Real-Time Drainage Basin Modeling Using Concurrent Compute Shaders

James E. Mower

ABSTRACT: This paper explores the use of graphics processing units (GPUs) for combined numerical analysis and visualization. Using an implementation of the O'Callaghan and Mark (1984) drainage accumulation algorithm, it explores the advantages and challenges that this environment brings to real-time, frame-by-frame production of spatial modeling products and their visualization. Modern GPUs employ massively-parallel processing architectures primarily configured for 3D rendering applications. However, the same environments also support general purpose numeric processing relevant to large scale simulations using languages like CUDA or OpenCL. Unfortunately, these languages do not interact directly with the 'rendering pipeline,' the GPU hardware responsible for transforming vertex, shape, patch, and fragment data into rendered pixels. In contrast, the most popular rendering APIs, OpenGL and Direct3D, have recently incorporated optional 'compute shader' processing stages that share the same data structures, memory models, and overall syntax used by the existing vertex, tessellation, geometry, and fragment shaders that feed the rendering pipeline. This addition allows developers to insert one or more purely numerical processing stages between graphics stages within the same API framework. Execution times for the implementation on a GPU with over 2,000 cores are analyzed to determine which operations are conducive to concurrent execution on the GPU as well as those processes and data access operations that lead to processing slowdowns.

KEYWORDS: Drainage basin models, concurrent, parallel, OpenGL

Introduction

This paper explores the use of graphics processing unit (GPU) computing to both analyze and display spatial data seamlessly in a unified programming environment. Although best known for their usage as graphics rendering engines, GPUs are also used extensively as parallel computing platforms in modelling and in other numeric applications, especially for problems that partition easily into independently-analyzable clusters. As GIS developers begin to implement concurrent algorithms in these environments, this paper proposes techniques for combining analytical and rendering operations conducive to real-time surface analysis and display (Meza, 2015; Carr, 2015). It implements the O'Callaghan and Mark (1984) drainage accumulation procedures as an example of how such a combination could occur.

The evolution of GPU architectures and rendering libraries is characterized by increasing programmatic access to internal functionality. As manufacturers continue to expose underlying chip resources to developers, new opportunities continually arise to exploit them. In 2010, OpenGL version 4.0 exposed access to GPU tessellation units that accelerate 3D rendering of polynomial surfaces (Khronos Group, 2016). Polynomial surface models have a long history of exploitation in GIS, including their use as DEMs going back to the 1960s (Maxwell and Turpin, 1968). While such early applications were limited by the computing power and storage capacities of the time, current tessellation shaders enable fast surface evaluation supportive of further numerical analysis. This paper will show how one such analytical process, a drainage accumulation model, can be constructed and displayed in a hybrid rendering pass.

Method

An overview of GPU programming. Unlike their 'black box' predecessors, modern graphics application programming interfaces (APIs) expose relatively low-level programmable elements of GPU architectures. Two such APIs, OpenGL and Direct3D, provide interfaces for client-side (CPU) and server-side (GPU) processing. Client-side operations are largely responsible for configuring the rendering environment, building shader programs that launch from the client and run on the server, and providing channels for copying data between the client and server. OpenGL server-side programs are written in a set of GPU programing languages referred to as GLSL while Direct3D uses HLSL. A program written in GLSL or HLSL targeting a particular stage of the programmable pipeline is referred to as a shader. The client compiles and links a set of shaders at runtime to form a shader program (Figure 1).

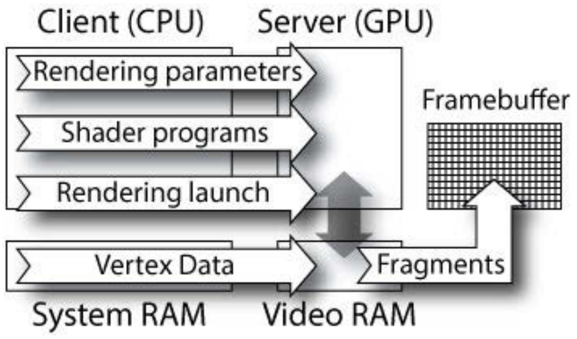


Figure 1. Client/Server relationships and responsibilities.

For the rest of this paper, the discussion of GPU programing will be limited to OpenGL and GLSL, the API and shader language set used to create the products discussed here. Although OpenGL client-side API methods are accessible from many programming languages; C++ is somewhat more convenient than others for desktop development environments and is the client-side language used for this project.

The server-side OpenGL rendering pipeline is summarized in Figure 2. After copying vertex data to video RAM, the client issues OpenGL commands to specify the desired rendering primitive output (lines, triangles, patches, etc.) and launch pipeline processing on the server. It is very important to note that vertex rendering is 1) performed in parallel with, but independently of and asynchronously with respect to all other vertices and 2) that asynchronous parallel processing is the norm in OpenGL to ensure high graphics throughput. The pipeline is composed of mandatory and optional programmable stages, as well as fixed mandatory stages. During a pass of the rendering pipeline, the vertex shader code first operates on vertex data that has been copied, or 'bound' to server RAM from the client. Each vertex is processed as an 'instance' of the vertex shader, hosted on an individual processor (virtual or real, depending on the number of available processors with respect to the number of vertices). The vertex shader code set is shared by all vertices. Generally, but not always, a vertex shader instance performs coordinate transformations on its data, possibly including perspective projection. Next, if tessellation is desired, an instance of the tessellation control shader (TCS) combines a user-defined set of vertices from the vertex shader output, assembles them into a patch, and performs user-specified operations such as determining the positions of control points (coefficients) for a polynomial surface through the patch. An instance of the subsequent tessellation evaluation stage (TES) interpolates the surface created in the TCS at user-specified locations (through the use of tessellation level parameters, defined below) and outputs 1 vertex for each instance at a single evaluation site. An instance of the optional following geometry shader can assemble outputs of TES instances into primitives (triangles, lines, etc.) and create yet more new vertices or structure the TES output into useful topological forms (e.g. triangles). At this point, any available output data is passed to several mandatory, non-programmable stages for final assembly into vector primitives, projection to screen coordinates, clipping to the viewport, and rasterization. Finally, an instance of the programmable, mandatory fragment stage is called for each image fragment output by the rasterizer. A fragment shader is usually responsible for coloring the fragment but can also perform other useful programmatic operations. Fragments map to pixels in the on-screen framebuffer or to textures (2D grids of fixed image storage types) attached to on- or off-screen framebuffer objects for deferred rendering.

