is represented on the TGIS website as WPVaryMinRendScore.mpg).

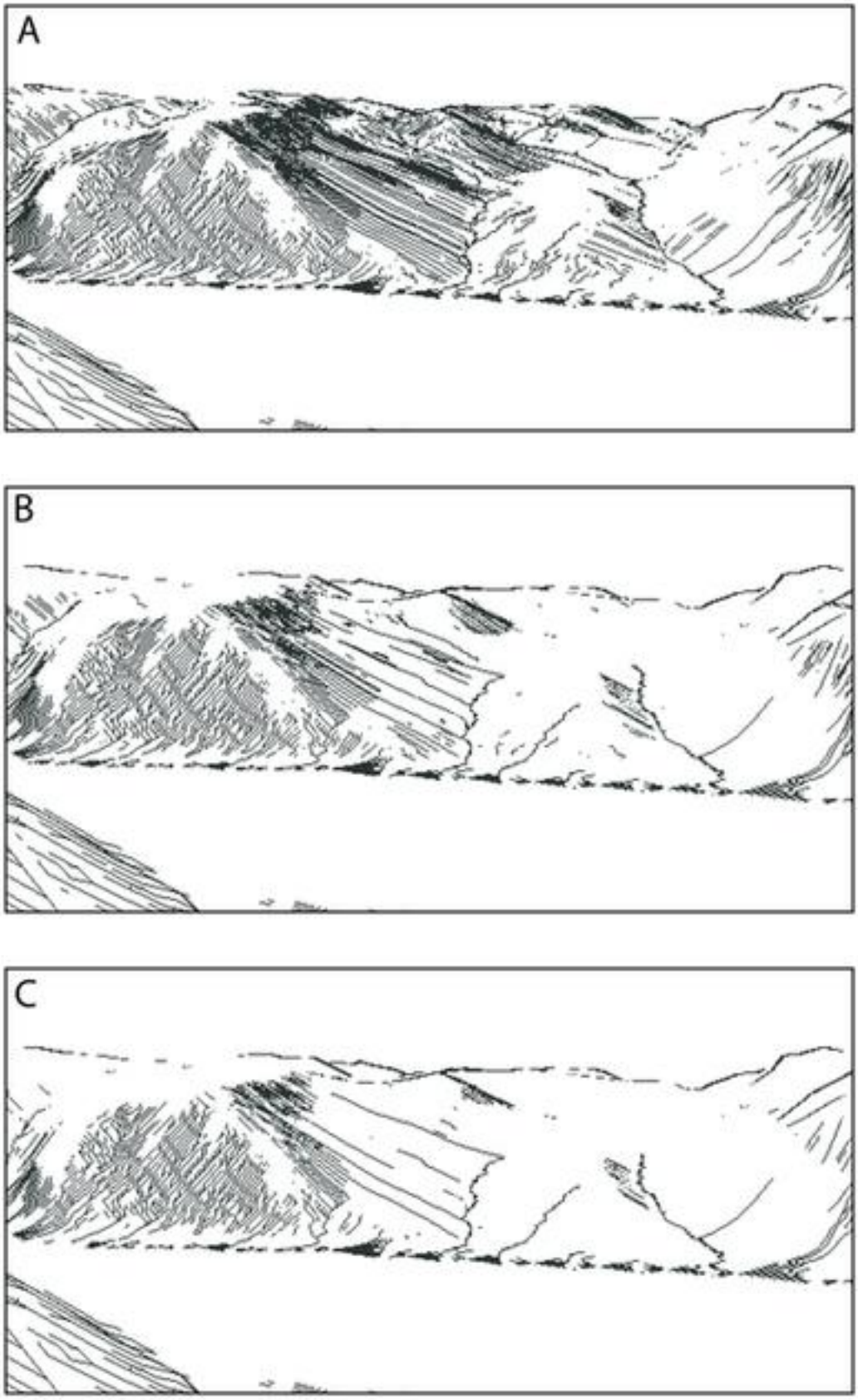
A
B
C

Figure 15. In A, the minimum rendering score = .10. In B, .19, and in C, .25.

Figure 15A was rendered with the lowest minimum rendering score, resulting in several areas on the image with coalescing linework. In B, those areas have been reduced while still retaining evidence of surface curvature on the northern (right-most) side of the image. In C, the area of coalescence has been further reduced but at the expense of some suggestion of curvature on the right side of the image.

