Nintendo
Software
Development
Support Group

4820 150th Ave. NE
Redmond, WA 98052

Phone:
(425) 861-2715

Phone Menu:
1 → AGB
2 → CGB
3 → N64
4 → GCN
5 → Lot check
6 → Licensing / How to become a developer
7 → Other questions

Fax:
(425) 882-3585

E-Mail:
support@noa.com



Edited and produced by
Yvette Kirby Waters, NTD

NINTENDO GAMECUBE™
Development News

Volume 1

©2000 Nintendo. All rights reserved.

CONFIDENTIAL

NINTENDO GAMECUBE™ Development News

December 1, 2000
Volume 1, Issue 1

Of Special Interest:

- HW1 to HW2 transition
- Racing game graphics
- Direct access to EFB

In This Issue:

Introduction	1
HW2 Update	1
Anisotropic Filter	4
Version Corner	5
CPU-EFB Access	6
Contact Us	6

©2000 Nintendo. All rights reserved.

Introducing the GCN developer newsletter

Welcome to the first issue of *NINTENDO GAMECUBE Development News*! We hope you'll find this newsletter to be a useful addition to Nintendo's developer support. Each month or so, the newsletter will address common development questions, alert you about bug fixes and current workarounds, and share tips to make your game development experience easier and more efficient.

Nintendo of America's support team has undergone some changes. The Engineering Tools group and the Developer Support group have joined forces to create the Software Development Support Group (SDSG). Over the next few months, we plan to reshape the face of support for Nintendo developers. These periodic newsletters are just a start; we are developing an NNTP newsgroup server for NINTENDO GAMECUBE (GCN) open forums, and we are working on consolidating our two websites to provide a more consistent and user-friendly experience.

Since we expect this newsletter to grow and adapt to the needs of the GCN developer community, we encourage you to send us your questions and suggestions of topics you'd like to see covered. Please e-mail your comments to support@noa.com.

Enjoy!

—The Software Development Support Group



What's new in HW2

By Dante Treglia II, SDSG

ATI and NTD have been working hard on a new version of the ATI Graphics Processor that supports some exciting new features and bug fixes (listed inside). We use HW1 and HW2 macros to differentiate between the two versions of the GX API. HW2 development kits are in production and should be ready by the holidays. This article outlines a few of the new changes to the graphics chip and how they will affect your games.

New TEV compare functions

We put a lot of effort into keeping the new API as backward compatible as possible. The new TEV compare functions, however, will have a small impact on porting programs, because the compare functionality of `GXSetTevClampMode` has been removed in the HW2 API. Please be aware of this change when porting programs from HW1 to HW2. The functionality has been improved with some new TEV compare functionality. You now have the option of comparing 8-bits, 16-bits, 24-bits, and there's even a per-component compare con all three color channels. This provides the functionality for per-pixel conditional shaders.

(*continued next page*)

HW2 update

New Features in HW2

- ***NBT indices separated***
- ***Fractional shift works with 8-bit vertex attributes***
- ***Renormalization and "post-transform" matrices added for texgens***
- ***Line aspect ratio fixed for field-mode rendering***
- ***New TEV compare functions added***
- ***More flexibility in TEV for texture and raster color component swaps***
- ***New TEV "constant" color registers, component selectable***
- ***Subtractive "blend" mode***
- ***New texture copy formats (for both color and Z)***
- ***Scissor-box offset***
- ***Bug fixes***

(*continued from previous page*)

Renormalization and "post-transform" matrices

This is a very exciting new feature. We've added additional functionality and matrix memory to the texture coordinate generation unit in HW2 (shown below). Whereas the HW1 path stops after the first matrix-vector multiply, the HW2 path has an optional re-normalization step followed by a second "post-transform" matrix-vector multiply, with 64 new matrix rows.

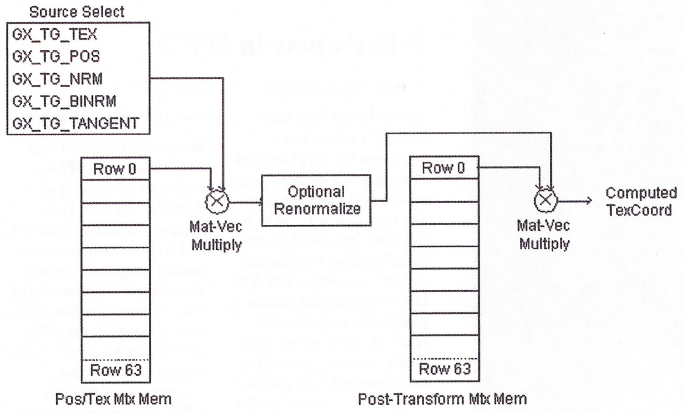
There are various ways to make use of these extra features. You can choose to provide twice as many matrices for position transformation and use the extra memory for texture coordinate generations. This is handy for optimizing skinned characters. To get the extra matrices, you would select an identity matrix for the first transformation, then load the texture coordinate matrices into the new memory. This feature will also optimize projected textures on skinned objects. Since position matrices are loaded in the first memory, the re-projection matrix (and the transformation into texture space) can be loaded in the second memory. This relieves the CPU from having to do additional, redundant work. Also, since the second memory can be used to transform coordinates into texture space, normal matrices can simply be copied into the position matrix memory for environment mapping. This saves the CPU from having to manipulate the matrix. Use the function `GXSetTexCoordGen2` to access this feature on the HW2.

Bypass bug on HW1

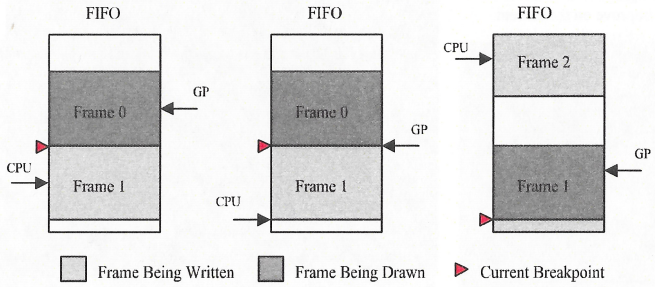
A few bugs managed to make their way into the first version of the chip, but these have been successfully eliminated on the second version (see "HW1 & HW1_DRIP Errata" in the *NINTENDO GAMECUBE Hardware Transition Guide* for more details). Most of the bugs are pretty minor, except for the insidious bypass bug.

(*continued next page*)

"We've added functionality and matrix memory to the computation path for texture coordinate generation in HW2."



HW2 update



Using breakpoints helps bypass the bypass bug.

(*continued from previous page*)

This bug manifests when a lot of bypass registers (raster-related commands), or a mixture of bypass registers and XF registers (lighting-related commands), are sent down the FIFO. Large vertex sizes also increase the likelihood of this bug occurring. The bug causes the graphics processor to crash randomly, frequently, and unpredictably. We have seen cases where adding one line of code would change the frequency of the bug. Fortunately, this bug has been fixed on HW2 and will no longer be a problem.

So how do you avoid the bypass bug if you have a HW1 development kit? We've added a few workarounds to the Graphics library to prevent the bypass bug from occurring, but with little success. It is possible, however, to abort the frame, restore affected registers, and then continue drawing. You will lose frames using this method, but your program will continue to run. The DEMO library has been modified to perform this workaround for you. Check out the Dolphin Reference Manual page for `DEMOEnableBypassWorkaround`, or see the source for more details. And there is one more thing you can do…

GX breakpoints

We have found that the bypass bug occurs less often if FIFO breakpoints are used. Actually, we recommend this feature for any game, regardless of the bug. The breakpoint feature allows the application to have two frames of graphics in one FIFO, while staggering the CPU and GP by one frame. The CPU and GP write and read from the same FIFO, but the CPU can instruct the GP to stop processing at a certain FIFO address until the CPU is done drawing the current frame (see figure above). Once the CPU is done with the frame, the CPU disables the current breakpoint (allowing the GP to process the new frame) and sets the new breakpoint to the beginning of the next frame. This process maximizes the bandwidth of both processors. The drawback is that you may need to double-buffer your textures and indexed data.

"Fortunately, [the bypass] bug has been fixed on HW2 and will no longer be a problem."

Conclusion

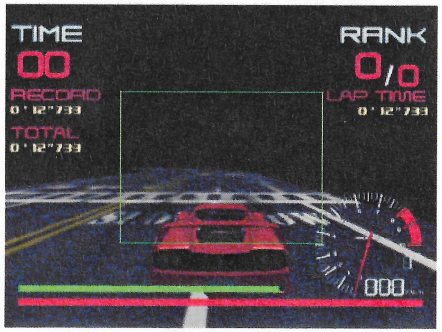
We hope that you find the new features and bug fixes as useful as we have. All our tests have come back positive on the new chip. Eventually, all development kits will be based on HW2. Stay tuned for more…

Anisotropic filtering: Improving the racing game

By Ryan Woodland, NTD

The big question on everyone's mind right now is "How is NINTENDO GAMECUBE going to improve on the current generation of games?" Here at NTD, we recently set out to answer part of that question by tackling the monster that is the street-racing game. In this article, I'm going to talk about the single best improvement we've come up with so far.

