Wii Will ROAM: Optimizing for Low-End Systems

Eric Raue Daniel Truong
Simon Fraser University
CMPT 464 Geometric Modelling

Abstract

In this paper we discuss our implementation of a terrain rendering demo using split-only ROAM with optimizations for low-end systems such as the Wii video game console. We examine the performance and visual appearance of two different algorithms to tessellate the terrain, a recursive one and another based on a distance priority queue. Furthermore, we examine some unique uses of view frustum culling by not only using it to reduce the number of triangles sent to the GPU but also integrating it into our splitting algorithm.

1 Introduction

Rendering large terrain using height maps efficiently requires algorithms that are very fast and only render the absolute minimum number of triangles given an error bound. We found that this sometimes comes with a trade off of memory which is a significant bottleneck of the scalability of low-end systems. We define a low-end system to have less memory, CPU, or GPU performance than modern computers.

First we provide background information about ROAM proposed by Duchaineau et. al. in '97 and then we discuss data structures and algorithms we use to implement it. We then look at a case study of implementing ROAM on the Wii and how it influenced our decisions during implementation. Next we look into optimizations using view frustum culling and fog to hide detail.

2 Background

In *ROAMing Terrain: Real-time Optimally Adapting Meshes*, Duchaineau et al. discuss a unique approach to viewing high levels of details in very large terrain sets. The general idea is that by using view-dependent error-metrics we can optimize the terrain by focusing more polygons and providing more details where the user will need/notice it. In addition using a bintree structure we guarantee that T-vertices are not possible.

One of the many useful applications for ROAM is it's usefulness in video games with large terrains. Some form of level of detail algorithm is required because it is not optimal for video game consoles with various constraints to render the entire terrain. The goal of ROAM is to render the scene in the least amount of polygons while maintaining a certain level of visual acuity.

In the paper, Duchaineau et al. specifically mention their wedgie-based error metric. The idea behind the error metric is that for each possible triangle we want to know how much volume it takes in world space. The wedgie is the bounding volume that encapsulates all the terrain geometry and the associated triangles. The larger this wedgie is the more likely we will have to split that triangle.

3 Data Structures and Algorithms

The authors of ROAM state that ROAM is simple to implement but we have found otherwise. Our data structures attempt to minimize memory usage and maximize algorithmic performance.

3.1 Wii Will Roam Data Structures

3.1.1 Terrain Height Map

A height map is loaded from TGA image files and normalized between 0 and 1. Once loaded it is scaled to an appropriate height that looks right for what the terrain is supposed to represent. The height values are stored in a compact array of floats where one float represents one height value. Looking up a height value can be done in constant time.

3.1.2 Terrain Texture

We store the RGB values of each point in the height map in a compact array of floats loaded from a separate texture file. It is three times larger than the height map because each colour has 3 components. If no texture is specified the height of the terrain at that point is used instead. Looking up a texture value can be done in constant time.

3.1.3 BinTree

The most important data structure used in ROAM is the bintree. Our BinTree structure is designed to reduce the amount of space being used by storing only relations to neighbours and children.

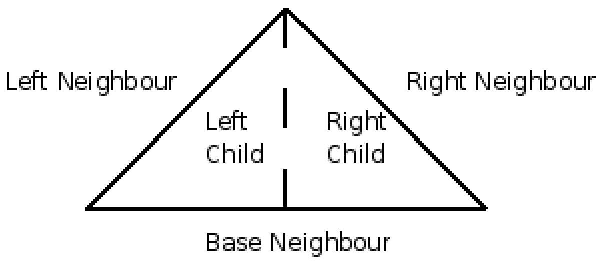

Figure 1: BinTree relations that are stored.

By storing the relationships of the triangle instead of triangle coordinates we can have an implicit representation of where a triangle is in world space coordinates which is useful for rendering. Recursive functions that require world space coordinates are passed indexes into the Terrain Height Map. This saves us memory from storing world space coordinates for each Triangle in the BinTree. We adopted this idea from [2].

3.1.4 ErrorTree

ROAM is a top-down algorithm however the wedgie based error metric is bottom up. In our implementation the terrain does not change so it is possible to store an ErrorTree which stores the error for every possible triangle in the terrain. Since it only stores the error, this data structure is useful for any error metric such as wedgie or variance. The data structure is a full-height binary tree implemented in a sequential array as mentioned in Brian Turner's article about ROAM. Finding a child or parent can be done in O(1) time. The error tree is calculated bottom up using a recursive function. The current implementation stores the variance of the center of the hypotenuse and the actual height at that point. The variance of the parent is calculated to be the largest of either the variance of it's children or itself.