View Az°	View Alt°	FOV°	Light Az°	Light Alt°	Vert exag	Norm slope weight	Norm illum weight	Norm flow weight
225	5	22	180	45	1	.33	.33	.33

Table 6. Viewing parameters held constant for Figure 15.

Figure 16 demonstrates the effect of moving the position of the light source on surface shading. A low minimum rendering score of .10 was chosen to exaggerate shading contrasts. In Figure 16A, the light is coming from the left side of the image (azimuth of 135°) with a view azimuth is 225°. In B, the light is coming from the right (315°). In both cases, slopes facing away from the light source select more lines for rendering than do slopes facing toward the light source, creating a shading effect. An animated sequence of 180 frames, with illumination azimuth angles increasing in 2° clockwise increments around a circle from 135° to 133° (with a constant 45° illumination altitude angle), is available on the TGIS website as WPVaryingIllumDir.mpg.

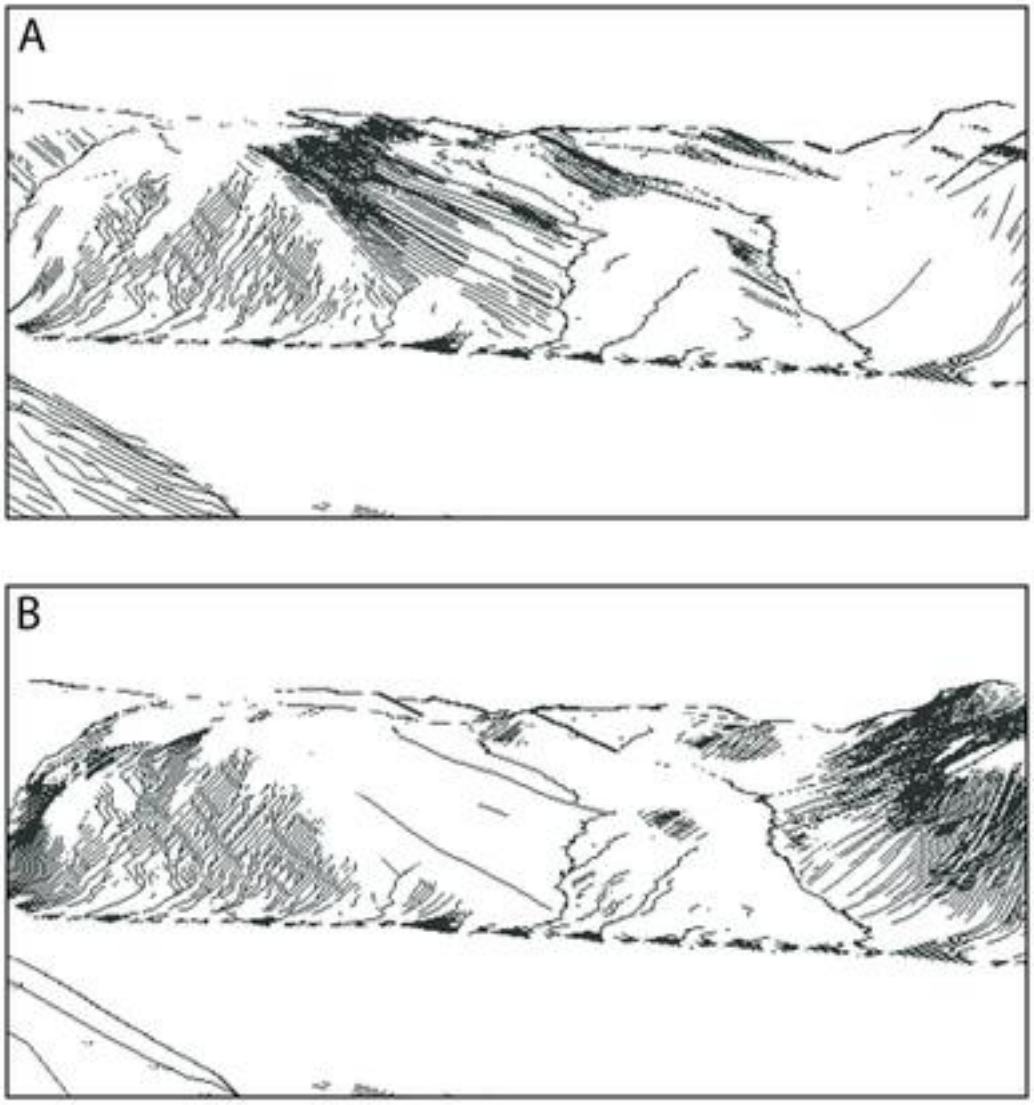

Figure 16. 2 images of the same scene, looking at an angle of 225° across the Hudson River from a position above West Point, NY with varying light source azimuths. In A, light is coming from 135° (the left). In B, the light is coming from 315° (the right). The light source altitude angle in both cases is 45°. The minimum rendering score was set to .10 to intentionally oversample lines in shaded regions.