The choice to delve into the street-racing genre was an obvious one for us. Simply put, we have limited art assets and we had already done quite a bit of car work by developing the now infamous car demo. The first major challenge we faced was developing a track that looked good enough to truly demonstrate the GCN's graphics capabilities.



Mipmapping with no anisotropic filtering (enlarged area of detail on page 5)

"How do we make a road that actually looks like one would in real life?"

In looking at other racing games, we noticed a nasty similarity in track design: basically, roads have little or no detail! In most racing games, the yellow and white lines, crosswalks, and markings to control traffic are replaced by flat, boring, dark gray asphalt. The few games that have attempted to add road detail (Ridge Racer V on PS2, for example) suffer from nasty sampling problems. Our first problem became quickly apparent—How do we make a road that actually looks like one would in real life?

Our first step was to take advantage of the GCN's seemingly unlimited texturing capabilities by texturing the living daylights out of our road. We figured that the presence of large textures coupled with mipmapping would make a more than convincing presentation… Boy, were we wrong!

The problem is simple. Mipmapping suffers terribly in this situation because the road polygons appear on-edge relative to the viewer. Consequently, there is a large sample space in the *s* direction and a very small sample space in the *t* direction of road textures. This causes a very small texture LOD to be chosen for a polygon that is very close to the viewer. In our initial attempts, the resulting image was so blurry that it sent me running for the aspirin bottle—obviously, this just wouldn't do! So, after trying a few things like hand-generating mipmap levels and turning off mipmapping altogether, we were left scratching our heads.

Then it occurred to us—we had yet to really test out anisotropic filtering! Anisotropic filtering increases the number of texture samples taken per pixel, thereby reducing blurring.

Check out the difference in the screenshots on the next page. (Please pay no attention to the frame bars in the picture above, as our screen-capturing process renders them inaccurate.)

The difference in drawing time is a little less than 1% between ANISO1 and ANISO2. The time difference between ANISO1 and ANISO4 is less than 3%—a small price to pay for the visual difference!

Finally, we decided to play with the mipmap bias to improve things further. We artificially biased mipmap LOD selection so that larger LODs would be selected closer up, which prevents further blurring.

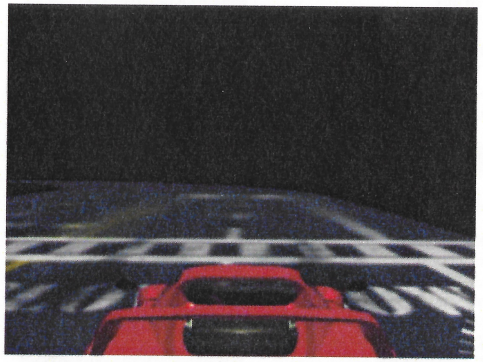
As a result of all of this, we are able to apply very convincing textures to our road without suffering the effects of blurring or flickering! We've added crosswalks, traffic arrows, road lines, and even writing to our track with awesome results. Say goodbye to the days of boring gray roads—NINTENDO GAMECUBE's graphics will convince even the best state transportation inspector!

SDK Version Corner

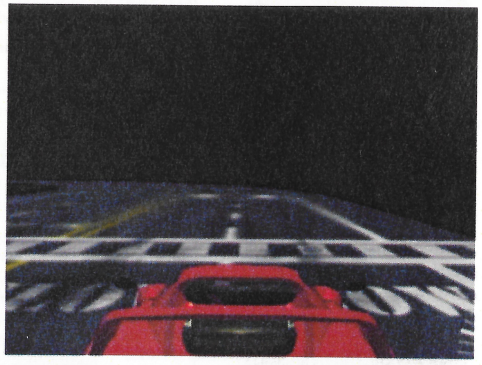
As many of you may have noticed, we are using the install image date as the version number until we finalize the NINTENDO GAMECUBE Software Development Kit.

The latest release is dated **November 10, 2000**. We expect to issue a new release in the coming week.

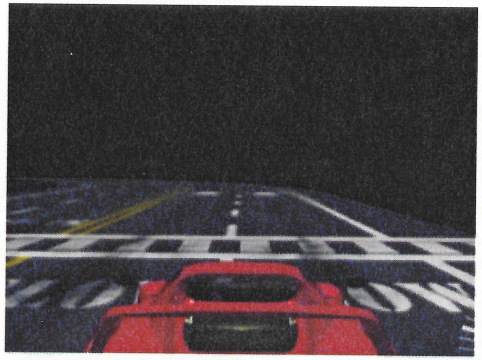
Anisotropic filtering in action!



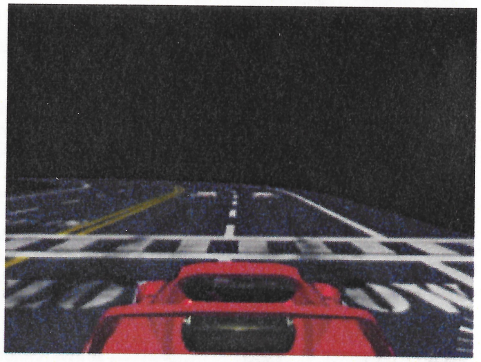
Mipmapping with no anisotropic filtering (enlargement from previous page)



Mipmapping with anisotropic filtering set to ANISO2



Mipmapping with anisotropic filtering set to ANISO4



Mipmapping with anisotropic filtering set to ANISO4 and LOD Bias enabled

Nintendo Software Development Support Group

4820 150th Ave. NE
Redmond, WA 98052

Phone:
(425) 861-2715

Phone Menu:
1 → AGB
2 → CGB
3 → N64
4 → GCN
5 → Lot check
6 → Licensing / How to become a developer
7 → Other questions

Fax:
(425) 882-3585

E-Mail:
support@noa.com



Edited and produced by Yvette Kirby Waters, NTD

Direct access to EFB

By Satoru Hosogai, SDSG

Quite a few people have asked us about CPU direct access to the Embedded Frame Buffer (EFB), so I've put together some tips for those of you who want to try this. Please keep in mind, however, that the EFB was not designed to be accessed by the CPU directly, so this type of access is very limited.

Originally, `GXPeekARGB` and `GXPokeARGB` were designed to verify the data in the EFB for debugging purposes. The CPU has uncached access to the EFB and can obtain pixel information in ARGB (8:8:8:8) format. This means that once the EFB is mapped to the CPU address, you can reference the data only as a sequence of ARGB (8:8:8:8). Here's some pseudocode for `GXPeekARGB` and `GXPokeARGB` to illustrate the process:

```
#define EFB_ADDRESS 0x08000000

void GXPeekARGB(u16 x, u16 y, u32* color)
 {
     u32 addr;

     addr = (u32) OSPhysicalToUncached(EFB_ADDRESS) + offset(x, y);

     *color = (*(u32 *)addr);
 }

void GXPokeARGB(u16 x, u16 y, u32 color)
 {
     u32 addr;

     addr = (u32) OSPhysicalToUncached(EFB_ADDRESS) + offset(x, y);

     *(u32*)addr = color;
 }

where offset can be obtained from:

offset = ((y)<<12) + ((x)<<2);
```

As you can see, `GXPeekARGB` and `GXPokeARGB` are just copying the 4 bytes of data. Notice that there is room for optimization if you want to implement a multi-chunk copy version of `GXPeekARGB` and `GXPokeARGB`. Instead of calculating the uncached base address of `EFB_ADDRESS` each time, you can reference the pixel by the offset from the uncached base address of `EFB_ADDRESS`. Doing so will avoid duplicate calculation of the uncached base address of `EFB_ADDRESS`; however, it will not improve the performance to any great degree. The table below shows the results that I got from my test program.

If you are looking for a faster copy function, `GXCopyTex()` is the one you should use. This function will copy the data from EFB to main memory and take care of the data conversion by the hardware at the same time. This is a hardware copy that Flipper can execute directly. You can also select the destination format from the texture formats supported by the CGN. We'll talk about the performance and format issues of this function in the next newsletter.

Function	*Drip System (40/400)*	*Orca System (150/375)*
`GXPeekARGB`	34.6 MBPS	28.2 MBPS
`GXPokeARGB`	132.3 MBPS	121.5 MBPS

NINTENDO GAMECUBE™ Development News

January 23, 2001
Volume 1, Issue 2

©2001 Nintendo. All rights reserved.

In This Issue:

Fast CPU skinning 1
SDSG news server 5

SDK Version Corner

The latest release is dated **January 22, 2001**. We expect to begin sending out this release later this week.



Fast CPU skinning

By Tian Lim, NTD

Several developers have asked us how they should approach skinning on the NINTENDO GAMECUBE (GCN). In researching the problem, we have implemented the simplest and most flexible scheme—CPU-based soft skinning. We have pulled almost every CPU optimization trick, and we think you'll be happy with the resulting performance. Even better, the source code for our implementation will be shipped with the next release of the SDK!

In this article, we'll outline our approach to skinning and present some early performance numbers. Here's a sneak preview of the results… We expect your next 60Hz fighting game to be able to put two characters—each with 30K skinned vertices (and normals) and 60K faces—on the screen, and still have 50K+ faces and 50% CPU left over for background, effects, and AI.

What is CPU soft skinning?