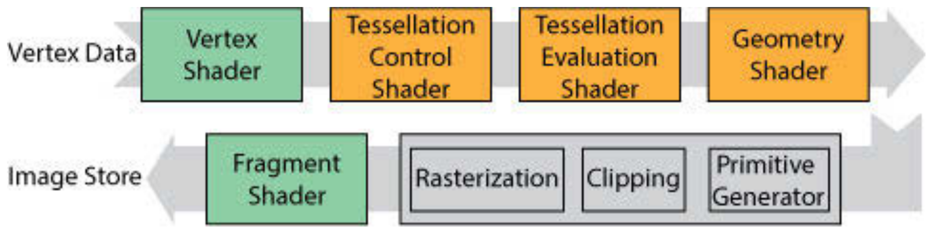


Figure 2. The OpenGL rendering pipeline. Mandatory programmable stages are drawn in green, optional stages in orange, and non-programmable stages in gray.

Besides providing mechanisms for rendering operations, versions of OpenGL from 4.3 onward also permit access to GPU computing operations that are independent of and operate outside of the vertex processing pipeline. These so-called compute shaders use the same data types and channels as graphics shaders but share characteristics of ‘numeric’ GPU programing languages such as OpenCL and CUDA. The addition of compute shaders to OpenGL allows a server-side program to combine modeling and rendering elements within the same framework as a series of GPU ‘passes,’ where each pass begins as a request from the client to initiate data processing (whether through the rendering pipeline or with a standalone compute stage). Importantly, the developer need only understand and link his or her client code with 1 API. The project described in this paper uses multiple rendering and compute passes to create a drainage accumulation model on a frame-by-frame basis, built on real-time tessellations of a cubic polynomial surface. All of the stages of the rendering pipeline and compute shaders can both read and write ‘global’ data, in the form of textures, shader storage buffer objects (SSBOs—lists of arbitrarily structured data) and other formats. These global data channels are also visible to the client (CPU), affording a communication mechanism between server processors and generally between the server and client.

Characteristics of drainage network models. This project builds local polynomial surfaces from gridded elevation samples while maintaining visual continuity along edges and at vertices. The resulting surfaces can be evaluated at any required sampling density to create a mesh of patch-local triangulated irregular networks (TINs) that form the connected source and drain pairs for the drainage model. For a surface described by a mesh of 3-dimensional elevation samples, a drainage direction map determines the path of steepest descent from each point in the mesh to a connected neighbor. The mesh itself may be arranged in a regular grid or TIN format. A TIN mesh is frequently constructed from a Delaunay triangulation of an irregularly distributed set of elevation samples. Meshes constructed on grids typically connect each interior sample to its 8 surrounding samples. In both cases, the drainage direction can be specified as the steepest downhill path from an uphill cell (source) to a connected downhill cell (drain).

Any drainage basin modeling techniques that operates on grid cell elevation samples with constant horizontal sampling density is prone to modeling artifacts sometimes referred to as interior or false pits (O'Callaghan and Mark, 1984). TINs largely avoid generating internal pits by allowing the data developer (or automated procedures) to select samples for inclusion at topologically-significant locations: at peaks, pits, and passes, and along ridges, course lines, and slope breaks. A judicious selection prevents the occurrence of internal pits in subsequent analyses.

For the sake of simplicity in this exploratory project, the Bézier patch-local control grid is drawn from grid cell elevation data (Piegl and Tiller, 1997, pp. 9-34.). This structure maps intuitively to the multidimensional processor grid layouts promoted by OpenGL and is especially convenient for connecting drainage sources and drains between adjacent patches. However, even though the gridded samples serve as coefficients of cubic polynomial patches, the grid itself remains prone to the creation of interior pits. To reduce their number, existing pits within the control grid can be modified by several techniques including weighted smoothing or projection to the plane of 3 of their 8 surrounding neighbors. Although such techniques remove most interior pits, some may remain after multiple passes. O'Callaghan and Mark (1984) provide a 'flooding' procedure to replace all elevations less than the lowest point on the edge of the basin with its value and use that location as a 'pour point.' This creates a drainage path between the basin containing an interior pit and its immediate downhill neighboring basin (see Mower (1994) for a parallel implementation of the basin flooding procedure). Rather than implement this process here, future work will avoid the problem by switching to TIN models (see Future Work below).

Building PN triangles. This project uses a methodology known as PN (point normal) triangles to automatically generate patch-local TINs through the evaluation of local Bézier polynomial surface patches within a triangular mesh (Vlachos et al., 2001). PN triangles provide a useful framework for constructing drainage networks since:

- they are formed from continuous surfaces that suppress local noise;
- they can be constructed efficiently with tessellation shaders on modern GPUs; and
- they are adaptable to multiple resolution and scaling within a single frame.

