The FEV Frontend: A Terrain and Water Simulation Client

Donald Brent Sharrow
University of Mississippi

The FEV Frontend: A Real-Time Terrain and Water Simulation Client

Donald Brent Sharrow

A thesis submitted in partial fulfillment
of the requirements for the degree of

Master of Science
Engineering Science
Computer Science

University of Mississippi

2011

Abstract

The FEV (Flood Emergency Visualizer) Frontend is a Java and OpenGL-based 3D visualizer of flood simulation results in real-time. Simulations can be recomputed based on user-defined terrain edits made directly on the rendering surface. The frontend is the client software of the FEV system that includes a backend for solving water flow equations using CUDA. To render large datasets, level-of-detail and other optimizations are needed to render terrain and water surfaces with interactive frame rates.

Acknowledgments

The FEV Frontend was developed as part of the *War Games for Flood Emergency Managers (WGFEM)* project, funded by the Southeast Region Research Initiative (SERRI) project, part of U.S. Department of Homeland Security.

Table of Contents

List of Figures

Figure Number Page

List of Tables

Chapter 1
Introduction

Widespread floods often happen unexpectedly. As floodwaters cover a large geographical area, devastation to homes, land, and cities is a real and expected possibility. Once a flood is identified, predicting and tracking its future trajectory is essential and time-sensitive. Flood emergency managers must use this information to make critical decisions in order to mitigate damage by (among other methods) strategically placing protective barriers or trenches in the floodwater's path. A major issue for these flood emergency managers is that current methods of prediction are too error prone to reliably use or too slow to create.

Using a virtual environment to address flood prevention can provide a substantial improvement for flood emergency managers. A simulation ”sandbox” based on real-world data gives the user free reign to view and navigate an area while manipulating various points of interest to closely predict future floodwater behavior. This level of interaction would be impossible in the real world.

There are two major problems facing the creation of such a system. First, suitably-accurate mathematical models for water flow equations are complex and require a lot of processing power to solve not just in a reasonable time frame, but fast enough for the *computation time* (real time) to exceed *simulation time.* Second, a 3D visualization of a data-heavy simulation needs to be run on a computer with suitable graphics hardware so interaction with the scene is smooth and without delay. Also included in this visualization is the ability to make changes to the environment and subsequently see an updated simulation appear in real-time. This document addresses the second point by describing the theory and implementation of the *FEV Frontend.*

Chapter 2

Background

2.1 The Flood Emergency Visualizer (FEV) System

The frontend was developed as the client component of the *Flood Emergency Visualizer (FEV)* system. Figure 2.1 illustrates the primary components of FEV. The goal of FEV is to provide flood emergency managers (the user) with software to run a simulation *faster than real-time* and do so with *interactive frame rates*. The frame rate, measured in *frames per second (FPS)*, is generally considered to be interactive at around 15 FPS (Akenine-Möller *et al.*, 2002).

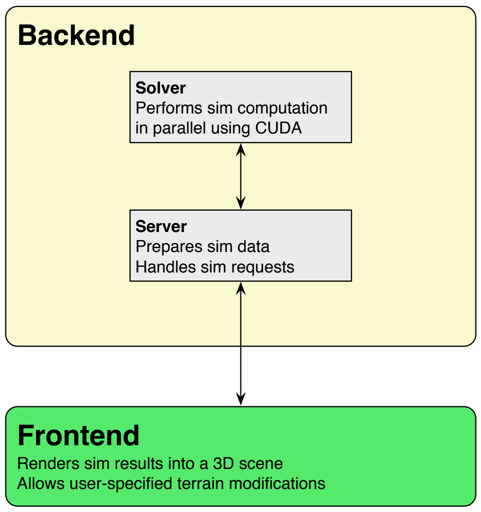
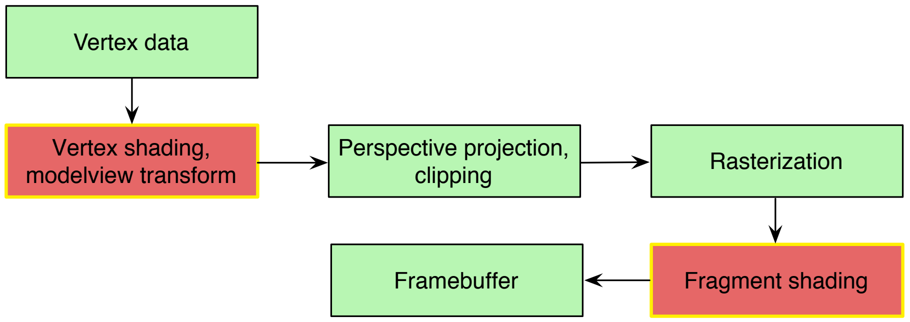

Backend
Solver
Performs sim computation in parallel using CUDA
Server
Prepares sim data
Handles sim requests
Frontend
Renders sim results into a 3D scene
Allows user-specified terrain modifications

Figure 2.1. FEV modules. The frontend sends user-specified simulation requests to the server who prepares this information for the solver.

2.1.1 Frontend Requirements

The frontend serves two purposes: visualizing the results of a user-specified simulation and allowing the user to make changes to a simulation by navigating a 3D scene in real-time. These changes are done by editing the terrain surface. Terrain edits are sent to the backend so it can recalculate a simulation. As the recalculation is computing, new simulation data is streamed to the frontend and subsequently visualized.



Vertex data
Vertex shading, modelview transform
Perspective projection, clipping
Rasterization
Fragment shading
Framebuffer

Figure 2.2. OpenGL graphics pipeline. Programmable aspects of the pipeline used in the frontend are indicated in red.

2.2 Fundamentals

This section introduces key computer graphics concepts and terms important in describing the frontend's implementation.

2.2.1 The Graphics Processing Unit (GPU)

The *graphics processing unit (GPU)* translates data representing computer graphics into a viewable image for the display. This rendering process uses a group of coordinate points (called *vertices*) to create a triangle, which is the main rendering primitive. A group of connected triangles represents a single *model* or *mesh* in the scene. Data is stored based on the type of GPU: *dedicated* GPUs have their own memory that is directly accessed, while *integrated* GPUs share memory with the CPU and are less powerful. A modern GPU contains many cores which allows computation to be done in parallel. When sending data from CPU to GPU, performance tends to be at its best by infrequently sending large chunks of data rather than frequently sending small chunks of data.

2.2.2 OpenGL

The frontend was developed using OpenGL, a popular API to facilitate graphics rendering. A vertex's journey from a set of coordinates to a pixel on a screen involves several stages, as shown in Figure 2.2. Modern versions of OpenGL allow some stages of the graphics pipeline to be programmable via the OpenGL Shader Language (GLSL), notably vertex processing and fragment processing. The frontend uses custom vertex and fragment shaders for rendering terrain and water surfaces. Shaders take advantage of the GPU's many cores to execute commands in parallel. A primary goal to ensure good GPU performance is to have all data associated with an object processed by the same set of instructions. Conditionals in a SIMD environment such as a shader tend to reduce performance and are avoided wherever possible.

2.2.3 Coordinate Spaces

OpenGL uses a right-handed coordinate system, meaning the x and y axes point right and up respectively, while the positive z axis points out of the screen. Models exist in one of several *coordinate spaces*. *Object space* is simply the local space in which a model's coordinates reside. To be rendered to the screen, a model goes through several *transformations*. There are two matrix stacks for transforming vertices: the *modelview* matrix and the *projection* matrix. The modelview matrix transforms vertices in object space coordinates to *world space* coordinates and ultimately to *view space* coordinates. World space contains all models in the scene. View space transforms vertices so that the camera is centered at the origin (0.0, 0.0, 0.0) which makes camera-relative operations easier. The projection matrix applies perspective to the scene by transforming view space coordinates to *screen space* coordinates resulting in the complete rendered image.

2.2.4 Creating a Camera

OpenGL does not have a concept of a camera moving through the scene; the view is always centered at the origin through a view space transformation. Camera functionality is implemented either by manually translating and rotating models in world space or by creating a custom camera matrix to use. Both methods result in identical modelview matrices but the latter option is a more intuitive solution; this creates an abstraction for a camera which allows it to be treated like any other model that can be transformed in world space. The frontend uses a custom camera.

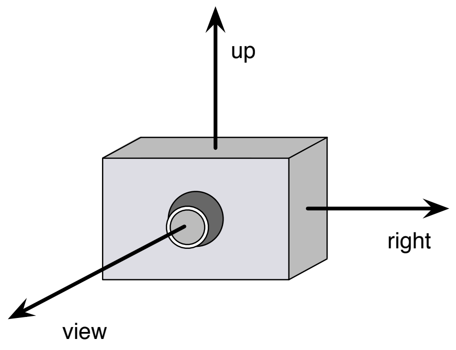


Figure 2.3. Orientation vectors for the camera: the *up vector*, *right vector*, and *view vector*.

Orientation Vectors

The *eye position* is the camera's position in world space. The eye position uses three vectors in object space to orient itself in the scene, illustrated in Figure 2.3.

Movement

Camera movement is done via a third-person orbiting view. In this mode, the eye position rotates around a point called the *target*. The view vector always points towards the target. The target itself can be moved which causes the eye position to be translated as well. The distance from the target to the eye position is the *orbit distance*, which can be manipulated by the user.

Calculating how much to move the camera on each frame is achieved in one of two ways. The first, *frame-based movement*, uses a static value to move the camera a particular distance each frame. This method is simple but can result in jagged movement if the scene is rendered at varying frame rates. The second method is called *time-based movement* because it uses the actual time between rendered frames as a scalar value in conjunction with a static value. This method assures uniform movement regardless of frame rate. A weighted average of past frames provides additional smoothing. The frontend's camera implementation uses time-based movement.

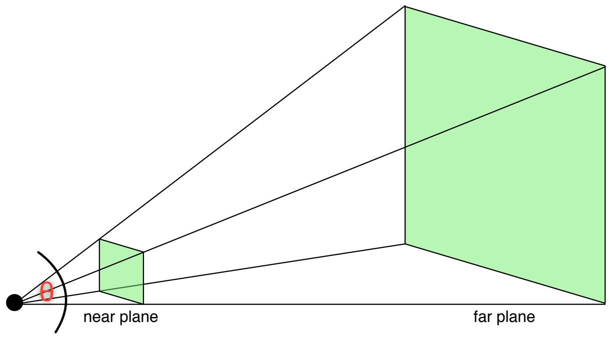


Figure 2.4. A view-frustum. The viewable area is located between the near and far planes. Angle θ is the field-of-view.

View-Frustum Culling

There are often areas of the terrain or water mesh that are not seen by the camera on a given frame. Regardless, OpenGL renders every mesh in the scene. This is wasteful as many thousands of triangles (or more) are drawn, yet are hidden from view. *View-frustum culling* is a method to address this issue. The *view-frustum*, shown in Figure 2.4, is defined by six planes indicating the viewable volume as seen by the camera. From these planes, intersection tests are performed with the mesh to indicate areas that lie inside or outside the frustum.

