

To the Graduate Faculty:

I am submitting herewith a project written by Christopher Stephen Jones entitled “Terrain Streaming on the Nintendo GameCube”. I recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Interactive Technology in Digital Game Development, with Specialization in Software Development.

Gary Brubaker, Faculty Supervisor

We have read this Project
and recommend its acceptance:

Dr. Wouter van Oortmerssen, Academic Advisor

Colt McAnlis, External Reader

Accepted for the Faculty:

Dr. Peter Raad, Executive Director, The Guildhall at SMU

TERRAIN STREAMING
ON THE
NINTENDO GAMECUBE

A Project Presented to the Graduate Faculty of
The Guildhall at Southern Methodist University

in

Partial Fulfillment of the Requirements

for the degree of

Master of Interactive Technology

in Digital Game Development

with

Specialization in Software Development

by

Christopher Stephen Jones

(B.A., Fort Lewis College Durango Colorado, 2001)

September 15, 2007

Terrain Streaming on the Nintendo GameCube

Faculty Supervisor: Professor Gary Brubaker

Master of Interactive Technology degree conferred September 15, 2007

Project completed September 13, 2007

The goal of this project is to answer the question “What are the best practices for streaming level of detail based terrain on the Nintendo GameCube?” There are three key problems when wanting to display a portion of a much larger piece of terrain. The first is the streaming of data. While I will limit the exploration of the problem to terrain data specifically, the nature of the problem will be the same on any system that wishes to stream data. The second problem is that of level of detail. When viewing a large expanse of terrain, the number of triangles used to display geometry in the distance can be significantly less than up close, with no loss of detail. The third problem is that of compression. Compression is useful in that it both reduces the amount of storage for data and it reduces the amount of data that needs to be transferred. The streaming of data benefits from compression since more data can be buffered in the same amount of space and more data can be read from disk in the same amount of time. It does however require CPU time to decompress the data. This project provides a solution to displaying large expanses of terrain using streaming, level of detail, and compression.

Efficient streaming on the GameCube requires that an application intelligently stream data at a rate of no more than 31000 bytes per second and buffer sufficiently large amounts of data to compensate for 67ms seek times. The level of detail system must limit the number of triangles on the screen to less than 72900 triangles per frame if 60 frames per second is to be maintained and a low memory footprint is required. For this project it is limited to no more than eight megabytes. Compression must be used to allow the 31000 bytes per frame to keep up with the viewable area and to increase the amount of buffered data to compensate for any read rate slow downs.

TABLE OF CONTENTS

TABLE OF CONTENTS	II
COMMON TERMS.....	III
1 INTRODUCTION.....	1
1.1 PROBLEM STATEMENT	1
1.2 MOTIVATION	1
1.3 ABOUT THE GAMECUBE.....	2
1.4 PROJECT OVERVIEW.....	3
2 LEVEL OF DETAIL SYSTEM.....	4
2.1 LOD OVERVIEW	4
2.1.1 <i>Chunk Based Tiled Geometry</i>	4
2.1.2 <i>Geometry Clipmaps</i>	5
2.2 GAMECUBE SPECIFIC DETAILS	5
2.3 LOD METHOD USED	6
2.3.1 <i>First Attempt</i>	6
2.3.2 <i>Hybrid Method</i>	7
3 STREAMING SYSTEM.....	9
3.1 STREAMING OVERVIEW.....	9
3.2 GAMECUBE SPECIFIC	10
3.3 STREAMING METHOD USED	12
3.3.1 <i>Data Baking</i>	12
3.3.2 <i>First Method</i>	13
3.3.3 <i>Second Method</i>	13
3.3.4 <i>Final Method</i>	14
3.3.5 <i>Read Buffering</i>	16
4 COMPRESSION	18
4.1 COMPRESSION OVERVIEW	18
4.2 FOR THE GAMECUBE.....	19
4.3 PROBLEMS ENCOUNTERED.....	19
4.4 WHAT I USED	20
5 FINAL SOLUTION	23
6 CONCLUSION.....	25
6.1 RESULTS.....	25
6.1.1 <i>Best Practices</i>	25
6.2 FURTHER IMPROVEMENT.....	26
7 APENDIX 1 READ SPEEDS	28
8 REFERENCES.....	31

COMMON TERMS

Chunk	A small piece of the over all terrain
Chunk Size	The number of quads that make up a row of a chunk
Total Chunk Size	Is the Chunk Size squared
Chunk Number	The number of the chunk in the over all terrain.
Window	The viewable portion of the terrain. Made up of a number of chunks
Cell	This is a single entry into the window.
Terrain	Refers to the complete set of chunks that make up the terrain
Terrain Size	Is the number of vertexes in one row of terrain
Total Terrain Size	Is the Terrain Size squared

1 INTRODUCTION

1.1 Problem Statement

The goal of this project is to determine the best way to stream level of detail terrain on the Nintendo GameCube. To accomplish this task, three key elements must be in place. The first is a level of detail component to take some of the burden off the streaming component by ensuring that terrain that is only partially loaded can still be displayed and to maximize the amount of terrain that can be displayed to make the viewer feel they are looking a large expanse of terrain rather than a small island. The second is a streaming component that can manage a pre-allocated section of memory to pull in new terrain data and discard old. The third component is data compression. This is necessary to reduce the amount of data that is stored on disk, minimize the amount of data that needs to be read from the disk, and maximize the amount of data that can be buffered in memory. The reduced read amount and increased buffered amount reduce the risk that a read delay will impact the visible terrain.

While using the GameCube adds a number of risky elements, part of the project is to work in a limited resource environment. While that could be accomplished on a PC though limiting the available memory, putting in sleep commands to slow the system down, and other tricks, there would always be questions such as, “Did something get missed?” or “Is the OS doing something behind the scenes I don’t know about?” By using the GameCube as the development environment, it guarantees that the resources are as limited as they are claimed to be and there is no interference from external sources.

1.2 Motivation

The system is geared towards the GameCube, but it should work well in any limited resource environment. Even when working without the hardware requirements of the GameCube, having an efficient terrain component will free up resources that can be used for other aspects of a game.

Since a lot of the research being done in the area of terrain focuses on making the algorithms GPU friendly [VTERRAIN], this project is a divergence from the norm. Since

the GameCube does not have a programmable shader pipeline, most of the newest research doesn't directly apply, but the concepts used are useful. However with mobile devices becoming more powerful and more popular a system that doesn't use a GPU should still be useful.

While the GameCube is older technology, the Nintendo Wii is a popular item at the moment [WII]. Since the Wii and GameCube share a similar architecture, a terrain engine developed for the GameCube would give a good starting point for one on the Wii. The "Broadway" CPU used in the Wii is the successor of the "Gekko" used in the GameCube. One of the major bottle necks, data transfer from the disk, is the same for the Wii as the GameCube and the second bottle neck, memory constraints, is somewhat lifted with the Wii's 88 megabytes of main memory.

