Hindawi Publishing Corporation
International Journal of Computer Games Technology
Volume 2008, Article ID 753584, 7 pages
doi:10.1155/2008/753584

Review Article

Real-Time Optimally Adapting Meshes: Terrain Visualization in Games

Matthew White

Department of Computing and Mathematics, Manchester Metropolitan University, All Saints, Manchester M15 6BH, UK

Correspondence should be addressed to Matthew White, mattwhite06@googlemail.com

Received 27 September 2007; Accepted 21 December 2007

Recommended by Kok Wai Wong

One of the main challenges encountered by interactive graphics programmers involves presenting high-quality scenes while retaining real-time frame rates on the hardware. To achieve this, level-of-detail techniques can be employed to provide a form of control over scene quality versus performance. Several algorithms exist that allow such control, including the real-time optimally adapting mesh (ROAM) algorithm specifically aimed at terrain systems. Although ROAM provides an excellent approach to terrain visualization, it contains elements that can be difficult to implement within a game system. This paper hopes to discuss these factors and provide a more game-orientated implementation of the algorithm.

1. INTRODUCTION

Efficiently rendering meshes within a virtual environment requires the use of a level-of-detail (LOD) algorithm. This helps ensure that the number of primitives (triangles) used to represent the mesh is kept as close to an "optimal" level as possible. As graphics developers, we measure this level as a compromise between both scene detail (triangle count) and frame rate. The optimal level is then defined as the highest number of triangles we can render, while retaining an acceptable frame rate for our application.

Traditional level-of-detail methods begin by defining several versions of the scene's meshes, each differing in triangle count. As the application renders the scene, a version of each mesh is chosen in relation to factors, such as the meshes' onscreen size and overall scene importance. As meshes become closer or further from the viewer, their onscreen size changes and thus the number of triangles required to render them effectively. The result is a form of control over the scene triangle count and thus a more optimal detail level of the scene.

However, when applied to "massive" meshes, such as terrains, this technique breaks down. By massive, we mean a mesh whose size is so large that it is common for it to contain both very close and very distant sections from the viewer at one time. Simply put, we cannot just pick a distance from this vast range, apply a single detail level across the entire landscape, and expect reasonable results. Instead we need to implement a more specialized LOD algorithm that takes this range of distances into account. One of the first of these methods was introduced by Lindstrom in his continuous level-of-detail (CLOD) paper [1], which was then expanded upon by Duchaineau to produce the original ROAM algorithm [2].

ROAM works by defining a mesh as a hierarchal bintree structure of renderable triangles, dubbed by Duchaineau as a *binary triangle tree*. In this tree, each node represents a triangle that is a lower detail version of its two children nodes. Leaf nodes represent the highest LOD's triangles, while the root node represents the lowest. The rendering procedure then becomes a recursive task where we transverse the tree and decide which nodes to render for the current frame. When testing each node, we can choose to either tag the relevant triangle to be rendered this frame, or step a level deeper into the tree, and perform the same test upon the child nodes. Because each node represents 3 vertices (a triangle), a 3D location in the virtual world can be defined for the node and thus a distance from the viewer can be found. With this distance, we can perform the same distance test as the more traditional LOD algorithms, except that this test is now performed at a per-triangle level instead of the entire mesh. The result is that we can spread the LOD across the entire visible terrain and thus solve the problem of the "massive mesh."

Although ROAM produces a range of detail levels for a terrain that can be tweaked to a specific triangle level, the algorithm itself does not translate to graphics hardware that well [3]. Because the graphics processor can only process data in its local graphics memory, any change to the renderable dataset requires an upload to this graphics memory. This upload can be considered expensive, and overuse of it can result in a problem known as "thrashing," causing the graphics processing unit (GPU) to stall as it waits for graphics memory to be written to. For high performance graphics, we prefer to load the required data onto the graphics card at initialisation time, and then attempt to minimise any further uploads during the runtime of the application. Duchaineau's ROAM relies heavily on changes to the mesh vertices, which are built to describe the current tessellation of the mesh. Uploading this buffer to the graphics memory can cause the mentioned thrashing effect and thus a performance hit, something that has caused criticism from games developers and led to simplified variations of the algorithm appearing in several games [4, 5].