Skinning involves taking model data (vertices and normals) in the start pose, animating the skeleton that influences the mesh, and transforming all the vertices and normals for the new pose. Since a given vertex may have multiple bones affecting it, a weighted average is taken between all of its potential new locations. The CPU performs all of the transformations. To render, it simply gives the graphics processor (GP) a model-to-world matrix.

Why CPU skinning?

There are many alternatives to CPU skinning on the GCN. For example, it is possible to load matrices on the GP and assign a matrix index per vertex. These matrices could be pre-blended to apply multiple bone influences. Although this approach is certainly feasible, it does present its own challenges, namely:

1. *Converter complexity will increase.* Not only do you have to create strips/fans based on UVs and material, you now have to consider the boundaries between bone influences.

2. *Matrix loads will stall the GP's transform unit briefly.*

3. *Finely-skinned characters will have many meshes with a wide variation in weight distribution.* The converter will have to perform some quantization on the weights to reduce the number of matrices. This may change how models appear on the GCN.

We feel that using the CPU allows you to neatly separate the skinning problem from the traditional conversion problems. Ultimately, we expect games to use a combination of CPU and GP skinning.

Our approach

We use vertex-normal (VN) pairs throughout. VN pairs are pre-sorted according to the number of matrices (bones) that affect them. Three lists are generated:

(continued on page 2)

Fast CPU skinning

"We expect your next 60Hz fighting game to be able to put two characters—each with 30K skinned vertices (and normals) and 60K faces—on the screen, and still have 50K+ faces and 50% CPU left over for background, effects, and AI.."

(continued from page 1)

1. ***SK1*** – lists VN pairs with only one matrix affecting them.

2. ***SK2*** – lists VN pairs with two matrices affecting them.

3. ***SKAcc*** – lists VN pairs with any number of matrices affecting them. Also known as the "accumulation" list. Note that there may be some duplication of VN pairs between this list and the *SK2* list.

We use a single destination array to store results from all of the lists.

The *SK1* and *SK2* lists are pretty self-explanatory. *SK1* applies matrices directly to the VN pairs and stores the result in a destination array that is used by the display lists. *SK2* applies two matrices to each VN pair and computes a weighted average. Although we could use the GP to transform the *SK1* list, we chose to avoid the matrix load stalls that this approach would incur.

The *SKAcc* list is for all the leftover vertices. It works by applying a matrix to the source vertex, scaling by a weight, then accumulating the result with the data currently in the destination array.

The *SKAcc* list uses a list of destination indices because vertices may be transformed multiple times by different matrices. To avoid random access on the source vertex data, we duplicate them.

It is important to note that the *SK2* list may have transformed the destination VN pairs of the *SKAcc* list once already. For example, all the VN pairs affected by three and four matrices may have two of the matrices applied by the *SK2* list. The rest of the weights will be applied by the *SKAcc* list.

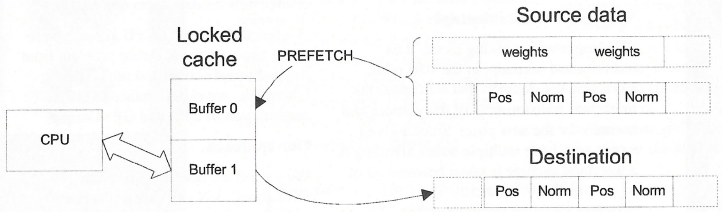
Optimizations

We implemented all inner loop code in paired-single assembly. The code for the *SK1* list is based on the `PSMTXROMultVecArray` code, which is already highly optimized. We made modifications to ensure that the normals were not translated. The code for *SK2* and *SKAcc* received only a few iterations of code scheduling.

NOTE: We do not have an SK3 list because there are not enough registers to hold three matrices and a VN pair.

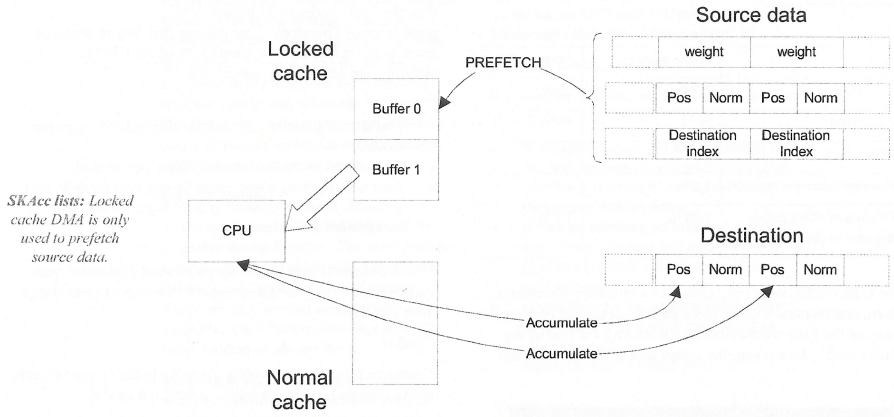
For the *SK1* and *SK2* lists, we use the locked cache DMA to bring in source VN data and weights. Transformations occur in-place and the new data is DMA-ed directly into the destination array.

(continued on page 3)



SK1/SK2 lists: *CPU operates on one buffer while the locked cache DMA prefetches the next data.*

Fast CPU skinning



SKAcc lists: *Locked cache DMA is only used to prefetch source data.*

(continued from page 2)

For the *SKAcc* list, we cannot use the locked cache DMA for destination data, since it is indexed randomly (although it is conceivable to sort destination indices by DMA buffer).

Instead, we DMA the source VN data, the weights, and the destination indices, but rely on the normal cache for destination data. The cost for this is not bad, as the length of the destination data for the *SKAcc* list is relatively short. Moreover, since the destination data had to be initialized to zero anyway, we could use the PowerPC `dcbz` instruction to allocate cache lines and zero them at the same time, without faulting in the old data.

Tricky spots

Because the locked cache DMA is 32-byte aligned, we had to insert dummy vertices in the source and destination data between DMA boundaries.

Getting data out of the cache (so that the GP can see it) is also a problem for the *SKAcc* list. Theoretically, all data that was touched should be flushed out. The inner loop code currently embeds cache store instructions, but only for one of the cache lines, and vertex data may cross cache line boundaries. To handle this, we currently do a `DCStoreRange` at the end of the loop. This is faster than eating up multiple load-store slots in the inner loop with two cache store instructions.

"...using the CPU allows you to neatly separate the skinning problem from the traditional conversion problems..."

For the vertices that are partially transformed by the *SK2* list, we created a list of indices that needed to be flushed out. This list is the minimal set (i.e., one per cache line) required to ensure that all data is flushed.

Early performance numbers

For early development and measurement of our code, we used data from a title currently in development (thanks, guys!). We integrated the inner loops into the Character Pipeline runtime, and we used a model of Link built by one of our artists.

The table on the next page shows the data for each model, with the amount of time required to skin each one (measurements do not include time to animate the bones).

(continued on page 4)

Fast CPU skinning

(continued from page 3)

	Game model	*Link model*
Vertex count	2949	2550
Face count	5810	4660
Bone count	24	103
Average bones per vertex	1.89	1.7
Fraction of 60Hz frame required to skin	1.98 %	1.8 %

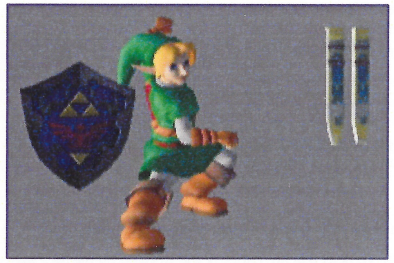
The Link model, with 103 bones and a relatively low vertex count, represents a slightly more pathological case because for many of the lists, we may not be amortizing the cost of the matrix load. As a result, the vertex rate of the game model (4.47 million VN pairs/second using 50% of CPU) is higher than the Link model (4.25 million VN pairs/second).Therefore, if we assume that you're writing a game at 60Hz that uses (at most) 50% of the CPU for skinning, we realistically expect that:

- You should achieve 4.25 million skinned VN pairs per second.
- That means 8+ million skinned faces per second.
- Your next fighting game could feature two models, each with 60K faces, with another 50K faces for the background and effects.

This is conservatively assuming you do not find some neat tricks specific for your game engine. We expect great things from you now!

Credits

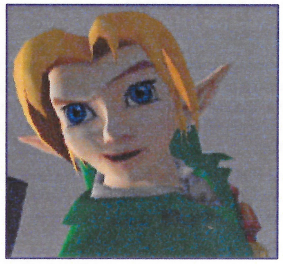
Thanks to John Cho for adding skinning to the converter path, and to George Henion for building the Link model.



Our Link model uses bones to animate many things, such as his hat, tunic...



...And his face!





SDSG news server launched

By Dante Treglia II, SDSG

We are very excited to present a news server hosting an online interactive support forum. This is the newest feature in Nintendo's Developer Support Plan. With this service you will have access to everyone in the Software Development Support Group and Nintendo Technology Development, and to developers around the globe to answer questions and exchange information on current topics.

How to connect

There are several news (NNTP) readers available on the Internet. The most widely used are those provided with Microsoft Outlook and Netscape Communicator. There are also several alternatives; for example, www.tucows.com has over 20 news readers to choose from.

Please note that we do not recommend using Netscape 6. We have had several problems using its bundled newsreader. Please use Netscape Communicator 4.76 instead.