1.3 About the GameCube

At first glance the Nintendo GameCube looks like a limited system; however it holds an amazing amount of potential. It has a 486 MHz "Broadway" CPU with 24MB of 1T-SRAM for main memory, 16MB of audio memory [GC], and uses a fixed function pipeline. The hardware supports native asynchronous data reads from the optical disc, which is capable of transferring data at around 2MB/s.

With the development hardware used for this project, the DVD drive is emulated and all the data is actually stored in a remote location and transferred over Ethernet. Since the broadband adapter is actually slower, reading from the emulated drive takes longer than from the actual optical drive [DM]. This means that if streaming works with the emulator it should also work with the actual drive, but the results using an actual drive could be somewhat different.

There are two ways in which the GameCube API allows rendering of geometry. The first is immediate mode. In this mode, instructions are sent to a buffer individually and then processed by the video hardware. The system is very similar to OpenGL's immediate mode. The second way is through display lists. These lists are compiled lists of graphics instructions that get loaded into memory. Part of a display list is an array of

locations within the display list that hold pointers to data. When using a display list, just the pointers need to be modified to change what the list is drawing. Display lists provide a performance boost over immediate mode, but are more complicated.

1.4 Project Overview

First the entire terrain is divided up into chunks where each chunk contains a portion of the overall terrain. This is almost like cutting up the terrain into a lot of smaller pieces which makes loading specific chunk of terrain fast since the entire chunk is stored sequentially. The level of detail system makes sure that all the different chunks of terrain fit together seamlessly regardless of what level of detail a specific chunk or its neighbors are being rendered

at. The LOD system is covered in chapter 3. The streaming system works by monitoring a window made up of a number of cells where each cell contains one terrain chunks. In figure 1.1 the entire image is the terrain but the visible window is everything within the white box. As the user moves about the window moves and the data that is stored in each cell of the window is streamed in. This system is covered in more detail in chapter 4. The data compression portion of the project helps keep more data buffered in main memory and also reduces the amount of data that is stored and read from disk. Compression is covered in chapter 5.

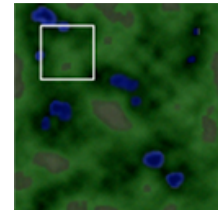


Figure 1.1:
Terrain
Window

2 LEVEL OF DETAIL SYSTEM

2.1 LOD Overview

Just within the terrain arena there are a huge number of level of detail algorithms [VTERRAIN]. At their heart, all level of detail algorithms try to reduce the number of triangles rendered without reducing the visual quality of the object being drawn. The two level of detail methods that mine is based on are chunk based tiled geometry [WOUTER] and Geometry Clipmaps [HOPPE]. For this project, level of detail is necessary to keep the number of triangles being displayed at a reasonable level for the GameCube and to make sure even partially loaded chunks mesh nicely with chunks of different levels of detail.

2.1.1 Chunk Based Tiled Geometry

With this system the over all terrain is split up into a number of chunks (Figure 2.1). If the terrain were an image, the effect would be to create number of smaller images. The exact size of each chunk is chosen to maximize speed for a particular system. If drawing extra triangles is less expensive than sorting through a large number of chunks, then larger chunks would be used and visa versa.

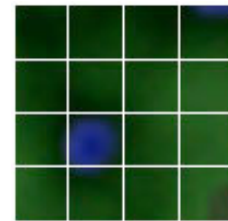


Figure 2.1:
Terrain Chunks

Index lists are created to draw these chunks at different levels of detail. For example if a chunk size of 32x32 quads is used, index lists will be generated to draw the chunks at that resolution and every power of 2 down to a single quad. For every level of detail additional index lists are generated to adapt a chunk to chunks of lower detail. Figure 2.2 shows a 4x4 resolution adapting to a 2x2 resolution.

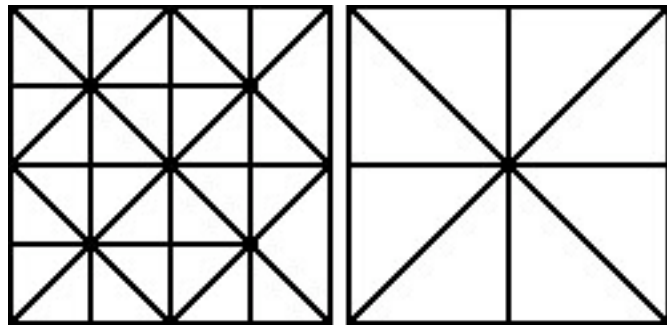


Figure 2.2: Four by Four resolutions adapting to a two by two

What level of detail a particular chunk is drawn at is determined by a level of detail bias based on how different one level of detail is compared to the next and the distance to the viewer. A very good bias can eliminate popping caused by switching from one LOD to another. This also means that chunks that are very far away from the camera can have just as high a level of detail as chunks that are very close.

Since at any given point in time a lot of terrain will be at the lowest resolution, a performance boost may be seen by dynamically creating a list of all the lowest LOD chunks and drawing them at the same time. This significantly reduces the number of draw calls for trivial numbers of triangles and therefore can increase triangle throughput. The viability of this step is explored in section 2.2.

2.1.2 Geometry Clipmaps

This method “caches the terrain in a set of nested regular grids centered about the viewer.”[HOPPE] As the view moves about the terrain, old data is pushed out of the regular grids and new data pulled in. The method uses highly compressed terrain data that allows the entire terrain to be stored in memory and specific regions decompressed just enough to give the desired amount of detail. The system creates blending regions between different levels of detail and uses geometry morphing to eliminate any seams. This method is very different from the previous and what I will be using since it is not chunk based. However the level of detail bands and the way it pulls in new data and pushes out old are concepts I will be working with.

2.2 GameCube Specific Details

The GameCube has an estimated limit of 23 million vertexes or 7.5 million triangles per second using position, a single texture, and generated texture coordinates [DM2]. This translates into about 125 thousand triangles per frame running at 60 frames per second. Testing the GameCube using immediate mode resulted in 72900 triangles per frame running at 60 frames per second.

Tests using display lists showed just a few microseconds improvement in rendering speed. Since their use adds a certain amount of additional complexity and requires extra memory to store them (two bytes per triangle to be displayed using 16 bit indexing and triangle lists), display lists were not used for this project.

Drawing a 32x32 quad chunk takes about 0.183 ms per call where drawing a 96x96 quad chunk takes about 1.700 ms. The larger chunk has about 9 times as much data and takes about nine times as long to draw. This is to be expected since in immediate mode commands are sent one at a time, and so long as they are sent fast enough the size of a given object doesn't particularly matter. Based on this, the idea of reducing draw calls by dynamically creating buffers as presented in section 3.1.1 would yield no performance boost in immediate mode. Since the data being generated would need to be flushed every frame, it could in fact slow things down.

Two other factors when dealing with a LOD system for the GameCube are that it has a very limited amount of RAM and that it uses a fixed function pipeline. The low amount of RAM makes loading the entire terrain set into memory, such as with Clipmaps, inadvisable even if it is only a reduced set and using large amounts of RAM for low detail areas, such as the chunk based system does, is very wasteful. The fixed function pipeline forces any sort of geometry morphing to be done on the CPU rather than the GPU.

