Sega Dreamcast™

Dreamcast Roadmap and Overview

Table of Contents

Preface

This document is a roadmap to the Sega Dreamcast development system and an overview of its hardware, with special emphasis on its graphics and sound hardware and its CPU and memory architecture.

This is not a programming manual; it won't show you how to write code or how to create and organize Dreamcast programs. Its goal is far more modest; it merely introduces you to a fair number of the components of the Dreamcast development system, and tries to give you an idea of how they all work together.

Once you know what the parts of the Dreamcast development system are, and what it can do, you'll be ready start reading about Dreamcast programming. Then it will time to turn to the programming and user manuals that came with your Dreamcast development system.

1. The Sega Dreamcast Development System

The Sega Dreamcast Development System is an integrated hardware and software package for programmers who want to develop software for the Sega Dreamcast console. This chapter lists and describes the components that make up the Dreamcast development system.

1.1 Dreamcast Development Components

The Sega Dreamcast development system includes the following components:

- *The SH4 high-performance RISC engine* is a 32-bit reduced instruction set microprocessor featuring object-code upward compatibility with the SH1, SH2, and SH3 microprocessors. Software supplied with the SH4 includes a C compiler, a cross-assembler, and a linkage editor/librarian/object converter package. To learn how to use the programming software supplied with the SH4 microprocessor, see the *Dreamcast SH4 Program Manual*, the *Dreamcast SH4 Compiler User's Manual*, the *Dreamcast SH4 Cross Assembler* manual, *and the Dreamcast SH4 Linkage Editor, Librarian, Object Converter User's Manual*.
- The *Shinobi library* is a C-language library that provides four categories of system-level functions: a set of *system functions*, a set of *file-system functions*, and a set of *functions for obtaining data from peripherals*. The system functions provided in the Shinobi library initialize the Dreamcast hardware and make the Shinobi library ready for use. The file-system functions are used to open, read, and close files; to provide directory access; and to access the Dreamcast GD-ROM drive. The library's memory-management library contains functions for allocating and releasing memory, and its peripheral-interface functions are used to interface the Dreamcast console with various kinds of user-operated controls. For more details, see the *Katana Shinobi Library Specification*.
- The *Kamui graphics API* is a low-level library of graphics functions that directly manipulate the Dreamcast hardware. The Kamui library provides a wide range of low-level graphics-related functions, including functions for surface handling, setting scene parameters, setting vertex parameters, recording vertex data, and handling textures. All the functions provided in the Kamui API are listed and described in the *Dreamcast Kamui Specification*.
- *The Ninja library* is a higher-level of C-language library that is designed to provide all the basic graphics functionality needed to develop a title. The Ninja library contains matrix, collision, and mathematical functions; 2D and 3D graphics functions; light, scrolling, modeling, and view functions; sprite, special-effects, drawing, and special-effects functions; and more. To learn more about the Ninja library and how to use it, see the *Dreamcast Ninja Guide* and the *Dreamcast Ninja Library Specification*.

- *The Dragon SDK* is a software-development system built around Microsoft Windows CE, an operating system that is built into the Dreamcast console. The Dragon system is designed for developers who want to create titles using the Microsoft Visual C++ compiler rather than the Dreamcast SH4 compiler, and who want to create and manage graphics and game action using the Microsoft Direct X package. Dragon does not interface with any of the other Dreamcast programming software described in this document; it is a separate development path in its own right, and it is documented in a separate set of manuals.
- *The Dreamcast utilities package* is a set of tools for converting files generated by various kinds of graphics software packages into formats recognized by the Dreamcast development system. The Dreamcast utilities kit includes tools for converting Studio 3D Max and Lightwave-format files into the Ninja file format; a compression tool for converting PC bitmap files into compressed textures recognized by the Dreamcast graphics hardware; and more. To learn more about what files are provided in the Katana utilities package and how to use them, see the *Dreamcast Utilities* manual.
- *The Dreamcast Visual Memory System (VMS)* is a small peripheral that can be plugged into the Dreamcast controller. The VMS has an LCD screen and can function either as a small handheld game console or as an external backup memory. The VMS can also be plugged into a second VMS unit being held by another player and then used as a multiplayer game module. When the VMS is used as a data-storage cartridge, it can exchange information with other VMS cartridges. To find out more about the VMS cartridge and to learn how to program it, see the *Dreamcast (VMS) Visual Memory System* manual.
- *The Codescape debugger* is a powerful and versatile windows-based debugger. To learn how to use it, see the *Dreamcast Codescape User Guide*.

1.2 Summary

The Dreamcast development system has a number of different interlocking parts. There are two different compilers to choose from, as well as several different sets of graphics APIs and utility package. So at first glance, it might appear that the Dreamcast development system is made up of a large number of tools can be used together in many different ways. But that is not strictly true. The next chapter tells why.