The surface patch mesh is generated from a regular triangular tessellation of a fixed resolution grid cell elevation model in a 1-time preprocessing step (Figure 3). Each elevation sample is shared by 6 PN triangles. An M x N sample grid produces 2(M -1)(N-1) PN triangles.

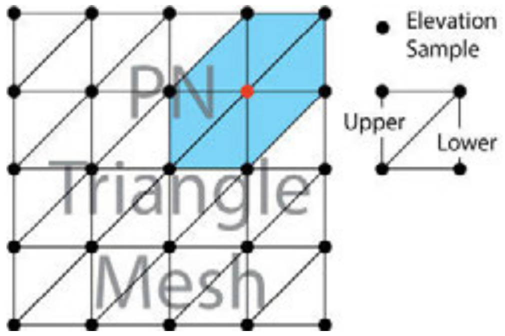


Figure 3. A triangular tessellation of a fixed resolution grid cell DEM is the framework for a PN triangle mesh. In later processing stages, a distinction is made between lower and upper triangles with respect to barycentric edge and vertex coordinate matching between adjacent triangles. Each interior mesh vertex (such as the one specified by the red dot) is shared by 6 PN triangle patches.

Each instance of the TCS is responsible for creating control points for a single patch-local cubic Bézier polynomial surface at the patch's corners (here, a corner refers to one of the 3 vertices defining a PN triangle), at interpolated vertices along its edges, and at its center. To establish a curved surface within a patch, and to assure that patch edges maintain visual continuity with respect to adjacent patches, the coefficients (control points) for the polynomial depend upon the establishment of normals at the corners across the mesh (Figure 4). These 'point normals' are constructed in a 1-time client preprocessing operation by summing the surface normals of the patches sharing the vertex, and then normalizing the summed vector (by dividing each of its 3 components by its length).

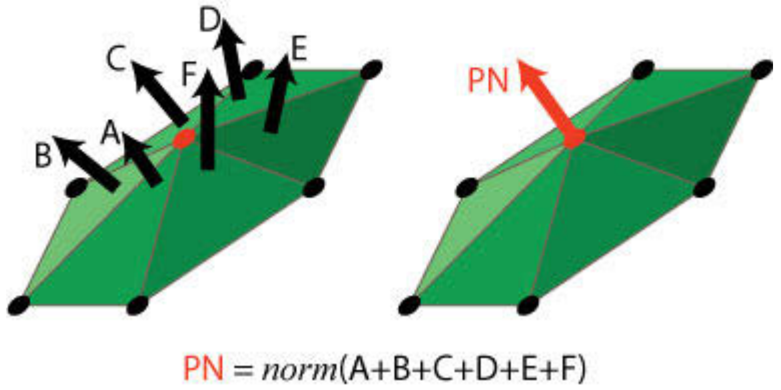


Figure 4. A point normal is constructed at the central vertex as a normalized vector sum of its vertex-adjacent face normals. This operation is performed for all mesh-internal vertices; others require edge cases.

The point normals carry curvature information between patches by determining the world positions of interpolated control points along the edges and at the center of each PN triangle within the mesh. Following Vlachos et al. (2001), 2 control points are established along each edge at 1/3 and 2/3 the distance from one of the corners (Figure 5). An additional control point is established at the PN triangle center (although this project uses a regular right triangle tessellation for simplicity, PN triangles may be irregular). It is convenient to use barycentric coordinates to refer to the control point positions (Bradley 2007).

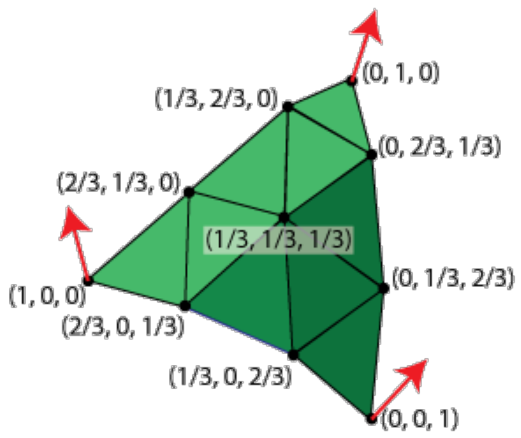


Figure 5. A PN triangle including edge and center vertices with their barycentric coordinates. The vertices at (0,0,1), (0,1,0), and (1,0,0) are PN triangle corners and correspond to mesh samples. Other vertices along the edges and at the center are interpolated with respect to the point normals (red arrows) at the corners.

The world coordinates of the samples at the corners are retained without interpolation or projection for their corresponding control points. The control points along the edges and at the center are calculated in 2 stages. First, a linear interpolation is performed between the bracketing corners. Next, each control point is projected onto the plane determined by the closest corner and its point normal. See Vlachos et al. (2001, section 3.1) for details.

The TCS is also responsible for determining the number and placement of evaluation sites for the tessellation evaluation shader (TES) stage. As its name suggests, an instance of the TES is called for each evaluation site to find its world coordinates with respect to the polynomial surface defined by the patch's control points. For this project, evaluation sites are established at 8 equally spaced subdivisions along each PN triangle edge (related to the user-defined 'outer tessellation level' parameter) and at 12 interior points, defining 2 nested interior triangles (derived from the 'inner tessellation level' value). Each instance of the TES outputs 1 vertex. Figure 6 shows the tessellation sites for 2 adjacent PN triangle patches. Although they share the same topology, the coordinate system for the PN triangle rendered in blue (the 'upper' triangle) is rotated 90° counterclockwise with respect to the lower triangle (rendered in green). This pattern is consistently enforced by the TCS through the user-selected vertex winding order parameter. The tessellation sites define the basis of the drainage network for each patch. Since patches share a common topology, patch drainage inputs and outputs can be connected along edge and at corner adjacencies. A 1-time client-side operation defines a mapping between upper and lower triangle edge and corner vertices. Future versions of this program will allow varying tessellation levels between adjacent patches (see Future Work below). For now, tessellation levels are held constant for all patches.