2.3 LOD Method Used

2.3.1 First Attempt

The first attempt at an LOD system was using the chunk based tiled geometry method. A terrain of 64x64 chunks at 32x32 quads per chunk resolution required about 13 megabytes of data (assuming 3 bytes per vertex). The problem is that a chunk being displayed at the lowest resolution still takes the same amount of memory space as chunks being displayed at the highest. With memory at such a premium, this wasted space is very costly. When loading a chunk it also has to be fully read even if it is never displayed at anything but the lowest resolution. A different method was needed, mainly to combat the large memory overhead.

2.3.2 Hybrid Method

The method implemented takes the idea of chunk based tiling and combines it with different regions of detail similar to how Clipmaps work. The allocated terrain memory is broken up into chunk distributions based on how many chunks at what level of detail are desired. An example distribution would be 2x2 chunks at maximum resolution, 4x4 at one less and 6x6 at the next lower. Figure 2.3 shows an example of this. As can be seen in the image the higher resolution bands are in the middle where the viewer would be. Instead of using geo-morphing to blend the different regions together, the stitching method of chunk based tiling is used between different detail regions.

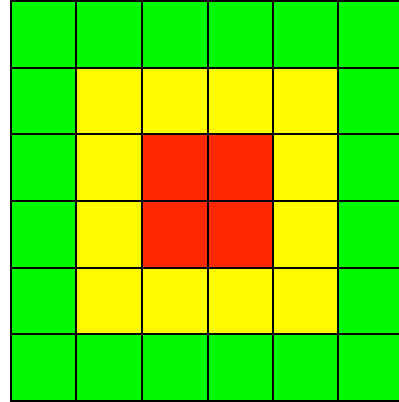


Figure 2.3: Green is the lowest level of detail and red is the highest.

Happily the outer bands don't have to worry about chunks that fall into inner bands. So in the example in Figure 2.3, there are 4 highest level chunks, 12 medium level (as opposed to 16) and 20 chunks at the lowest level (as opposed to 36). Table 2.1 shows the memory savings of using different LOD regions versus a single fixed size LOD. With this chunk distribution there is already a 25% memory savings. As the distribution gets larger and more low detail chunks are added the savings get even larger. The downside of this method is the ability to have high detail chunks far away is removed since there is no memory for the additional detail. This side effect can be partially compensated for by choosing sufficiently large bands based on the complexity of the terrain to be displayed. The cost however is in the sheer number of chunks that can be displayed.

With this system calculating the maximum number of triangles and draw calls possible for a given chunk distribution is very easy. Since it is often important for a system to work within a certain time slice, the maximum number of triangles and draw calls works as a worst case scenario. If it turns out that drawing the entire set of chunks can take more time than is available, the distribution can be modified to better accommodate time restrictions.

Number of Chunks	Resolution	Size in bytes	Triangles	Time to Draw in ms
20	9x9	4860	2560	
12	17 x 17	10404	6144	
4	33x33	13068	8192	
		28332	16896	2.556
vs				
36	33x33	117612	73728	8.888
Note: The resolution is the number of vertices and each vertex takes 3 bytes.				
Table 2.1: LOD System's Memory Comparison				

3 STREAMING SYSTEM

3.1 Streaming Overview

Streaming is “a technology for transferring data so that it can be received and processed in a steady stream” [DICTIONARY]. Streaming is what allows internet videos to start playing shortly after they are clicked rather than when they are completely downloaded. The benefit for videos is that the amount of time the user has to spend waiting before the video starts is minimal. In games, this translates into reduced or eliminated loading screens. Anyone who has sat looking at a loading screen for two or three minutes knows how much they interrupt the suspension of disbelief. With streaming, an entire game could be playable with no more than a single initial loading screen.

Any game with vast regions to explore and no loading screen is undoubtedly using some form of streaming. *Metroid Prime* by Retro Studios needs bidirectional region based streaming [WILSON] and the game does a good job of hiding any delays by keeping doors closed a few extra frames, or through short cut scenes. Games such as *Word of Warcraft* [WOW] and *Dark Ages of Camelot* [DAOC] allow a player to travel the entire length of a continent without ever looking at a loading screen. *Oblivion* [OBLIVION] is a single player example of a large world game that only uses loading screens when the player teleports or enters a city or dungeon. *Grand Theft Auto 3* [GTA3] is an example of a modern large world with little loading times. The omnidirectional region based streaming method [WILSON] has all the characteristics these large world games would need from a streaming system.

A few different streaming options are presented by Kyle Wilson [WILSON], and of these, terrain streaming falls under omnidirectional region based streaming where each chunk is a region. As with Wilson’s description of region based streaming, knowing the worst case scenarios is important in calculating just how fast the viewer can

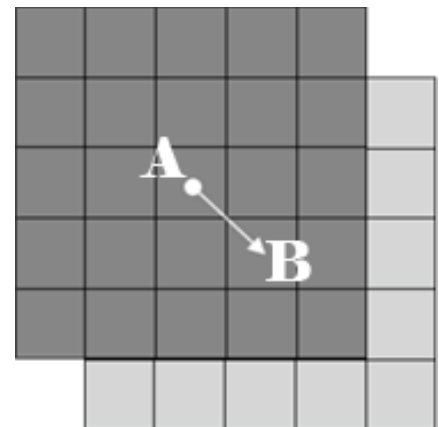


Figure 3.1: Loaded and needed

move. If the terrain window is 5x5 cells then the system must be able to load 9 chunks worth of data in the time it takes to move diagonally across one cell. In figure 4.1, the dark cell represent data that has already been loaded and the lighter colored cells represent data that needs to be loaded in the amount of time it takes for the viewer to travel from A to B. In addition, since the inner cells of the window are at a different level of detail, they too have to have information steamed in. The exact amount of data is dependant on the window's level of detail distribution.

While this project is mainly focused on terrain, another use of streaming is to load low level of detail models or textures first and then load the higher levels of detail as soon as they are needed. Gears of War [GOW] used this method. After loading a saved game, occasionally some pieces of the scene would suddenly pop with more detail.

Providing an uninterrupted gaming experience is a huge benefit of streaming. Being able to explore an entire continent without ever waiting for a loading screen provides a much better overall experience and adds to the idea that the whole game is one big world, rather than a whole lot of smaller pieces that you teleport between.

3.2 GameCube Specific

The GameCube supports both standard and asynchronous DVD reads. With both types the data being read must be read into a 32 byte aligned memory location, the read size must be a multiple of 32 bytes, and the file offset must be a multiple of 4. The terrain data to be read must take these restrictions into account.

The GameCube comes with built in support for asynchronous DVD reads where a read is issued and when the system finishes reading, a callback function is activated. Unfortunately, the only information provided to the callback is information about the file that was read and the number of bytes read. In addition, the callback function can have no commands that could potentially stall a thread, including debugger break points.