NOTE: If you use Microsoft Outlook, please be aware that "Delete news messages 5 days after being downloaded" is *enabled by default*. You may want to disable this feature. To do so, go to "Tools > Options..." on the Menu and select the "Maintenance" tab.

Account administration

For security reasons, we require accounts for each user.

- If you have a user ID and password for our web servers (www.warioworld.com and www.noa-engineering.com), then please use that account.
- If you do not have an account, go to http://www.warioworld.com/public/devapps/registerform.html to set up a new user account.
- If you forget your account information, or would like to update your account, please e-mail register@noa.nintendo.com.

NNTP Address: news.sdsg.nintendo.com
NNTP Port: 119

For more information, check out http://news.sdsg.nintendo.com!

Newsgroup	*Topic*
gamecube	General NINTENDO GAMECUBE
gamecube.announce	NINTENDO GAMECUBE news
gamecube.audio	Audio DSP, AX, etc.
gamecube.audio.musyx	MusyX audio tool (monitored by Factor5)
gamecube.cpu	CPU, enhancements, assembly, etc.
gamecube.emulator	PC and MAC emulators
gamecube.graphics	General graphics, chip features, GX library, etc.
gamecube.graphics.advanced	Advanced techniques
gamecube.os	OS, VI, etc.
gamecube.peripherals	Controllers, memory card, AGB link cable, etc.
gamecube.random	Open forum
gamecube.test	Test message postings
gamecube.tools	NINTENDO GAMECUBE development tools
gamecube.tools.artpath	Art path tools (such as the Character Pipeline)
gamecube.tools.metrowerks	Metrowerks Compiler Suite (monitored by Metrowerks)

Nintendo Software Development Support Group

4820 150th Ave. NE
Redmond, WA 98052

Phone:
(425) 861-2715

Phone Menu:
1 → AGB
2 → CGB
3 → N64
4 → GCN
5 → Lot check
6 → Licensing / How to become a developer
7 → Other questions

Fax:
(425) 882-3585

E-Mail:
support@noa.com

HTTP:
www.warioworld.com,
www.noa-engineering.com

NNTP:
news.sdsg.nintendo.com



Edited and produced by Yvette Kirby Waters, NTD

NINTENDO GAMECUBE™ Development News

March 8, 2001
Volume 1, Issue 3

In This Issue:

Antialiasing 1
News Server 4
Texture Cache 4

SDK Version Corner

The latest release is dated **February 6, 2001**. We expect to send out a new release later this month.

©2001 Nintendo. All rights reserved.



Antialiasing

By Carl Mueller, NTD

This article discusses antialiasing methods available on the NINTENDO GAMECUBE (GCN). We'll go over some antialiasing theory, then focus on its implementation on the GCN. We'll also look at some of the issues that may affect your decision to use antialiasing in a game.

Antialiasing basics

Aliasing is the set of all the various artifacts or abnormalities that you can see in an image due to the fact that your sampling rate is lower than the highest frequency content in the image. Most images contain sharp edges, which essentially require an infinite sampling rate to display without any artifacts. Since display systems don't offer infinite resolution, we are stuck with aliasing. However, we can use antialiasing in order to reduce the visual impact of aliasing artifacts. Antialiasing (AA) refers to any technique which does just that.

Our primary weapon against aliasing is proper filtering. Filtering lets us reduce the frequency content of the image we are generating such that it more closely matches the output capability of the display.

Some people consider this type of antialiasing as just blurring the output image. While this isn't quite correct, it isn't completely wrong, either. Antialiasing does require that sharp edges be blurred; however, they must be blurred correctly. Doing this right always involves getting a more accurate description of the true underlying image, not a less accurate one.

Getting a more accurate description of the underlying image that you are trying to generate is typically accomplished by gathering more samples. In normal, non-antialiased rendering, you can think of the final image as if it were generated by the following steps:

1. Sampling: Cast a single ray through the center of each pixel. Find the first polygon it hits. Compute the color at that point.
2. Filtering: For the one color sample we have, expand it up to the size of the entire pixel.

With super-sampling (or, equivalently, multi-sampling), the steps are:

1. Sampling: Cast several rays through each pixel. Compute the color for each one.
2. Filtering: For all the samples in the area of the pixel, average them together in an appropriate way.

To complicate things further, we have to deal with interlaced display. Proper (deflickered) display of interlaced images requires that information from the whole frame image be displayed with every field. Thus you must always render a whole frame for every field you wish to display, and each displayed pixel contains a mixture of its own color plus that from the pixels above and below (which reside in the "opposite" field).

(continued on page 2)

Antialiasing

(continued from page 1)

NINTENDO GAMECUBE antialiasing

With the basics covered, let's look at how antialiasing is accomplished on the NINTENDO GAMECUBE. As we mentioned, AA involves two distinct steps: sampling, then filtering. These steps happen at different places in the pipe. Let's talk about sampling first.

In AA mode, the GCN computes and stores three samples per pixel. Since the total amount of frame buffer memory is fixed, AA mode has several consequences:

1. The number of bits per pixel sample is reduced by one third (i.e., it offers only 16-bit color and 16-bit Z versus 24 bits each in non-AA mode).
2. The pixel memory footprint is still twice as big as that of non-AA pixels, and thus they require two memory cycles to read/write (versus just one for non-AA mode).
3. Only half the total number of pixels can be stored in the frame buffer. Therefore, you must either render images with half the vertical resolution, or render a frame in two passes (top and bottom halves).

We will discuss each of these consequences in more detail shortly, but first let's focus on the AA process itself. With the GCN, you can specify the sample locations for each sample within a pixel quad (a group of 2x2 pixels). This gives you more flexibility vs. specifying one layout for all pixels. By varying the layout across the different pixels within the quad, you can create patterns that work better for different images. We'll return to this issue in just a bit.

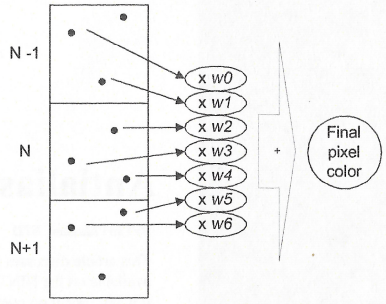
As we've noted, the second step of antialiasing is filtering the samples together. In the GCN, this phase happens when the embedded frame buffer (EFB) is copied out to produce the external frame buffer (XFB). In fact, the very same hardware that performs the deflicker filter is used to combine the various AA samples together. (See figure above right).

Choosing sample locations and filter weights

Now that the GCN sampling and filtering processes have been explained, how do we go about choosing a good sample layout and filter? For the former, we have these guidelines:

1. Since we have few samples available, each one should count for as much as possible (i.e., each one should represent an equal area). Thus the samples should be as far apart from each other as possible.
2. In order to reduce stair-stepping and jumping edges as much as possible, you'll want to lay out the samples such that a slowly moving straight line will most likely cross only one sample at a time and will cross the samples uniformly in time (see figure below).

Vertical Filter Blending
Define 3 weights from current pixel, 2 above, 2 below.



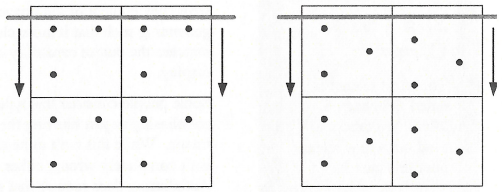
With regard to the filter, we have two different considerations to tackle: it is both an antialiasing filter and a deflicker filter. Normally, the ideal antialiasing filter kernel resembles a gaussian function. With discrete samples, you weight each sample by the volume under the curve represented by that sample. With equal-importance sampling, you weight each sample equally.

With a deflicker filter, you normally think of a tent filter with primary weight given to the current scan line and less to the adjacent ones.

(continued on page 3)

This line will cross the quad's samples in 4 increments

This line will cross the quad's samples in 12 increments



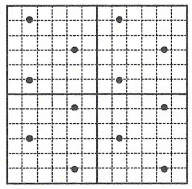
However, we still need to consider other lines (such as vertical). We also need to consider how evenly the increments are spaced.

Antialiasing

(continued from page 2)

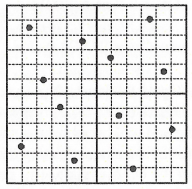
This next figure illustrates the default GX AA setup and an alternate one from the demo `frb-aa-full`:

GX default



weights: 4, 8, 12, 16, 12, 8, 4

frb-aa-full



weights: 4, 11, 11, 12, 11, 11, 4

You'll notice that the default GX AA setup doesn't seem to follow all the guidelines we've talked about. It turns out that it still works quite well for many types of scenes, and so we've left it as is. As we've said, you should choose the layout that works best for your scenery.

NINTENDO GAMECUBE AA trade-offs

Now let's return to discussing the various trade-offs of the GCN's AA mode.

Less color and Z resolution

In AA mode, the EFB only stores 16 bits each for color and Z. Dithering is an option to help make up for the loss of color resolution. To help make up for the loss of Z resolution, various floating-point Z buffer formats are available. These formats are briefly discussed in "Graphics Library (GX)" in the *NINTENDO GAMECUBE Graphics Programmer's Guide*, and we'll cover this topic in greater detail in a future article.