2. Sega Dreamcast Roadmap

As explained in "Dreamcast Development Components" on page 1, the Dreamcast development system is an integrated hardware and software package with a number of different interlocking parts. There are two different compilers to choose from, as well as several different sets of graphics APIs and utility package. So at first glance, it might appear that the Dreamcast development system is made up of a large number of tools can be used together in many different ways.

Actually, that isn't strictly true, because the tools that make up the Dreamcast development kit can't be intermixed with each other in just any old way. Each component in the system is designed to be interfaced with some of the system's other parts, but not with others. In fact, once you understand what each tool is designed for, you'll discover that there are actually only three development pathways to choose from when you want to start creating a Dreamcast title.

Furthermore, you'll probably find that it it's quite easy to choose the path that suits you best. Generally speaking, unless your boss rules differently, the development trail you take will probably depend on your background and your preferences. Are you accustomed to programming in a console environment, or is your background mostly in Windows? Are you a member of the C++ generation, or a hardcore C programmer? Do you like the elegance and versatility of high-level graphics libraries, or do you like to think up ways to squeeze every ounce of performance out of the platform you're programming for by getting as close to the metal as possible?

2.1 Three Examples

Your answers to questions like those will help determine which Dreamcast programming path is best for you. Here are general descriptions of three fictitious developers, along with the development trail that each might choose as he or she prepares to start work on a Dreamcast title:

- *The Windows developer migrating to Dreamcast*: Justin has experience programming for Windows, is most comfortable working in C++, and has a popular library of graphics code written in C++ for the Windows environment. He also has a collection of tools for converting 3D models generated by programs such as 3D Studio Max and Maya to file formats recognized by Direct X 5. And he has been asked to leverage some of his company's extensive Windows/Direct X code base into a collection of Dreamcast titles that his boss wants to get out the door by Christmas of 1999.

  The solution: As a short-term solution, Justin should probably start his career as a Dreamcast developer by using the Dragon SDK (described in ChapterTitle 1., "The Sega Dreamcast Development System.") with Direct X 5 immediate mode and the Microsoft Visual C++ compiler. But when the great rush for Christmas delivery is over, he should consider migrating away from Dragon and toward the Ninja development platform. Why? Because Sega provides the Dragon system mainly as a stepping stone to help Windows programmers migrate to the native Dreamcast development. By moving away from Dragon and toward Ninja, Justin can make great strides in improving the performance of his titles and develop higher-quality games.

- *The object-oriented game developer*: Gramercy works for a company that has a robust library of high-performance graphics code, some written in C and some written in C++. She comes from an object-oriented background -- and, like Theodore, she has experience developing graphics programs using Visual C++. But now she wants to start developing higher-performance games than she has been able to using Direct X. She wants to learn how to optimize triangles for stripping and how to perform other kinds of low-level operations that can reduce the overhead and enhance to performance of her games.

  The solution: Gramercy sounds like she may be ready to move from the Windows/Direct X programming environment to a more powerful, lower-level environment such as Ninja (described in ChapterTitle 2., "Sega Dreamcast Roadmap."). When she has finished migrating from Direct X to Ninja, she can also start using most of the other non-Microsoft tools that the Dreamcast development system provides, such as the Shinobi system-level library and the Dreamcast utilities package.

- *The console C developer*: Theodore is an experienced game developer who has proven that he can handle the challenges of working in a console environment. He has a robust library of high-performance 3D graphics code written in C. He also has a good set of tools for converting models generated by programs such as 3D Studio Max and Lightwave into code that a graphics pipe can parse. Theodore knows how to optimize triangles for stripping when the need arises, as well as how to handle most other kinds of low-level textbook-variety graphics programming tasks. Like Gramercy, Theodore likes to get as close to the metal as possible so he can grab and make use of every spare cycle that's available -- and, unlike Gramercy, he has made his living doing just that for some time now.

  The solution: Theodore should definitely consider developing his first Dreamcast title using the Kamui graphics API (see Chapter 1, "The Sega Dreamcast Development System,"), along with the Shinobi system-level library, the Dreamcast SH4 Optimizing Compiler, and -- on days when he really wants to rock -- the Dreamcast SH4 Cross Assembler. By taking this development path, Theodore can pare his system's overhead down to its lowest theoretical limit, and can maximize the most powerful package of programming resources that's currently available to the Dreamcast developer.

3. Sega Dreamcast Hardware

At first glance, the hardware used in the Dreamcast system looks much like what you may have seen used in other high-performance 3D game consoles. But as you move in closer, you begin to see that the Dreamcast system is also equipped with a number of powerful new hardware features that clearly set it apart it from earlier 3D game consoles.