Many of these variations have one thing in common. Instead of checking every triangle of the mesh for a correct LOD, the mesh is split into a collection, usually a grid, of terrain tiles. Each tile then contains several sets of geometry, each representing a different LOD for the tile, much like the more traditional LOD algorithms. Because each tile has a finite number of detail levels, they can all be uploaded to the graphics memory at initialisation time, minimising the thrashing effect. Therefore, better performance can be obtained by these ROAM variations, which can make them more desirable for games applications.

This performance increase has its cost however. By replacing the per-triangle LOD test with per-tile tests, we lose the tessellation accuracy of the algorithm. No longer can we increase or decrease the triangle count by a single triangle, and thus lose the near-perfect optimal detail level provided by the original ROAM method. Also, the effect known as "popping" can become much more apparent in these variations. Popping is the graphical artifact created when a visible part of the terrain changes its detail level. The geometry literally changes in front of the users eyes, and can become very distracting if large areas of the landscape suddenly switch. This effect is unavoidable, but can be reduced if the changing sections of the terrain are relatively small on screen. Since ROAM tessellates on a per-triangle basis, this area is usually sufficiently small for combating popping, but when entire terrain tiles change LOD, the effect can be much more noticeable.

Not all game systems that use ROAM implement this style of approach however. Treadmarks [6], an action game by Longbow Digital Arts, is probably the most well known of games that implement ROAM-based terrain. Instead of using the simpler versions of the algorithm, like those mentioned in Snook's and Ulrich's papers, Treadmarks uses a split-only approach, along with a technique called *Implicit Binary Trees* to increase performance [7]. Split-only means that the terrain's detail level is recalculated for each frame, without the *frame coherence* feature mentioned in Duchaineau's original paper. Although this requires more per-frame processing time, it greatly simplifies the algorithm making it much easier and quicker to implement into a game system.

The remainder of this paper will describe a new variation of ROAM that combines the ideas discussed in these previous variations into a new system, aimed mainly towards games and real-time graphical applications.

2. OVERVIEW

Originally presented previously at the Manchester CyberGames Conference [8], the "GEOmancy" terrain engine uses a version of the ROAM algorithm that overcomes the problems discussed. The system works by dividing the terrain geometry into a collection of tiles, each represented by a pair of ROAM triangle bintrees. The classic ROAM split-merge algorithm is then applied to each tile individually to produce an optimal detail level. To retain speed through hardware optimisations, the vertex buffer for each tile remains static and is uploaded to graphics memory at the application's start. The detail level of each tile is then described, instead, via an index buffer, which is created through transversing the tile's bintrees. Because the per-frame change in viewpoint position is usually a small fraction of the terrain size, the amount of LOD changes is also very small, resulting in very few updates to the separate tiles' index buffers. This allows the accuracy of the original ROAM algorithm to be maintained, while minimising the amount of data that must be posted to the graphics device per frame.

Although the algorithm tries to provide both high accuracy and high performance, it is liable to two major limitations. First, the algorithm only works on grid-based terrain geometry. That is, vertices that are spaced along the x-z plane at regular intervals with only their height values differing. This is not too much of an issue for games as this is by far the most popular terrain representation method, allowing the dataset to be compressed to a map of height values (a heightmap) and a single float that defines the distance between vertices. Secondary, due to the use of static vertices, only heightmaps of specific sizes can be used.This limitation can be overcome by using the next largest viable size and "voiding" off the unwanted extra vertices with water or walls, and so forth. The geometry may still be there, but techniques can be used to ensure that the player never sees it.

