4.2

Simplified Terrain Using Interlocking Tiles

Greg *Snook*

gregsnook@home.com

With recent advancements in 3D rendering hardware, it seems that everyone is bringing his or her game to the great outdoors. Far horizons and mountainous landscapes, once hidden by fog and far clipping planes, are now an obtainable reality. Game programmers once consumed with the BSP tree and span-based rendering methods are now trading in tired old buzzwords for shiny new acronyms like ROAM and VDPM.

ROAM *(Real-time Optimally Adapting Meshes)* [Duchaineau] and VDPM *(View Dependent Progressive Meshes)* [Hoppe98] are outlined elsewhere in great detail (see the References), so I'll just give them a quick overview here. Both of these methods do an amicable job of reducing the polygon count (and therefore the rendering load) on those parts of the terrain that do not require a great deal of geometry, such as reasonably flat plains or areas far off in the distance. In turn, they allow more detailed terrain to exist closer to the camera, or on very rough surfaces, where additional polygons are needed. Essentially, they are two rather complex ways to achieve the same simple goal: more polygons where you need them, less where you don't.

The trouble for some applications is that methods such as ROAM and VDPM tend to rely on procedurally generated geometry to achieve a smooth transition from low- to high-detail areas. ROAM uses a binary tree of triangle intersections to construct the actual terrain geometry from a given height field. VDPM achieves a similar effect by using a coarse mesh to represent the low-detail terrain and applying a set of successive vertex splits to further divide the terrain polygons where necessary. In most cases, these continuous triangulations disrupt the speed advantage of hardware transform and lighting, which relies on static geometry for optimum speed.

The main reason for this is that diese methods work almost too well. They have the power to analyze die terrain down to the poly level, hand-picking tliose that stay and those that get collapsed. This can result in many minute changes to the terrain geometry over time, and requires reprocessing of the entire method should the terrain change. By sacrificing that finite level of control over the geometry, we can remain hardware friendly by working over larger areas of static geometry and remain flexible to changes in the terrain over time.

What this gem proposes is a far simpler method that most applications can take advantage of with a minimal amount of coding. It is not intended to wrestle with the visual quality that ROAM or VDPM methods can produce; instead, it serves to create a simple terrain with the benefits of dynamically adapting detail levels and animation flexibility. It does this while maintaining a data system that is perfectly suited for hardware transform and lighting.

Tiles Revisited

Many moons ago, game programmers used 2D tiles to represent the playfield. This was done for one simple reason: less artwork was easier to create and manage. Early games simply did not have the memory to afford a large amount of pixel data, so smaller pictures were tiled to create the illusion of a larger area. These small tiles were also easier to draw and push around, so smooth-scrolling 2D games could easily be created out of little 32x32 pixel tiles.

The terrain method presented here works on the same basic principle by dividing the terrain into smaller, reusable tiles. The same advantages apply: smaller bits of data are easy to push around, drawing can be optimized, and memory is used more efficiently. The obvious difference is that we are no longer dealing with pixel data within the tiles. The terrain tiles are represented as index buffers that link together the vertices of the terrain.

Think of the 3D tiles as a grid being projected down on the landscape from above. Each grid square represents a single tile in the terrain system. On the surface, it may not appear as if the terrain tiles ever repeat, given that terrain is a pretty random set of geometry. The truth is the terrain may never repeat *on the surface,* but behind the scenes, there is ample data to tile and reuse.

Consider each terrain tile in the form of a vertex and index buffer. While each tile may contain a unique set of vertex data, the index buffers used to draw the tiles can be made to repeat rather frequently. In fact, by careful planning of the vertex data, we can create a finite set of index buffer "tiles" to use throughout the entire terrain.

We do this by taking advantage of a few refinements in the geometry of our tiles. First, each tile must contain an identical number of vertices, sharing the edge vertices with its neighbors. These vertices represent the highest level of detail possible for the tile. Second, the vertices of the tile are arranged in a regular grid on the *x-y* plane, using *z* to represent the vertices' height above sea level. Last, we store the vertices of each tile in an identical order so our index buffers can be used on any tile. Have a look at the sample tile shown in Figure 4.2.1. Here we have a 17x17 vertex tile showing the grid-aligned positioning of each vertex, each of which has a unique height value sampled from the terrain bitmap.