This chapter introduces and briefly describes some of the most important graphics and sound features of the Dreamcast hardware. More details on the Dreamcast graphics and audio hardware are presented in Chapter 4, ″Introducing Dreamcast Graphics,″ and Chapter 5, ″Introducing Dreamcast Audio.″

Major topics discussed in this chapter are:

3.1 The Dreamcast CPU

The Dreamcast CPU is the Hitachi SH4 microprocessor, customized for Dreamcast, with an internal processing speed of 200 MHz and a 100 MHz external bus. In the Dreamcast system, the SH4 handles processing involving the game sequence, artificial intelligence, 3D calculation, and issuing 3D graphics instructions. The SH4 also provides a general-purpose serial port with an FIFO buffer for use by external I/O devices.

The Dreamcast system is also equipped with an FPU (floating-point unit) module that can perform floating-point calculations at high speed. This module is dedicated to numeric processing that would take a great deal of time if unaided, such as the calculation of 3D coordinates.

The SH4 includes the following functions: FPU (floating-point calculation module) MMU (virtual memory management module) DMA controller Timer 8K instruction cache, 16K data cache 5-stage pipeline, two superscalar units

3.2 The Dreamcast Graphics Hardware

To handle graphics, Dreamcast uses the PowerVR module of the VL/NEC Holly graphics chip. The PowerVR processor is optimized to handle complex 3D models, and has many preprogrammed graphics-related functions onboard. The Dreamcast graphics system can render approximately 1 million polygons per second (a figure that can vary according to the sizes and other characteristics of the polygons being processed).

The Dreamcast graphics hardware not only offers the standard capabilities of any high-performance 3D graphics system, but also provides a number of special features and capabilities that distinguish it from the graphics hardware used in earlier game systems. These special features include translucent and other special kinds of polygons, a tiling system that splits the screen up into small squares for faster rendering, onboard hardware for producing fog effects, and much more.

For more about the Dreamcast graphics system, see Chapter 4, ”Introducing Dreamcast Graphics.”

3.3 The Dreamcast Sound Hardware

The sound chip used in the Dreamcast system is the Yamaha AICA, which relies on the ARM7DI -- manufactured by Advanced RISC Machines (ARM) -- to be its sound controller. The Dreamcast audio system generates sound using a 64-channel PCM/ADPCM sound source that is built into the AICA microprocessor.

The AICA chip digitizes sound using the ADPCM format, a proprietary modification by Yamaha of the standard PCM digital audio format. Instead of expressing volume at any given moment as a measured value -- as a standard PCM system does -- the ADPCM method records as volume data the difference between the volume that is expected (based on previous results) and the actual measured value. Because the degree of change in the difference from the expected value is smaller than the change in the actual measured volume, less data needs to be recorded for a waveform, and processing efficiency increases accordingly.

For a closer examination of the Dreamcast audio system, see Chapter 5, ”Introducing Dreamcast Audio.”

3.4 The GD-ROM Drive

The Dreamcast GD-ROM drive: The Dreamcast console uses a GD-ROM (Gigabyte Disk ROM), a special kind of CD-ROM that can contain both standard CD-ROM data and game data stored in a proprietary, high-density storage format.

The GD-ROM drive, unlike a conventional CD-ROM drive, rotates at a fixed speed. But it uses two read speeds: a 12X high-density speed for game data, and a 4X single-density speed for reading data that complies with the conventional CD-ROM format. To store these two kinds of data, the Dreamcast GD-ROM uses two different areas of an ordinary CD-ROM disk: a high-density band on the outside of the disk, nearest the spindle, for storing game data, and a smaller single-density band on the inside of the disk, nearest to the spindle, for storage of audio information and other data that complies with the conventional CD-ROM format. This band can hold about four minutes of audio information.

3.5 Dreamcast Peripherals

Dreamcast has four peripheral slots that permit the connection of input devices such as analog controllers and guns. The Dreamcast console also supports the use of a plug-in Dreamcast Visual Memory System (VMS), a small peripheral device with an LCD screen. The VMS can function either as a portable handheld LCD game console or as an external backup memory for the Dreamcast system.

When it is used for games, the VMS can function as a portable game player similar to the Tamagotchi console. When two Dreamcast controllers are connected, separate messages and graphics screens can be displayed on the VMS peripherals connected to each player’s controller. This capability makes it possible for multiple players to compete in a game, with each individual player receiving different information on his or her VMS screen.

3.6 Dreamcast Memory

Dreamcast has 16MB of main system memory, plus 8MB of texture memory, including memory used for drawing and display functions. There’s also 2MB of memory used for sound. All memory uses synchronous DRAM, a type of RAM that allows high-speed access. The Dreamcast Visual Memory System (VMS), described in the preceding section, can provide the system with expansion memory.

Dreamcast is equipped with a memory management unit (MMU) that assigns virtual addresses to addresses in physical memory, allowing the system to behave as if it has more memory than it actually has. The Dragon operating system (described in Chapter 1, ”The Sega Dreamcast Development System,”) uses this MMU module to conceal the physical memory space to a certain extent and make segmented memory appear to be continuous memory. The user does not need to be aware that memory is actually segmented.