3. IMPLEMENTATION

3.1. *Tiled geometry*

The GEOmancy algorithm begins by converting a heightmap into a grid of terrain tiles. For each tile, a vertex array is created by sampling the relevant heightmap entries and scaling these values to produce terrain heights and thus vertices. These vertex arrays can then be placed in the graphics memory ready for future render calls.

For each tile, we need to create two bintrees, each represented by an index buffer. When we tessellate our bintree, this index buffer will contain a description of which triangles to render to provide the current LOD of the tile. As stated previously, each node of a triangle bintree represents a renderable triangle. Because we are using an index array to reference which vertices to render, a triangle can be represented using three integers that can be used to index the appropriate vertex array. As well as this, we also need to store an error metric for the triangle, similar to Duchaineau's ROAM, so that we can perform LOD tests at runtime for each node in the tree. Since we cannot know the distance to the viewpoint at initialisation time, we need to store a value that can assist us during the runtime LOD decisions. For this, a technique from the Treadmarks engine is used called variance.

Since every non-highest detail level triangle is an approximation of its children, a difference for it can be calculated by finding the distances between it and the actual height of the geometry that it covers. When we run our LOD tests, we can say that triangles with a high variance are bad representations of the geometry they cover, and should receive a higher "split" priority than those with lower variances. When our algorithm is deciding where to add triangles to the frame, the variance measure helps ensure that rougher sections of the terrain will receive more detail than the flatter parts, which is exactly what we require.

As stated previously, our terrain tiles must be of a specific size. This is because an existing vertex at the correct point is required to split a triangle in two. Because of this, only specific sizes will allow us to split triangles down to the lowest level possible. As can be seen in Figure 1, there is a limited number of tile dimensions that allow this situation.

For each increase in usable detail levels for a tile, we are required to double the number of triangles along their edges. The size of the tile, in vertices, required for this can, therefore, be defined as $[(2^n) + 1]$, where n is the depth of the tile bintrees. For the demo, we used a dimension of 9×9 vertices per tile, as it provided a good balance between bintree depth and number of tiles.

3.2. *Implicit bintrees*

Now that we have divided our terrain into tiles, we need to create our version of the ROAM triangle bintrees. As mentioned previously, we will be using an updatable index buffer to describe which triangles to render from our vertex array. To help boost performance, a technique, again from the Treadmarks engine, called Implicit Bintrees, will be used. Because our trees will never add or remove nodes after the initialisation phase, we can represent our bintrees through a fixed-sized array, providing an abstract interface that accesses it like a bintree. The result is that all memory allocations are done at initialisation, improving the performance of the runtime part of the algorithm. An excellent explanation of this process was presented by Bryan Turner on the Gamasutra website [9].

The first index of our array stores the root node of the tree. Transversing the tree can be quickly achieved via bit-shift operations as follows.

Left-Child Index: $\text{curIndex} \ll 1$.
Right-Child Index: $(\text{curIndex} \ll 1) + 1$.
Parent Node: $\text{curIndex} \gg 2$.

These macros enable a parent or child index to be found from any other array index, through the use of very fast operations, as well as removing the need for each node to store pointers to its neighbours.

Perhaps the biggest advantage of implicit bintrees (other than removing the need for dynamic memory allocation), is that any triangle in the tree can now be described using a single integer index. As will be covered later, this fact is particularly useful for implementing the ROAM split and merge queues, as well as solving the CLOD problem known as cracks.

In our algorithm, each tile contains two of these implicit bintrees, one for the "top-left triangle" and one for the "bottom-right one." For each bintree node, we store three indices that describe the triangle vertices, along with a variance value for the triangle. We define our root node as a triangle with vertices at the relevant corners of the tile. Every child can then be defined by dividing the parent triangle down its centre, creating the two half-sized child triangles. This process can be repeated recursively through the tree to create all potential triangles for each tile.