The reason for this vertex organization is simple. Since the vertex data always appears in a regular grid and in an identical order, a fixed set of index buffers can be created for the entire terrain. Using the proper index buffer, a given tile can be rendered at any level, ranging from full detail down to a simple pair of triangles. Index

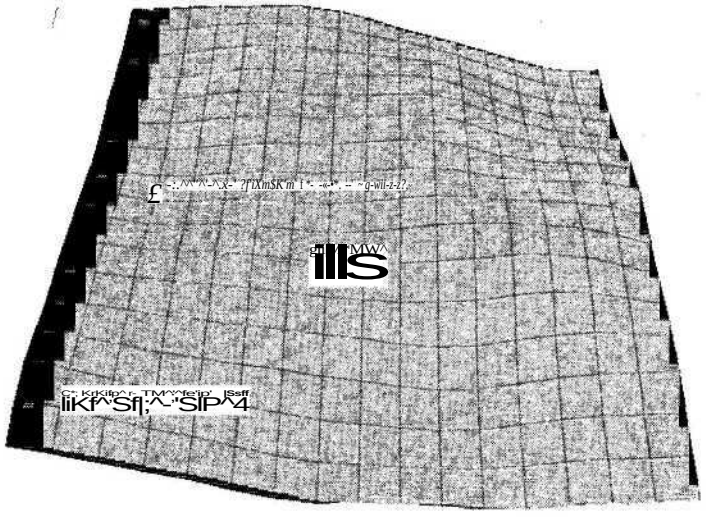


FIGURE **4.2.1** *A sample terrain tile of 17x17 vertices.*

buffers that use more of the available vertices create a higher detailed representation of the tile. Similarly, index buffers using less vertices render tiles with reduced triangle counts. Figure 4.2.2 illustrates this by showing a sample tile rendered at different detail levels.

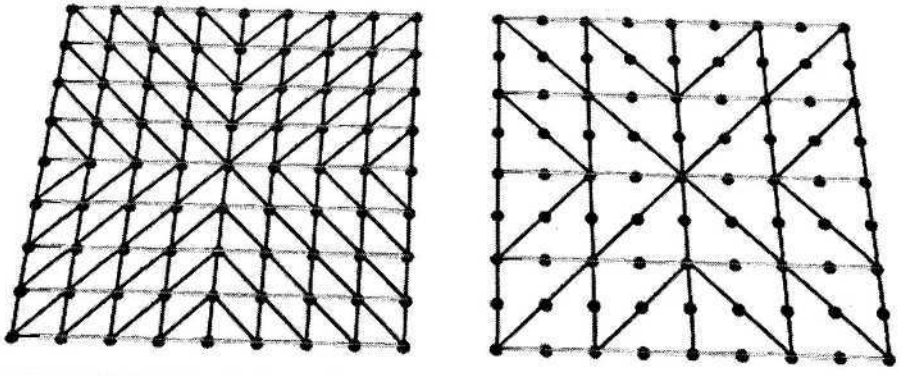


FIGURE **4.2.2** *Using index buffers to create two separate detail levels from the same set of vertices.*

Map Making

In order to create the landscape tiles, we need a set of source data from which to pull. A common method is to read elevation data from a height map. This map is simply a grayscale bitmap of the terrain, where the luminance of the pixel is used to represent the elevation at a given position. The height map has the added advantage that it is

already arranged in a regular grid, so it can be easily translated into terrain vertex data. It can also serve as an animation resource, adjusting the pixel values to afTect the terrain height at different locations.

Creating the tile vertices is simple. Since each tile vertex has a known 2D position on the *x-y* grid, all that remains is to sample the corresponding height pixel from the terrain bitmap and translate it to a *z* value for the terrain vertex. For each terrain tile, a corresponding block of pixels in the height map can be sampled to create a unique vertex buffer for the tile. In the case of an animating height map, this process must be repeated periodically to update the terrain vertices.

Tile Templates

The index buffer can be thought of as a drawing template cast over the tile vertices. As we saw in Figure 4.2.2, the index buffer defines how we pull triangles out of the tile, controlling how detailed a version of it we draw. Following our key rules, each tile's vertex buffer has been laid out in an identical order so the index buffers can be used interchangeably. For an example 9x9 vertex tile, we can create a global set of index buffers to draw all possible detail levels for any 9x9 set of vertices, skipping vertices in the grid to create the level geometry. The top-level index buffer uses all 81 vertices to draw 128 triangles, while the lowest level uses only the four corner vertices to draw a two-triangle quad. In addition, there are two additional detail levels representing 32 and 8 triangles, respectively.