Pixels require twice as much memory

Since each pixel in AA mode occupies twice as much memory as a non-AA pixel, it takes twice as long to read and write each pixel to or from the frame buffer. However, the pixel-rendering rate is not halved across the board. "Graphics Library (GX)" in the *NINTENDO GAMECUBE Graphics Programmer's Guide* mentions that going beyond one TEV stage somewhat equalizes the pixel performance of AA and non-AA modes. How is this so? As you may guess, only a single color sample is actually computed per pixel. However, the color sample is Z-buffered and stored only in the affected samples (i.e., those within the polygon boundaries). This works out well in most cases, since polygon edges are one of the main causes of aliasing.

Half as many pixels are available

With AA pixels being twice as big as non-AA pixels, only half the total number of pixels will fit in the EFB. Because of this, AA mode requires you to choose one of the following solutions:

- Use double-strike video mode (halving your vertical resolution).
- Use interlaced rendering mode (eliminating the possibility of proper deflickering as well as requiring strict adherence to a 60Hz update rate).
- Use full-frame deflickered mode (requiring two rendering passes in order to draw the top and bottom halves of the screen).

It's unlikely that most developers would choose the double-strike option, since the goal of AA mode is to make the picture look better in the first place. Using interlaced rendering mode is a reasonable choice for games where:

- The images don't feature many contrasting horizontal edges.
- The viewpoint is rapidly changing.
- A 60Hz strict update rate is required already.

Note that interlaced rendering requires a different filtering setup than full-frame rendering (since deflickering is not possible in this mode). Check out the demo `frb-fld-int-aa` for an example.

Full-frame AA mode may be the better choice for games that display images which demand deflickering, or for games that don't depend upon a strict 60Hz update rate. Drawing the two halves of the screen can be done by shifting the frustum or by shifting the rendered area. The demo `frb-aa-full` shows both methods in action. The cost of rendering the scene in two halves does not have to be twice as much if you use view frustum culling (alas, this is not shown in the demo).

Changing modes

Before we leave the topic of antialiasing, we should also discuss changing modes. Entering or exiting AA mode always requires calling both `GXSetPixelFmt` and `GXSetCopyFilter`.

When changing the pixel format, you should note that any data currently within the EFB will **not** be changed. Therefore, you may need to perform another copy/clear operation after changing the pixel format. You may be able to eliminate this extra clear by always clearing to all zeroes, which requires using an inverted Z range with 0 representing the furthest Z value. ■

What's new on the news server

By Dante Treglia, SDSG

As most of you already know, we enabled SSL (Secure Sockets Layer) on news.sdsg.nintendo.com. This service encrypts all data that is exchanged between you and the server. A few users encountered some problems, but for the most part, it works well. If you run into any problems using the news server, please e-mail support@noa.com.

NOTE: Microsoft Outlook users should keep in mind that, by default, Outlook expires articles after 5 days. For more info, go to http://news.sdsg.nintendo.com/news/newsgroup.htm.

DDH drive problems with Win2K. Developers using Windows 2000 on their DDH host PCs were experiencing problems where the emulation drive partition would "disappear." Some cases even exhibited emulation drive data corruption. Eugene Kwon (NTD) posted a fix to the vanishing drive problem. See gamecube.tools -- *NEW TOOL: AMC Dev Kit and Windows 2000 *** URGENT ***.*

How to fry your DDH. Takayuki Hashida (NTD) discovered that it is possible to destroy your dev kit by hot swapping EXI devices (such as the USB2EXI) and controllers. Please beware! See gamecube.peripherals -- *IMPORTANT: please turn off when you disconnect/connect EXI devices.*

GX State in Display List. Several developers have asked about putting GX state commands in display lists. Carl Mueller (NTD) has been working on this information. See gamecube.graphics.advanced -- *Information on putting texture commands into display lists & Putting GX state functions into Display Lists*.

Latest Software. See gamecube.announce for the latest software updates. Currently:

- SDK 1.0 (Feb 6, 2001).
- MusyX SDK v1.5 (Feb 6, 2001).
- gameOptix DDK 1.0 (Jun, 2000).

gameOptix DDK v2.1? In various articles, we've mentioned the next release of the DDH firmware. The software is currently in testing and will be released soon. It is crucial that we thoroughly test the DDK, since updating the firmware incorrectly could render the DDK useless. This release promises better-than-real DVD emulation speed, as well as more stability.

Screen Captures. Interested in capturing screen shots? See gamecube.tools -- *Grabbing Screen Shots from GameCube.*

Metrowerks IDE Global Prefix Files. The Metrowerks IDE requires a prefix file to declare global defines for a project. Several developers have had problems updated to new SDK versions because of inconsistent prefix files. Steve Wells from Metrowerks has posted a few articles about prefix files and where to get them. See gamecube.tools.metrowerks. ■

Understanding the texture cache

By Carl Mueller, NTD

The texture cache on the NINTENDO GAMECUBE (GCN) is perhaps unlike the texture cache of any previous graphics system. This is because it is very flexible and it incorporates many of the different kinds of functionality found in previous systems. The main purpose of the texture cache is to provide the bandwidth for looking up 8 texels per pixel (for 4 pixels) in just one clock cycle. Normally in other systems, this is achieved with a very small on-chip texel cache whose existence is typically unseen by the programmer. However, the GCN texture cache is very large, which offers several possibilities.

Caching textures

In general, we expect the programmer to use the texture cache simply as a large on-chip cache, similar to the on-chip L2 caches in modern CPUs. With this usage, textures are always stored only in main memory, and the texturing system automatically loads the necessary texels into the cache to be ready by the time that they are needed. The benefit of having a large cache is that main memory bandwidth is significantly reduced, leaving more bandwidth available for other tasks. Given the high performance and relatively low latency of the GCN's main memory system, this method of using the texture cache results in very little degradation compared to loading the entire texture into the cache beforehand. Cached textures may even operate faster than preloaded textures when part of the texture is not visible, since only visible texel tiles are ever loaded into the cache.

Preloading textures

Preloading textures is the alternative use of the cache, also made possible by its large size. By preloading a texture into the cache, the entire texture is immediately available for random access. The cost of this method is that it requires the programmer to manually load the texture into the cache prior to using it. With normal cached usage, cache loading is completely automatic. Texture preloading may be more desirable with frequently-used textures where texel access is more random in nature. Random lookups

(continued on page 5)

Understanding the texture cache

(continued from page 4)

tend to be more prevalent when a texture is addressed by an indirect lookup, or when a texture is addressed by a transformed surface normal (i.e., when using environment maps).

Storing lookup tables

The texture cache may also be used to store color lookup tables (CLUTs or TLUTs) for color-indexed textures. With color-indexed textures, the texture itself is stored in the low half of texture cache, while the TLUT is stored in the high half. Because only half of the bandwidth is available to look up the texture itself, only bilinear filtering can occur in one cycle (i.e., you can't do trilinear filtering with color-indexed textures). Even so, this still means that 16 TLUT entries (4 texels for 4 pixels) must be looked up in the next cycle. To provide this bandwidth from the TLUT, 16 copies of the TLUT are stored in the texture cache. This replication takes place as TLUTs are loaded and is the reason why the maximum TLUT size is 2^{14} entries. Such a TLUT occupies the entire upper half of the texture cache. (Just as an aside, you might note that a large TLUT can be used to do another type of indirect texturing. With 2^{14} entries, you can think of the TLUT as a 128 x 128 texture map.)

Partitioning texture memory

On the NINTENDO GAMECUBE, the texture cache can be partitioned into multiple regions, with each region operating independently. When used as an automatic cache, a region can be 32KB, 128KB, or 512KB in size. With different textures using different cache regions, cache conflicts during multitexturing are eliminated. When using a region to preload textures, the region need only be as big as the texture. (One restriction is that regions can only begin at 2KB boundaries.)

Allocating LODs

In order to do trilinear filtering, the odd and even LODs of a texture must be assigned to (or loaded into) opposite halves of texture memory. Together with the various types of texture requirements, this results in the parameters in the table below for how textures must be allocated in the texture cache. (This table is adapted from the one found in the `GXInitTexCacheRegion` man page in the Dolphin Reference Manual (HTML).)

When allocating the LODs, you should keep in mind that the even LODs are four times larger than the next smaller odd LODs. ■

	Allocation rules
Planar, non-32b-RGB, non-CI	Uses only one bank (a bank is half of texture cache). May be in either bank.
Planar, 32b-RGB, non-CI	AR color components must be in low bank. GB color components must be in high bank.
Planar, color-indexed	Must be in low bank (The corresponding TLUT will be accessed from the high bank.)
Mipmapped, non-32b-RGB, non-CI	Even LODs must be in one bank (either one). Odd LODs must be in the opposite bank.
Mipmapped, 32b-RGB, non-CI	Even (AR) + odd (GB) LODs must be in one bank (either one). Even (GB) + odd (AR) LODs must be in the opposite bank.
Mipmapped, color-indexed	Even LODs must be in low bank. Odd LODs must be in low bank. Note that trilinear filtering is not possible for this texture type.