An additional video file, WestTempleFlyThroughTGISCreases.mpg, represents a scene located in Zion National Park, centered on a view of the West Temple feature,

the subject of the silhouette rendering tests discussed in Mower (2014). Crease production has been added in this video and provides the user with an opportunity to evaluate the quality of the implementation with regard to a different type of landform, in this case one consisting predominantly of rock faces. The starting point of the 539-frame fly-through is 326404mE 4117405mN 12N along a view azimuth of 300° at a constant view altitude of 0°, extending to 323922mE 4118865mN 12N. To catch subtle changes between frames, the world stepping resolution along the flight path has been set to approximately 5.3m distance per frame. The illumination azimuth and altitude angles are 225° and 45° respectively. As in the images of the Hudson Valley region, surface detail increases as the distance to the viewpoint decreases.

4. Discussion

Although `SPI` is a test implementation and is therefore not written in optimized, production quality code, it nonetheless renders creases at an overall average of more than 101 fps, over 4 times the minimum acceptable 24 fps motion picture rendering standard (Table 4). `SPI` reaches this level of performance by sampling from a preexisting high-resolution drainage accumulation model instead of constructing a custom model during each rendered frame. Silhouette construction requires more analytical operations per rendered frame than do creases and accordingly does so at lower, but still acceptable frame rates. Currently, the combined output of crease and silhouette production falls just short of the minimum standard animation rate, averaging an overall 23.54 frames per second. It is reasonable to assume that optimization and hardware improvements in the near

future will increase performance by the 2% necessary to achieve true animation frame rates.

Not all of the features found on a hand drawn pen and ink landscape illustration are currently rendered in SPI. Landscape illustrators often use 'form lines' to represent visually significant folds in the surface. Imhof (2007, pp. 214-216) uses the term 'form line' to identify lines that delineate features lying in a horizontal plane (such as the edge of an escarpment or a plateau). Figure 17 shows a small section of a contemporary hand-drawn illustration by Rudy Slingerland of the Goose River delta in Labrador with a form line identifying the edge of a bluff (Rudy Slingerland, personal communication).

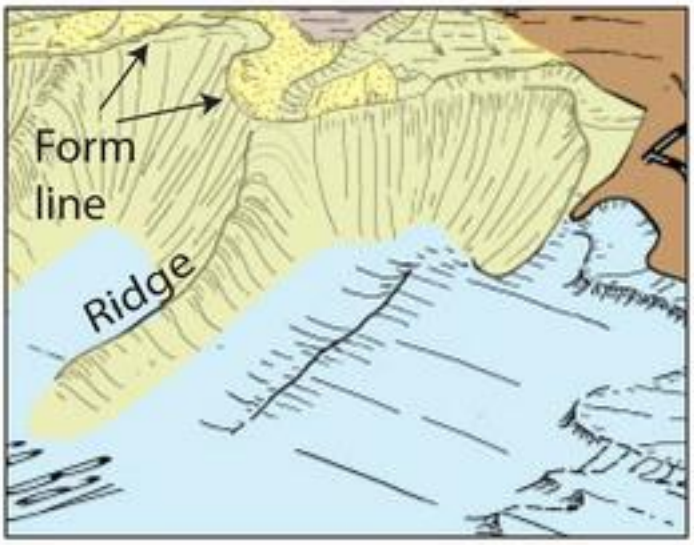

Figure 17. A hand-drawn pen and ink illustration by Rudy Slingerland of a portion of the Goose River delta, Labrador, Canada. Annotations have been added by the author of this paper. (Rudy Slingerland, personal communication)

Representing neither a silhouette nor a drainage feature, the form line in this image instead marks a relatively sharp change in surface orientation from near vertical to near horizontal. Musuvathy et al. (2011) propose techniques for extracting principal curvature ridges from B-spline surfaces that may be useful for discovering escarpments on the evaluated PN triangle surfaces used for this project.