3.3. *Error metrics*

To complete our bintrees, we need to find the variance value for each node of the tree. This is a recursive process that starts at the leaf nodes and works up to the root node. Because each leaf node represents our highest LOD, their variance value is 0. For each node above these leaves, we sample the height value from the heightmap where their hypotenuse's midpoint aligns to. We also find the average of the two hypotenuse's vertex heights to find the approximate rendered height at this point of the triangle. Variance is then the maximum of either the difference of these values, or of the two children's variance values. This max operation helps prevent a situation where a low detail triangle midpoint happens to fall at the same point as, or near to, the original height data. Whereas the variance for this would be near zero, the actual triangle itself could still be a bad approximation for the other points of the terrain that it covers.

At run-time, we can perform an error test per node based upon the relative variance value. To make the algorithm view-dependant, we take factors concerning the virtual camera into account when making this test. As mentioned, these factors are usually in relation to the triangles' onscreen size, and thus the viewpoint distance. A typical test divides the variance value with this distance and checks the result against a threshold. This ensures that closer, rougher terrain is split with more scrutiny than distant, flatter parts. If this test fails, then we can "split" the triangle by stepping down one level of the tree and repeating the test on the two child nodes. Once we find a node that passes the test, we can add the three indices stored for the triangle to the terrain tile's main index array. When all the tests have been completed, the tile's index array will describe an optimal tessellation of the mesh for that frame, and can be used to reference the vertex array when rendering.

Although this works and allows per-triangle tessellations on a frame-by-frame basis, it is not entirely performance

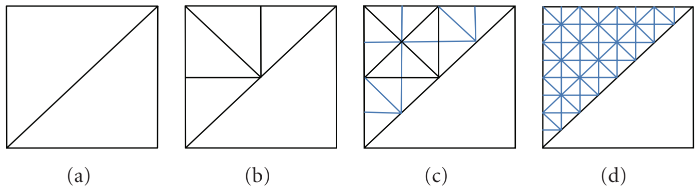


Figure 1: Static tile-size restrictions.

friendly. For each frame, we must recursively test every tile's bintrees from their root node to find the optimal detail level. Since our variance values do not change after initialisation, the only varying factor for our tests is the viewpoint itself. In most applications, it is not usual for the camera to move far between frames, so the results of the majority of tests will be identical to the previous frame's results. Therefore, instead of transversing the bintrees from root node down, we can "pick up" from where we left off last frame, testing each bintree from its previous optimal state. This optimisation is known as *frame coherence* and is a very effective part of the original ROAM algorithm.

3.4. Split-merge queues

In ROAM, frame coherence is achieved by using two queues called the split and merge queues. The split queue is used to store the next nodes that can be "split," thus increasing the bintree's effective LOD, whereas the merge queue stores nodes that can be "merged" to decrease the LOD. Splitting a triangle is the process of converting it into its two child triangles, and therefore incrementing the mesh's triangle count, whereas merging is the reverse process of converging two triangles into their parent.

Implementing these data structures is relatively straightforward. Because our system is using static vertex arrays, the number of potential triangles is also constant, and thus the maximum number of triangles that could be on either queue. We can therefore implement each queue as a fixed length array of this size, with each array containing the indices to relative nodes within the implicit bintree. We can then use markers to store the effective starts and ends of the active parts of these queues, and never have to reallocate memory during run-time.

These queues represent the detail level state for a single bintree. We create two operations that allow the increase and decrease of this detail level called Split and Merge, respectively. The following is the pseudocode for a typical implementation for these operations.

Split operation

```
Pop top node index from the Split Queue.
Push this index to the Merge Queue.
Find node's child indices using the implicit bintree bit-
shift macros.
Add child indices to the end of the Split Queue, in or-
der of Variance values.
```

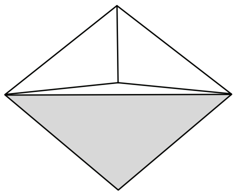


Figure 2: Cracks between triangles.