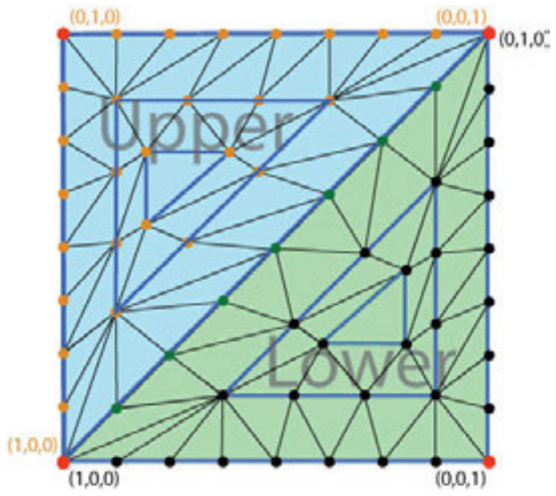


Figure 6. The PN triangle mesh is composed of upper (blue shaded) and lower (green shaded) triangles, each tessellated with the same topology (here, outer tessellation level = 8, inner tessellation level = 5).

Creating a drainage network. The creation of a drainage direction map requires that each tessellation site know its connected neighbors. Since this project keeps tessellation level parameters constant for all patches, determining the tessellation pattern of 1 defines them all. To establish the pattern and create a connected vertex list, an initial rendering pass is performed over a single patch (Figure 7).

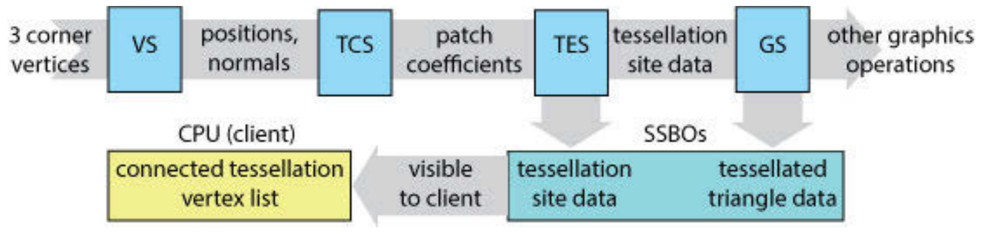
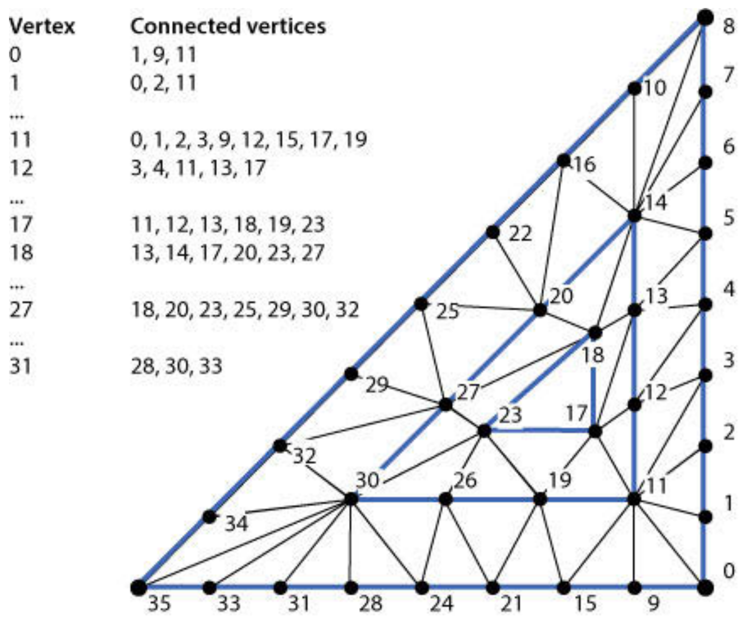


Figure 7. Programmable pipeline for creating a connected tessellation vertex list common to all patches.

Each of 3 vertex shader (VS) instances takes a single 3D sample position (in world coordinates) and point normal (as a normalized vector) and passes them to the TCS. A single TCS instance in turn assembles the 3 input vertices and their point normals to calculate a set of 3D patch coefficients and supplies it to the following TES. A TES instance for each tessellation site outputs an index (used as an ID value later) and a barycentric triple to a shader storage buffer object representing its site's location in the given PN triangle patch. The TES also outputs a vertex to the following geometry shader (GS). One GS instance for each tessellated triangle assembles 3 vertices from TES instances to create a triangle with vertices represented by barycentric coordinate triples. Each GS instance writes the IDs of its triangle and its component vertices to another SSBO. Although OpenGL requires a fragment shader stage in every rendering pass, the remaining products from the pipeline are ignored on this pass. After the pass has completed, a client-side operation builds the connected vertex table by finding all triangles sharing a given vertex and noting the other 2 vertices in each matching triangle (Figure 8).



Vertex	Connected vertices
0	1, 9, 11
1	0, 2, 11
...	
11	0, 1, 2, 3, 9, 12, 15, 17, 19
12	3, 4, 11, 13, 17
...	
17	11, 12, 13, 18, 19, 23
18	13, 14, 17, 20, 23, 27
...	
27	18, 20, 23, 25, 29, 30, 32
...	
31	28, 30, 33

Figure 8. Connected vertices for tessellation sites. The numbers appearing next to the sites are indices that are valid for all patches. The connected vertex table is shown here as an abbreviated list.