The optical drive is advertised to deliver around 2MB of data per second. Reading 1 megabyte from a single file took 581 ms, so 2MB per second is about right. . Running tests on the DVD showed that from 128 bytes to 8192 bytes required about the same amount of read time, so any read buffers used should be at least 4096-8192 bytes in size. Increasing the size of reads beyond 8192 did not significantly increase the amount of data

read compared to the time to read it. Reading from different files proved to be a bit inconclusive. Certain read sizes actually received better read rates when reading from one file then the other. This could be due to getting lucky and being able to start a read from the second file as soon as the first read is completed. Worst case however was that the read time was increased by around 5ms. The mean read time for reading 8k when seeking between every read was 71ms. Reading 8k at a time for a total of 1MB took 4ms. This shows the seek time to be around 67ms. This all means that within a single $1/60^{\text{th}}$ of a second frame 31k can be read directly, but for every file switch this amount is reduced some what and at 60fps for every seek approximately 4 frames of loading are lost.

In a simple scenario of one level of detail, moving at an angle would cause a new row and new column of data to be loaded. This means one seek at 67ms for the row data and if the column data is also stored in a format another seek for it. If using the 4x4 resolution with each chunk 4 byte aligned as the lowest resolution over 900 chunks of data could be read in one frame ($31000/32$), but since an optimal buffer size is around 8k only 256 chunks will be read in taking approximately 4ms. Total time to read a new row and column is 142 ms or 8.5 frames. Which means it must take the viewer longer than 8.5 frames to move diagonally across on chunk of data or the streaming will be outpaced.

In a more complicated scenario, every level of detail requires one seek for the row data and a number of seeks equal to the height in chunks of that level of detail for the column data plus 4ms. The row data also requires 4ms for every 8192 bytes worth of data needed.

The fact that streaming new data takes at least 8.5 frames to load, means that an amount of data equal to however much content the viewer can get to in 8.5 frames must be loaded ahead of time. In the worst case, the viewer may have stopped right at a loading boundary and when they start moving again multiple rows and columns will be needed with the 8.5 frame window.

The calculations presented here are based on the idea that only the terrain streaming engine will be accessing the DVD, in a real game situation there would possibly be many different systems all trying to read data from the DVD. At the very least audio will be getting streamed off. It is important to keep these other systems in mind when determining just how much terrain can be streamed.

3.3 Streaming Method Used

3.3.1 Data Baking

The first step of my streaming method was to pre-bake the data into a format that fit better with the LOD system. Since the system breaks the terrain up into a series of chunks, it made sense to do the same with the data to be streamed. The terrain data was converted to a chunk format where every chunk was stored so that the entire chunk was sequential rather than spread out. While this added an amount of redundant data at the right and bottom edges of every chunk, any chunk could be read all in one go. In addition, padding was added to keep all chunks aligned along a four byte boundary so they could be offset to when reading. While this could have been fixed by simply reading a little extra data and then ignoring the first few bytes, the extra padding is useful for storing special information about the chunk.

To make reading the different levels of detail easier, the data is stored in different files for each level of detail. If the maximum level of detail uses a chunk size of 32x32 then files are created for 16x16, 8x8, 4x4 and 2x2. To keep the amount of redundant data down, only the unique data is stored for each LOD could be stored. This does mean to build a 32x32 chunk the system first has to read in the 2x2 data, then the 4x4 data, etc.

The read rates from the GameCube specific section show that 7 32x32 chunks could be read in one frame. Likewise so long as the amounts per read were the same 327 4x4 chunks could be read in the same amount of time. Of course since the chunk data is stored in a row major format, any time a column of chunks is needed they would have to be read one at a time incurring the 67 ms seek delay. To fix this, the lowest resolution data is stored in column major format as well. Only the lowest resolution is stored both because it takes up the majority of the displayed terrain and because storing the higher resolutions as well would take too much space. If the total terrain is $2^{14} \times 2^{14}$ and each point is one byte, the final size is around 272MB. If the maximum resolution is 32x32 then the 4x4 representation would take 8MBs. The extra size is due to the duplicated edge data and padding chunks that are required to be 4 byte aligned.

An unfortunate side effect of setting up the data in the manner outlined above is that it makes compressing the data difficult. The chunks are all expected to be the same size,

and the edges of each chunk have to match exactly. See section 5 for more detail on these problems.

3.3.2 First Method

My first attempt at streaming terrain data used a single level of detail, asynchronous loading, and toroidal addressing of the allocated memory. Each memory space was 32 byte aligned so it could be read directly into. The terrain engine kept track of where in the memory space new data was to be added and pushed those positions and the chunk numbers to load into a list that the asynchronous loader would use to determine what was to be loaded and where it should go.

Unfortunately, the system had problems with data coherency. Since the loading thread was asynchronous and could not use a mutex, because they can stall a thread, trying to keep the main thread and the loading function in guaranteed synchronization wasn't possible. It also had a bug where the same memory location could be in the list twice if the user was moving in a certain direction. This leads to chunks being missed. The complexity of the toroidal addressing and updating of the chunks made this system very hard to maintain.

In addition, since only height data was being read in, the data had to be converted to X, Y, Z component vertices to be drawn. The method was flawed since any benefit from reading directly into the memory space was lost. Attempting to draw by either not using index lists, thus removing storage of X and Z, or by copying the data into a fixed array with X and Z already in place lead to very low performance. Due to the difficulty of the toroidal addressing and since any speed benefits from reading directly into a memory location were lost, a different system was needed.

3.3.3 Second Method

Changing the asynchronous loading over to an actual thread that used standard loading was easily accomplished. Since the thread could be debugged and allowed mutexes this change had the additional benefit of allowing the loading thread to deal with all aspects of loading and processing new data. Prior to this change the loading thread was kept very simple since it was a blind spot as far as debugging was concerned.

The method of dealing with memory locations was also changed. Rather than using the toroidal addressing, the system was switched to use a window made up of a number of cells equal to the number of chunks to be displayed. Each entry into the window consists of a pointer to the location of the chunk data, the fixed level of detail for the cell, the currently loaded level of detail, and the chunk number. As the camera moved, each window entry was copied to a neighboring entry based on which way the camera was moving. The entries with the oldest data also pushed their locations into a list of free memory. It's important to note that while the window entries are copied, the data they pointed to stayed in the same place. The final row or column had its pointers set to null to indicate that they needed new data. To keep the main thread from trying to draw the window as it is being shifted, a mutex was used.

For example, if the camera is moving left to right, all the locations pointed to by the left most column of the window are pushed into a list, each window entry then sets its self equal to the location to its right. The rightmost column of entries all set their positions to null and increased their current chunk number. When the window finished updating, a loading function was called that would then loop through all the window cells looking for null pointers. If a null pointer is found, it would then read in data based on the chunk number of the window entry.

3.3.4 Final Method

The above system worked fine so long as only one stored level of detail was used. I.E. all chunks are stored at 32x32 resolution. Once multiple stored levels of detail were used, the locations pointed to by the window entries were no longer of the same size and simple shifting of the terrain window no longer worked since different levels of detail required different amounts of memory. As mentioned in Chapter 2 the terrain window is actually made up of a number of bands of different maximum levels of detail. As the camera moves, low LOD data needs to be moved into high LOD positions and visa versa. In the first case, the low detail data is spread out and copied into a higher level's memory location and the window's loaded LOD is set to the lower level's loaded LOD. If the chunk is drawn before the higher level data is loaded, it is simply rendered at the lower

level. This works since all the data needed for the lower LOD is in place. In the reverse direction, the lower level of detail samples the higher one's data and only copies what it needs. After the window is finished being moved, the loading function scans through it again for any cells that have a loaded LOD that is less than the fixed LOD.