Figure 17 also contains a ridge line that marks a drainage divide but does not coincide with a silhouette. Fortunately, this type of feature is relatively straightforward to extract from the drainage basin model described in this paper. Referring back to Figure 9, drainage points that have no upstream inputs lie on ridges. A fragment in a ridge texture would be rendered once wherever 2 adjacent drainage points from 2 adjacent basins occur. The author will explore both form line and ridge extraction in future work and begin evaluating the quality of the imagery produced by the implementation. Feedback from user comments will guide the automated tuning of feature weights and other rendering and selection criteria.

5. Conclusion

Data from the `SPI` test runs demonstrate that `Creases` is an acceptable algorithm for rendering linework representative of surface curvature at animation frame rates. Furthermore, the test results show that silhouettes and creases can be rendered together at near animation frame rates with the expectation that short-term program optimization and modest hardware speed increases will raise overall performance above 24 fps.

Automated pen and ink style landscape renderings have as many potential applications in physical surface visualization as do their hand-drawn counterparts,

whether in the service of geomorphological study or as artistic ends unto themselves. Now that modern GPUs and software rendering environments are powerful and flexible enough to support complex numerical analysis and visualization in the same computing context, it is time to explore new ways of exploiting this technology to create more interesting and informative landscapes than are currently available on commercial systems.

References

Brassel K 1974 A model for automated hill shading. *The American Cartographer* 1, 1:15-27

Buchin K, Sousa M.C, Dollner J, Samavati F, Walther M 2004 Illustrating terrains using direction of slope and lighting. *4th ICA Mountain Cartography Workshop in Vall de Núria, Catalunya, Spain, September 30-October 2 2004*, http://www.mountaincartography.org/ publications/papers/papers_nuria_04/buchin.pdf

Fernlund K J 2000 *William Henry Holmes and the Rediscovery of the American West*. Albuquerque, University of New Mexico Press

FFmpeg 2014 *About FFmpeg*. WWW document https://www.ffmpeg.org/about.html

Imhof E 2007 *Cartographic Relief Presentation*. Redlands CA, ESRI Press. Reprint of Imhof E 1982 Cartographic Relief Presentation. Berlin, Walter de Gruyter and Co.

Isenberg T, Freudenberg B, Halper N, Schlechtweg S, Strothotte T 2003 A developer's guide to silhouette algorithms for polygonal models. *IEEE Computer Graphics and Applications* 23, 4: 28-37

Kennelly P J 2008 Terrain maps displaying hill-shading with curvature. Geomorphology 102: 567-577

Kennelly P J and Kimerling A J 2006 Non-photorealistic rendering and terrain representation. *Cartographic Perspectives* 54: 35-54

Lesage P L and Visvalingam M 2002 Towards sketch-based exploration of terrain. *Computers and Graphics* 26: 309-328

Lobeck, A K 1958 *Block Diagrams and Other Graphic Methods Used in Geology and Geography*, Amherst, MA, Emerson Trussell Book Company

Markosian L, Kowalski M A, Goldstein D, Trychin S J, Hughes J F, and Bourdev L D 1997 Real-time nonphotorealistic rendering. *Proceedings of the 24th Annual Conference on Computer Graphics and interactive Techniques International Conference on Computer Graphics and Interactive Techniques*. New York, ACM Press/Addison-Wesley Publishing Co pp. 415-420

Maxwell, D A and Turpin, R D 1968 Numeric Ground Image Systems Design, *Research Report 120-1, Numerical Ground Image, Research Study Number 2-19-68-120*. College Station, TX, Texas A&M University

Meiri E 2014 Modern OpenGL Tutorials. WWW document, http://ogldev.atspace.org/

Mower J E 2011 Supporting automated pen and ink style surface illustration with B-spline models. *Cartography and Geographic Information Science* 38, 2: 175-184

Mower J E 2014 Fast Image-Space Silhouette Extraction for Non-Photorealistic Landscape Rendering. In press, *Transactions in GIS*

Musuvathy S, Cohen E, Damon J, and Seong J 2011 Principal curvature ridges and geometrically salient regions of parametric B-spline surfaces. *Computer-Aided Design* 43, 7:756-770

O'Callaghan J F and Mark D M 1984 The extraction of drainage networks from digital elevation data. *Computer Vision, Graphics, and Image Processing* 28: 323-344

Raisz, E. 1938 *General Cartography*. New York, McGraw-Hill Book Company, Inc

Samsonov T 2014 Morphometric mapping of topography by flowline hachures. *The Cartographic Journal*, 51, 1:63-74