Efficiency is maintained by having the ErrorTree only calculated once after loading our height map. During our splitting algorithms we would simply look up the error of the current node.

3.1.5 Priority Queue

The Priority Queue maintains the BinTree with largest error metric to be split by keeping it at the top of the stack. An unfortunate downside of implementing our priority queue was that we had to store the world space coordinates for each queue node we pushed into the priority queue. This is because the distance needs to be calculated to that node from the camera. Due to time constraints we did not have time to implement a depth-first search of the ErrorTree which would remove the need to store world space coordinates.

3.2 Wii Will ROAM Algorithms

3.1 Recursive Split

Recursive split is an idea adapted from an idea proposed by Byran Turner in his article about ROAM [2]. His implementation is based on dividing the world into equally sized patches and calling a recursive split algorithm on each patch. A positive aspect of this is that the whole terrain maintains a minimum level of detail equal to the size of the patch. Another benefit is that their error metric based on the variance in height from the center of the hypotenuse to the actual height performs computationally better than a wedgie based error metric.

The problem with such an algorithm is that it relies heavily on the patches to be already moderately detailed or small in size. In our opinion this almost defeats the purpose of ROAM. Our implementation of recursive split on a large terrain triangulation shows that it is not very adequate in terms of providing detail where it is needed. Our implementation also shows a lot of random popping when crossing base triangles due to the recursive nature of the algorithm. Figure 2 demonstrates this.

Once the algorithm chooses a triangle to recurse based on the frame variance, it force splits that triangle and recurses to the child nodes to check their frame variance. Based on distance it will choose either the left child or right child and recursively splits those until either a minimum frame variance is met or we have reached the finest level of detail. The issue here is that on a large terrain triangulation with a low polygon constraint such as that on the Wii, we recurse until the minimum frame variance is met, but we run out of triangles to split the other large base triangulation leaving us with one detailed triangulation and the other half bare.