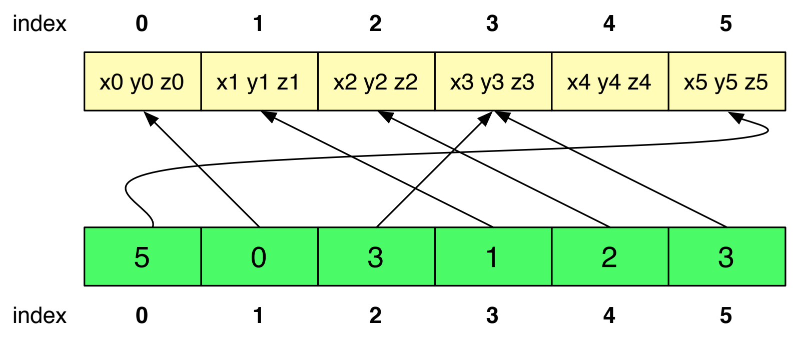


Figure 2.5. Vertex and element arrays. Element arrays are simply instructions to triangulate a mesh.

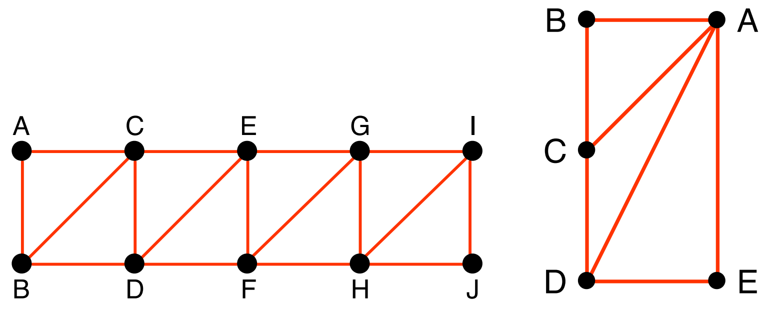
2.2.5 Storing/Accessing Vertex Data

Format

All vertices for a model are normally stored in a single *vertex array.* Sometimes these vertices are stored in triangle order so the GPU only needs a vertex array to produce a mesh from the data. This method is undesirable in many cases. The existing order of vertices is often important; in these cases, an *element array* is used. Shown in Figure 2.5, an element array stores indices into a vertex array in the correct triangle order.

Storage Options

There are two ways to provide vertex data to the GPU. The first is to keep vertex data in CPU memory and send it to the GPU every frame. This method, referred to as *immediate mode*, is costly due to the overhead of sending data from CPU to GPU every frame. A more efficient method is to store vertex data directly in GPU memory through the use a *vertex buffer object* (VBO). A VBO stores a collection of vertices in a contiguous block of GPU memory. For static data, uploading needs to be done only once. Dynamic data is updated by specifying the relevant subset and new data values. Similarly, an element array can be stored on the GPU as an *index buffer object* (IBO). An IBO is treated in the same manner as a VBO.



(a) Triangle Strip (b) Triangle Fan

Figure 2.6. A triangle strip is constructed by listing each vertex in alphabetical order. This results in a total of 10 elements, rather than 24 elements that would otherwise be needed. A triangle fan is constructed first from the central vertex *A* followed by the remaining vertices in counter-clockwise order, requiring only 5 instead of 9 vertices.

2.2.6 Triangle Types

The primary type of triangles used to construct the terrain and water meshes is called *triangle strips*. A single triangle strip is made up of two neighboring rows of vertices, as shown in Figure 2.6(a). Once the first three vertices of a triangle are given, additional triangles on a strip are created by specifying the remaining vertices in alternating fashion among the two rows. Memory storage is efficient because a triangle strip with n triangles can be defined by $n + 2$ vertex references, rather than $3n$ references needed if each vertex in each triangle is specified.

A second triangle type called *triangle fans* (shown in Figure 2.6(b)) is used on the terrain mesh. A triangle fan consists of several triangles that are connected by a single vertex. Memory storage with triangle fans is similar to triangle strips; n triangles are defined by $n + 2$ vertex references.

2.2.7 Dataset Format

Datasets for terrain and water surfaces used by the frontend are represented by a *height map*. A height map is a collection of evenly-spaced data points distributed across two dimensions. Each data point corresponds to an elevation (or height, represented in meters) at a two-dimensional grid location. To create a three-dimensional mesh, some metadata is also required, including width, height, and grid spacing. Grid spacing defines how far apart (in meters) each elevation value is from its neighbors. A vertex is created for each data point of a height map; its x and z coordinates are equal to the two-dimensional grid location of the data point multiplied by the grid spacing. The y-coordinate for a vertex is the data point's height value which *warps* the surface into three dimensions.

2.2.8 Level of Detail Overview

A brute-force approach to rendering a height map based terrain surface builds triangle strips that extend across the entire domain. This works when rendering with relatively small datasets but real-time rendering of large datasets is barely achievable. Rendering performance suffers once the number of data points reaches the order of several million, especially with machines containing low-to-modest GPUs. *Level-of-detail (LOD)* algorithms provide a method to solve this problem by removing unnecessary triangles while still providing an accurate model for display.

LOD algorithms aim to increase rendering performance by reducing the number of triangles in an object while maintaining a level of accuracy that sufficiently represents its original appearance. LOD behavior depends on how far an object is from the camera. When far away, an object's triangles are drawn on a small number of pixels on the screen. If there are many thousands of triangles, computation is wasted on unnecessary detail—unnecessary because the detail is not discernible.

Determining an appropriate resolution for an object requires a compromise between accuracy and performance. Removing too little detail slows performance, causing sluggish or uneven interaction. Removing too much detail introduces noticeable artifacts. This balance is important when integrating an LOD algorithm because it must be run "behind the scenes" to maintain an illusion of smooth animation to the user.

Terrain LOD

Initially, LOD algorithms applied to terrains operated on a per-triangle basis (Lindstrom *et al.*, 1996)(Duchaineau *et al.*, 1997)(Röttger *et al.*, 1998). The goal of these algorithms was to build an optimal or perfect set of triangles for a mesh on every frame. This process is inherently CPU bound and quickly creates a bottleneck that slows down other parts of an application. As GPUs rose to prominence in the early 2000s, minimizing CPU to GPU communication and keeping the GPU as busy as possible became far more important than building a perfect set of triangles (De Boer, 2000)(Pajarola & Gobbetti, 2007).

2.2.9 Texture Mapping

One method of coloring a mesh is through the use of *texture mapping.* A *texture* is stored on the GPU and is commonly represented in two dimensions (although one and three dimensional textures are also possible). Textures store an image and its metadata: width, height, pixel format (i.e. RGB), and data format (i.e. unsigned short). The base unit of a texture is a *texel.* A texel can be stored as an 8-bit value up to a 32-bit floating point value on modern GPUs. Texture coordinates are vertex attributes that define where texels are placed on a mesh; this provides freedom to stretch a texture across a mesh or repeat it an arbitrary number of times.

2.2.10 Phong Shading Model

$$I_p = k_a i_a + k_d i_d + k_s i_s \tag{2.1}$$

Shown in Equation 2.1, the *Phong Shading Model* (Phong, 1975) defines the illumination I_p of a point on a surface as the sum of the *ambient*, *diffuse*, and *specular* terms. The ambient term ($k_a i_a$) is a fixed color representing the "base" lighting of a scene. The diffuse term ($k_d i_d$) represents light that reflects in all directions on a surface; diffuse light is view-independent. The specular term ($k_s i_s$) is view-dependent and represents light reflecting in one particular direction. These terms are commonly represented as three separate RGB values. Values of k are simply constant values such that $k_a + k_d + k_s = 1$.

$$i_d = N \bullet L \tag{2.2}$$

This model (Equation 2.1) is computed on either a per-vertex or per-pixel basis. Both methods start with a vertex normal N and a light direction L. The diffuse term calculation is shown in Equation 2.2. With per-pixel shading N and L are interpolated across each pixel before the calculation is made. Per-vertex shading interpolates colors across pixels after evaluating the shading equation only at the vertices. For some surfaces (i.e. terrain), it is sufficient to ignore the specular term. Other surfaces (i.e. water) are heavily influenced by the specular term as it reveals important visual characteristics.

2.2.11 Depth Buffer

A *depth buffer* is a collection of depth values for each pixel. Each depth value (in screen space) is set according to the object(s) closest to its associated pixel. As objects are rendered, a depth test for each pixel determines if new depth values are closer to the screen than the current value and if so, the depth value is overwritten. The precision of a depth buffer is generally 16 or 24-bit, and depth values range from 0 to 1 where 0 is closest to the screen and 1 is farthest away.

Chapter 3

Frontend Foundations

3.1 Terrain

The frontend is required to run on machines with modest GPUs. Real-time rendering of a terrain surface is difficult on the frontend without the use of LOD, as described in Section 2.2.8.

3.1.1 LOD Implementation

Block-based LOD algorithms operate on groups or blocks of triangles. Operating with a large set of triangles (rather than individual triangles) minimizes CPU computation and CPU to GPU communication which in turn keeps the GPU as busy as possible.

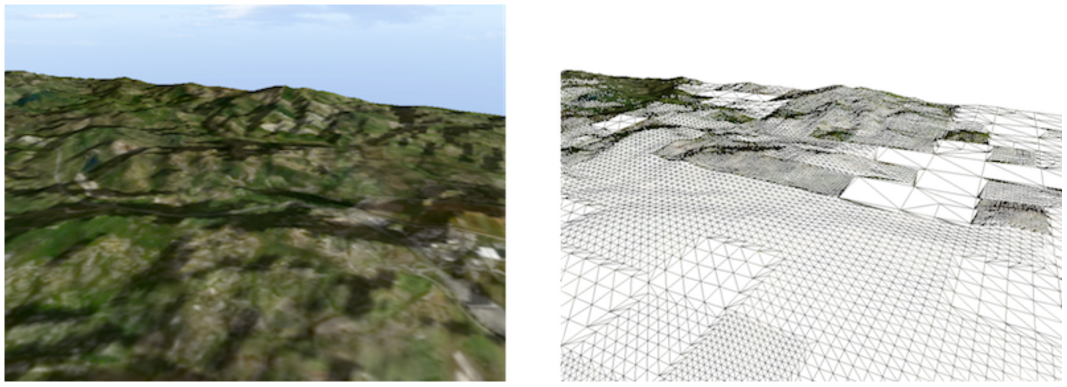


Figure 3.1. An LOD terrain surface in textured and wireframe modes.