Sellers G, Wright R S Jr, Haemel, N 2014 *OpenGL SuperBible, 6th ed.*, New York, Addison-Wesley

Shreiner D, Sellers G, Kessenich, J, and Licea-Kane B 2013 *OpenGL Programming Guide, 8th ed.*. Upper Saddle River, NJ, Addison-Wesley

US Geological Survey 2014 National Elevation Dataset. WWW document http://ned.usgs.gov

Vlachos A, Peters J, Boyd C, and Mitchell J L 2001 Curved PN triangles. In *2001 ACM Symposium on Interactive 3D Graphics.* New York, ACM Press: 159–166

Wikipedia 2014 OpenGL Shading Language. WWW document http://en.wikipedia.org/wiki/OpenGL_Shading_Language

Appendix A. `Creases`—an algorithm and methods for rendering creases on programmable pipeline GPUs

To render creases from a drainage network model:

1. On CPU (one time data preprocessing operation):
    1.1. Assemble an input elevation grid into PN triangle vertices; compute face and vertex normals;
    1.2. Read a high-resolution drainage network model as a list of stream segments, each of which is composed of vertices. Each vertex will have an easting/northing/elevation position triple, links to attached vertices, and a drainage accumulation value. Call the drainage model the DV list;
    1.3. Bind (copy) the PN triangle vertices, vertex normals, and drainage vertices to the GPU memory store.
2. On GPU (for each frame):
    2.1. Pass 1—TFT (render toward-facing triangles):
        2.1.1. Read PN triangle vertices and normals from the vertex input stream (vertex shader);
        2.1.2. Assemble PN triangle patches from input vertices, vertex normals, and interpolated interior control points and tessellate the PN triangle interior (tessellation control shader);
        2.1.3. For each patch, write the patch coefficients to a shader storage buffer object (tessellation control shader);
        2.1.4. Evaluate tessellation vertices to a Bezier surface (tessellation evaluation shader);

2.1.5. Render any surface fragments facing toward the viewpoint to a texture attachment of a framebuffer object (fragment shader).

2.2. Pass 1a—AFT *(render away-facing triangles (only used for silhouette processing):*

2.2.x. Perform the steps for Pass 1 but render only away-facing fragments to a separate framebuffer.

2.3. Pass 2—DM (render drainage map):

2.3.1. Multiply drainage vertices by their containing patch normals to extend them beyond the surface mask (vertex shader);

2.3.2. Render the DVs as line segments to a texture attachment of a framebuffer object if the combined weight of their evaluation criteria exceeds the minimum rendering score (fragment shader).

2.4. Pass 2a—SIL *(render silhouettes):*

2.4.1. Create 2 screen-filling triangles (vertex shader);

2.4.2. Find edges of toward and away-facing surfaces and write edge fragments to a silhouette texture (fragment shader).

2.5. Pass 3—LAY (render layers to on-screen framebuffer):

2.5.1. Create 2 screen-filling triangles (vertex shader);

2.5.2. Sample fragments from the toward-facing triangle texture, drainage texture, *and the silhouette texture* and write them to the on-screen framebuffer (fragment shader).

Supporting File Captions

WPCreasesOverall.mpg. An MPEG video of the 99 frame sequence that generated the timing data for tables 3 and 4. This and the following video files are built with the Lavf56.0.100 video codec using the FFmpeg utility (FFmpeg 2014). Stepping along the flight path between frames is approximately 49.7m.

WPVaryMinRendScore.mpg. An MPEG video of 101 frames that vary the minimum rendering score from 0 to 1 in .01 steps and hold the viewpoint of Figure 15 and the remaining view parameters listed in Table 6 constant.

WPVaryingIllumDir.mpg. An MPEG video sequence of 180 frames, with varying light source azimuths through a circle in 2° clockwise increments from 135° to 133° using the same viewing geometry as WPVaryMinRendScore.mpg. The minimum rendering score was set to .10 to intentionally oversample lines for exaggerated shading effects.

WestTempleFlyThroughTGISCreases.mpg. An MPEG video sequence of 539 frames representing a fly-through from 326404mE 4117405mN 12N in Zion National Park along a view azimuth of 300° toward the West Temple feature. Stepping along the flight path between frames is approximately 5m.

PenAndInkCreasesFlightPathOrigin.kml. A KML file representing the viewing coordinates of the first frame of WPCreasesOverall.mpg. For best use, download and open in Google Earth to see a pushpin of the flight path origin for the video WPCreasesOverall.mpg.