Merge operation

```
Pop top node index from the Merge Queue.
Push this index to the Split Queue.
Find node's child indices using macros.
Remove these child node indices from the Split Queue.
```

By restricting access to the queues to these 2 operations, we can ensure that the queues are always ordered by the node's variance values. This helps ensure that higher variance areas of a bintree are split and merged before the lower variance parts, which is exactly what we want.

Because the split queue of a bintree contains a list of all visible triangles, an index buffer for the tile mesh can be built up by iterating through it and referencing the appropriate structures directly. Furthermore, we can tag which bintrees have had their split or merge methods accessed for each frame, and only upload the new index buffer for them. Because the number of per-frame tessellations is usually a small percentage of the visible terrain tiles, this results in a great reduction in the amount of data being transferred to the graphics memory.

3.5. Avoiding cracks

3.5.1. Overview

One problem that all level-of-detail algorithms have to deal with is that of cracks appearing between different LODs. In ROAM, this occurs whenever a triangle is split. As can be seen from Figure 2, the extra vertex at the children's heads is at a different height than the second triangle, thus resulting in a gap. To solve this, the triangle at the base of the splitting triangle must also be forced to split.

There are three possible arrangements of triangles when splitting: base-to-nothing, base-to-base, and base-to-edge.

Base-to-nothing occurs when our split target triangle's hypotenuse (the base) is at the edge of the mesh, and thus no geometry. In this situation, we simply do nothing and split the triangle as normal.

Base-to-base is when the triangle shares its hypotenuse with its neighbour. In this situation, we simply force the neighbour triangle to split before the target triangle splits. The result is that the crack is covered up as the neighbour triangle's children reference the same vertex at that point.

Base-to-edge is perhaps the most complex scenario. In this case, our triangle's base-to-base partner is one of the neighbour's triangle's children. We essentially need to split twice, once for the neighbour and once for its appropriate child. The reason why this can become complex is that this initial split can encounter the same base-to-edge scenario as the original split. The result is that forced splitting can be propagated across the mesh, as triangles force other triangles to be split.

However, because a base-to-edge scenario can only occur between a triangle and a triangle of a detail level that is one less, this propagation seldom travels very far, so this rarely becomes an issue in practice.

To implement this forced split, each triangle needs knowledge of its *diamond partner*, which is the triangle in the mesh that shares a base-to-base relationship. With this knowledge, the triangle can inform its partner that it too needs to split. In the original ROAM, a pointer to this partner was stored on a per-triangle basis. However, in a terrain mesh that can contain hundreds of thousands of potential triangles, this extra memory requirement can soon mount up. GEOmancy takes advantage of the implicit bintrees and uses a *neighbour map* to significantly reduce these memory requirements.

3.5.2. *Neighbour map*

As mentioned previously, we can describe any potential triangle in our mesh with a pointer to a bintree and an index that defines which slot of the implicit bintree array to look at. We also know that, apart from the underlying height data, the structure of every bintree in our system is the same. The consequence of this is that every node in a bintree is also surrounded by, *relatively,* the same neighbours as simular nodes in other bintrees. With this similarity in mind, instead of storing a diamond partner pointer for each triangle, we can create a static *map* that when queried can return a description of the required partner.

Because a diamond partner shares its base with its partner triangle, it can only be in either the same bintree or a neighbouring bintree. During initialisation, we store three pointers for each bintree, each of which point to the relative neighbouring bintree. These pointers can even be null in the case that the tree is at the edge of the mesh. Upon querying, the neighbour map returns a partner's array index and also a flag that denotes which bintree neighbour the index refers to (left-edge, right-edge, or hypotenuse-edge neighbour). With this information, a bintree can call the split function through the relevant pointer, passing in the index to produce a forced split. In the case of a "same bintree" flag, the bintree class simply calls its own split method. In the event that a neighbour pointer is null, then the split can be assumed to be a base-to-nothing scenario, and the forced split can be ignored.

The end result is a fast look-up system for finding diamond partners that does not require per-triangle pointer storage. The neighbour map's size remains fixed regardless of the size of the terrain, which can prove very beneficial for systems that require vast landscapes.