Block-based LOD

The frontend uses block-based LOD based on the *Geomipmapping* algorithm (De Boer, 2000). This algorithm divides a height map into equally-sized square blocks (referred to

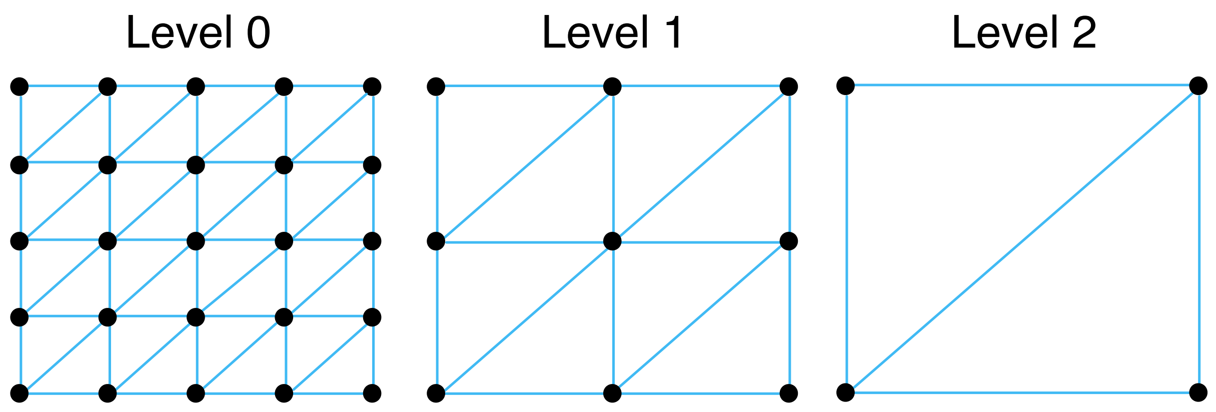


Figure 3.2. An LOD block at several resolutions, from the highest resolution to the lowest. Note how every other vertex in a row and column are removed as the resolution decreases. The number of resolution levels is related to an LOD block's dimension; an LOD block size n has $log_2(n-1)$ levels.

as *LOD blocks*). Each LOD block is independent in that its resolution is determined using only the data points (height values) it contains. At run-time, each LOD block calculates its appropriate resolution for the next frame. An LOD block's resolution is adjusted by adding or removing every other vertex from each row and column. This means not only must an LOD block's dimension be square, but it must have dimensions $(2^n + 1) \times (2^n + 1)$ to accommodate the pattern described. Figure 3.2 provides a visual example of triangulating an LOD block at several resolutions.

An important consideration to be made of an LOD block is its shape. The *compactness* of an LOD block refers to the ratio of its sides' lengths. An LOD block square in shape has a high compactness as each side is of equal length. A more disproportionate rectangular shape is not compact as it may cover the same total area as a square but it extends further in one dimension.

When choosing the dimensions for an LOD block, a balance must be made between block overhead and visual accuracy. A small set of large LOD blocks creates less overhead but visual "popping" can occur due to large resolution differences between neighboring LOD blocks. Additionally, because contents of an LOD block change uniformly, decreased locality increases the risk of unnecessarily setting coarser areas of an LOD block to a high resolution, which is wasteful. On the other hand a large set of small LOD blocks may create too much overhead which increases the number of CPU to GPU transfers. Minimizing these transfers is a high priority.

Calculating Resolutions

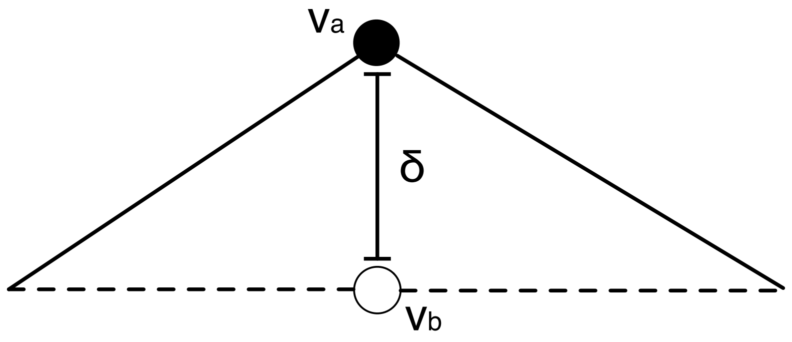


Figure 3.3. Difference (error) between a vertex's actual height value v_a with its interpolated value v_b.

Selecting an appropriate resolution for an LOD block requires a measurement of error against some globally-defined error threshold. For each resolution level $l > 0$ in an LOD block, an error value ϵ is computed. Figure 3.3 shows the calculation of a difference value δ; a vertex's y-coordinate (height) v_a is subtracted from an interpolated value v_b at the same xz position. This is done for all removed vertices of l. The *maximum delta* δ' is then chosen and projected into screen space using the distance from the LOD block's center to the camera's current eye position. This projection, ϵ, represents the visual error in pixel units. The global threshold τ is also in screen space; a typical value for τ is 3 or 4 pixels. The value of l is incremented as long as the condition $\epsilon < \tau$ holds. Once this condition breaks, l is used as the LOD block's resolution for that frame.

Calculating ϵ for every l in every LOD block is far too CPU intensive to maintain real-time rendering frame rates, and ϵ cannot be precomputed with the camera's current eye position as it is only accessible during run-time. Despite this, ϵ can be partially pre-computed if the camera's view vector is assumed to be fixed parallel to the xz plane; in this scenario, values of δ (which can be precomputed as well) are at their maximum. The distance approximation in screen space, D_l, is then made using a fixed camera view vector and δ'. This value is stored for each l in an LOD block. Finally, during run-time, the distance from the LOD block's center to the camera's current eye position is computed and compared with D_l to select a resolution.

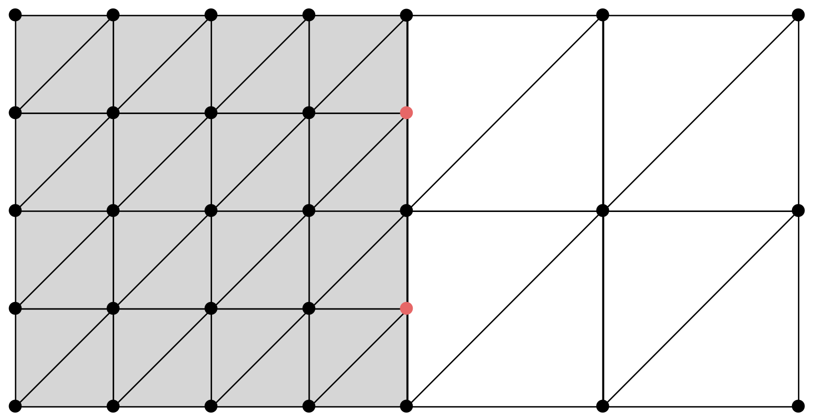


Figure 3.4. Two neighboring LOD blocks at different resolutions. The gray LOD block uses two vertices at a neighboring edge (indicated in red) that create cracks in the mesh because they are not connected to the right LOD block's triangles.

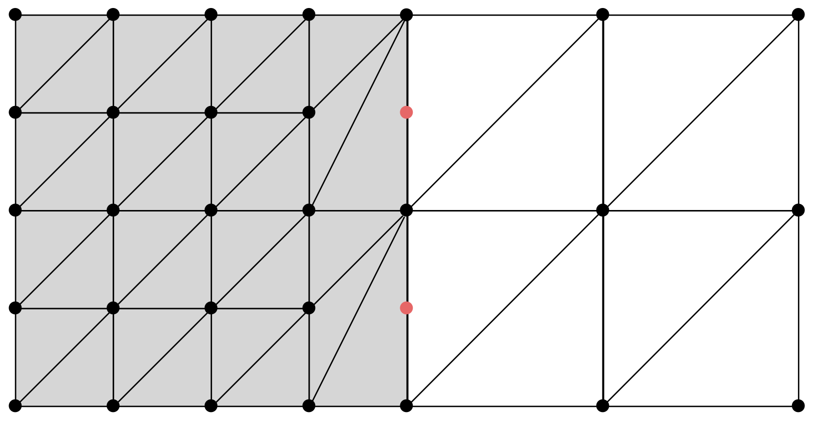


Figure 3.5. Solving the cracks problem. Triangle fans are created on the higher resolution LOD block to "sew" together the neighboring edge. The red vertices are not rendered.

Rendering a Block

Despite an LOD block's ability to calculate its resolution independently, it does require some information about its neighbors in order to properly render the mesh. Neighboring edges of LOD blocks share vertices, but a low resolution LOD block does not use all the vertices available. Cracks or gaps appear on neighboring edges where neighboring LOD blocks differ in resolution, creating a discontinuity in the mesh, as shown in Figure 3.4. The cause of this problem is due to interpolated height values at the location of missing vertices; they are not equal to the actual height value in a vertex from a neighboring higher resolution LOD block. The procedure to solve cracks in the mesh (shown in Figure 3.5) involves using triangle fans, as described in Section 2.2.6, to effectively "sew" neighboring LOD blocks at different resolutions.

3.1.2 The Quadtree

During rendering there must be a method to quickly traverse the terrain's LOD blocks so their resolution can be calculated and afterwards drawn. A common two-dimensional structure for spatial subdivision is a quadtree (Samet, 1984). A quadtree works by subdividing its area into four equal areas, each of which continue to subdivide until a specified limit is reached. The dimensions of a quadtree in its typical form are $2^n \times 2^n$, where n is a positive integer. As previously mentioned, the LOD algorithm's blocks are square. Additionally, a terrain mesh can be seen as a two-dimensional surface. Thus a quadtree is well-suited to contain a terrain mesh, and LOD blocks become the leaf nodes in the tree. A quadtree is also important for fast view-frustum culling, as described in Section 2.2.4.

3.1.3 The Non-Square Case

Not all datasets used by hydroscientists have square dimensions, so the ability to support datasets with arbitrary dimensions is an important feature for the FEV frontend. Such a feature gives the frontend flexibility to handle a wide array of datasets prepared by hydroscience engineers for specific flood scenarios. This feature presents a problem for LOD rendering. Block-based LOD techniques typically use LOD blocks with dimensions $(2^n + 1) \times (2^n + 1)$ in order to correctly triangulate the mesh at different resolutions (as previously shown in Figure 3.2). For this problem of the non-square case with an LOD algorithm, two approaches were considered. We could use rectangular blocks, or we could alter the dataset.

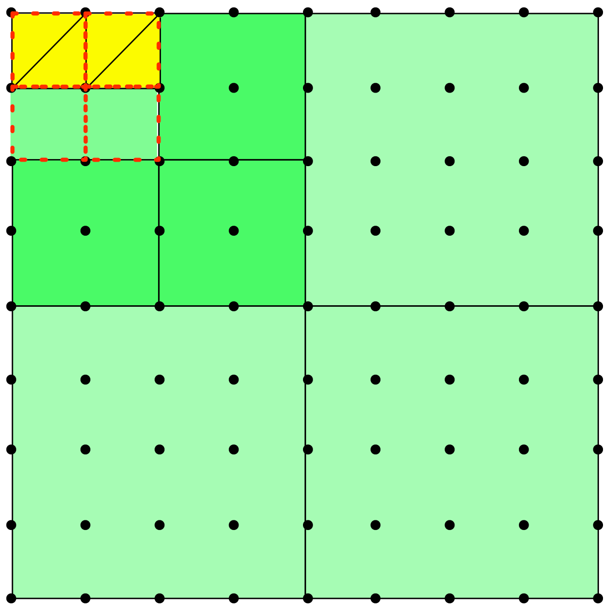


Figure 3.6. A possible rectangular LOD block is shown in yellow. The quadtree node's children, shown in red, are the leaves.

Using Rectangular Blocks

The first approach is to expand an LOD block's dimensions to $(2^n + 1) \times (2^m + 1)$ where $m \neq n$. For instance, an LOD block may be of the size 33×17, where $n = 5$ and $m = 4$. This allows a greater range of datasets to be used without modification, but there are two consequences to consider when allowing LOD blocks of this type.