Chapter 6, ”The SH4 and Memory Management,” examines the Dreamcast memory components in more detail.

3.7 Summary

This chapter introduced and described some of the most important special features of the Dreamcast hardware. More specific information about individual Dreamcast hardware components is provided in subsequent chapters.

4. Introducing Dreamcast Graphics

Although the Dreamcast graphics hardware has all the standard capabilities of any high-performance 3D graphics system, it also has a number of special features and capabilities that distinguish it from the graphics hardware used in other game systems. This section introduces the unique features of the Dreamcast graphics hardware, and provides some general information on how you can use those features and capabilities in your own graphics titles.

Keep in mind, though, that this chapter is only a brief introduction to the Dreamcast graphics system. For more detailed information on the topics introduced in this chapter, and for guidance on how to write the code that it takes to implement Dreamcast programs, please refer to the various programming and user manuals listed and described in Chapter 1, ”The Sega Dreamcast Development System.”

Major topics examined in this chapter are:

4.1 Special Features of Dreamcast Graphics

The Dreamcast graphics hardware has a number of built-in features that distinguish it from other high-performance 3D graphics systems. Here's a list that describes just a few of the most noteworthy graphics features and capabilities of the Dreamcast system:

- *Translucent and punch-through polygons*: Along with standard opaque polygons, the Dreamcast system supports the use of *translucent polygons*, which can give effects such as those of colored class, and *punch-through polygons*, which have both transparent and opaque areas. For more details, see "How Dreamcast Handles Polygons" on page 11.
- *Tile partitioning*: When the Dreamcast hardware processes graphics, it partitions the screen into 32-pixel by 32-pixel tiles, each of which can be processed separately. When Dreamcast renders a polygon, the only tiles it redraws are those in which part of the polygon appears; it does not have to redraw the entire screen. This distinctive Dreamcast feature dramatically reduces overhead and results in an enormous increase in processing speed. For more information on this topic, see the section headed "Tile Partitioning" on page 13.
- *Modifier volume*: This unique Dreamcast feature answers the age-old question: How do you render shadows in a 3D scene? It also allows you add other interesting effects, such as spotlights, to scenes in your games. The section headed "Modifier Volume" on page 16 explains modifier volume in more detail and explains how it can be used in Dreamcast titles.
- *Mipmapping*: The Dreamcast graphics hardware also has built-in support for mipmapping, a commonly used mechanism for making textures look right when they are viewed from various distances. To read more about how mipmapping works in the Dreamcast environment, see "Mipmapping" on page 17.
- *Texture blending*: Dreamcast supports three kinds of texture blending: point sampling, bi-linear filtering, and tri-linear filtering. See "Texture Filtering" on page 18 for details.
- *Hardware-controlled fog effects*: The Dreamcast graphics hardware has built-in support for creating various fog effects. Just store values in a table, and the hardware will extract them to give you just the kind of fog you need. For more information about this feature, see"Fog" on page 22.
- *Bump mapping*: Dreamcast has a special built-in feature that you can use to create *bump maps*, or simulated bumps on a surface. For more information about bump mapping, see "Bump Mapping" on page 23.

In the remaining sections of this chapter, each item in this list is examined in more detail.

4.2 How Dreamcast Handles Polygons

Dreamcast supports the use of both individual triangle polygons and individual quad polygons. In addition, Dreamcast supports the use of *triangle stripping*, a well-known technique for speeding up the calculation of vertices during polygon processing. By combining triangles into strips, you can reduce both processing overhead and the amount of polygon data that must be stored in memory. For example, in Figure 4.1, “Stripping Triangles,”, stripping has reduced the number of vertices needed to render a set of triangles by more than half: from 15 to 7.

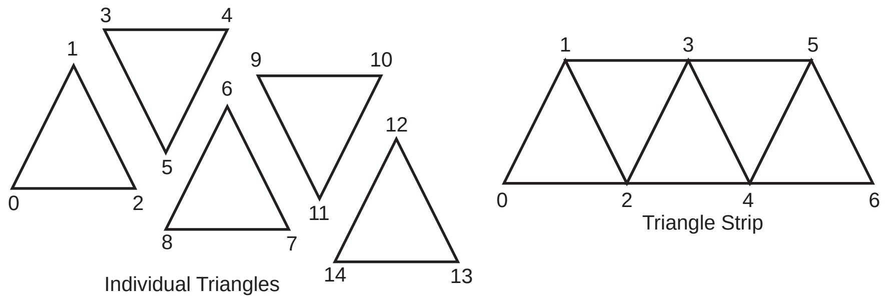


Figure 4.1 *Stripping Triangles*

Dreamcast supports the use of infinitely long strip data. But be careful: When triangles are combined into a strip, Dreamcast must calculate all the vertex data for the entire strip whenever it needs to render even a single polygon.To avoid inordinate slowdowns in processing speed, be sure to keep this requirement in mind when you start laying out strip structures.