The next requirement is a method to determine which of our four detail levels each tile needs to use when being drawn. This determination can range from a simple function of the distance between the tile and the camera, to a full heuristic taking into account die viewing angle and perceived roughness of the tile. The best method to use depends on die game terrain and camera movement. Hugues Hoppe's paper on the VDPM mediod [Hoppe] sheds more light on heuristic ideas that can be used to select detail levels for each terrain location. The sample application on the companion CD-ROM, *SimpleTerrain,* uses the distance from the tile to die camera for simplicity. Once the detail level is known, drawing is a simple matter of sending die tile's vertex buffer along widi die desired index buffer into your chosen rendering API for drawing.

Ugly,.Ugl*_Ugly

We have the basic terrain system in place, but it is by no means a smooth terrain. What we have now is a terrain that changes abruptly as tiles of different detail levels are drawn side by side. In addition to that, seams can appear in the gaps created by two tiles of different detail levels. In short, we have made a mess, but there is still hope.

The key to this method is having tiles that interlock. That is, creating tiles that mesh together perfectly, regardless of the differences in detail levels between neighboring tiles. To do this, a different set of index buffers is required to merge tiles of different levels together without gaps and seams. These index buffers can be broken into

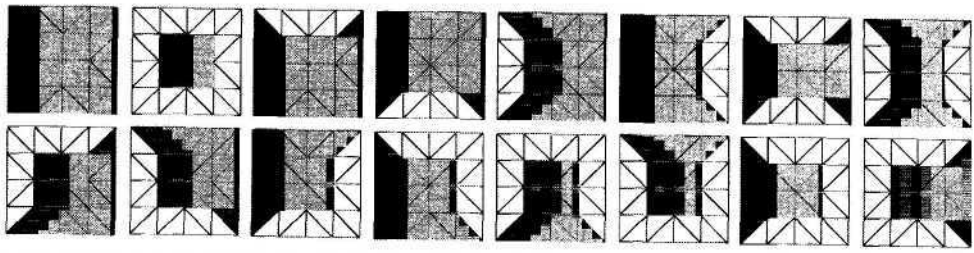


FIGURE **4.2.3** *The 16 basic tile bodies. Unshaded areas show where linking pieces must be placed.*

two groups: bodies and links. Bodies represent a major portion of a tile at a given detail level, with areas removed to provide space for linking pieces. Links, as the name suggests, link the bodies of different detail levels together seamlessly.

Figure 4.2.3 shows the 16 possible body types for a tile of any given detail level. To keep things under control, we specify that tiles only link downward, meaning that tiles at higher detail levels must use link pieces to fit together with lower-detail neighbors. Looking at Figure 4.2.3, the unshaded areas of each body type then represent spaces where links are required to connect to a neighbor tile at a lower detail level.

Linking pieces are smaller index buffers that fit into the spaces left vacant by the body tiles. These index buffers arrange triangles to step down from a tile using a higher number of vertices to an adjacent one using less. Figure 4.2.4 shows an example link tile used to connect two body tiles. Since we only link downward in detail levels, each detail level needs enough link pieces to connect to the details levels below it. For the example 9x9 vertex tile, we would need three linking pieces for each side of our highest-detail level, since it must be able to link to three lower-detail levels. Our lowest-detail level, the simple quad tile, needs no linking pieces, since all higher-detail levels must do the work to link down to it.

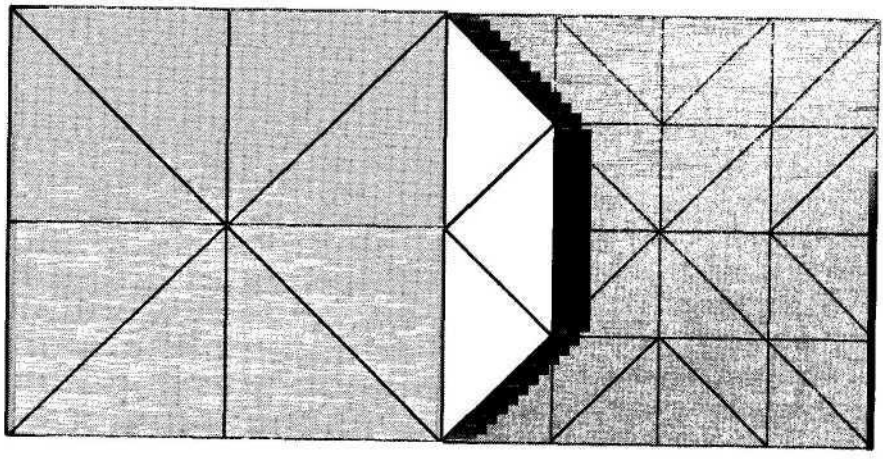


FIGURE **4.2.4** *An example link piece used to join two tiles of different detail levels.*