Consider an LOD block shape A with dimensions 17×17 and another LOD block shape B with dimensions 65×5. Both blocks have a similar number of vertices (289 and 325 respectively). When the camera's view vector is positioned in parallel with a set of LOD blocks of shape B, some vertices in LOD blocks near the camera are distant from view while other vertices are closer. Because an LOD block is set to one resolution at any given time, high-resolution shape B LOD blocks near the camera may have additional wasted triangles extending further from view. This can also be a problem for square blocks however the issue is certainly exaggerated as LOD blocks become narrower. The cause of this problem is the lack of compactness of an LOD block in relation to the camera. Square LOD blocks extend the same distance in all directions so they have a much better locality and uniformity from the camera's standpoint, resulting in fewer visual inaccuracies.

Let an LOD block shape C have dimensions 17×9. A 1025×1025-sized dataset with shape A LOD blocks results in a total of 4096 blocks. LOD blocks of shape C on the same dataset results in 8192 blocks, double the total from type A. This problem stems from the need to balance an LOD algorithm between visual accuracy and performance. In this case, doubling the number of blocks doubles CPU to GPU communication cost while halving the number of vertices per block. Despite an increase in accuracy due to more LOD blocks, performance will likely suffer.

The sole reason for implementing more rectangular LOD blocks is to fit a set of these blocks on rectangular datasets without resorting to altering the dataset. Because each dimension of an LOD block must be of the form $2^n + 1$, the variety of datasets able to perfectly fit LOD blocks is still limited. Additionally, rectangular LOD blocks tend to be either too narrow to display without noticeable inaccuracies or too small, thus creating too many blocks. Adapting an LOD algorithm to use rectangular blocks for the non-square case is not a desirable situation for the frontend to manage.

Altering the Dataset

The second approach modifies the dataset to adapt to non-square datasets. The scope of alteration on a dataset is dependent on the size of an LOD block. That is, a dataset using LOD block dimensions of 33×33 alters at most 32×32 values per block.

There are two options in altering the dataset, the first of which is by *padding*, where LOD blocks at the edge of the dataset add extra data so they adhere to the defined square dimension; these LOD blocks are filled with null height values. One drawback to this approach is a possible performance hit from the addition of more vertices which creates more triangles for the GPU to draw. A second drawback to padding is that special care must be done during rendering to identify such areas. We do not want the user to believe padded height values are actual height values; hence, there must be code to visually distinguish padded areas. Detecting "special" vertices in a SIMD shader is not ideal; it creates conditional code that reduces performance.

The second option to alter the dataset is *trimming*. In this case, actual height values are removed from the edges of the domain until the LOD blocks at the edge are of the defined dimensions. This process is done by simply "fitting" as many LOD blocks as possible on the height map; this maximizes the coverage on the dataset and limits the data lost. Remaining data points outside the LOD blocks are discarded. Removing data can be troublesome as it may be of interest depending on the simulation.

The number of removed vertices when trimming depends on the dimensions of an LOD block. For instance, LOD blocks with size 33×33 on a dataset with dimensions 1100×440 results in only 2.9% of the vertices being removed. Even this small penalty is avoided completely if the dataset is composed of an integer number of LOD blocks. An advantage of trimming is that shading the mesh uses the same operation for all vertices, so it does not need a conditional case in a shader as padding does. By default, the FEV frontend uses trimming to adjust non-square datasets.

Adapting the Quadtree

A main advantage of using a quadtree for subdivision is its simplicity and fast traversal. If a quadtree's leaf dimensions are adjusted to precisely fit a non-square dataset, a problem occurs. The frontend wants a quadtree's leaf nodes to correspond to the dimensions of an LOD block. For this to be true, a quadtree's dimensions must be some factor of an LOD block's dimensions. By simply "fitting" a quadtree onto a dataset, there is a very strong possibility that leaf nodes will be misaligned with LOD blocks, as shown in Figure 3.6. As a result, blocks are detected in multiple nodes, requiring additional work to determine which leaf node should be associated with a particular LOD block.

To handle the non-square case, a quadtree's dimensions are determined by using the closest power-of-two greater than a dataset's largest dimension. The resulting quadtree covers a significantly larger area than the dataset in many cases which requires additional, unnecessary node visits. Additional node traversals are limited by adding some metadata to each node of a quadtree. Each node contains a flag indicating whether a node's area contains data. This results in nodes with no data stopping traversal of their child nodes. A handful of node visits outweighs a quadtree that must spend additional time determining the nodes assigned to LOD blocks. The quadtree maintains its simplicity while guaranteeing LOD blocks are grid-aligned.

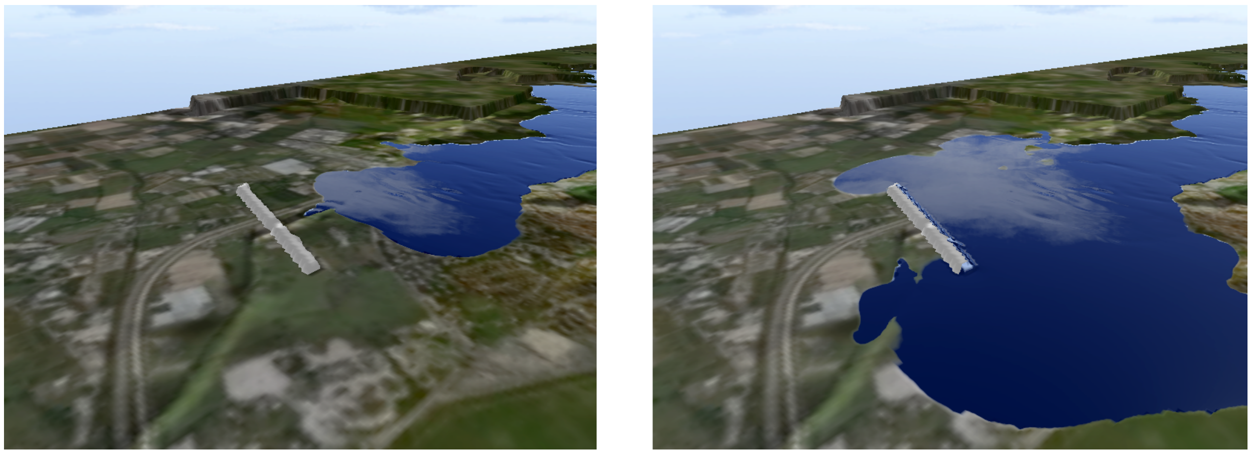


Figure 3.7. An example terrain edit. On the left, an edit is drawn in the path of oncoming water. On the right, the water's path is changed as a result of the terrain edit.

3.2 Terrain Editing

A core feature of FEV is the ability to change the outcome of a simulation by editing the terrain surface. An example terrain edit can be seen in Figure 3.7. The basic requirements set forth for this task are simple: the user must be able to draw “edit lines” of a specified width that raise or lower the terrain surface by some defined value. This action is performed in real-time and in 3D. Additionally, the ability to discard or bring back unwanted terrain edits must be an option. Furthermore, these edits are color-coded based on whether they raise or lower the terrain. From the listed requirements, several data structures are needed to store the information on an individual terrain edit, manage the collections of past and present edits, and render the present edits on the terrain surface.

3.2.1 Creating an Edit

The area of the terrain surface a particular edit represents is stored as a list of two-dimensional grid coordinates. These coordinates are the terrain cells intersected by a user-defined two-dimensional line. Determining intersected cells of the terrain is done using Bresenham’s line algorithm (Bresenham, 1965), an algorithm originally created to determine which pixels should be drawn on a screen to represent a line in raster form. Since the terrain mesh is a regular grid, much like pixels on a screen, this algorithm can be applied in a similar fashion. Edits are drawn by the user in screen space by drawing a two-dimensional line across a desired area of the terrain. To translate the user’s line to world space coordinates, each two-dimensional point (in screen space) is first used to retrieve the closest depth value from the depth buffer. This value and the other two coordinates are then projected from screen space to world space using the current modelview and projection matrices. The result is a three-dimensional point that is used by the Bresenham algorithm.

How an edit behaves on the terrain surface is specified in two ways: an edit *type* makes a distinction on whether the terrain’s current height values are modulated or replaced and an edit *operation* specifies if the edit should raise or lower the existing height values. An edit also contains the magnitude at which it alters the terrain surface.

To support ”undo” and ”redo” functionality, two linked lists must be maintained. A terrain edit is contained in only one of the two lists at a given time. Recently-created edits are stored in an *active* list. If an edit from the active list is discarded, it is transferred into the *undo* list. Edits from the undo list are restored into the active list at the user's request, allowing an arbitrary number of undos and redos.

3.2.2 Adapting Edits to LOD

As stated earlier, the LOD algorithm in FEV is based on a static mesh. A major reason the LOD algorithm can be effective is its ability to keep error value calculation in the initialization phase so run-time performance does not falter. Allowing edits on the terrain surface, however, changes the characteristic of the mesh to be somewhat dynamic. This presents a problem when edits are performed on flatter areas of the terrain; LOD blocks believe they should be at a low resolution when in fact the edit should cause an LOD block to be at a much higher resolution. A recalculation of error for affected LOD blocks does not work as it removes the sense of real-time interactivity to the user. If the LOD block simply ignores the edit and proceeds normally, there is a high probability the user will see an inaccurate representation of the terrain edit, so the LOD algorithm must be modified.

The adaptation is straightforward: for terrain blocks containing edited areas, LOD is turned off. This may appear to create a significant issue as LOD blocks containing just a single edited value are set to full resolution regardless of its "flatness" property. In practice, terrain edits tend to be localized and normally there are only a few areas of real interest. The amount of editing on the terrain is done on quite a small percentage of the total set of blocks so the performance cost is minimal.

3.3 Water

In common simulation scenarios, the water mesh is updated twenty times per second or more, cycling through hundreds of height maps or *timesteps.* Technically, a static LOD algorithm could be applied to all the timesteps in a simulation. The only data unique to each timestep would be the results of the error calculations.

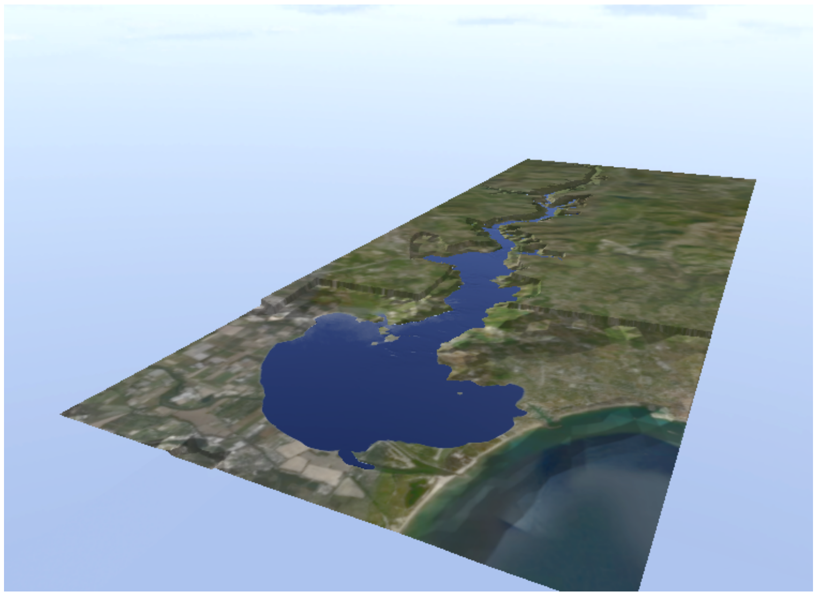


Figure 3.8. The water surface of the Malpasset dam break case.