It is important to note that when copying the window cell data between cells with the same fixed LOD, the location pointers are actually swapped but the data they point to is unchanged so only a small amount of data is exchanged. Swapping the pointers causes the oldest data to bubble up to the window edge that is receiving new data. All the pointers along the edge are then added to the free list.

Another note is that the order in which the window cells are copied is very important. When shifting the window right entries are copied right to left starting on the left side of the window. When shifting the window left entries are copied left to right starting on the right side of the window. The same can be applied to shifting up and down. It is important for the window to be able to shift in any direction since the viewer can move in any direction. The more complicated approach for different LODs is to ensure that the only entries in the viewing window that can not be drawn are on the edges.

Figure 3.2 shows a row of the window being updated as the viewer moves from left to right. Since new data will be coming into the right side of the window, updating starts on the left. First cells A and B swap information, but the actual terrain data they point to doesn't need to be moved. Then B copies the higher LOD data from cell C into itself. Since B and C are not the same LOD they have different memory requirements so their locations points can not be swapped. Cells C and D then exchange information just like cells A and B did. Cell D then copies the lower LOD data from E into itself. D also sets its loaded LOD to that of E so it will be drawn at E's LOD until new data is loaded. E and F now exchange their information. New data is then loaded into F and the update is done.

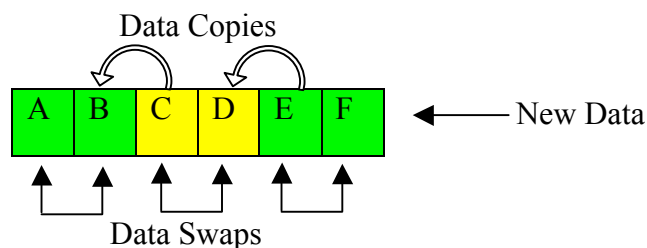


Figure 3.2: The viewer is moving left or right so new data comes in on the right, but the updates start on the left.

3.3.5 Read Buffering

One of the major problems with the streaming method is that it has to do one read for every missing chunk of data. The apparently 2D height map data is really just a very long 1D array of values. Knowing how wide the height map is allows it to be displayed in 2D by inserting the equivalent of line breaks every width distance in the 1D array. For purposes of read from disk however, the 2D list of values are stored sequentially as a 1D array. The speed of each read is dependant on a number of issues.

The first is the physical distance on disk from the data previously read to new data to be read since it takes time for the DVD's laser to move into position. Switching files can be particularly expensive since it guarantees the laser is going to have to move to a new position or at best at least wait for the disk to come around again. The second is the location of the file on the disk. For every rotation, more data can be read from files on the outer edges of the disk than files towards the center simply because there is more room on the outer edge [BRUBAKER]. All of these issues mean how data is laid out on disk is very important to streaming performance. Files that need to be read fastest should be put on the outer edge of the disk. Something that must be compensated for, but cannot be easily predicted is extra read time because of damage to the DVD.

When dealing with a window that's 128x128 there is the potential for 256 reads every few frames if the viewer is moving at an angle. The vast majority of these reads are only going to read a small amount of data. A 5x5 vertex chunk is only 25 bytes after all. Testing revealed that the time it took to read a 5x5 chunk was about a third of the amount of time to read a 33x33 chunk despite the fact the 33x33 contains roughly 43 times more data! Obviously reading in small amounts is a bad idea. A method to combat this is to read in many chunks if the read size is small and then check to see if any of the other chunks are needed. For rows this works very well since the data is stored row by row. By always reading in 33x33 bytes worth of data, the time it took to read in all the data for a 128x128 chunk terrain was reduced by a factor of 10.

For columns the situation isn't so simple. If the user moves left or right, it will take a number of reads equal to the window size just to get the lowest detail, but as the user continues to move, each subsequent read can make use of the extra data read in. So while the first few reads may only read one usable chunk of data, the following reads will make

use of more and more of the buffered data. Since this problem only has a significant impact when there are many many chunks of data to read, it could also be solved by storing the lower level of detail (which most the chunks are) in a column major format as well. While this would take more space on disk, it would only be a small subset, less than 1/100 of the full size when going from a 32x32 resolution to a 2x2 resolution, of the overall terrain being duplicated.

Using the $2^{14} \times 2^{14}$ terrain size with a maximum resolution of 32x32 and a minimum of 4x4, the minimum resolution is 7MB. The amount of duplicated data will always be $((\text{terrain size}) / (\text{max resolution}))^2 * \text{chunk size at resolution}$. Using a window size of 60, reading a new column one chunk at a time would take 4260ms ($60 * (4\text{ms read} + 67\text{ms seek})$). Using a single column read would only take 71ms. A 4189ms savings at the cost of 7MB of storage.

Using just one level of detail stored as column and row major would allow the viewer to move diagonally across the terrain at a speed of nine frames per chunk, or six chunks per second, at 60 frames per second so long as each row and column could be read in one read. If the distance between points were one meter and a chunk is made up of 33 points, this would be a speed of 276 meters per second ($\sqrt{33+33} * 6 \text{ chunks}$), 993.6 kilometers per hour, or .276 millimeters per millisecond which translates 4.6 millimeters per frame ($.267 * 16.67 \text{ ms per frame}$). For every additional read per new row and column of data, the maximum speed is reduced by 20 millimeters per frame ($4.6 \text{ mm per frame} * 4.25 \text{ frames per read}$) or 1.2 meters per second.

4 COMPRESSION

4.1 Compression Overview

Compression methods come in two flavors: lossy and lossless. With lossy compression some quality is sacrificed by discarding information for greater amounts of compression. The exact amount of compression is partly based on the amount of error that is deemed acceptable. With lossless compression methods, no quality is sacrificed, but less significant amounts of compression are achieved. For compression of terrain data using the level of detail system presented in Chapter 2 a lossless method is needed or at least lossless enough so that terrain right at the near clip plane would not be moved.

Data is generally first transformed in some way to make it more easily compressed. The HAAR transformations are a way of transforming data that then allows for that data to be quantized, entropy encoded, and/or run length encoded [SALOMON]. While the transformation itself is lossless, the method used to encode the data can be lossy or lossless. One of the nice things about the HAAR transformation is that one method of doing it uses a matrix to encode the data and then use the inverse of that matrix to decode the data. This does rely on the size of the data being a power of two, but extra zeros can be added if it is not.

Difference encoding is another transformation method where rather than storing the exact value of a given point, the difference between a point and the previous point is stored [SALOMON]. This method is useful in situations where the data changes gradually. An exact starting value is stored and then for every point after that the difference between it and the previous point is saved. For example, the sequence 10, 8, 11, 5 would be changed to 10, -2, 3, -6. To reconstruct the original data, start with the second to furthest left point and add the value of the previous point and then move to the next point and add the value of the previous point, etc. As long as the differences are small enough, they can be stored as fewer bits.