Nintendo Software Development Support Group

4820 150th Ave. NE
Redmond, WA 98052

Phone:
(425) 861-2715

Phone Menu:
1 → AGB
2 → CGB
3 → N64
4 → GCN
5 → Lot check
6 → Licensing / How to become a developer
7 → Other questions

Fax:
(425) 882-3585

E-Mail:
support@noa.com

HTTP:
www.warioworld.com,
www.noa-engineering.com

NNTP:
news.sdsg.nintendo.com



Yvette Kirby Waters, NTD

NINTENDO GAMECUBE™

DEVELOPMENT NEWS

July 31, 2001
Volume 1, Issue 4



©2001 Nintendo. All rights reserved.

In This Issue:

Development tools 1
Final specs 10

SDK Version

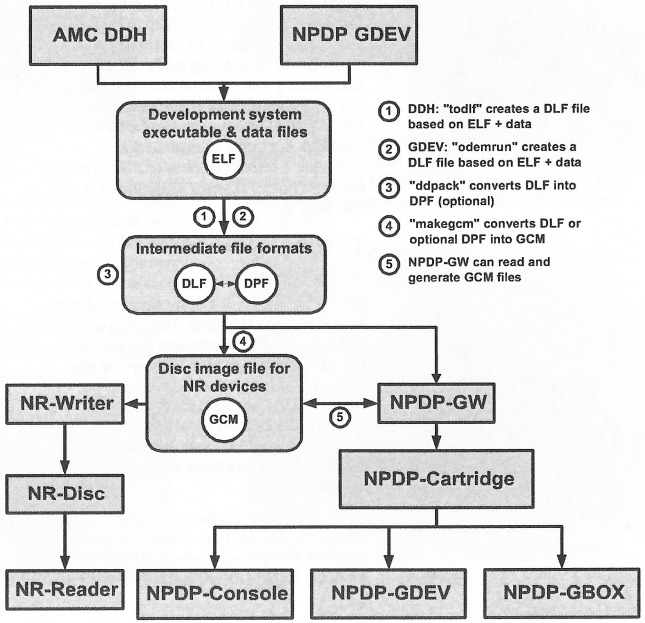
The current release is dated **May 22, 2001**. Look for **Patch 5** at www.warioworld.com.

GCN hardware lineup

By Greg McBride & Dante Treglia v.II.1, SDSG

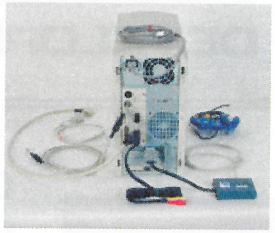
As more game development tools come online, all developers should be aware of the entire range of development solutions available for NINTENDO GAMECUBE.

In this issue, we present a snapshot of the current state of NINTENDO GAMECUBE development tools, including descriptions of all the various equipment and how each contributes to the NINTENDO GAMECUBE development process. Our goal is to help you become better informed about all of the NINTENDO GAMECUBE development hardware options available and how they can fit your game development needs.





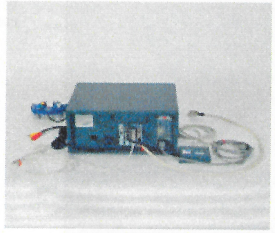
DDH front



DDH back



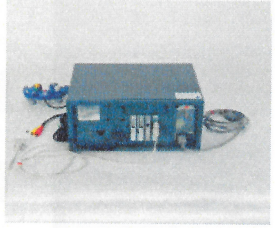
NPDP-GDEV front



NPDP-GDEV back



NPDP-GBOX front



NPDP-GBOX back

Development solutions

DDH

AMC's Dolphin Development Hardware (DDH) is a NINTENDO GAMECUBE development and optical disc emulation system. The internal DDH controller board ("Marlin") and an internal hard disk emulate the NINTENDO GAMECUBE optical disc. The emulation parameters are determined and configured by the firmware installed on the Marlin. The DDH provides a real-time debugging channel via an Ethernet connection supported by both Metrowerks CodeWarrior and SN Systems ProDG debuggers. As with all development systems, the maturity of the DDH is determined by the ORCA board (see page 5) and the disc emulation software.

NPDP-GDEV

The NPDP-GDEV is another NINTENDO GAMECUBE development and optical disc emulation system. The NPDP-GDEV uses the hard disk and CPU of the host PC to perform NINTENDO GAMECUBE optical disc emulation. The SCSI host connection channels real-time debug information supported by both Metrowerks CodeWarrior and SN Systems ProDG debuggers. The NPDP-GDEV also provides an NPDP-Cartridge slot for an alternative disc emulation method. The heart of this system is the NINTENDO GAMECUBE ORCA board (see page 5).

Testing solutions

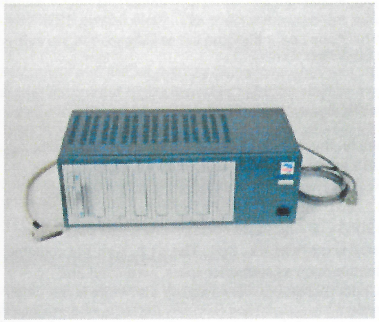
NPDP-GBOX

The NPDP-GBOX is a cost-effective solution for testing and debugging. This stand-alone unit accepts NPDP-Cartridges that emulate the NINTENDO GAMECUBE optical disc. Up to four disc images can be stored on a single NPDP-Cartridge (see page 6). The system provides front panel controls for emulating game disc cover opening and closing, and for emulating game disc swapping.

The NPDP-GBOX is the only testing solution that offers two real-time communication channels for optional connection to a PC. First, the NPDP-GBOX can be connected to a host PC using the built-in USB2EXI device. With this connection, your development team can fine-tune your game by exchanging textures, sounds and other data. Second, a built-in serial cable provides debug output via the OS libraries.

The other major benefit of the NPDP-GBOX is that it has 48 MB of main memory. This allows your testing team to utilize any additional functionality that your developers have implemented.

The NPDP-GBOX cannot write to an NPDP-Cartridge. Game disc images must be written to NPDP-Cartridges using the NPDP-GW cartridge writer.

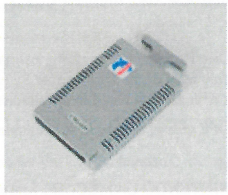


NPDP-GW cartridge writer

NPDP-GW

The NPDP-GW is a high-speed gang writer capable of writing game disc images to as many as eight NPDP-Cartridges at once. By daisy-chaining up to four NPDP-GW systems, you can write to a maximum of 32 NPDP-Cartridges simultaneously. Burning full disc images onto NPDP-Cartridges takes about five minutes.

The NPDP-Writer GUI-based software mounts and writes data to the NPDP-Cartridges. It sets all available NPDP-Cartridge settings, such as the default boot disc image, and even updates firmware. Using this software, you can compare the disc image on an NPDP-Cartridge with the original image on your PC. There is also a GUI for generating a master binary disc image (GCM; see page 4) for NR-Readers and game submission.



NPDP-Cartridge

NPDP-Cartridge

The NPDP-Cartridge is a removable, rewritable NINTENDO GAMECUBE optical disc emulation cartridge. It is the main medium of the NPDP series of products. Each cartridge stores up to four 1.4GB optical disc images. (The actual size of the NINTENDO GAMECUBE Game Disc is 1,459,978,240 bytes, or 1.46 billion bytes.)

The NPDP-Cartridge consists of a high speed drive and controller. The controller firmware manages all the emulation parameters and can be modified via the NPDP-GW.



NPDP-Console

NPDP-Console

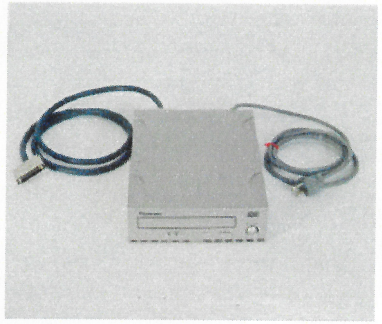
The NPDP-Console is essentially a production NINTENDO GAMECUBE, except that its optical disc drive mechanism has been replaced with a NPDP-Cartridge interface. Front panel controls emulate game disc cover opening and closing and disc swapping.



NR-Reader

NR-Reader

The NR-Reader provides your testing team with a hardware environment that is as close to production as possible without requiring the creation of final discs. The NR-Reader is identical to a production NINTENDO GAMECUBE console, except that the NR-Reader's optical disc drive mechanism accepts only proprietary NR-Discs written using the NR-Writer. NR-Discs are not the same as production NINTENDO GAMECUBE optical discs. NR-Discs will not work in production hardware, and production discs will not work in NR-Readers. (An NR-Disc is shown in the photo of the NR-Reader on page 3.)



NR-Writer

NR-Writer

The NR-Writer is an optical disc writer that creates NR-Discs. Although it looks like a standard DVD-R with SCSI interface, it's much cooler! The NR-Writer is specifically designed to produce 8cm NR-Discs. The NR-Writer software only accepts GCM files (see below). The NR-Writer can burn a GCM disc image to an NR-Disc in about 20 minutes. You can daisy-chain up to four NR-Writers on a single SCSI card, allowing multiple disc images to be burned simultaneously.

Testing path

Game development produces an ELF file (the executable) and a collection of game data files. This collection of files is called a game image, which is represented by a DLF/dvdroot, DPF, or GCM file. There are advantages and disadvantages to each file format.

DLF

A Disc Layout File (DLF) is a text-based list of all of the files that your game requires, as well as the offset from the beginning of the disc.