Adapting a static LOD algorithm to a dynamic water surface presents some problems. First, the results of a simulation's computation are not known until its ultimate arrival at the frontend. Timesteps are streamed in one at a time from the FEV backend so error calculations have to be done during run-time. These error calculations normally take on the order of a few seconds to compute. A few seconds of computation for 200 timesteps adds roughly 15 minutes delay the user has to sit through. In our minds, this crosses the line between interactivity and semi-interactivity; it hurts the user experience we wish to provide.

A second problem concerns the appearance of an LOD water mesh. Visual properties of a water surface and terrain surface differ significantly. Terrains tend to be diffuse with dull colors. A water surface has a significant amount of reflection and high specular coefficient which reveals more of the surface texture. Generally, reducing detail on a water surface is far more noticeable than reducing detail on a terrain surface. Low-resolution LOD blocks of water simply do not look good enough to be of use.

3.3.1 Water Blocks

Even though we cannot use LOD with the water surface, the frontend can still use the same block structure as the terrain to subdivide the water mesh. The only difference is every block of water must be rendered at the highest possible resolution since there is no LOD. Subdividing the water surface presents three primary advantages. First, the water mesh's blocks (i.e. *water blocks*) can be set to the same dimension as LOD blocks, meaning they are represented as the leaves of a quadtree exactly like the corresponding terrain LOD blocks. Secondly, view-frustum culling can be applied in the same manner as the terrain. Third, observing the water surface's coverage area allows further culling to be done.

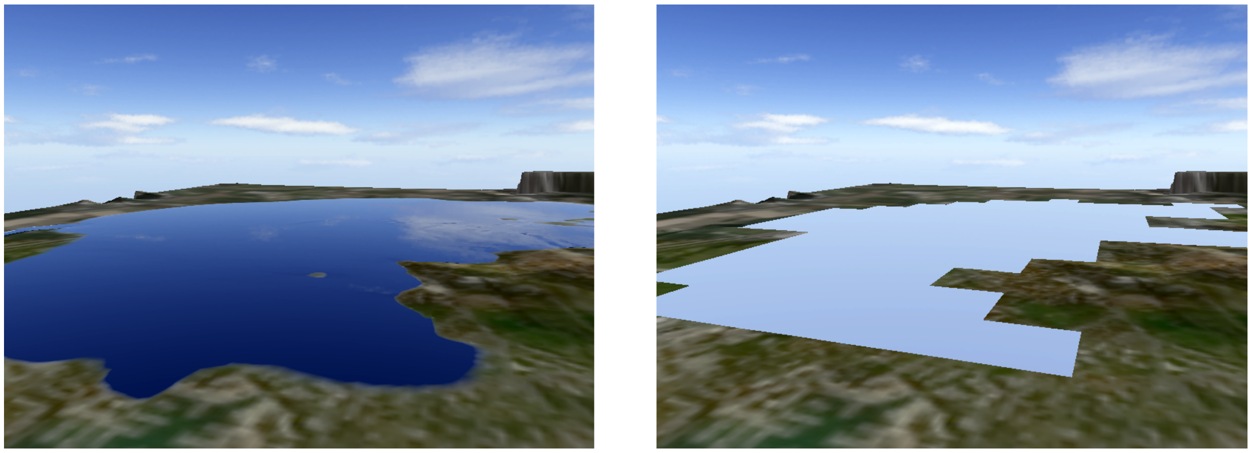


Figure 3.9. Large pool of water on the left. On the right, the water surface is hidden from view to reveal terrain blocks culled from Wet Block Classification.

3.3.2 Wet Block Classification

This advantage of subdividing the water mesh is best seen by first observing the amount of water coverage existent during a typical simulation. Rarely does a water surface cover even half of the overall domain in tested simulations. In most scenarios, the water surface area is a small percentage of the domain. A consequence of this is that in many areas of the domain, the water mesh is not visible. Thus, water blocks can be culled. Determining whether a water block is a wet block (has non-zero depth values) or a dry block is simple and fast to calculate so it is done as timesteps are received at run-time. The water renderer stores this metadata per block for each timestep which results in a memory increase of mn bytes, where m is the number of blocks in a timestep and n is the number of timesteps. Overall this is a small memory hit; for example, the metadata for a 1025×1025 dataset with water block dimensions 33×33 and 200 timesteps takes up just 204KB of space in RAM.

We classify water blocks with the following terms: *dry* blocks do not contain any positive water depths, *partial* blocks have some positive water depths, and *wet* blocks are completely filled with positive water depths. Water blocks have the same dimensions as terrain LOD blocks, so they are given access to its associated terrain LOD block at the same two-dimensional grid location. The benefit is that a wet water block fully covers its associated terrain LOD block; in this case, the wet water block can notify the terrain LOD block that it is not visible and does not need to be rendered. Culling terrain LOD blocks in this manner has an added advantage of further balancing performance as the water surface area increases. While not perfectly balanced (water blocks are at full resolution), both extremes of water coverage are optimized; when coverage is low, the majority of water blocks are culled and when coverage is high, many terrain LOD blocks are culled.

Chapter 4

Implementation

To satisfy the requirement of portability to Windows and Apple machines, the frontend is written in Java. The OpenGL 2.1 API is written in C, so a third-party library is used. Java Bindings for OpenGL (JOGL) (jog, 2011) provides Java access to all the functionality of OpenGL via several object-oriented interfaces.

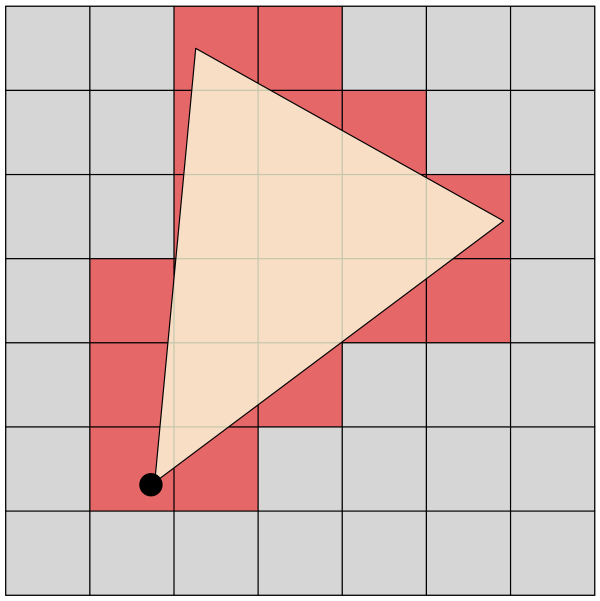


Figure 4.1. View-frustum culling on a terrain surface. The camera's view-frustum is indicated in yellow. Red blocks are visible terrain blocks that are rendered, gray blocks are not seen and therefore culled.

Each quadtree node contains a *bounding sphere* to be used for view-frustum culling. The bounding sphere defines the area enclosed by a node by using a point indicating a node's center along with a radius that extends along the dimensions of a node. As the quadtree is traversed, each full node performs an intersection test with the view-frustum. This intersection test is performed until the complete set of visible LOD blocks is determined, shown in Figure 4.1.

4.1 Shading Terrain

The terrain surface is shaded using one large texture attached as part of a dataset's metadata provided by hydroscientists. Additionally, lighting is performed on the texture itself. An LOD terrain cannot perform real-time lighting without inaccuracies—many vertices are unused so lighting is performed at varying resolutions, resulting in a displeasing appearance. For this reason, lighting is precomputed.

4.1.1 Shading Edits

Because edits are color-coded on a terrain surface, an auxiliary data structure is needed to identify these areas. The frontend refers to this structure as an *edit mask*. The edit mask is an array of bytes where each element represents one cell of a terrain surface. When the active or undo list is updated, the edit mask must be updated as well. In order for the terrain shader to correctly color edited areas, the edit mask must be organized and uploaded as a texture for the shader to read.

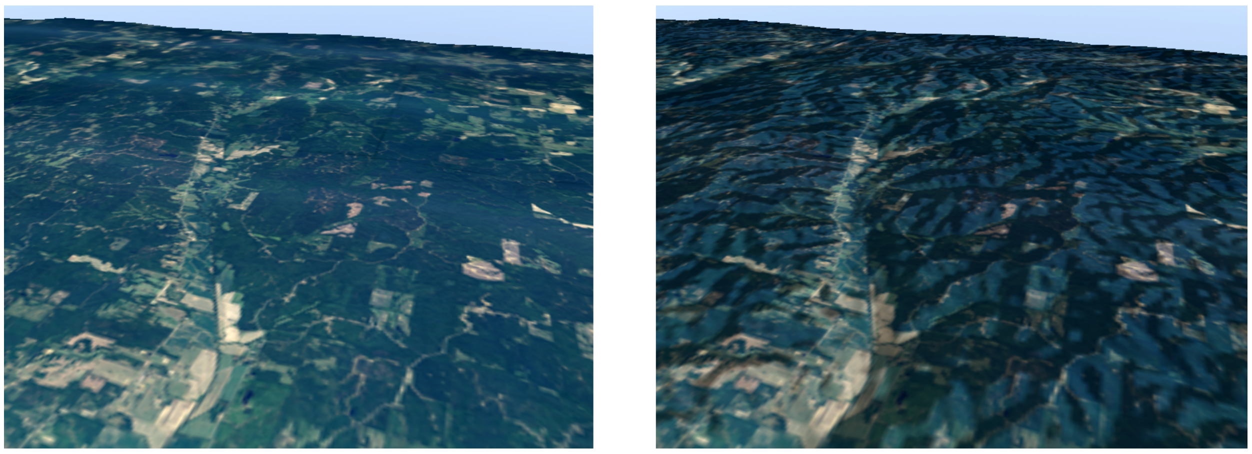


Figure 4.2. The difference between a terrain surface with and without lighting calculations.

4.1.2 Lighting

Vertex normals are calculated on the CPU. There is one static directional light used. Lighting calculations are done in the CPU and uploaded as a floating-point texture. This value is read in the vertex shader but processed in the fragment shader; this interpolates the value so per-pixel lighting is possible. The resulting light value is "baked" onto the texture (see Figure 4.2), resulting in both a textured and lit terrain surface.

4.2 Shading Water

Shading the water surface puts more burden on the shaders than the terrain does. Due to the differences between terrain and water surfaces (as discussed in Section 3.3), the water surface requires additional effort.

4.2.1 Vertex Texture Fetch

The x and z coordinates of a vertex are set by a particular two-dimensional grid location in a height map; these two coordinates never change. Uploading vertex data through a Vertex Buffer Object calls for all three coordinates from each vertex to be specified. For the water surface, the current timestep is updated often; two-thirds of the data uploaded via a VBO are unnecessary (because the x and z coordinates never change). Today's GPUs are at a real advantage here: they are optimized in creating and handling textures and are much faster than the process of uploading VBO data.