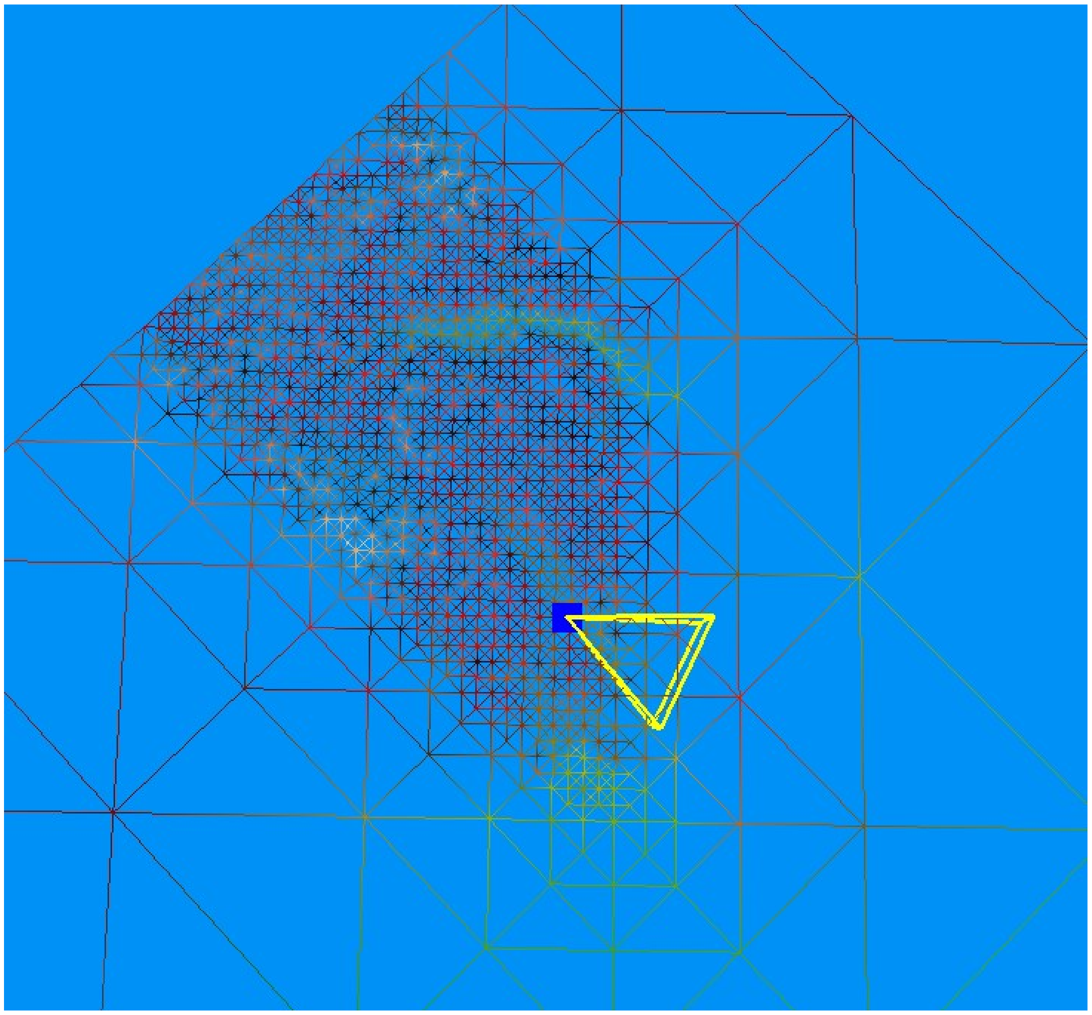

Figure 2: Recursive Split where detail added is concentrated on the wrong area.

3.2 Splitting with Priority Queue

As mentioned in the ROAM paper by Duchaineau et al. they maintain a priority queue of triangles with their error metric. A triangle is popped from the queue, forced split with the newly created children added to the priority queue. Rather than be recursive like the previous mentioned split algorithm, our implementation will keep popping the queue until we either run out of of triangles or we no longer can split as a result of running out of BinTree allocations. Another aspect is that we have defined a the finest level of detail to be the length of the hypotenuse. Once it reaches a certain length of detail, the triangle can no longer be split. The hypotenuse length is a variable we consider to change the distribution of triangle as discussed in section 4.4.

With the priority queue, we maintain a level of fairness unlike the recursive split algorithm. Triangles that are very inaccurate are at the top of the queue and will get first priority when splitting. This ensures a higher level of accuracy when splitting the terrain that falls in line with our memory constraints. For example if we can only split 10000 triangles, then with our priority queue the 10000 triangles with the highest error will be guaranteed to be split either explicitly or implicitly in force-split.

In our implementation of priority queue split we have a naive error metric: distance of the triangle to the point. We also do not add nodes to the priority queue based on the frustum culling technique described in section 5.3 which removes the detail not seen behind. At this time of writing we were unable to integrate our ErrorTree with our priority queue to use a better error metric such as variance of the height approximation to the actual height.