Creation of the neighbour map is a recursive task much like the creation of the bintrees. It was found that, with the exception of the root triangle, every triangle's neighbours could be found by examining their parent. Root triangles have no parent node, so their neighbours must be defined upon the bintrees creation, through the use of the neighbour pointer class members mentioned previously. The neighbour map itself is an array of the same size as the system's bintrees. At each slot, we store a flag, indicating the root triangle's bintree neighbour, and another index, describing the specific triangle from this bintree.

Figure 3 shows the graphical representation of the first 3 levels of a bintree. As we can see, the bintree (triangle 0) has the neighbours L, R, and H denoting left-edge, right-edge, and hypotenuse neighbours, respectively.

The left child of this triangle is triangle 1. As can be seen, its neighbours are as follows:

(i) left neighbour: triangle 0's right child;
(ii) right neighbour: triangle 0's base neighbour;
(iii) base neighbour: triangle 0's left neighbour.

The right child of the root (triangle 2) shares a similar relationship:

(i) left neighbour: triangle 0's base neighbour;
(ii) right neighbour: triangle 0's right child;
(iii) base neighbour: triangle 0's right neighbour.

The next level down (triangles 3, 4, 5, and 6) follows the same pattern depending on if they are the left or right children of their parent. Using this information, we can use a recursive method to fill the neighbour array with a flag and index number, describing the relative location of the diamond partner for any node in one of our bintrees.

3.6. *Summary*

At the end of the initialisation, we have converted our heightmap into a grid of terrain tiles. Each tile represents a square of geometry of our terrain, represented via vertex buffers, and also two bintrees. These bintrees represent the current tessellation of the terrain tile, using split-merge ROAM to produce an index buffer that denotes which triangles to render from our geometry. These bintrees offer split and merge methods to increase or decrease the tree's LOD by a single triangle.

During run-time, we test each tile against an error threshold using both its split-queue's top node's variance and the distance to the tile from the camera. These factors insure that the worst approximations and the closest triangles are split at a higher priority.

To maintain an optimal level of performance, we also use frame-by-frame coherence offered by the split and merge queues. Because of this, and the segmentation of the geometry due to the tiled terrain, only a small proportion of the

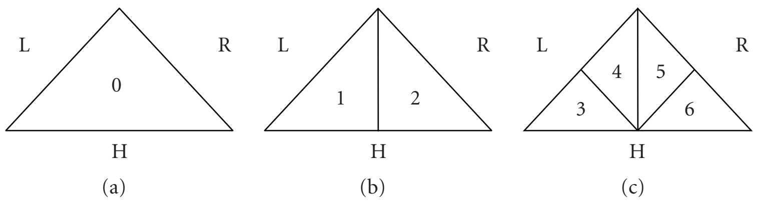


Figure 3: Neighbourhood map structure.

terrain's entire index buffer requires changing each frame, minimising the effect of thrashing.

Finally, any bintree can query the neighbour map for a description of a specific triangle's diamond partner, from which it can force another local bintree to split a triangle, avoiding cracks from appearing within the mesh.

4. RESULTS

To test the final implementation of the system, a sample heightmap was used and frame rates were observed. A heightmap of size 512 × 512 was chosen for these tests, and thus provided just over 522 000 potentially renderable triangles in the mesh. The system used for the tests was a typical desktop system; 2.0 GHz CPU, 512 MB RAM with an ATI Radeon 9600 graphics card. Different error metrics were tested to see the difference between performance and scene quality during the rendering. Table 1 shows the average frame rates achieved for several error metrics.

Table 1: Frame rates for specific error metrics.