Entropy encoding is a method where values are assigned to symbols based on the frequency at which they occur [ENTROPY]. The more common a value is the smaller the symbol used to represent it is. There are adaptive methods that adapt based on the data

being encoded and static methods that require the data to be pre analyzed and the frequencies determined before encoding begins. Two common forms of entropy encoding are Huffman and Arithmetic both of which can be adaptive or static.

A compression technique could use one or many of the techniques mentioned above. For example the data might first be transformed using difference encoding to reduce the number of different values and then entropy encoded to reduce the overall size. Each extra layer of encoding, however, adds time when decompressing.

A good example of compression is Hughes Hoppe's Clipmap demo [HOPPE]. In this demo the entire United States is compressed from 40GBs down to 350MBs using a pyramid compression scheme and quantizing the results [HOPPE, MALVAR]. The compression method used here is particularly interesting since it supports partial and region of interest decompression. The first allows data to be decompressed just enough to get the amount of detail desired and the second allows only a specific part of the overall file to be decompressed.

4.2 For the GameCube

The compression method needs to work in a small amount of memory and allow fast decompression. The GameCube has support for S3TC compressed textures and all decompression of such textures is carried out in hardware [GPG]. While this project is not focusing on texturing a terrain beyond a very basic level, the use of compressed textures is certainly appealing given the limited amount of memory.

There is some appeal to using the compression on heightmap data. The method is very good for textures that have smooth changes in color and gradation [GPG], much like a height map would. The encoder provided with the Dolphin SDK did not work as intended. Were the encoder to have worked, there may have been potential for compressing the internal components of a chunk and then decompressing them in hardware.

4.3 Problems Encountered

The system designed required the compression used on the terrain to be nearly lossless and to compresses every chunk by the same amount. Being nearly lossless was required so that the edges of each chunk line up and no seams form due to precision loss

from compression. This removed some common compression techniques such as quantization. It would be possible to just quantize the inner portion of a chunk, thus leaving the edges uncompressed.

The streaming system required that every chunk be of the same size so they could be found quickly. Requiring the same size for every chunk meant statistical methods and simple run length encoding could not be used unless all chunks were padded to be the same size. This makes the compression only as good as the best it can do on the worst chunk, which potentially means no compression overall and a lot of time wasted decompressing. Using Amir Said's Huffman and Arithmetic encoding implementations [SAID], most chunks were compressed to about half size, but the worst cases were slightly larger than the uncompressed chunks so the overall file size was larger because of padding. With varying sizes there would be no easy way to simply offset to the correct chunk. The streaming system heavily relies on being able to calculate offsets into a file based on nothing more than a chunk number. Were the chunk sizes to be variable, the streaming system would have to be redesigned to take this into account.

The lesson learned from this process is that all elements of a system must be taken into account when developing any piece. If that is not possible then time must be provided for iterations so that previously developed systems can be retooled to work with newly developed ones. Compression is a very integrated part of an overall system. If it is not built in from the beginning, trying to add compression in after the fact leads to problems with the design decisions made for other systems. In this instance a few of the design decisions about how to stream and draw data made finding a suitable compression method very difficult.

4.4 What I used

Breaking the terrain up into separate files for each resolution and into chunks, allows for sections of the terrain to be only partially loaded. This is very useful since chunks that the viewer never goes close to will only need a minimal amount of data transferred from disk. Since height map based terrain falls onto a predictable grid. Only the height data need be stored and the position of it can be calculated based on where in the height map

the point was read from. This alone reduces the size of the height map by 2/3 since only one of three components needs to be stored.

The terrain was smoothed out to allow no more than a seven unit change between any two points in a row. The smoothing was done by first checking the distance between every two points in a row and if greater than seven, adding half the distance between them beyond seven to the smaller point and subtracting the same from the larger. If the difference contained a fraction, the difference rounded down is applied to the first point and rounded up and applied to the second.

Since the first pass could cause previous points to fall out of the seven distance range a second pass was done just like the first, but the difference beyond seven was only added to the left point. While lossy, the change was applied the terrain as a whole, before it is chunked, so any loss of data would not lead to misaligned seams and still allow for a slope of 81 degrees. For example given the three points 10, 19, 28, the first pass would convert the points to 11, 19, 26 and the second pass would convert them to 12, 19, 26. This method would not work for all terrain types, however the terrain used with this project resulted in having less than 0.001% of the data points changed and the average change was under 2.

After the terrain is broken up into chunks, the maximum resolution data is then difference encoded by storing the first point of a row exactly and then storing the difference of the rest. Since the terrain was smoothed to differences of no more than seven, each point could be packed into just four bits. Only the highest resolution could be stored in this way since the difference between a point and points more than one away could be greater than seven. Encoding in this way reduced the size of a 32x32 quad chunk from 1089 bytes to 561 bytes.

In addition, the terrain data is restricted to a maximum distance of 255 between the lowest and highest points in a chunk. The value of the lowest chunk is then subtracted out of every chunk when storing the data and applied as a position modification when displaying the data. For example if a chunk's lowest point is 750 then the highest possible point in that chunk is 1005, and the modifier saved would be 750. When drawing the chunk, 750 would be added to every point's height value. Doing this allows each point to be stored as one byte but the total range in heights is limited only by the size of the

modifier. With a two byte modifier, a height of 65k can be reached over the course of 256 chunks. The slope restriction reduces the maximum variation along a row to 231 per chunk.

5 FINAL SOLUTION

The system starts by filling in all the entries of the terrain window based on the cameras starting position. The fixed LOD for each window entry is determined by a constant array of resolution and number of chunk pairs. The number of chunks for each resolution must be even and larger than or equal to the size of the resolution one step higher. Table 5.1 is an example chunk distribution.

Resolution	# of Chunks
32x32	4
16x16	12
8x8	48
4x4	512
2x2	3520

Table 5.1: Example Chunk Distribution

The highest resolutions can be skipped by setting them to 0 chunks. Once the window is filled, each cell in the window has its location pointer set to an appropriate memory space and the main thread kicks off a secondary thread. At this point the main thread is tasked with updating the camera and drawing the terrain. Control of the terrain window is completely given to the secondary thread.

The secondary thread consistently monitors the camera position and as the camera moves it moves the terrain window. Since the camera is updated in a different thread from the window, updating the camera position and reading the position are placed within a mutex. The exact point when the window will move is different depending which way the camera is going. This is to eliminate any places where the viewer could cause the system to load and unload data by moving back and forth over the same point.

After updating the terrain window, the secondary thread loops through the terrain window and looks for any chunks that need new data. If any are found it reads the appropriate chunks. Whenever a chunk is read, extra data is also read. These extra chunks are used as buffer data to reduce the number of over all reads. The exact number of buffered chunks depends on the resolution of the chunks being read and the buffer size.