Individual DLF files are small, but they must be accompanied by all the files listed in the DLF file. Multiple files need to be transferred, so it's easy to make mistakes. You may also encounter relative and absolute path issues when transferring game images.

There are two methods for generating DLFs, depending on which development system you use. If you are using a DDH, then you must use the `todlf` tool (available on our website). If you are using NPDP-GDEV, the `odemrun` utility automatically creates a DLF.

If you are using a DDH, the DLF for the GX demo `tg-shadow.elf` (which only uses one data file) would look like this:

```
v1.00
0x00000000,"boot.bin"
0x00000440,"bi2.bin"
0x00002440,"appldr.bin"
0x00020440,"fst.bin"
0x00060480,"dvdroot\gxTextrs.tpl"
0x000b8e40,"dvdroot\default.dol"
0x000dfe40,""
```

In this example, the `default.dol` file is the NINTENDO GAMECUBE target executable converted from the ELF file. (For a complete layout of the executable, please refer to the `eppc.HW2.lcf` file in `DolphinSDK1.0/include/dolphin/`.) The only demo data file listed is `gxTextrs.tpl`. The `bi2.bin` file contains state information such as controller specs, command line arguments, boot modes, etc. The `appldr.bin` file is the application loader binary. The `fst.bin` file is the File System Table that contains the directory structure used by the DVD library. Finally, the empty quotes are a file marker providing a sanity check for file sizes.

If you are using a NPDP-GDEV, then the DLF for `tg-shadow.elf` would look like this:

```
v1.00
0x00000000,"tg-shadow.dsf"
0x00040700,"c:\DolphinSDK1.0\dvddata\gxTextrs.tpl"
0x040549e0,""
```

In this example, all of the system and executable files are in the `tg-shadow.dsf` file.

DPF

The Disc Package File, or DPF, compensates for the disadvantages of the DLF format by packaging the DLF file, along with all files listed in it, into one file.

The DPF utility, `ddpack`, is available at www.warioworld.com.

GCM

The NINTENDO GAMECUBE Master (GCM) file format is a exact binary image of your final NINTENDO GAMECUBE disc image. The precision of this format makes it ideal (and required) for final submissions. The obvious drawback is the large image size of 1.4 GB.

GCM files can be produced from DLF files by using the NPDP-GW software or the `makegcm.exe` utility. Both are available at www.warioworld.com.

Region encoding

Currently, console game production and development systems support two regions, Japan and North America. Each NINTENDO GAMECUBE production console will be hard coded for a specific region, as will the NINTENDO GAMECUBE optical discs. The production consoles will reject discs that are not encoded to the proper region.

Development and testing systems provide mechanisms for changing the current region encoding. The development systems are mostly indifferent to region encoded disk images; however, the NINTENDO GAMECUBE memory cards are region-sensitive.

The `setcountrycode` script sets the country code of the DDH or NPDP-GDEV. This script is included in the NINTENDO GAMECUBE SDK.

Final submission

Nintendo will only accept the GCM file format for final submission of your games. We will provide more details on this process shortly.

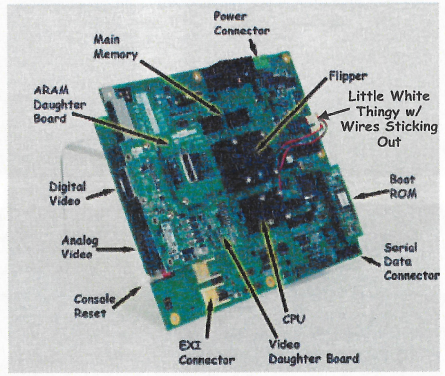
Internals

ORCA

The ORCA board is the heart of every development solution. It contains the CPU, the graphics processor, and the main memory. Although there are different versions of the ORCA board, they are all designed to work in the DDH, NPDP-GDEV, and NPDP-GBOX.

D1	**HW1_Drip (4MB RAM prototype)**
D2	**375/150 clocked HW1**
D3	**375/150 clocked HW2**
D3.4, D4.4	**486/162 clocked HW2**
D5	**Final spec**

Here are the final specifications for the ORCA boards:



The ORCA board

CPU clock speed	**486**
GP clock speed	**162**
Audio DSP clock speed	**81**
Main SRAM	**48MB**
Boot ROM	**0.93A**

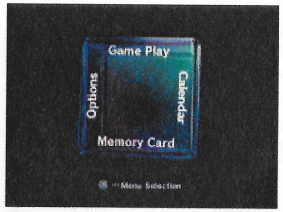
ORCA boards are supplied with the following daughter cards:

- **Video card.** The final video card has the component video connector and a final-spec 9-bit video DAC.
- **EXI boards.** These provide a hardware interface for EXI devices.
- **Boot-ROM.** This contains the Initial Program Loader software.
- **Serial connector.** This provides debug output directly from the ORCA board, and accounts for one of the communication channels of every ORCA-equipped system.

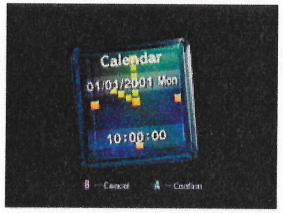
Boot-ROM

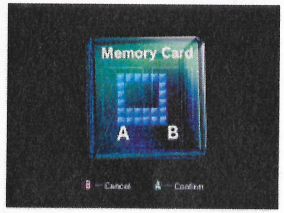
The boot-ROM determines the boot sequence for any NINTENDO GAMECUBE system. Early versions of the boot-ROM chip on the ORCA board simply boot the application. The 0.93A boot-ROM, however, can operate in two modes: *production* and *development*. The mode is determined by a small amount of non-volatile RAM (NVRAM) located on the ORCA board. The 0.93A boot-ROM reads a bit in the NVRAM to determine the boot mode. This mode can be set using the `bootmode.elf` program (see page 7).

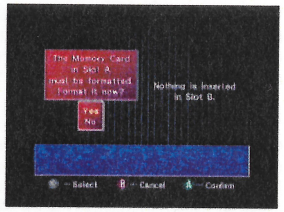


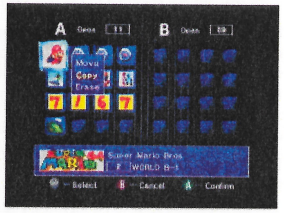


0.93A bootROM Initial Program Loader screens (production mode)



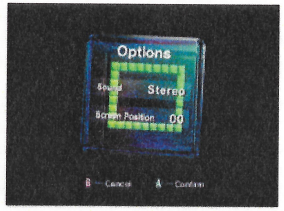








NINTENDO GAMECUBE motherboard (shown actual size)





Development mode is the default and is meant to be used during development. It enables debugging functionality and also takes less time to boot than the production mode.

Production mode is for your reference only. Note that debug functionality and output are disabled in this mode. When a system is booted or an application run, you will see the official NINTENDO GAMECUBE boot sequence and be able to access the system menu. Here you can browse files on your memory cards, set the date and time, or set stereo settings.

The first time a boot sequence is run, you will be asked to set the time and date. To get to the main selection screen on subsequent boots, hold down the A button during the boot sequence.

All settings provided in the Initial Program Loader (IPL) menu are stored in NVRAM. Some non-final ORCA boards do not have the proper circuitry (and battery) to retain these settings after a power-down.

You can determine whether your ORCA board has the 0.93A boot-ROM by observing the debug output. In development mode, the boot-ROM version prints when applications are loaded. The 0.93A boot-ROM prints "`--- DEVKIT BOOTROM v0.93a (DEVELOPMENT MODE) ---`".

In order to switch the boot mode, you must run the `bootmode.elf` program.

```
% loadrun bootmode.elf -a 1  # EAD IPL mode
// with file browser, etc.

% loadrun bootmode.elf -a 0  # NTD IPL mode
// much faster, default
```

This program determines the boot mode for the next boot sequence. Because the setting is stored in NVRAM, all subsequent boot sequences will be affected. If the NVRAM boot setting is set to 1, then please make sure you see a black screen for a few seconds before running another application.

Accessories

Component cable

The NINTENDO GAMECUBE supports one digital output format, 480 Progressive Scan video output. Standard interlaced NTSC televisions draw two 240-line frames every 60th of a second, whereas 480 Progressive Scan televisions can draw all 480 lines every 60th of a second. This produces a much smoother and less flickered image.



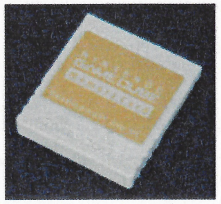
Component cable

In order to output this video mode, you must have the component cable shown. Please note that this is a prototype cable and that production component cables will have a plastic housing around the connector. The metal housing contains a 10-bit video DAC (as opposed to the default 9-bit DAC).

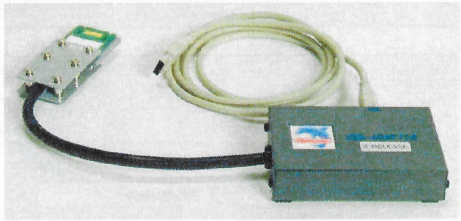
As a side note, all the games shown at E3 were running 480 Progressive Scan on Panasonic CT-32HX40 and Panasonic PT-61HX4 televisions.