In a process commonly known as *vertex texture fetch* or *vertex displacement* (Kryachko, 2005), textures (often floating-point) are used in conjunction with a shader as a means to displace incoming vertices. A height map suits a texture well as both are two-dimensional regular grids. For the frontend, a single floating-point texture is stored on the GPU and read in the water surface's vertex shader. Vertex data is initialized once as a VBO for the water surface; the y-coordinates initially specified in the VBO are not meaningful as the vertex shader overrides these values.

4.2.2 Calculating Normals

If vertex normals for each timestep were to be computed on the CPU, there are two viable options: either a significant increase in memory to store them all or a reduction in performance to do calculations in real-time as the current timestep is updated. Computing normals on the GPU has (like many other calculations) a definite speed advantage. Despite this, a shader still needs to perform lookups of neighboring vertices and several cross products for just one vertex normal.

$$n_v = \frac{\sum_{i=0}^{n} n_i \, a_i}{\sum_{i=0}^{n} a_i} \tag{4.1}$$

$$n_v = \{w - e, 2, n - s\} \tag{4.2}$$

(Shankel, 2002) describes a more efficient method of calculating vertex normals in a height map. Equation 4.1 is the general method of computing a vertex normal n_v; for each neighboring face from i to n, a face normal n_i and its angle a_i with the vertex is averaged together. This computation requires many cross product calculations that are costly during run-time. Because a height map is regularly-spaced, computing normals can be simplified to using only the differences between neighboring heights as shown in Equation 4.2. A vertex normal n_v is computed from the elevation values of its west (w), east (e), north (n), and south (s) neighbors.

4.2.3 Reflection/Refraction

$$R_i = 2I(N \bullet I)N \tag{4.3}$$

Modeling the reflective behavior of a water surface is a view-dependent task; the amount of the specular contribution to intensity changes as the camera moves in the scene. Figure 4.3 shows the relevant vectors involved in reflection and refraction. The reflection vector R_i

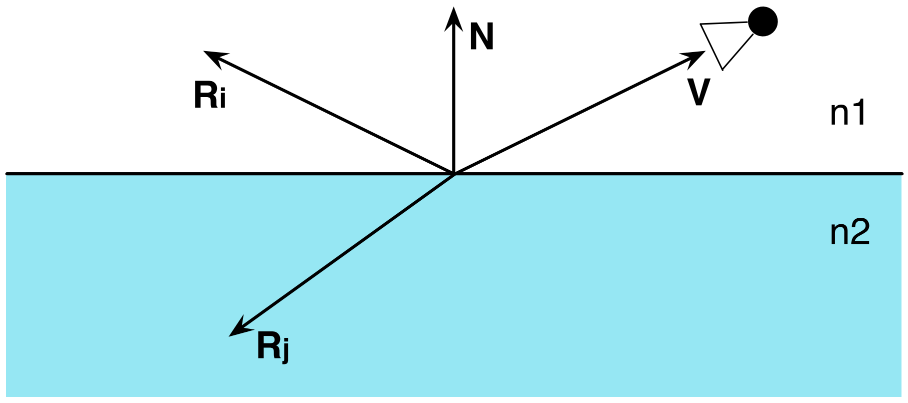


Figure 4.3. Reflection and refraction between two surfaces with indices of refraction $n1$ and $n2$. Vector V is formed from a point on the surface to the camera. V is projected onto the surface normal N to find the reflection vector R_i.

is solved by Equation 4.3, where N is the vertex's unit-length normal and I is the incident vector found by subtracting the camera's eye position from the vertex position. Functions for solving reflection and refraction are provided by GLSL and need only N and I as inputs.

Fresnel Equations

$$f + (1 - f) * (1 - V \bullet N)^5 \tag{4.4}$$

$$f = \frac{(1 - \frac{n1}{n2})^2}{(1 + \frac{n1}{n2})^2} \tag{4.5}$$

The Fresnel equations use the angle of incidence between two boundaries (in the frontend's case, air and water) to approximate the amount of reflection and refraction present. These equations show that the reflectivity of a surface increases as the angle of incidence decreases; this explains why a water surface is more reflective standing next to it versus observing it from an airplane.

The frontend uses Schlick's approximation (Schlick, 1994) of the Fresnel equations which provide "good enough" visual results with relatively fast computation. This approximation is solved by Equation 4.4, where the reflectance of a surface f is given in Equation 4.5. Variables $n1$ and $n2$ are the indices of refraction for the two surfaces.

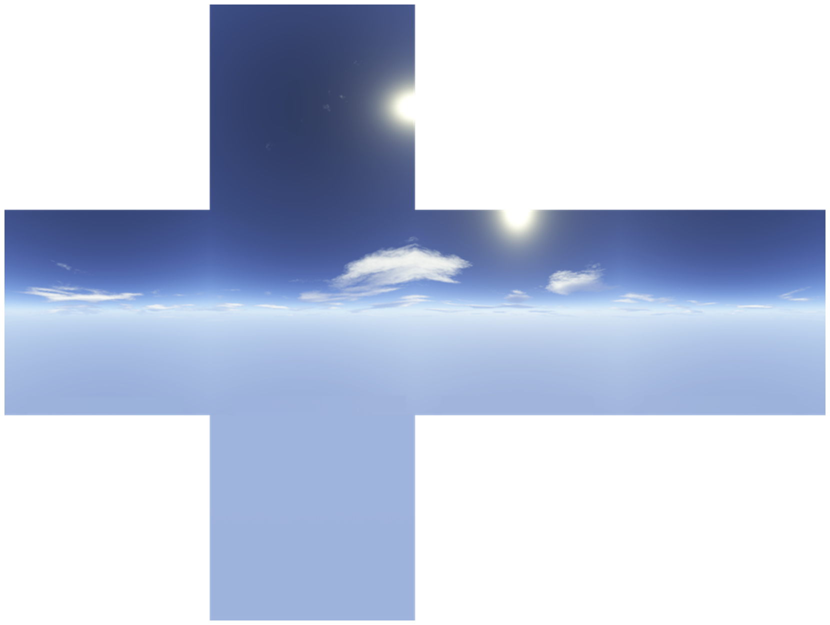


Figure 4.4. Six images in a cube map layout representing a skybox.

Skybox

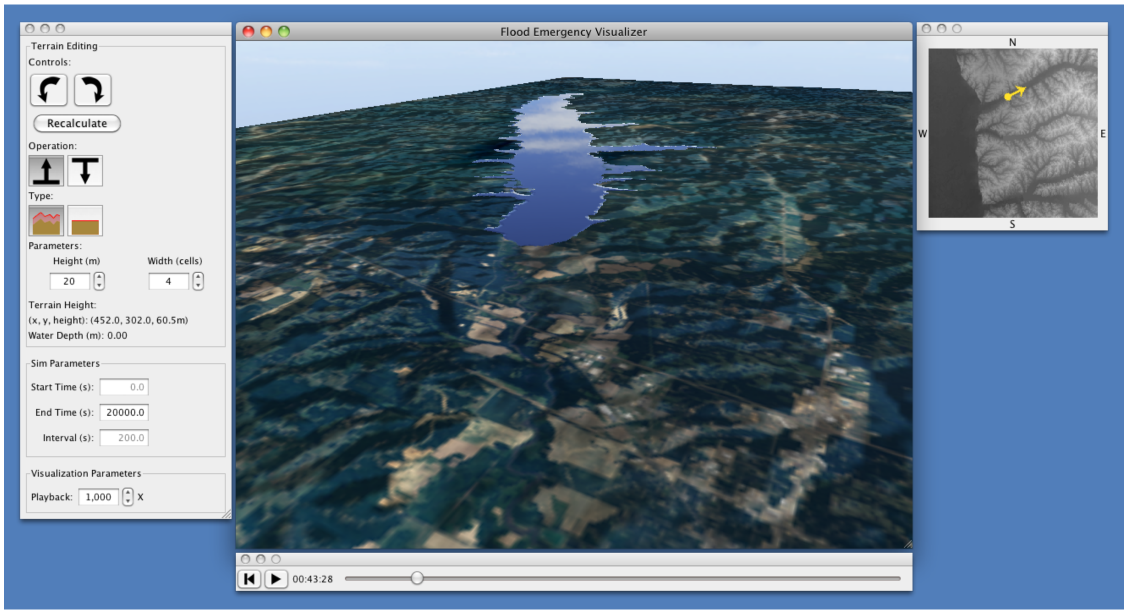
Real-world reflection is simulated with the help of a *skybox*. A skybox is a set of seamless images that surround the scene to give the appearance of a true world. A *cube map* neatly stores the skybox as a single texture (see Figure 4.4). A skybox must give the illusion of being infinitely far away from view to appear realistic. To appear stationary, the skybox is rendered after the camera has rotated but before it has been translated. Additionally, a skybox is rendered with depth testing turned off so it remains in the background of the scene.

The reflection vector is used to do a 3D texture lookup on the skybox so the water surface reflects the sky. The refraction color must come from the terrain's own texture so the refraction vector is resolved to 2D texture coordinates for a texture lookup.

4.3 Terrain and Water Coherency

A common computer graphics issue known as *z-fighting* is created when planes on the verge of intersection are too close for the depth buffer's precision (described in Section 2.2.11) to handle, resulting in distracting visual artifacts between the two surfaces. In the frontend, z-fighting can arise when areas of the water surface are located over low-resolution terrain areas, especially when the interpolated terrain heights may rise above the respective water heights (at full resolution). OpenGL's *glPolygonOffset* method suppresses the problem in very close cases, but if overused the artificial offsetting between the terrain and water surface becomes noticeable.

The partial and wet block classifications discussed in Section 3.3.2 can be used to the frontend's advantage in dealing with terrain and water coherency issues. For partially wet blocks, corresponding terrain blocks have their LOD turned off to minimize the problems with z-fighting. Since fully wet blocks cover their corresponding terrain blocks, these areas of the terrain can be culled.