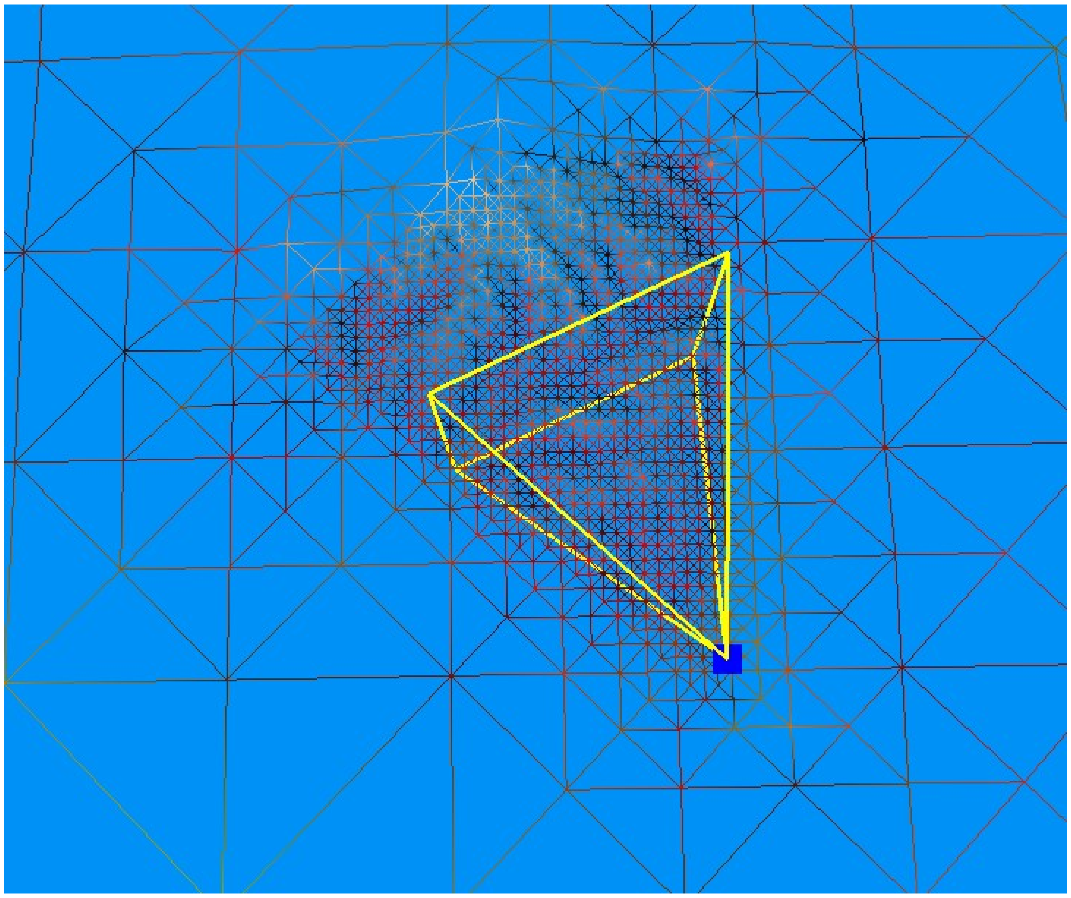

Figure 3: Priority Queue with frustum culling

4 Wii Implementation

We chose to implement ROAM on the Wii because of the unique programming challenges it presents and also to gain experience programming on a video game console. It required us to learn a new API and deal with memory constraints which are normally not present when programming for modern computers.

4.1 Overview

The Wii is Nintendo's latest home console and weakest in terms of hardware power of the current generation of video game consoles. It uses an innovative input device which utilizes accelerometers, a single-axis gyroscope, and an infrared camera. It has a 64-bit PowerPC-based CPU, which reportedly [6] clocks at 729 MHz. It has 88 MB of main memory with 24 MB used exclusively by the GPU leaving 64 MB general purpose use. It can output in 480p or 480i screen resolution in standard aspect ratio or anamorphic widescreen.

4.2 Libraries and Graphics API

Nintendo does not allow unlicensed developers like ourselves to have access to a development kit so we turned to the homebrew community for tools and support. In order to run our code on the hardware it requires installing a new Wii Channel called the *Homebrew Channel* [7] using a buffer overflow exploit found in a game. This allows anyone with a Wii to be able to develop software for it.

We use a compiler called *devkitPPC*, [5] which is based on the GNU Compiler Collection and a library called *libocg* provided by the homebrew community. The library provides a very thin layer over the hardware and we often found that it is not forgiving and causes our program to crash. There is minimal documentation and very few examples with source code using the 3D API so much of what we did was trial and error.

The graphics API is similar to OpenGL but the differences made it difficult to diagnose problems. Drawing primitives on the Wii works similar to OpenGL's glBegin and glEnd functions. However, the total number or primitives and the vertex format that will be used must be explicitly stated before drawing them. Vertex positions must be specified before specifying vertex colours; opposite to OpenGL. If an incorrect vertex format, number of primitives, or passing the vertex colour before position, it crashes the Wii.

4.3 Memory Issues

Unlike modern computers which usually have over 512 MB of RAM, the Wii only has 88 MB with 24 MB reserved for graphics. This means our data structures must be as efficient as possible. Sometimes this was at the cost of doing more computation. Many of our algorithms act on binary trees or perform constant look ups so this overhead in computation is minimal. The complexity of code increased and took hours to write, debug, and test as a consequence. It is better than not being able to load a reasonably sized terrain. In table 1 we found the first column to produce the best results on the Wii and the other two to run out of memory. In addition to these data structures we have code and the stack taking up memory.

	Size 512, Pool 10k	Size 1024, Pool 17k	Size 2048, Pool 50k
Height map	1 MB	4 MB	16 MB
Texture map	3 MB	12 MB	48 MB
BinTree pool	0.38 MB	0.65 MB	1.9 MB
ErrorTrees	4 MB	16 MB	64 MB
Total:	8.38 MB	32.65 MB	129.9 MB

Table 1: Approximate memory for large data structures

Table 1 describes the memory usage for terrains of a specific size in pixels and an appropriate pool size for that resolution. We see that the ErrorTrees take up the most memory because they store all possible errors of the terrain. As this data structure contains one float for every node, there is nothing we can do to reduce the memory required. The texture map also takes up a significant amount of memory compared to the height map but it is a necessary component to look visually appealing. It is clear that the memory used is dominated by the helper data structures which make the algorithms possible instead of the terrain map itself. Duchaineau et. al. claim that ROAM is memory efficient and based on these results we disagree.