Memory Cards

The NINTENDO GAMECUBE Memory Card is a 4-Mbit flash ROM storage device that fits into either of the two EXI ports located on the front of the system. The ports are located on the back of the NPDP-GBOX, NPDP-GDEV, and DDH.



GCN Memory Card



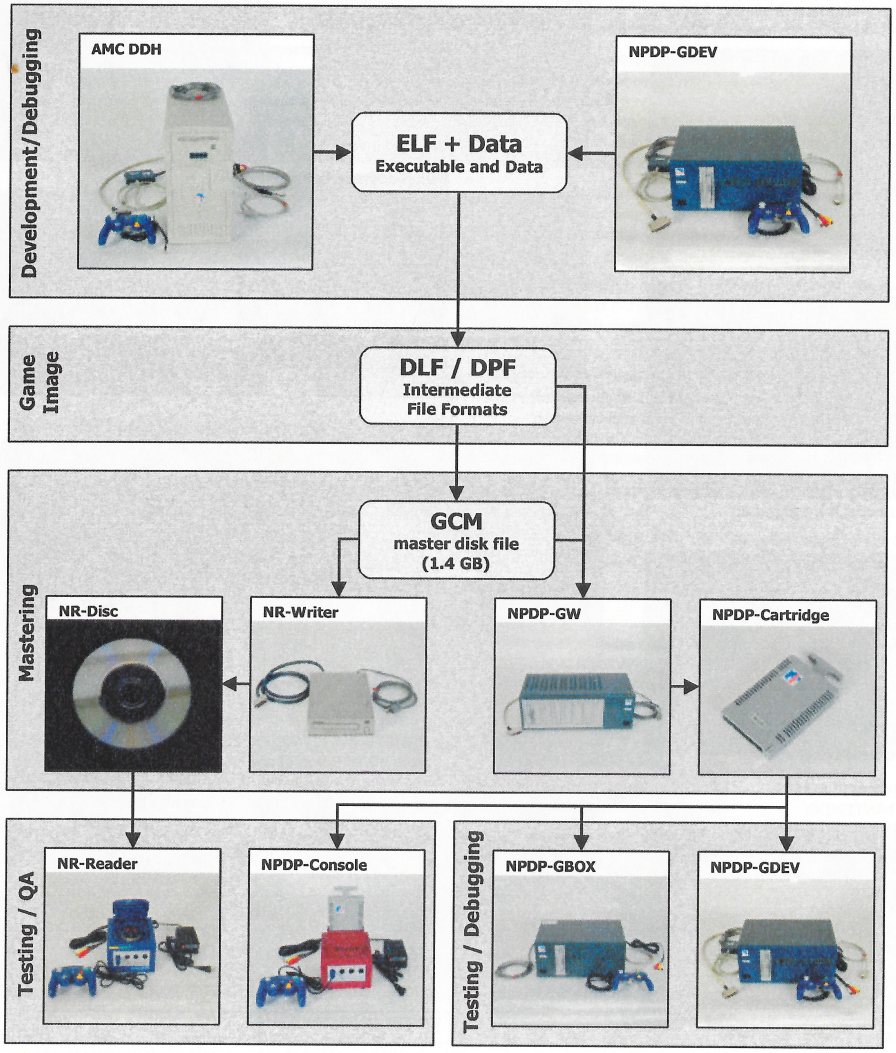
USB2EXI

USB2EXI

The USB2EXI device can be used as a communication link between NPDP-GDEVs, NPDP-GBOXs, DDHs, and a host PC. This device connects from an EXI port on the development system to the USB port of any USB-capable PC. Many developers have incorporated this device as a critical component in their game development process. Look for a screen shot utility using the USB2EXI device on our website.

Development Tool	*Availability*
NPDP-GDEV, DDH, NPDP-GBOX	Allocated until Sept. 2001
NPDP-GW, NPDP-Cartridge	Available
NPDP-Console	Q4, 2001
NR-Writer, NR-Reader	Q4, 2001

Summary: GCN development hardware tools



Hardware features at a glance

ORCA Board
• Main system board forms basis for AMC DDH, NPDP-GDEV and NPDP-GBOX.
AMC DDH
• 48MB of main memory. • Complete optical disc emulation. • Real-time debugging. • Serial interface outputs debug information.
NPDP-GDEV
• 48MB of main memory. • Complete optical disc emulation. • Real-time debugging. • Serial interface outputs debug information. • NPDP-Cartridge interface.
NPDP-GBOX
• 48 MB of main memory allows same debugging features as development team uses. • Stand-alone operation. • USB port allows optional communication between the GBOX and Host PC applications. • Requires NPDP-Cartridge for game disc image.
NPDP-GW
• Writes disc images in about 5 minutes. • Write to 8 cartridges at once. • Daisy-chain up to 4 devices to allow writing of up to 32 cartridges at once. • Create master binary disc image files for NR-Reader and for game submission. • Verify your cartridge data by comparing it to original disc image file on PC. • Control NPDP-Cartridge settings.
NPDP-Cartridge
• Stores up to 4 disc images of 1.4GB each. • Images are written using NPDP-GW. • Used with NPDP-GBOX, NPDP-GDEV, NPDP-Console, and NPDP-GW.
NPDP-Console
• Uses NPDP-Cartridge. • Front panel interface for simulating disc tray functions. • 24MB of main memory. • Production motherboard and interfaces.
NR-Writer
• Creates NR-Discs for use in NR-Reader. • Writes full disc in about 20 minutes. • Daisy-chain up to 4 NR-Writers to the same SCSI card for simultaneous writing of up to 4 discs.
NR-Disc
• Proprietary optical disc format used in NR-Writer and NR-Reader. • 1.4GB disc capacity. • 8cm diameter disc.
NR-Reader
• Run game images on NR-Discs, burned optical media. • 24 MB of main memory, the same as production NINTENDO GAMECUBE. • Same hardware as a production NINTENDO GAMECUBE, except for proprietary NR-Disc interface.

Nintendo
Software
Development
Support Group

4820 150th Ave. NE
Redmond, WA 98052

Phone:
(425) 861-2715

Phone Menu:
1 → AGB
2 → CGB
3 → N64
4 → GCN
5 → Lot check
6 → Licensing / How to become a developer
7 → Other questions

Fax:
(425) 882-3585

E-Mail:
support@noa.com

HTTP:
www.warioworld.com,
www.noa-engineering.com

NNTP:
news.sdsg.nintendo.com



Edited and produced by
Yvette Kirby Waters, NTD

NINTENDO GAMECUBE final specs

Microprocessor Unit (MPU)	Custom IBM Power PC "Gekko"
Manufacturing Process	0.18 micron IBM Copper Wire Technology
Clock Frequency	486 MHz
CPU Capacity	1125 Dmips (Dhrystone 2.1)
Internal Data Precision	32-bit integer & 64-bit floating-point
External Bus	1.3GB/second peak bandwidth (32-bit address space, 64-bit data bus, 162 MHz clock)
Internal Cache	L1: instruction 32KB, data 32KB (8-way); L2: 256KB (2-way)
System LSI	Custom ATI/Nintendo "Flipper"
Manufacturing Process	0.18 micron NEC Embedded DRAM Process
Clock Frequency	162 MHz
Embedded Frame Buffer	Approx. 2MB sustainable latency: 6.2ns (1T-SRAM)
Embedded Texture Cache	Approx. 1MB sustainable latency: 6.2ns (1T-SRAM)
Texture Read Bandwidth	10.4GB/second (peak)
Main Memory Bandwidth	2.6GB/second (peak)
Pixel Depth	24-bit color, 24-bit Z buffer
Image Processing Functions	Fog, subpixel antialiasing, 8 hardware lights, alpha blending, virtual texture design, multitexturing, bump-mapping, environment mapping, mipmapping, bilinear filtering, trilinear filtering, anisotropic filtering, real-time hardware texture decompression (S3TC), real-time decompression of display list, hardware 3-line deflickering filter
Sound Processor	Custom Macronix 16-bit DSP
Instruction Memory	8KB RAM + 8KB ROM
Data Memory	8KB RAM + 4KB ROM
Clock Frequency	81 MHz
Performance	64 simultaneous channels, ADPCM encoding
Sampling Frequency	48KHz
System Floating-Point Arithmetic Capability	10.5 GFLOPS (peak) (MPU, geometry engine, hardware lighting total)
Real-World Polygon	6 million to 12 million polygons/second (peak), assuming actual game conditions with complex models, fully textured, fully lit, etc.)
System Memory	40MB
Main Memory	24 MB MoSys 1T-SRAM, approximately 10ns sustainable latency
Auxiliary Memory	16MB (81MHz DRAM)
Disc Drive	CAV (Constant Angular Velocity) system
Average Access Time	128ms
Data Transfer Speed	16Mbps to 25Mbps
Media	3-inch NINTENDO GAMECUBE Game Disc (based on Matsushita's optical disc technology, approx. 1.4GB capacity)
Input/Output	Controller port x4 Memory card slot x2 Analog AV output x1 Digital AV output x1 High-speed serial port x2 High-speed parallel port x1
Power Supply	AC adapter DC12V x 3.5A
Main Unit Dimensions	4.3" (H) x 5.9" (W) x 6.3" (D)