4.2.1 Translucent and Punch-Through Polygons

In addition to ordinary opaque polygons, Dreamcast supports the use of *translucent polygons* and *punch-through polygons*. Figure 4.2, “Varieties of Polygons,” shows the characteristics of these three kinds of polygons.

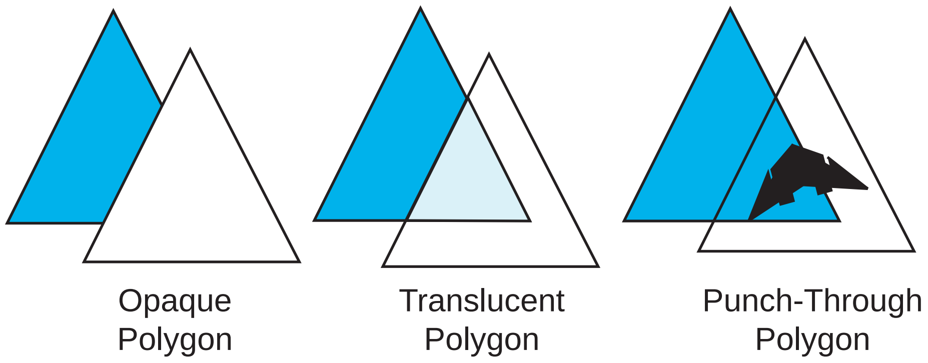


Figure 4.2 *Varieties of Polygons*

As you can see, an opaque polygon hides all objects behind it, making extremely high-speed processing possible. But translucent and punch-through polygons use special kinds of processing to allow parts of their backgrounds to show through.

A *translucent polygon* can dramatically change the appearance of objects that appear behind it, making it possible to achieve effects such as those of colored glass. But the rendering of a translucent polygon is considerably slower than that of an opaque polygon. And if multiple translucent polygons overlap, processing speed can grind down to an extremely slow pace. So for optimum drawing speed, you’ll probably want to use translucent polygons sparingly.

Another option is to use a *punch-through polygon* – that is, a polygon that is completely transparent in some places and completely opaque in others. Dreamcast can draw punch-through polygons much faster than it can process translucent polygons – in fact, punch-through polygons can be drawn almost as rapidly as opaque polygons. So feel free to used punch-through polygons for such things as sprites, trees, and other kinds of objects that look nice when they have opaque and transparent parts.

4.2.2 How to Sort Translucent Polygons

Dreamcast offers a feature that automatically sorts translucent polygons. But use it with caution. When you implement automatic translucent polygon sorting, Dreamcast draws your translucent polygons properly, even if they are layered on top of each other – but, unfortunately, the algorithm that makes this precision possible tends to be slow. If you sort your translucent polygons beforehand -- in your software -- and then pass them to the hardware for rendering, the process goes much faster, and the visual result is the same as if you had ordered an automatic sort.

4.2.3 Applying Special Effects to Vertices

The Dreamcast graphics hardware can apply various effects to vertices. For example, it can apply different colors to each vertex to change the color characteristics of polygons using Gouraud shading (color-complement shading), or to create specular effects which (highlighting achieved by increasing the color brightness of vertices that light shines on). With Dreamcast, you can specify original color information for each vertex in a polygon, as well as additional color information (expressed as color offsets). Using this capability, you can create very realistic images created by changing the light adjustment and position of models made up of polygons.

4.3 Tile Partitioning

One distinctive feature of the PowerVR processor used by the Dreamcast SH4 chip is a *Tiling Accelerator* that partitions the screen into 32-dot by 32-dot tiles for extraordinarily fast graphics processing. When Dreamcast renders a scene, it processes individual tiles rather that processing the entire screen. Thus, if only one small part of the screen changes prior to a refresh cycle, only the polygons in the region of the screen that has changed must be redrawn.

Figure 4.3, "Tile Partitioning," shows in a very simplified way how Dreamcast's tile partitioning system works. Suppose you have a scene in which only one triangle (the one shown in the picture) has moved since the last screen refresh. During the next refresh cycle, the Dreamcast hardware will redraw only the four tiles occupied by the changed triangle, and will leave the rest of the screen alone, resulting in an extraordinarily fast refresh of the whole screen.