In an earlier stage of our implementation we used a TGA loading library by [4] that would load a TGA in memory and then convert it to our own data structures. This meant having two copies in memory at a time. Depending on the size of the terrain, loading the texture map would fail because there was not enough space. We rewrote the TGA loader from scratch to read one pixel at a time and convert it to our data structure essentially streaming the data in.

See Future Improvements for how we propose to decrease memory usage.

4.4 Parameter selection

Perhaps the biggest advantage of developing programs for video game consoles is that all consumers will have the same hardware excluding the TV. This means we can select optimal parameters and expect good performance for all users.

The number of BinTrees we chose is 10k which does not crash on the Wii and looks visually appealing. Reducing memory usage of other data structures may yield a higher triangle count. Our height map is 512 pixels and this looks like an acceptable amount of triangles. This number is dependent on the error metric used.

We chose a larger minimum size for triangles in the priority queue algorithm described in section 3.2. This compensates a bit for the smaller pool of triangles which visually pushes detail further away from the camera. This distributes the details better.

5 Optimizations

5.1 View Frustum Culling

One very common way of optimizing a 3D program is to perform view frustum culling. Our view frustum is obtained by extracting the six planes of the frustum from our projection matrix and model view matrix. Initially we could easily test if a point was inside the frustum by calculating the signed distance between the point and a plane. However, to test if triangles are viewable in the plane, we need to perform a more robust test than simply checking if a single point of the triangle is in the view frustum. There could be cases where the triangle is very far apart and all three points are outside the frustum but the bounding area of the triangle is inside.

Our robust culling test checks to see if all three points are outside a particular plane, and if it is then the triangle is completely outside our view frustum. In our implementation we made it so that the recursive rendering of the BinTree triangles would check if the parent is outside of the view frustum. If it is then we do no not need to recurse any further to the children for rendering. This can be seen in figure 4.

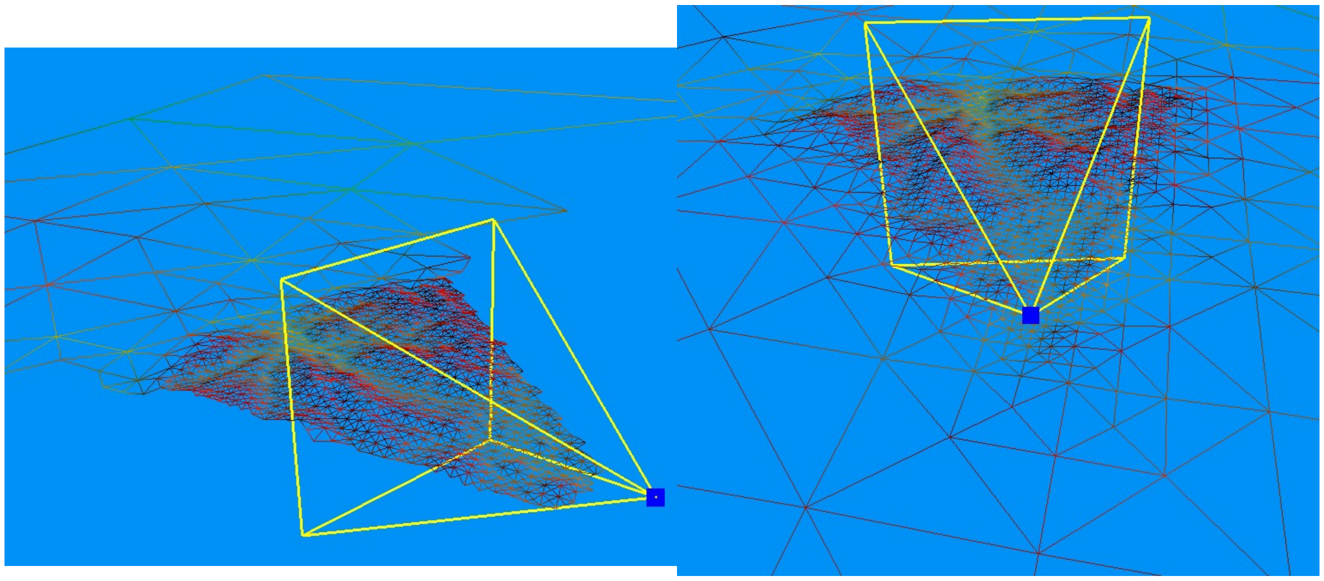

Figure 4: Frustum culling enabled (left). Frustum culling disabled (right).