Following the 1-time connected vertex pass, the program enters an event handling loop that responds to changes in user viewing parameters, including the initial viewing conditions on startup. Each such event initiates the following operations:

1. Render all patches (rendering pipeline):

    1.1. Write evaluated tessellation vertices to the drainage vertex list (DVL) SSBO, ordered by patch and tessellation site index

2. Create the drainage network (standalone compute stage):

    2.1. Create the drainage direction map for all tessellated vertices in the DVL

        2.1.1. Find drainage directions for non-edge, non-corner (patch-internal) vertices

        2.1.2. Find drainage directions at patch corners

        2.1.3. Find drainage directions along patch edges

    2.2. Compute drainage accumulation values for all evaluated vertices

Step 1 is accomplished through a single pass of the rendering pipeline over all patches in the PN triangle mesh. The main product of the pass is the creation of a SSBO containing all tessellated vertices in all patches (the ‘drainage vertex list’ or DVL), sorted by patch and tessellation site index. DVL vertices have been evaluated to the polynomial surface of their containing patch and are written out in world coordinates (UTM plus elevation in this case). Note that for the version of the program discussed here, tessellation levels are held constant for all patches and are unaffected by viewing parameters. Subsequent versions, however, will allow varying resolution through the use of multiple tessellation levels (see Future Work below).

A compute shading pass (step 2.1) then determines vertex drainage mapping for each tessellated vertex in each patch, with each vertex represented by its own compute shader instance. Each instance examines its vertex's barycentric coordinate values to determine if it is positioned within the patch, along an edge, or at a corner. Starting with the internal case (step 2.1.1), recalling that each DVL vertex is associated with a barycentric triple and thus a set of known connected vertices, an instance accesses each of the internal vertex's connected neighbors in the SSBO and then records the steepest downhill path among them. Each patch can establish drainage for its interior vertices independently from all other patches.

Vertices at PN triangle corners can drain to connected vertices in any of the 6 sharing patches. Figure 9 shows the possible drainage paths for a mesh-internal corner vertex. Since the barycentric coordinates of upper and lower triangles are rotated 90° with respect to one another, a mapping function determines the barycentric coordinate triple for *v* with respect to each of the 6 sharing PN triangles.

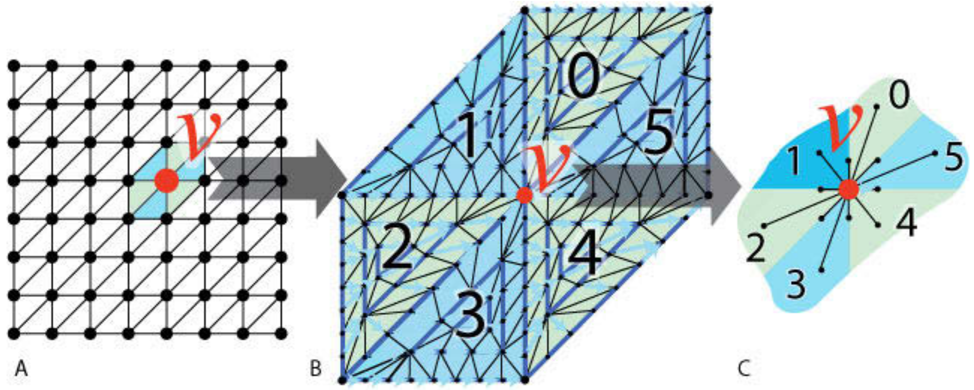


Figure 9. A mesh vertex *v* is shared by 6 PN triangles (A). B shows how *v* is connected to edge and internal vertices in each of the sharing patches 0 through 5. C details those connections.

Finding the drainage direction for a corner vertex amounts to finding the steepest of all possible downhill paths to connected vertices in each of the 6 sharing patches. Since the patches share the same tessellated vertex topology, the search reduces to finding the world coordinate values for the connected vertices in each patch in turn and selecting the vertex connected by the overall steepest downhill path.

Finally, an edge vertex in an interior patch can drain to another edge vertex or to an interior vertex in either of the sharing patches. Every edge vertex is shared by a lower and upper triangle but the barycentric coordinate systems for each is rotated 90° with respect to the other. A mapping function like that for corner vertices determines matching vertices in the sharing triangles and identifies the connected vertices for each. Finding the drainage direction for an edge vertex thus becomes the search for the steepest downhill patch to a connected vertex in the 2 sharing patches. Figure 10 shows the complete drainage direction map for a rendered sample DEM containing 2 patches.