The main thread draws chunks of data based on the terrain window. Since data at the edges of the window could potentially be missing, fog is applied and the outer few rings of chunks are completely obscured. Since the terrain window is modified in another thread, updating the window and drawing are mutexed. While there is some concern that this mutex could greatly reduce the draw speed, in practice the amount of time the render

spends waiting to get a lock is trivial. When drawing chunks, the main thread is also responsible for frustum culling and stitching one chunk to its neighbors. The main thread is also responsible for gathering user input. Table 5.2 shows the control flow for both threads. Table 5.3 shows the frame time spent on each step. In both tables, stylized text is used to show when that particular thread is waiting on a mutex lock. A mutex is used to make sure that if one thread is modifying data, another doesn't read that data while it's being changed. Waiting on a lock causes a thread to sit and wait until it can get a lock so making sure the wait time is minimal is important for performance. In this application, the terrain window and camera's position are both used across multiple threads so they must be within a mutex lock. The terrain window is modified in the loading thread, and the main thread uses it to determine what to draw and the level of detail to draw it at. The camera's position is modified by the main thread but used in the loading thread to update the want window.

Main Thread	Secondary Thread
Read Input	Get Camera Position
Update the Camera	<i>Update window</i>
<i>Draw Terrain</i>	Read new data
Repeat	<i>Copy new data from buffer</i>
	Repeat
Table 5.2: Control flow of each thread. The styled text is used to show when each thread needs a particular mutex lock.	

Main Thread	Time in MS	Secondary Thread	Time in MS
Read Input	0.018	Waiting on Camera Mutex	0 (0.035 max)
Waiting on Camera Mutex	0.08 max	<i>Waiting on Terrain Window Mutex</i>	0 (0.094 max)
<i>Waiting on terrain window mutex</i>	0.001 (0.833 max)	Updating Window	0.226
<i>Draw Terrain</i>	15.131	Read new data	16.129 (108 max)
		Processing New Data	0.048
Table 5.3: Frame Time Distribution. . The styled text is used to show when each thread need a particular mutex lock. Any times that are not listed as max are an average.			

6 CONCLUSION

6.1 Results

Using the final level of detail, streaming, and compression methods described above the system is able to efficiently render a terrain of using a distribution of 4 32x32s, 12 16x16s, 48 8x8s, 512 4x4s, and 3024 2x2s totaling 61056 total triangles at 16.68 ms per frame while streaming and 15.88 ms per second when not. This is roughly 88% percent of the number of triangles from the performance test in section 3.2. The viewer is able to move at one chunk every 120 frames or about 2 seconds per chunk at an angle without ever waiting for missing chunks to load.

To reduce the number of seeks, the 4x4 and 2x2 resolutions were also stored in column major format taking an extra 10 MBs for a $2^{14} \times 2^{14}$ terrain. Every new row and column pair requires 14 reads of 4096 bytes per read. For this distribution 4096 byte reads worked just as well as 8096 byte reads and required less room. According to section 4.3.5 this many reads and that much data should only take up about half (14 reads at 71 ms per read) of the 2000ms to travel a chunk. The rest of the time is spent processing and decompressing data. Only the 32x32 resolution data was compressed and decompressing it required 0.058 ms per chunk as opposed to the average time per read of 0.048 ms for the entire buffer. The compression resulted in nearly doubling the number of 32x32 chunks read in at one time.

6.1.1 Best Practices

The original level of detail system was not very efficient for such a low memory environment. Optimizing the memory used by low detail chunks allows a lot more terrain to be displayed with minimal loss of detail so long as the chunks are properly distributed.

Using the terrain window as a fixed array with pointers to data made determining what data needed to be loaded much simpler. The window constantly being scanned made loading new data very easy, even if the user suddenly changed direction. The window needs to be kept to a size of no more than 60 x 60 and the total number of triangles less than 62k.

Baking the data into a format that allowed chunks to easily be found and read made for simpler data loading. Using asynchronous loading is nice, but the additional headache of dealing with a function that cannot be debugged means it needs to be kept very simple, or needs to be tested in an actual thread using standard loading and then changed over to asynchronous loading. There is potential for using a combination of both to hide the seek times for data by processing a chunk in one thread while loading the next asynchronously.

6.2 Further Improvement

While the banded level of detail system is very simple it doesn't allow highly detailed chunks at a distance to be rendered with the amount of detail they deserve. It would be interesting to keep the idea of chunk distributions for memory purposes but allow chunks to request a desired level of detail and then get whatever is available. For example a chunk in the distance might request the highest level of detail but only receive a middle level of detail. That way if no close chunks needed the detail, ones further off could use it. Of course chunks would also need the ability to take a higher level of detail away from another chunk.

The constant looping through the terrain window to look for new data to load could be improved by keeping a list of chunks that need to be loaded. There is the additional complication of needing to make sure the list is kept current even if the view changes direction and certain chunks no longer need to be loaded.

The loading thread could be made to use a double buffer and asynchronous loading so that the next chunk could be being read in while the current chunk is being processed. This would speed up load times a bit at the expense of simplicity and memory for the extra buffer. Keeping the loading thread and asynchronous function in synchronization could be a bit difficult since mutexes can not be used.

Storing terrain data in both row and column major chunks is a big waste of space, but is necessary to keep the number of seeks down. An alternative solution would be to store the highest level chunks just as they are, but then store the lower level chunks as chunks themselves rather than in rows. For example if 32x32 is the highest level of detail then

the first 4 chunks in the 16x16 data set would contain the first two chunks from the first row and column of the terrain and the second two would be from the second row and column. In this way reading data would not be so biased to row major or column major.

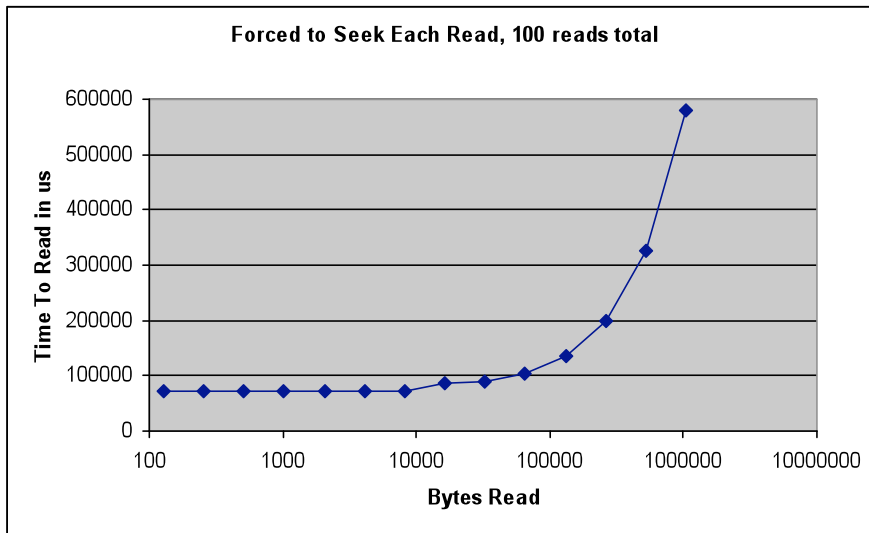
The compression of the data could stand for some major improvement. Using a method that allows region of interest decompression and partial decompression would totally eliminate the need to format the data into chunks and to save multiple resolutions. The system over all should be redesigned with a compression method in mind.