5.2 Fog

This is not so much a hardware optimization, but rather an optimization on the visual acuity of the terrain. Since we know that parts of the terrain and objects further in the background will have less detail we can hide this from the viewer by putting a light fog over it. As the view moves around, the terrain moves out of the fog and closer to the viewer, giving it a higher priority for it to be split.

5.3 View frustum Culling with Split

One interesting optimization we came up with was to integrate the view frustum culling into our priority split algorithm. The results have been amazing. When popping a triangle to be split, we check to see if it is in the view frustum, and if it is we force split. However, the optimization comes in when the triangle is not in the view frustum. We do not split it, or add it's children to the priority queue. Instead we save ourselves BinTree allocations that can be used for other splits for triangles in our view frustum. This is demonstrated in figure 4.

6 Future Improvements

Rethinking some of our data structures – Due to time constraints we decided to go with algorithms that first came to mind.

Patches – Instead of loading the entire terrain, paging in parts of it would be more memory efficient and perhaps increase the triangle count on the Wii.

Error Tree lookup integrated with our priority based splitting – Implementing this would really get our priority splitting to look good. The current problem is that since the algorithm is not recursive, it is a lot more difficult to find out what node you are in the Error Tree for the error lookup. Any triangle could be chosen to split at any time and we need to implement a function that will look up the triangle's error metric in the Error Tree.

Merge – If the view does not change much from frame to frame it would be beneficial to maintain the current tessellation and split and merge triangle accordingly. Currently every time we move we reset the terrain at the coarsest level and split. An optimization mentioned in the paper is to merge and split triangles. However we currently feel that implementing merge would require a large amount of code refactorization and using different data structures. It may not be that beneficial for fast paced games such as racing games and flight simulators.

Frustum culling with our recursive split – One way to make our recursive split algorithm is to check if the triangle about to be split is within our view frustum. If it is not then stop at that level.

7 Conclusion

With proper error metrics, real-time optimally adapted meshes can provide very good approximations of the terrain. However the biggest limiting factor for widespread use of ROAM on consoles seems to be memory issues. As our case study of implementing ROAM on the Wii shows, the biggest problem we faced was trying to get everything to fit into memory.

Programming for the Wii using the undocumented library has been a computationally expensive task for our brains. Not only were we programming a ROAM implementation from ground up for Windows and Linux, but also constantly having to port and refactor code for the Wii.

Recursive splitting algorithms work well on smaller terrain sizes such as the patches described in *Real-Time Dynamic Level of Detail Terrain Rendering with ROAM*, but are inaccurate and unsuitable for large terrain sets. On the other hand our View Priority Queue Splitting with integrated Frustum Culling has given us a noticeable boost in visual acuity over the recursive splitting algorithm.

Overall ROAM has been a very rewarding project to have worked on.

References

[1] Duchaineau, Mark and Wolinsky, Murray and Sigeti, David E. and Miller, Mark C. and Aldrich, Charles and Mineev-Weinstein, Mark B. 1997. ROAMing terrain: real-time optimally adapting meshes. *Proceedings of the conference on Visualization '97*, 81-88.

[2] Turner, Bryan. Apr. 2000. Gamasutra - Features - "Real-Time Dynamic Level of Detail Terrain Rendering with ROAM. 13 April 2009 http://www.gamasutra.com/features/20000403/turner_01.htm

[3] Fernandes, Antonio Ramires. April 2009. View Frustum Culling Tutorial. Accessed April 13th 2009 at http://www.lighthouse3d.com/opengl/viewfrustum/index.php?intro

[4] Fernandes, Antonio Ramires. April 2009. A TGA Library. Accessed April 4th 2009 at http://www.lighthouse3d.com/opengl/terrain/index.php?tgalib

[5] devkitPro. (n.d.) Retrieved April 14, 2009 from http://www.devkitpro.org/

[6] Wii. (2009, April 13). In *Wikipedia, The Free Encyclopedia*. Retrieved 10:21, April 14, 2009, from http://en.wikipedia.org/w/index.php?title=Wii&oldid=283562073

[7] The Homebrew Channel. (n.d.). Retrieved April 14, 2009 from http://hbc.hackmii.com/