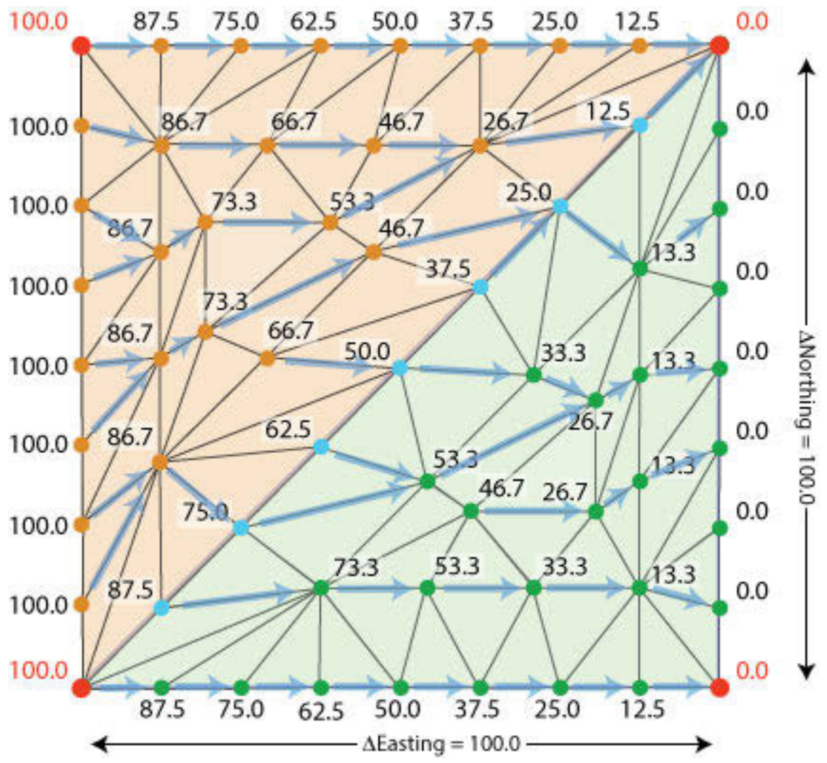


Figure 10. Drainage directions for tessellated evaluation sites in 2 patches. Elevations at sites are in meters.

Drainage accumulation modeling is an iterative process that requires information passing between patches. As in the drainage direction pass, each surface patch is processed separately from other patches on a dedicated processor workgroup. Each processor in the workgroup represents a single vertex. Because OpenGL does not provide a mechanism for workgroups to synchronize with one another during a compute shader pass, this project runs the drainage accumulation model as a series of compute shader passes on the GPU with intermediate data copying steps on the CPU (step 2.2). Before the first pass, the starting reserve and accumulation values for each vertex in all patches are set to 1 and 0 respectively, On each computing pass, any vertex that drains to another vertex in its own patch increments its downhill neighbor's reserve by its own reserve value and decrements its own reserve by the same amount (Figure 11). All cases in which a vertex instance modifies any value associated with another instance is performed by an atomic data operation that guarantees to complete without interruption before any other concurrent operator can access the data. The vertex instance then increments its own accumulation value by the reserve it sent downhill. Processing continues asynchronously at all vertices within a workgroup until no non-pit vertex has a reserve greater than 0. Even if a given vertex has a reserve equal to 0, it idles in case a vertex upstream has a reserve that has not yet made its way downhill to its position. This is necessary to assure the correct operation of the procedure on subsequent passes where patch-local ridges, but no other vertices, have reserves initialized to non-zero values.

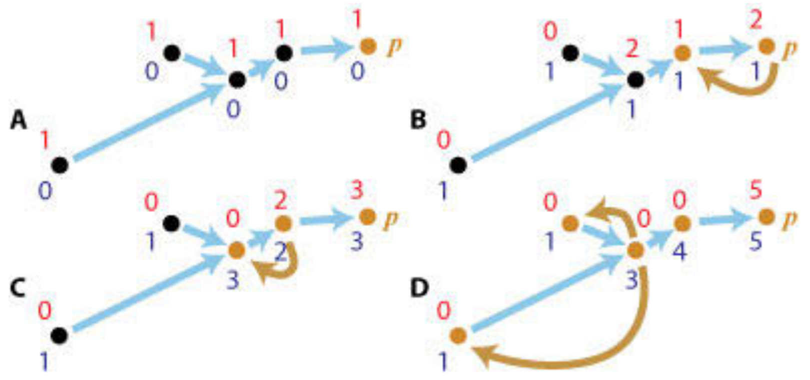


Figure 11. State of reserve (red), accumulation (blue), and basin label (brown) values after each accumulation modeling pass (A through D) for a portion of a drainage network.

After the completion of a pass, control returns to the client. Local pits associated with corresponding ‘ridge’ vertices (vertices with no patch-internal uphill neighbor) in adjacent downhill patches copy their reserve values to their counterparts and reset their own reserve values to 0 (Figure 12). On each subsequent pass, non-zero reserves at the ridge vertices propagate through the patch, incrementing accumulation values at each passing vertex. Accumulation processing ends when all reserves in the system have drained to global pits (patch internal pits or pits on the edges of the PN triangle mesh).

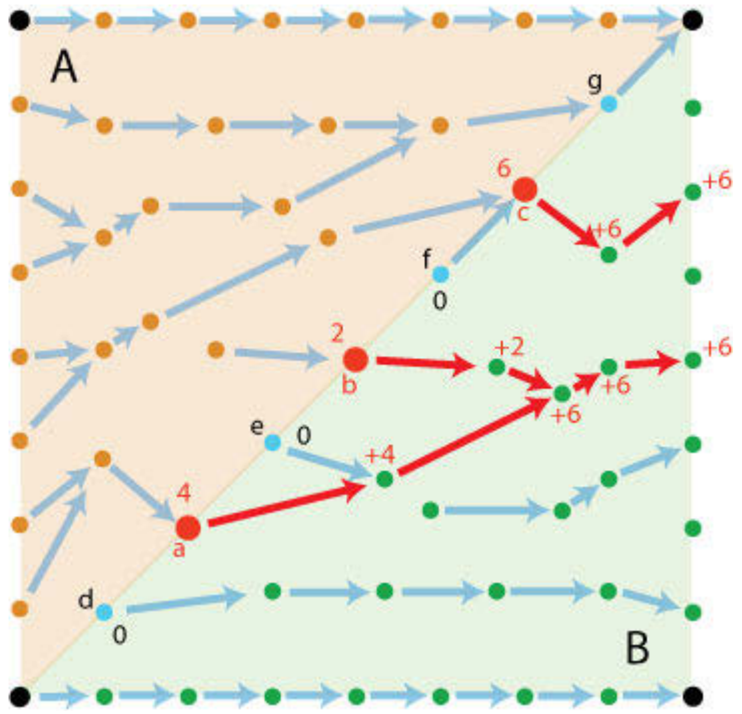


Figure 12. Local pit vertices a, b, and c in patch A have corresponding ridge vertices in adjacent patch B. At the end of a pass, a client process copies reserves at a, b, and c to their counterparts in B. On the subsequent server pass, ridge values in B drain to their local pits, adding vertex accumulation values (in red) along the way. Although vertices d, e, and f are ridges in B, they have no corresponding pits in A and so contribute no more accumulation within B. Vertex g drains along the edge to another vertex in A and therefore does not contribute to accumulations in B.