Table 4.2.1 All Index Buffers Required for Our Sample of Four Detail Levels

Detail Level	Body Pieces Required	+	Linking Pieces Required	=	Total
4	16		3 for each side		28
3	16		2 for each side		24
2	15		1 per side		19
1	1		0		1
				Grand Total:	72

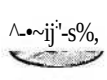
Given our example of a 9x9 vertex tile with four detail levels, we can calculate that the total number of index buffers required amounts to a grand total of 48 "body" pieces and 24 "linking" pieces. Table 4.2.1 shows the full table of index buffers required to smooth out our terrain. As can be seen, increasing the number of detail levels increases the index buffer count, but since these are relatively small in size and can be used throughout the entire terrain, they still remain rather efficient.

Better, Faster, Stronger

Using the new body and linking pieces means we need to change our rendering method. For each tile, we now need to examine the tile's neighbors. We choose a body tile that contains a notched side for each neighbor that is at a lower detail level than the current tile. Then, we select the necessary linking pieces that fill the notches and connect us with the adjacent tiles. Each of these index buffers is then sent to the rendering API along with the tile's vertex buffer for drawing. In the worst case, we send five index buffers per tile (one body, four linking), but in the best case, we still send only one (the full body tile).

Organizing the index buffers into triangle strips and fans can further optimize the method. For larger tiles (33x33 vertices and up), this will greatly reduce rendering time. In addition, the order of the vertices in the tile can be adjusted for better cache performance when rendering. The exact order will depend on which index buffers the tile will be rendered with most often.

Conclusion



ONJHCCP

Figure 4.2.5 shows the final output of the rendering method. The sample program *SimpleTerrain* demonstrates the method using DirectX 8.0. Full source code for the sample program is available on the companion CD-ROM. In this image, the wireframe of the terrain is exposed to show the various body tiles and linking tiles in use. Ground textures have been removed tor readability.

The intent of this gem was to provide an alternative to the popular procedural methods for rendering dynamic terrain while fully enabling hardware transform and lighting. By using this method, a dynamic terrain system can be up and running quickly without severely impacting the application's frame rate. While the final terrain may not rival that of a well-written ROAM or VDPM system, it does provide the

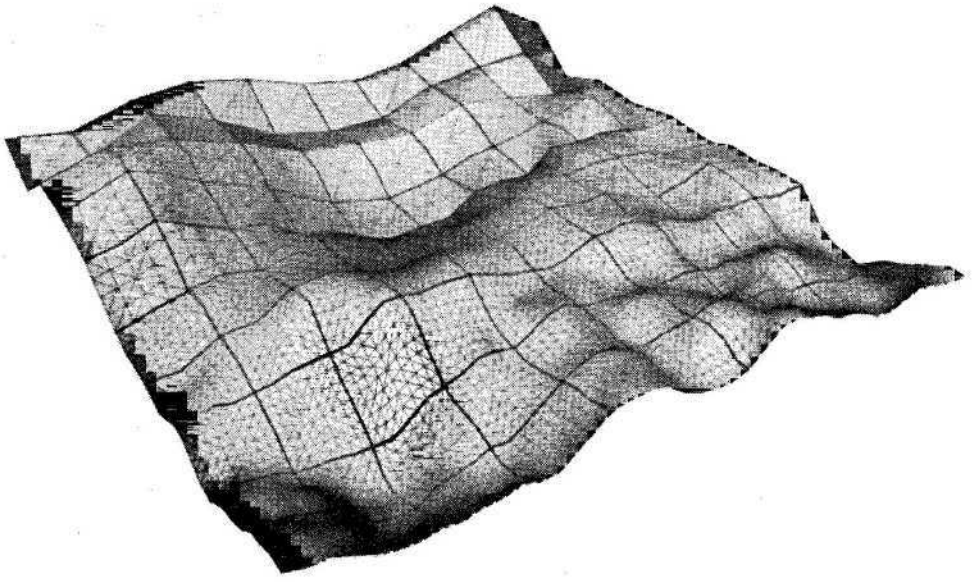


FIGURE 4.2.5 *Sample output of the* SimpleTerrain^rograzwz *showing tiles and linking pieces in use.*

same basic advantages of those methods with the potential of greater rendering speed in hardware.

References

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

[Hoppe98] Hoppe, H. "Smooth View-Dependent Level-of-Detail Control and Its Application to Terrain Rendering" *IEEE Visualization 1998,* October 1998, pp. 35-42. (www.research.microsoft.com/'-hoppe).