Error metric	Average frame rate (per second)	Triangles (per second)
1.0	90.5	47.26 million
3.0	106.7	55.72 million
5.0	112.2	58.59 million
7.0	113.9	59.48 million

Metrics above 7 pixels provided slightly higher frame rates, but also suffered from very noticeable popping. By using a small error metric, this popping effect was restricted to the smaller onscreen triangles, and was not as noticeable. Without some extra feature to deal with these artifacts, however, metrics over 7 are unlikely to be favourable for use within a games application.

For comparison, the most recent version of ROAM (ROAM 2.0) shows a performance between 40 million and 56 million triangles per second [10] depending upon the hardware being used. Depending upon the error metric chosen, our system can produce higher frame rates while maintaining an acceptable level-of-detail. The full source and an executable demo for the GEOmancy system can be found online at http://members.gamedev.net/rootevilgames/mwhite/GEOmancy.htm.

5. FURTHER WORK

At the time of writing, the GEOmancy algorithm provides a new variation of ROAM, aimed for implementation within a games-orientated system. However, there are further improvements being worked on that will be discussed in this section.

Memory can be a tight resource, especially in the development of console games. Storing an entire dataset for a landscape can hog up much of this resource. To get round this, we intend to make as much of the vertex data reusable as possible. The idea revolves around the use of vertex buffer streams. In one stream, we load the vertices' x and z positions. Because these are repeated for each tile, due to the grid nature of the mesh, we can create a single vertex buffer for each tile to reference. A second stream can then be used to reference other vertex data, such as the y position and texture coordinates.

For systems using pixel shader 3.0, an expansion of this technique can be applied using vertex textures. This way, each tile's vertex height can be referenced directly from the heightmap texture, moving part of the processing onto the GPU. Normal maps can also be used in the same fashion to provide fast per-vertex normals for dynamic lighting.

Perhaps one of the most exciting ideas for future development is the incorporation of DirectX 10's new geometry shader. This is a shader stage that allows the generation of new primitives within the rendering pipeline itself. As mentioned in my previous paper, the main reason that there has been no GPU-only implementation of ROAM is the inability to add and remove vertices in this pipe-line. With this new shader, this limitation should no longer apply and the creation of a full GPU ROAM algorithm could soon become a reality.

6. CONCLUSION

ROAM is a popular and very effective algorithm for the visualisation of terrains. However, several problems and performance issues can be encountered when trying to implement it into a performance-heavy application, such as a computer game. This paper has presented an overview of the original algorithm and discussed a possible implementation of a more games-orientated variation. By imposing size restrictions upon the input geometry, memory requirements can be precalculated during the initialisation stages, eliminating the need for dynamic memory allocations at run-time. Finally, a tile-based system has been incorporated, allowing us to treat each terrain tile as a separate mesh. This allows us to separate the terrain mesh's index buffer into more manageable sections, rebuilding only the parts that require it between frames.

ACKNOWLEDGMENTS

The author would like to thank Geoff Brindle for his assistance throughout his dissertation, which led to the development of this paper. He would also like to thank Edmund Prakash for general assistance during his dissertation, as well as allowing him to present his original paper at the Manchester CyberGames Conference 2007.

REFERENCES

[1] P. Lindstrom, D. Koller, W. Ribarsky, L. F. Hodges, N. Faust, and G. Turner, "Real-time, continuous level of detail rendering of height fields," in *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '96)*, pp. 109–118, New Orleans, La, USA, August 1996.

[2] M. Duchaineau, M. Wolinsky, D. Sigeti, M. Miller, C. Alrich, and M. Mineev-Weinstein, "ROAMing terrain: real-time optimally adapting meshes," Tech. Rep. UCRL-JC-127870, Lawrence Livermore National Laboratory, Livermore, Calif, USA, July 1997.

[3] G. Snook, *Real-Time 3D Terrain Engines Using C++ and DirectX 9*, Charles River Media, Hingham, Mass, USA, 2003.

[4] G. Snook, "Simplified terrain using interlocking tiles," in *Games Programming Gems 2*, pp. 377–383, Charles River Media, Hingham, Mass, USA, 2001.