On all passes, the total reserve values in a basin are constant. At the end of processing, the accumulation value at a pit equals its basin's total reserve. Basin labeling can be performed at the same time as drainage accumulation without requiring additional iterations (refer to Figure 11). Pits are initialized to a basin ID equivalent to their unique vertex identifier and all other vertices are initialized to null. Whenever a vertex writes its reserve to its downhill neighbor (DHN) (or queries it on an otherwise idle cycle), it copies the DHN basin label (either an assigned ID or null) to its own. Therefore, basin labels propagate uphill as reserves move downhill. Between passes, basin labels are copied uphill from local ridge cells to their corresponding local pits in uphill patches.

Results

Illustrations and performance statistics for the implementation 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.

At the end of drainage accumulation modeling, each vertex has an accumulation value equal to 1 plus the number of its uphill cells. This value can be used for further modeling operations or illustrative purposes. For this project, the accumulation values are used to shade drainage segments to illustrate accumulation values. Drainage basin labeling is used both to illustrate basin membership and also to normalize accumulation values within basins. Figure 13 shows perspective views of an 8 x 8 patch grid (with 2 PN triangles per square grid cell) representing a hemisphere with colors representing basin membership. Since pit accumulation values equal the total reserves in a basin, an accumulation value at a vertex can be normalized by its pit's accumulation, providing a consistent way to compare stream importance within and between basins. Figure 14 illustrates the same 8 x 8 x 2 mesh rendering segments in pure RGB blue with alpha (transparency) varying with normalized flow.

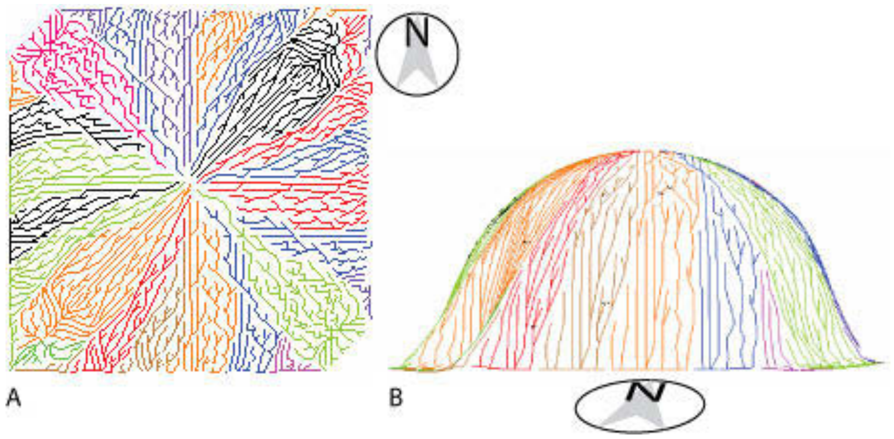


Figure 13. Perspective drainage basin views for a hemispherical sample database covering an 8 x 8 x 2 triangle mesh. A shows a view from directly overhead with north at the top. B is a view from the side, looking north. In both cases, segment color indicates its drainage basin membership. All lines are 1 pt. thickness.

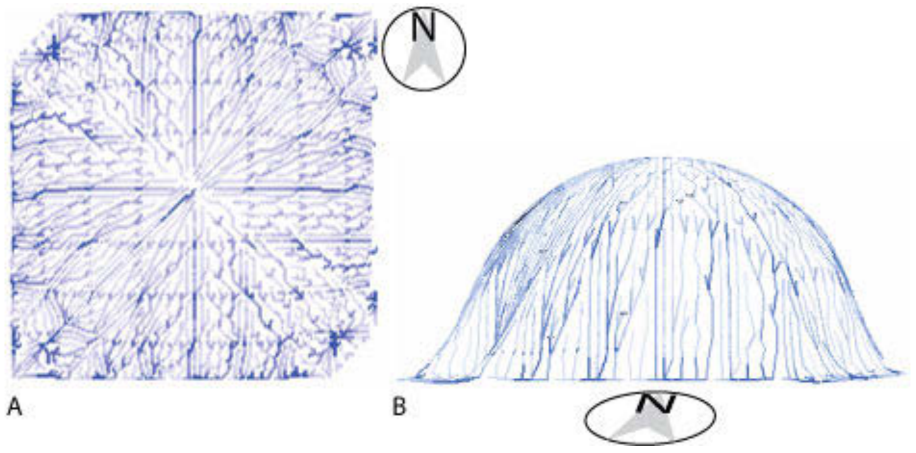


Figure 14. Stream segments shaded by normalized accumulation, using same data and viewpoints as in Figure 13. The normalized flow for a segment is shaded as a pure RGB blue color with alpha (transparency) varying with normalized flow. Segments are rendered in 2 pt. lines for clarity.