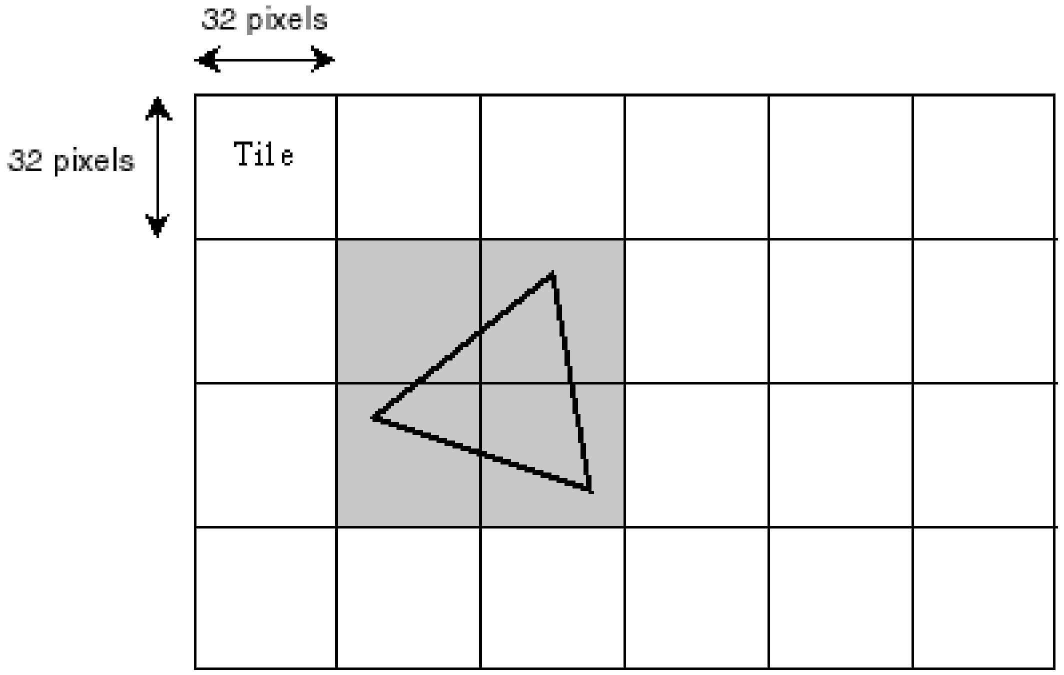


Figure 4.3 *Tile Partitioning*

What makes tile partitioning even better is the fact that the whole process is automatic. The Dreamcast graphics hardware is designed in such a way that the Tiling Accelerator automatically keeps track of which tile partitions have to be redrawn during each refresh cycle. So, each time the graphics hardware renders a scene, the Tiling Accelerator automatically redraws only the portion of the screen that needs to be redrawn.

4.3.1 Tile Partitioning and Polygon Clipping

Because Dreamcast segments the screen into tiles, the ordinary clipping techniques that 3D programmers are accustomed to have been supplemented by some additional kinds of clipping.

Briefly, Dreamcast supports two kinds of clipping: *Tile clipping*, which clip polygons displayed in adjacent tiles, and *pixel clipping*, the familiar kind of clipping that is applied when polygons extend past the edges of the screen.

There are two kinds of tile clipping: *Global tile clipping*, which affects the whole screen, and *user tile clipping*, which the developer can set to affect individual polygons.

To implement user tile clipping, you define a *user tile clip area* by specifying what tiles it will include. Then you can use your clip area in two different ways. You can either clip off the edges of polygons that extend beyond the edges of your clip area, or you can use your clip area as a mask, keeping the screen area outside the clip area intact but clipping off the edges of polygons that extend into it.

Figure 4.4, “The Global and User Tile Clipping Areas,” is a simplified illustration of these two uses of the user tile clip area.

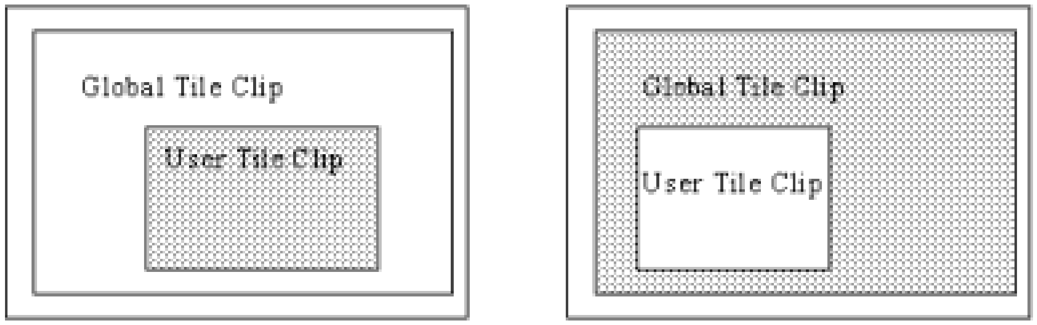


Figure 4.4 *The Global and User Tile Clipping Areas*

4.3.2 A Point to Remember about Tiling

Although the Dreamcast tiling mechanism generally works quite well without any particular assistance from the developer, there is one tip that can speed up the tiling process under one set of unusual conditions.