Parameters:
Height (m): 20
Width (cells): 4

Terrain Height:
(x, y, height): (452.0, 302.0, 60.5m)
Water Depth (m): 0.00

Start Time (s): 0.0
End Time (s): 20000.0
Interval (s): 200.0

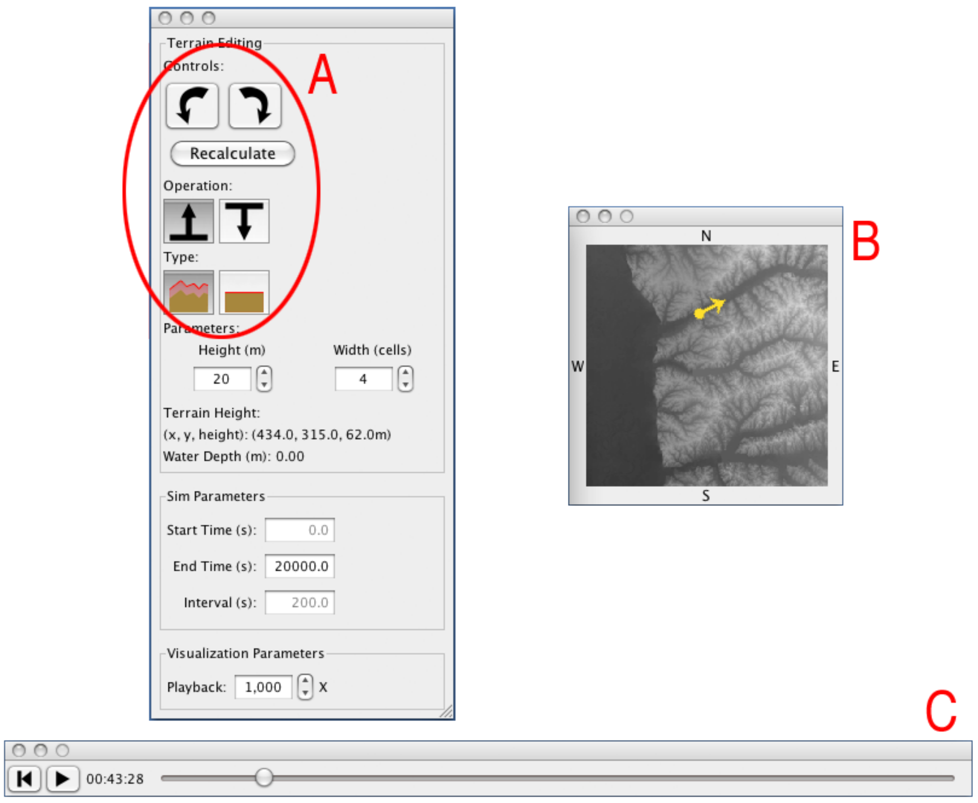
Playback: 1,000 X

00:43:28

Figure 4.5. **The frontend's graphical user interface (GUI).**

4.4 GUI

The frontend's user interface is shown in figure 4.5 and figure 4.6. Playback controls provide the user the ability to play, pause, or move the simulation to a particular moment in time. The GUI's terrain editing toolbar provides options for creating a terrain edit. Under "controls", buttons provide options to undo and redo created edits, while the button labelled "recalculate" sends edits to the backend and begins a new calculation from the current simulation time. Under "operation", options to raise or lower the terrain are given. The buttons under "type" determine whether the created edit keeps the existing surface intact or completely replaces the edited area. On the right, the map provides a point of reference for the viewer's current location and direction.



Parameters:
Height (m): 20
Width (cells): 4
Terrain Height:
(x, y, height): (434.0, 315.0, 62.0m)
Water Depth (m): 0.00

Sim Parameters
Start Time (s): 0.0
End Time (s): 20000.0
Interval (s): 200.0

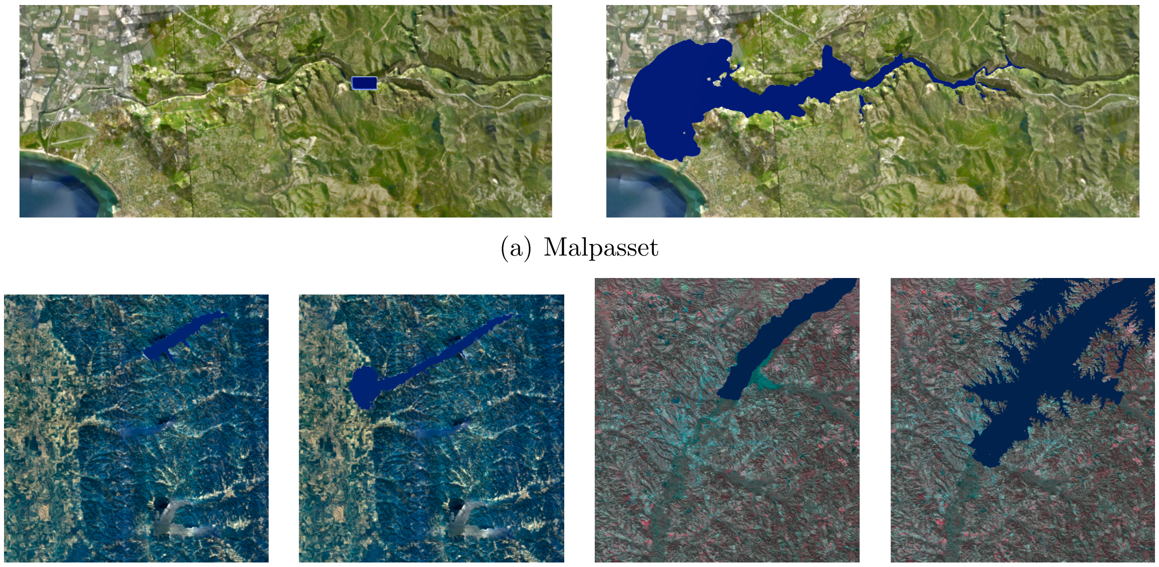
Visualization Parameters
Playback: 1,000 X

00:43:28

Figure 4.6. Closeup of the primary GUI elements. The terrain editing toolbar is shown on the left (A). A reference map is located on the right (B). At the bottom (C) are the simulation playback controls.

Chapter 5

Evaluation



(a) Malpasset

(b) Sardis

(c) RossBarnett

Figure 5.1. Datasets used for the tests. The first and last timesteps for each dataset are shown.

Measuring "success" in the frontend is done primarily by testing terrain and water rendering performance. Overall performance is measured in either the time to render a frame (in milliseconds) or in frames per second (FPS): the *rendering frame rate.* The first set of tests begin with a basic, naive rendering implementation then separately introduce LOD to the terrain and Wet Block Classification to the water surface. The last test case represents the complete rendering system in the frontend. Each test was run on three different datasets, during which the water surface animates at a rate of five times per second; this is the *animating frame rate* that defines how often the water surface updates to a new timestep. To help isolate the performance of terrain and water rendering among the test cases, unless otherwise noted, these results do not include the benefits of view-frustum culling.

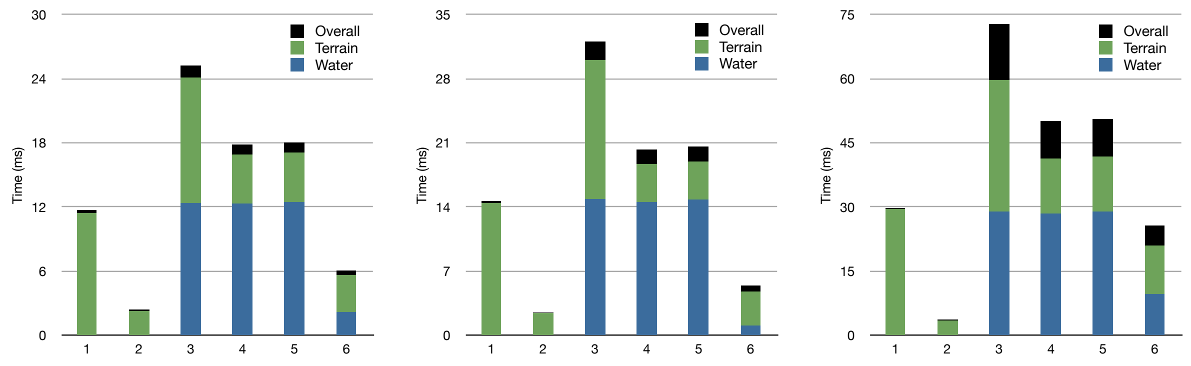
With the exception of Section 5.3, tests were gathered on a machine with a 2x2.26GHz quad-core Intel Xeon CPU and a 1GB NVIDIA GTX 285 GPU. Java's virtual machine was allotted 4GB of memory for all tests.

Dataset/Dim	Test	FPS	Overall Time (ms)	Terrain Time (ms)	Water Time (ms)	Triangles
Malpasset 1100x440	1. Terrain without LOD, No water	85	11.7	11.4	N/A	917,885
	2. Terrain w/ LOD, No water	416	2.4	2.3	N/A	96,070
	3. Terrain without LOD, Water at full res	40	25.2	11.7	12.4	1,853,726
	4. Terrain w/ LOD, Water at full res	56	17.8	4.6	12.3	1,193,065
	5. Terrain w/ LOD, Water at full res in tiles	55	18.0	4.6	12.5	1,193,065
	6. Terrain w/ LOD, Water at full res in tiles w/ WBC	166	6.0	3.4	2.2	368,862
Sardis 1026x1026	1. Terrain without LOD, No water	68	14.7	14.4	N/A	2,062,878
	2. Terrain w/ LOD, No water	400	2.5	2.4	N/A	258,903
	3. Terrain without LOD, Water at full res	31	32.1	15.2	14.9	4,156,692
	4. Terrain w/ LOD, Water at full res	49	20.3	4.1	14.5	2,517,448
	5. Terrain w/ LOD, Water at full res in tiles	49	20.6	4.1	14.8	2,517,448
	6. Terrain w/ LOD, Water at full res in tiles w/ WBC	184	5.4	3.8	1.1	524,557
RossBarnett 1949x2113	1. Terrain without LOD, No water	34	29.8	29.5	N/A	7,856,529
	2. Terrain w/ LOD, No water	272	3.7	3.5	N/A	782,801
	3. Terrain without LOD, Water at full res	14	72.7	30.8	28.9	15,692,054
	4. Terrain w/ LOD, Water at full res	20	50.1	12.9	28.4	10,828,256
	5. Terrain w/ LOD, Water at full res in tiles	20	50.6	12.9	28.9	10,828,256
	6. Terrain w/ LOD, Water at full res in tiles w/ WBC	39	25.6	11.4	9.6	5,124,584

Table 5.1. Terrain and water rendering performance statistics. Times are listed in milliseconds.

5.1 Terrain and Water Rendering

Table 5.1 and the charts in Figure 5.2 lists the terrain and water rendering performance results. The number of triangles in a frame directly correlates to performance. Terrain with LOD (test 2) causes a 90%, 87%, and 90% drop in triangle count from its full-resolution counterpart (test 1) in the Malpasset, Sardis, and RossBarnett datasets respectively, resulting in 5x to 8x gains in FPS. Test 3 performs the worst of all the test cases; this is expected as it is the unoptimized case. Comparisons of tests 4 and 5 show that dividing the water surface into tiles has a seemingly negligible effect on performance.



(a) Malpasset (b) Sardis (c) RossBarnett

Figure 5.2. Rendering performance charts for the datasets using the data from Table 5.1. Shorter bars indicate better performance.

Water Coverage		
Dataset	**Min**	**Max**
Sardis	5.86%	11.72%
Malpasset	3.05%	30.12%
RossBarnett	16.67%	37.08%

Table 5.2. Minimum and maximum percentage of wet blocks for each dataset.