One of the biggest challenges for running compute shaders in the OpenGL rendering pipeline is to complete an analysis before a video timeout can occur. Video systems, such as those based on the NVIDIA chipset used for this project, monitor the amount of time that a process blocks refresh attempts. When the process exceeds a preset value, the video driver crashes ungracefully, leading to a black screen or a system reboot. Although the preset timeout value can be extended, it is more desirable to find and fix the processing bottlenecks that led to the condition. In this project, building the drainage direction map dominated all other activities, accounting for approximately 773 milliseconds (ms). Since default video timeout values are often in the range of 2 seconds or less, analyses for larger meshes could easily exceed this value.

The expected running time of the drainage accumulation procedure depends on surface complexity. In the best case, a stream flowing across a monotonically sloping landscape parallel to a patch grid axis would require a number of passes linearly dependent on the grid dimension along that axis. However, as a stream channel oscillates across the grid, the number of passes increases with the number of patch crossings (Figure 15).

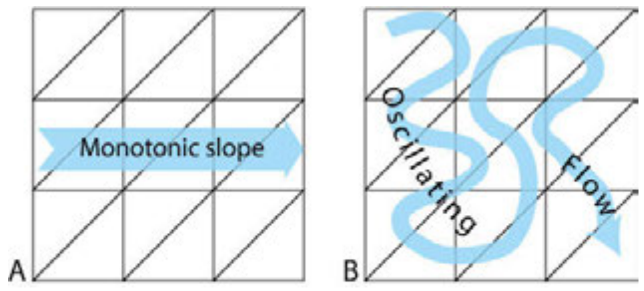


Figure 15. In A, a channel running from the left to the right crosses 6 patches, requiring 6 server passes to drain. In B, an oscillating channel makes 24 patch crossings, requiring as many passes to complete.

Each drainage accumulation run of the 8 x 8 x 2 patch dataset requires approximately 60 ms of processing time, including server side and client side operations. This is less than 1/10 the processing time required to create the drainage direction map.

Future Work and Conclusion

Clearly, the first step toward increasing the efficiency of this program must be to reduce the time required to create the drainage direction map. A relatively simple, short-term solution to this problem is to partition drainage direction mapping into regions that can be processed independently in multiple passes. By assuring that the number of tessellated vertices does not exceed the number of physical processors in a given pass, and that control is returned to the client (the CPU) between passes, processing will be as efficient as possible and video timeout detection will reset between passes to avoid system crashes.

However, this approach does not reduce the total workload required to make the drainage direction map. In this exploratory project, each patch determines edge and corner vertex drainage independently of all other patches. This leads to highly redundant drainage direction mapping for all mesh-internal edge and corner vertices in the mesh. This can be fixed easily by creating edge and corner data structures so that vertices in each category can be processed once and only once. Given that each internal edge vertex is currently processed twice, and each internal corner is mapped 6 times for every frame, the savings should be very substantial. Some more time can be saved by partitioning tessellation vertex data out to the processor workgroups that refer to them. In OpenGL server-side processing, dedicated workgroup memory provides faster data access than does global shader storage buffer object memory, the current repository for all tessellated vertex data.

Beyond the exploration of efficiencies, future versions will operate on TIN elevation models to avoid the generation of internal pits. Since a TIN mesh has irregular topology, supporting data structures will be required to explicitly state corner and edge adjacencies that are implicit in the regular grid model. Internally, all TIN patches have the same topology (an internal triangular tessellation with a fixed number and placement of edge and internal sampling points) but differ on their internal geometry. Since OpenGL tessellates all patches with the same winding order (clockwise or counterclockwise, as determined by user settings), consistent rotations of the barycentric coordinates of adjacent triangles will limit side adjacency coordinate matching to 3 cases, allowing for relatively simple lookup table construction and access. Fortunately, these data structures can be created in 1-time operations preceding the rendering loop.

Finally, future versions will allow for rendering meshes with multiple tessellation levels. Since 1 of the 3 barycentric coordinates for an edge vertex measures the ratio of its distance along the edge relative to the bounding corners, it can be used in real-time to identify the closest matching vertex on the same edge in an adjacent patch with different tessellation levels. Then it can access the connected vertices for that vertex and search for the steepest downhill path as for the fixed grid mesh model.

OpenGL compute shaders provides a mechanism for combining analytical operations and rendered output in real-time. Although concurrent programming on GPUs is often unintuitive, it is nonetheless important to consider its relevance for GIS and cartography applications. As future computing environments move away from monolithic CPU architectures and toward distributed, concurrent computing models, such models may also become the norm in coming application development cycles.

References

Bradley, C.J. (2007) *The Algebra of Geometry: Cartesian, Areal and Projective Coordinates*. Bath: Highperception.

Carr, C. (2015) How NVIDIA GRID Makes Geographic Info Systems Available in Any Geography. https://blogs.nvidia.com/blog/2015/07/20/nvidia-grid-gis-esri/ Last visited 6/22/2016.

Khronos Group (2016) History of OpenGL. https://www.opengl.org/wiki/History_of_OpenGL Last visited 6/22/2016.

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. Online at http://d2dtl5nnlpfr0r.cloudfront.net/tti.tamu.edu/documents/120-1.pdf Last visited 6/22/2016.

Meza, J. (2015) Virtualizing ArcGIS Pro. https://blogs.esri.com/esri/arcgis/2015/05/28/49638/ Last visited 6/22/2016.

Mower, J.E. (1994) Data Parallel Procedures for Drainage Basin Analysis, *Computers and Geosciences*, 20, 9, pp. 1365-1378.

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, pp. 323-344.

Piegl, L. and Tiller, W. (1997) *The NURBS Book*, 2nd ed. Berlin: Springer-Verlag.

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, pp. 159-166.

James E. Mower, Department of Geography and Planning, State University of New York at Albany, Albany, NY 12222