Those conditions are illustrated in Figure 4.5, "A Tiling Problem,", which shows a long, thin triangle that extends diagonally across the screen

Because the triangle shown in the picture is so long, Dreamcast's Tile Accelerator sees the triangle as affecting almost every tile show – even though it actually appears inside a small number of them. To render a scene containing such a triangle, the Tile Accelerator would redraw quite a few tiles that really don't have to be redrawn.

Once you are aware that this kind of problem can arise, it isn't difficult to avoid it. All you have to do is break the offending polygon down into a number of smaller polygons, as shown in the diagram on the right. Then the Tile Accelerator can redraw the screen using a much smaller number of tiles (those that are shaded in gray in the diagram).

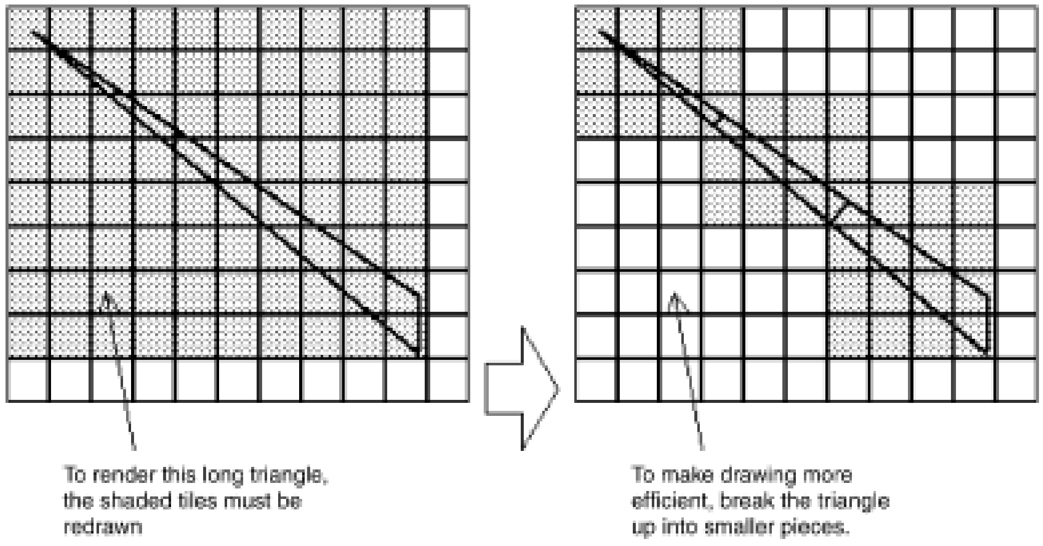

To render this long triangle, the shaded tiles must be redrawn
To make drawing more efficient, break the triangle up into smaller pieces.

Figure 4.5 *A Tiling Problem*

4.4 Modifier Volume

Modifier volume is a mechanism that compares a *virtual model* to an actual model to determine whether certain properties of the two objects – such as texture and material -- are the same or different. The modifier volume mechanism can then apply properties of the virtual model to the actual model at points where the two objects intersect – or at points where they don't. You can use modifier volume to create shadow effects or other special effects such as spotlights and window masking.

Figure 4.6, "Modifier Volume," shows how the modifier volume mechanism works. In the illustration, the texture of a virtual model is applied to Area 1, which is the portion of the actual model where the virtual model and an actual model intersect. The opposite is also possible; that is, the original model's texture data can be left intact in Area 1, while the virtual model's texture data is applied to the non-overlapping area of the actual model (Area 0).

Note that the transparent cube shown in the diagram is not visible on the screen; it is used in the picture merely to illustrate the operation of the modifier volume mechanism.