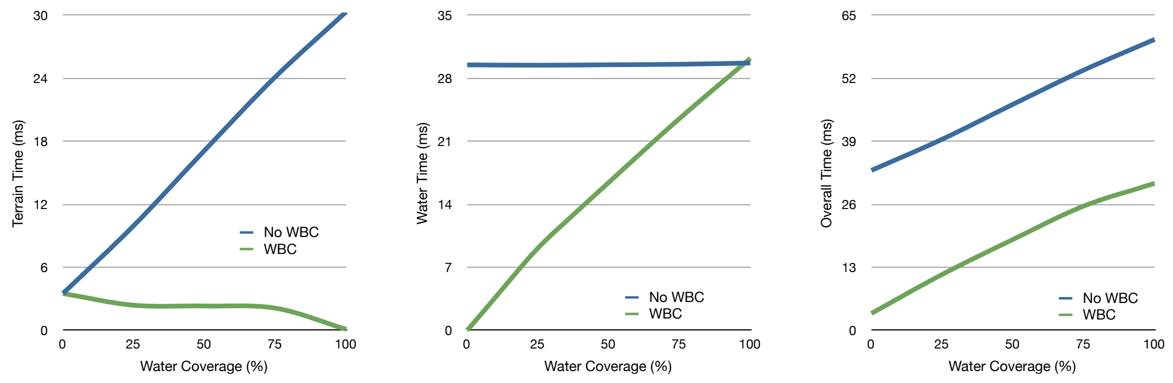
Test 6 (the fully-optimized case) benefits most from discarded triangles from the water mesh's dry blocks identified by WBC. As a whole, WBC causes a triangle count drop of 64%, 87%, and 47% from test 5 to 6 in the Malpasset, Sardis, and RossBarnett datasets respectively. Additionally, test 6 is the only case where water rendering performs better than terrain rendering for all datasets. Table 5.2 lists the water coverage percentages of these datasets. We can infer from these numbers that dry blocks are in abundance for the tested real-world datasets.

5.2 Wet Block Classification

To observe how water coverage relates to performance, the RossBarnett dataset was given an artificial "wall" of water covering an increasingly large area of the domain. Table 5.3 and figure 5.3 provide the results with this dataset. Test 6 differs from test 5 only in

Water Coverage (%)	Terrain Time (ms)		Water Time (ms)		Overall Time (ms)	
	Test 5	Test 6	Test 5	Test 6	Test 5	Test 6
0	3.52	3.52	N/A	N/A	3.52	3.52
25	9.98	2.40	29.46	9.16	39.44	11.56
50	17.14	2.33	29.51	16.44	46.65	18.77
75	24.14	2.12	29.57	23.54	53.71	25.66
100	30.34	0.11	29.71	30.24	60.04	30.35

Table 5.3. Performance statistics of Wet Block Classification. Smaller time values indicate better performance.



(a) Terrain (b) Water (c) Overall

Figure 5.3. Charts illustrating the effect of Wet Block Classification. Smaller time values indicate better performance.

that it includes WBC, so the comparison between the two cases is used to give some insight of WBC's performance. By increasing the maximum water coverage in 25% intervals, we can analyze the impact of three factors: the terrain mesh discarding more blocks that are submerged, the water mesh gaining more triangles, and wet terrain blocks rendering at high resolution.

With the rise in water coverage, water performance (figure 5.3(b)) decreases linearly while terrain performance (figure 5.3(a)) increases slightly until it is completely submerged. Ideally we would like terrain performance to more closely complement water performance, but the effects of discarding submerged terrain blocks cannot fully create such a balance. A closeup of the terrain's performance is shown in figure 5.4. The initial drop from 0% to 25%

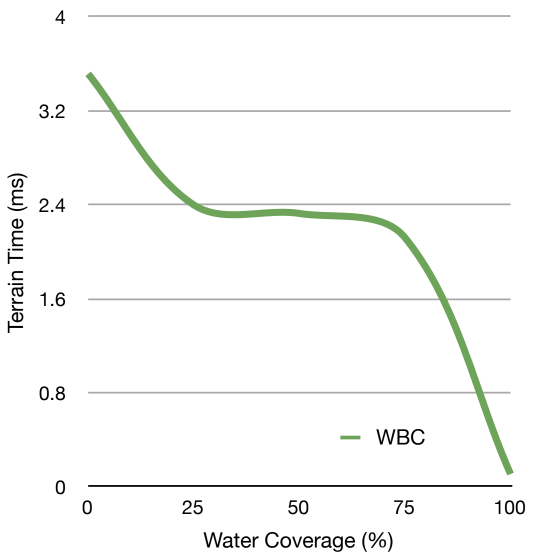


Figure 5.4. Rescaled chart from Figure 5.3(a) that only includes the WBC test.

is due to the introduction of the water surface which causes some terrain LOD blocks to be culled. From 25% to 75%, terrain performance stabilizes; while more terrain LOD blocks are culled, other terrain LOD blocks are set to full resolution as they are under partial water blocks. As the water coverage approaches 100%, little to none of the terrain surface is visible so all the terrain LOD blocks are eventually culled. Recall that the water mesh is always at full resolution, so as water coverage increases more terrain LOD blocks are culled but full-resolution water blocks take their place. While overall performance naturally decreases as water coverage rises, the effects of WBC allow rendering to never drop below 33 FPS for this particular dataset.

5.3 Using a Modest GPU

One requirement of the frontend is the ability to run on machines with more modest GPUs. To evaluate this, the complete frontend renderer (test 6) was run with two of the datasets on a laptop machine with a 2x2.53GHz Core 2 Duo and a 256MB NVIDIA GeForce 9400M GPU. Table 5.4 lists the results.

View-frustum culling is used for the first time with this test. When even a small portion

Dataset/Dim	FPS	Overall Time (ms)	Terrain Time (ms)	Water Time (ms)	Triangles
Malpasset 1100x440	**Without Frustum Culling**				
	101	9.9	4.8	2.7	380,597
	With Frustum Culling				
	250	4.0	1.6	1.1	124,413
Sardis 1026x1026	**Without Frustum Culling**				
	118	8.5	4.0	1.4	396,457
	With Frustum Culling				
	327	3.1	0.8	0.3	57,549

Table 5.4. The complete frontend system running on a more modest GPU.

of the dataset is out of the camera's view, view-frustum culling provides a great advantage. This is evidenced by a 2.5x and 2.8x performance increase from the datasets. Traversing through the scene resulted in roughly two-thirds of the domain being culled, causing a 67% and 85% reduction in triangles for Malpasset and Sardis. Overall, these datasets perform comfortably with interactive frame rates.

Chapter 6

Conclusions and Future Work

The frontend was developed to provide flood emergency managers with two primary functions: to control an interactive 3D visualization of real-world simulations and to create terrain edits that affect these simulations. This software must be able to run on modest GPUs to maintain interactive frame rates. To accomplish this goal, many optimizations are needed. A level-of-detail algorithm allows a terrain mesh to use far fewer triangles during rendering while still appearing sufficiently accurate. A water mesh cannot utilize LOD, but Wet Block Classification provides a way to cull unseen water and terrain areas.

The frontend has achieved the goal of allowing a user to visualize simulations of a wide array of datasets in real-time. The frontend has also successfully provided the ability to make modifications to the terrain surface which consequently updates the simulation in real-time. Additionally, performance results of the frontend show the ability to maintain frame rates well above what is considered interactive, even running on a laptop machine with a modest, integrated GPU.

The FEV backend may look to expand its computation to a cluster environment that would (theoretically) allow fast computation on even larger datasets. Out-of-core simulations are currently supported for storing timesteps but these future datasets may reach sizes such that a single height map cannot fit in memory. With this and other enhancements in mind, there are some opportunities for future modifications to the current frontend design. First, while the water mesh's performance scales well, it does not use LOD which can be a great benefit. A possible optimization to the water mesh is to implement the *Projected Grid* algorithm (Johanson & Lejdfors, 2004), which creates a per-frame LOD grid of vertices that only includes areas of a height map currently in view. Second, the terrain mesh could be adapted to another LOD algorithm, the *Geometry Clipmap* concept (Losasso & Hoppe, 2004). This algorithm uses a set of nested regular grids centered at the camera at varying resolutions. Compression of the height map is supported which may provide the solution to visualizing datasets that are normally deemed out-of-core.

The FEV frontend is a first step in accomplishing the goal of visualizing and manipulating flood simulations of any size. Because the computation is kept separate from the visualization, the frontend is able to run on a wide selection of machines, as long as a connection to a remote (or local) backend is available. As computation cost continues to decrease, the backend can continue to take a heavy load by moving further towards a computing environment in the cloud. Additionally, lightweight devices are including increasingly powerful integrated GPUs, thus future versions of the frontend may be running on tablets or even smartphones, allowing flood emergency managers to visualize and create terrain edits virtually anywhere.

Bibliography

Bibliography

(2011) "JOGL - Java Binding for the OpenGL API.".

Akenine-Möller, T.; Haines, E.; & Hoffman, N. (2002) *Real-Time Rendering.* AK, 3 edition.

Bresenham, J. (1965) "Algorithm For Computer Control of a Digital Plotter." *IBM Systems journal*, Vol. 4(1), pp. 25–30.

De Boer, W. (2000) "Fast Terrain Rendering Using Geometrical Mipmapping." *URL: http://www.flipcode.com/archives/article_geomipmaps.pdf*, Vol. .

Duchaineau, M.; Wolinsky, M.; Sigeti, D.; Miller, M.; Aldrich, C.; & Mineev-Weinstein, M. (1997) "ROAMing terrain: Real-time Optimally Adapting Meshes." In *Proceedings of the 8th Conference on Visualization'97*, pages 81–88. IEEE Computer Society Press.

Johanson, C. & Lejdfors, C. (2004) "Real-time Water Rendering." *Lund University*, Vol. .

Kryachko, Y. (2005) "Using Vertex Texture Displacement for Realistic Water Rendering." *GPU Gems*, Vol. 2, pp. 283–294.

Lindstrom, P.; Koller, D.; Ribarsky, W.; Hodges, L.; Faust, N.; & Turner, G. (1996) "Real-time, Continuous Level of Detail Rendering of Height Fields." In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 109–118. ACM.

Losasso, F. & Hoppe, H. (2004) "Geometry Clipmaps: Terrain Rendering Using Nested Regular Grids." In *ACM Transactions on Graphics (TOG)*, volume 23, pages 769–776. ACM.

Pajarola, R. & Gobbetti, E. (2007) "Survey of semi-regular multiresolution models for interactive terrain rendering." *The Visual Computer*, Vol. 23(8), pp. 583–605.

Phong, B. (1975) "Illumination for Computer Generated Pictures." *Communications of the ACM*, Vol. 18(6), pp. 317.

Röttger, S.; Heidrich, W.; Slusallek, P.; & Seidel, H. (1998) "Real-Time Generation of Continuous Levels of Detail for Height Fields." *Proc. WSCG'98*, Vol. pages 315–322.

Samet, H. (1984) "The Quadtree and Related Hierarchical Data Structures." *ACM Computing Surveys (CSUR)*, Vol. 16(2), pp. 187–260.

Schlick, C. (1994) "An Inexpensive BRDF Model for Physically-based Rendering." In *Computer graphics forum*, volume 13, pages 233–246.

Shankel, J. (2002) "Fast Heightfield Normal Calculation." *Game Programming Gems*, Vol. 3.

Vita

Name:

Brent Sharrow

E-mail:

brent.sharrow@gmail.com

Education:

Bachelor of Science, Computer Science

University of Mississippi (Aug 2005 - May 2009)

Work:

Research Scientist, Dept. of Computer Science

University of Mississippi (Jan 2009 - Aug 2011)