[5] T. Ulrich, "Chunked LOD," http://www.tulrich.com/geekstuff/chunklod.html.

[6] Longbow digital arts, Treadmarks, http://www.ldagames.com/treadmarks.

[7] S. McNally, "Treadmarks Engine (Binary Trees and Terrain Tessellation)," http://www.ldagames.com/.

[8] M. White, "Adapting ROAM for use within a games application," in *Proceedings of the 3rd International Conference on Games Research and Development (CyberGames '07)*, pp. 59–66, Manchester, UK, September 2007.

[9] B. Turner, "Real-Time Dynamic Level of Detail Terrain Rendering with ROAM," http://www.gamasutra.com/features/20000403/turner_01.htm.

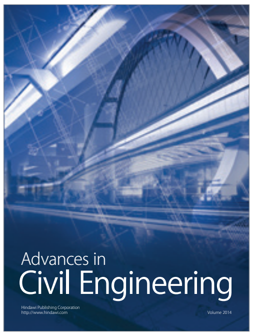
[10] M. Duchaineau, ROAM Algorithm Version 2.0, http://www.cognigraph.com/ROAM_homepage/ROAM2.

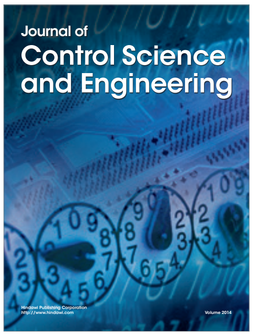
Journal of
Engineering

The Scientific World Journal

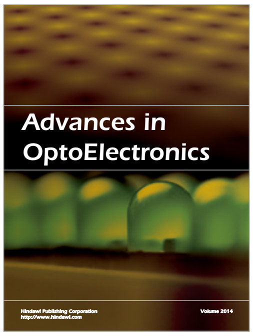
International Journal of
Rotating Machinery

Journal of
Sensors

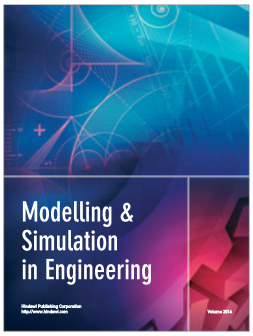
International Journal of
Distributed Sensor Networks

Advances in
Civil Engineering

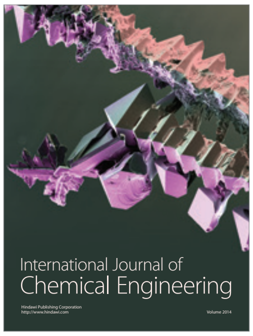
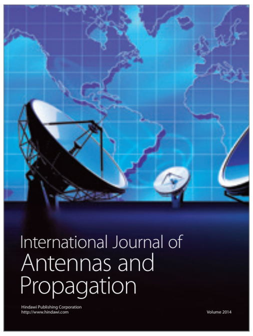

Journal of
Control Science and Engineering

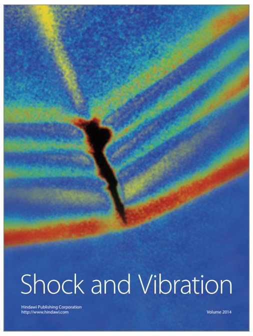
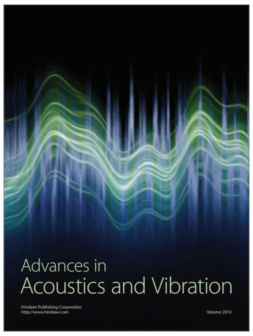
Journal of
Robotics

Journal of
Electrical and Computer Engineering

Advances in
OptoElectronics

VLSI Design

International Journal of
Navigation and Observation

Modelling & Simulation in Engineering

International Journal of
Aerospace Engineering

International Journal of
Chemical Engineering

International Journal of
Antennas and Propagation

Active and Passive Electronic Components

Shock and Vibration

Advances in
Acoustics and Vibration