Specifically to the GameCube, the Audio memory could be used as a temporary buffer to store data into. Using just one quarter of the available 16MB could provide enough space to store over 150k 4x4 resolution chunks. In addition, display lists could be used to boost performance.

7 APENDIX 1 READ SPEEDS

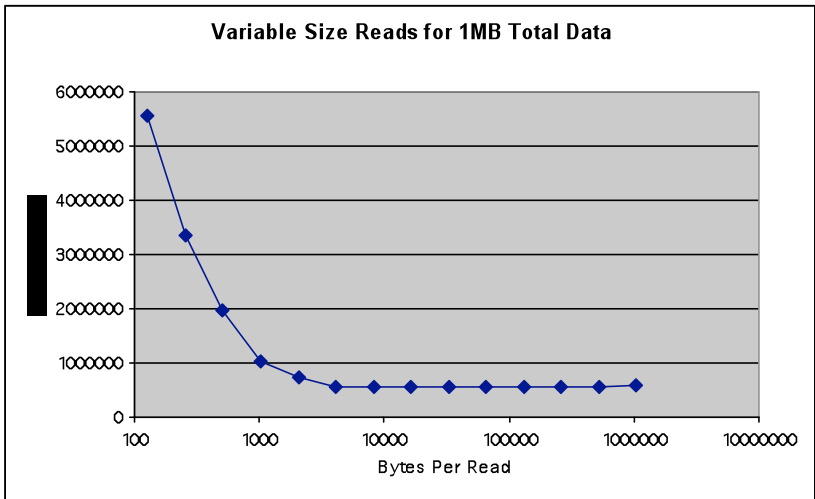
Forced to Seek Each Read, 100 reads total per size

bytes per read	Total Time	Mean	Bytes per us	Bytes Per Second
128	7163363	71633	0.00178687	1786.870217
256	7158667	71586	0.003576085	3576.084766
512	7178539	71785	0.007132371	7132.370528
1024	7188347	71883	0.014245278	14245.27781
2048	7220490	72204	0.028363726	28363.72601
4096	7226213	72262	0.056682525	56682.52513
8192	7262165	72621	0.112803826	112803.8264
16384	8758352	87583	0.187067156	187067.156
32768	8909847	89098	0.36777287	367772.8697
65536	10498530	104985	0.624239775	624239.7745
131072	13675198	136751	0.958465099	958465.0986
262144	20011226	200112	1.309984706	1309984.706
524288	32706634	327066	1.603002009	1603002.009
1048576	58107629	581076	1.804541018	1804541.018



Variable size reads for 1MB total data

bytes per read	Total Time	Total Reads	Mean	Bytes per us	Bytes Per Second
128	5546021	8192	677.00	0.189068163	189068.16
256	3349969	4096	817.86	0.313010658	313010.65
512	1973496	2048	963.62	0.531329174	531329.17
1024	1026650	1024	1002.59	1.02135684	1021356.84
2048	742179	512	1449.57	1.41283437	1412834.37
4096	562632	256	2197.78	1.863697763	1863697.76
8192	551628	128	4309.59	1.900875228	1900875.22
16384	551891	64	8623.30	1.899969378	1899969.39
32768	551032	32	17219.75	1.902931227	1902931.23
65536	548075	16	34254.69	1.913198011	1913198.01
131072	550514	8	68814.25	1.904721769	1904721.77
262144	550809	4	137702.25	1.903701646	1903701.65
524288	551225	2	275612.5	1.902264955	1902264.95
1048576	575256	1	575256.00	1.822798893	1822798.89



Reads From One File Then Another Every Other Read 100 Reads Total per size

bytes per read	Total Time	Mean	Bytes per us	Bytes Per Second
128	7315977	73159	0.001749595	1749.60
256	7285613	72856	0.003513774	3513.77
512	7255723	72557	0.007056499	7056.50
1024	7051050	70510	0.01452266	14522.66
2048	7316273	73162	0.027992394	27992.39
4096	7258321	72583	0.056431784	56431.78
8192	7339745	73397	0.111611507	111611.51
16384	7564778	75647	0.216582694	216582.69
32768	8828059	88280	0.37118012	371180.12
65536	10399788	103997	0.630166692	630166.69
131072	13986831	139868	0.937110057	937110.06
262144	20481805	204818	1.279887197	1279887.20
524288	32355291	323552	1.620408854	1620408.85

8 REFERENCES

- [OBLIVION] *Oblivion*, Bethesda Softworks 2006. <http://www.bethsoft.com/>
- [BRUBAKER] Brubaker, Gary. 2007. Streaming lecture at the Guildhall.
- [DAOC] *Dark Ages of Camelot*, Mythic Entertainment 2001.
<http://www.darkageofcamelot.com>
- [DICTIONARY] Streaming – Definitions from Dictionary.com,
<http://dictionary.reference.com/browse/Streaming>
- [DM] Dolphin SDK Manual 2004. DVDETH API Features List
- [DM2] Dolphin SDK Manual 2004. Vertex Performance Caluculator
- [ENTROPY] Planet Math: Entropy Encoding.
<http://planetmath.org/encyclopedia/EntropyEncoding.html>
- [GC] GameCube Specs, <http://www.segatech.com/gamecube/overview/index.html>
- [GPG] Nintendo GameCube Graphics Programmers Guide 2004.
Compression Texture Format
- [GOW] *Gears of War* Epic Games 2006. <http://gearsofwar.com/>
- [GTA3] Grand Theft Auto 3 Rockstar Games 2001. <http://www.rockstargames.com>
- [HOPPE] Hoppe, Hüge. Losasso, Frank. 2004.
Geometry Clipmaps: Terrain Rendering Using Nested
Regular, grids, <http://research.microsoft.com/~hoppe/geomclipmap.ppt>
- [ROAM] ROAMing Terrain: Real-time Optimally Adapting Meshes
http://www.cognigraph.com/ROAM_homepage/
- [SAID] Fast AC – FastHC Amir Said.
<http://www.cipr.rpi.edu/~said/FastAC.html>
- [SALOMON] David Salomon 2006. Data Compression 4th Edition
- [VTERRAIN] Terrain LOD Published Papers. <http://www.vterrain.org/LOD/Papers/>
- [WII] Nintendo Wii U.S. sales top Sony's PS3 2006,
<http://www.msnbc.msn.com/id/16115967/>
- [WILSON] Wilson, Kyle 2005. A Streaming Bestiary.
<http://gamearchitect.net/Articles/StreamingBestiary.html>
- [WOW] *World of Warcraft*, Blizzard Entertainment 2004.
<http://www.worldofwarcraft.com>

[WAVEREN] van Waveren, J.M.P. Real-Time DTX Compression

[WOUTER] van Oortmerssen, Wouter 2006. Lectures during term four

[XANTAX] http://wiki.xentax.com/index.php/List_Of_Compressed_Games