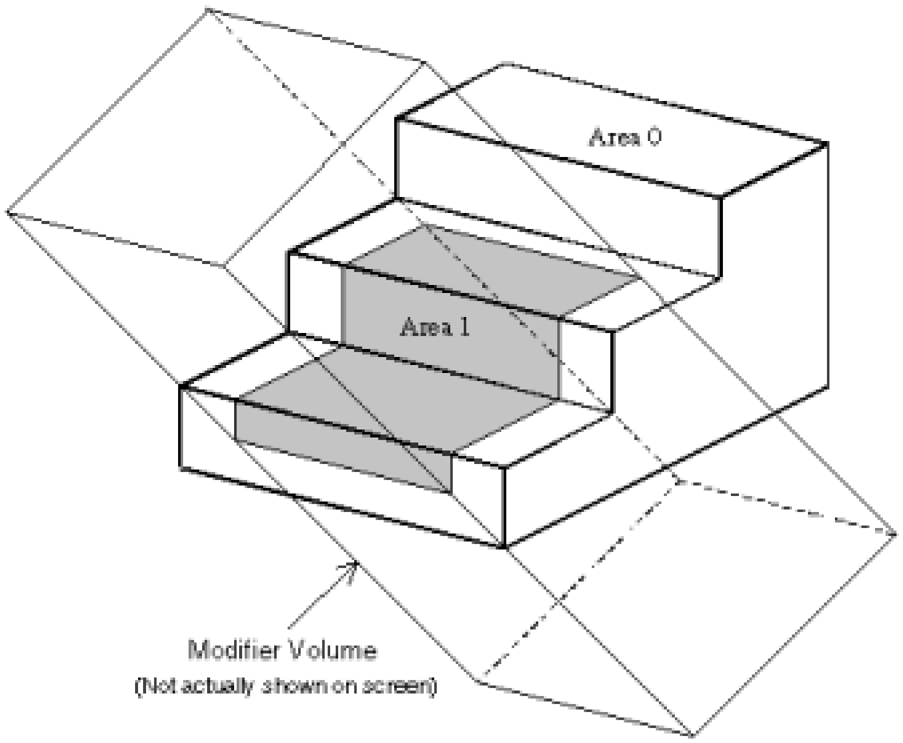


Figure 4.6 *Modifier Volume*

Modifier volume is an effective mechanism for applying shadows to models with uneven surfaces. For instance, when modifier volume is used to create shadowing effects, only the shadowed area becomes darker, so it is not necessary to set the other textures or materials.

And the property that is replaced in a modifier volume operation doesn't even have to be a texture; it can be derived from material data, vertex colors, and various other properties.

One very useful feature of the modifier volume mechanism is an *intensity mode* that you can use to control the amount of darkening or the intensity of texturing that is applied affected areas.For more detailed information about the modifier volume mechanism, see the various programmer's manuals supplied with the Dreamcast system.

4.5 Mipmapping

Mipmapping is a commonly used mechanism for making a texture look right when it is viewed from different distances. Dreamcast, like most high-performance 3D graphics systems, supports the use of mipmapping.

When a texture is displayed in a scene without the use of mipmapping, the becomes pixelated when it is viewed from a distance that is very close, and can look distorted when it is viewed from a long distance away.

Figure 4.7, “Mipmapping for Close Viewing,” shows how mipmapping works. The top left quarter of the picture shows a texture that was designed to be viewed from a certain distance. The top right quarter of the picture shows what happens when the viewer zooms in closer; as you can see, the texture becomes pixelated and distorted.

The bottom left quarter of the picture shows a *mipmap* of the same texture. A mipmap is simply a differently scaled drawing of a texture, specifically designed to make the texture look better when it is viewed from a different distance. The use of a mipmap can make a texture look much better when it is viewed from a different distance, as shown in the bottom right quarter of the picture.

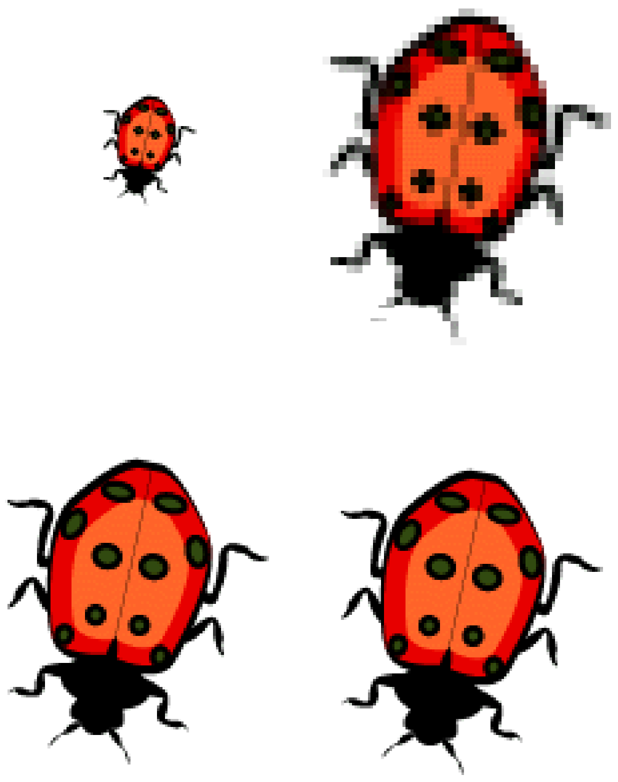


Figure 4.7 *Mipmapping for Close Viewing*

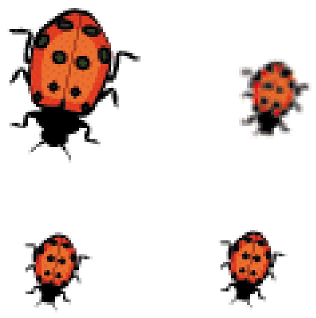


Figure 4.8 *Mipmapping for Distant Viewing*

Figure 4.8, “Mipmapping for Distant Viewing,” shows how a mipmap can improve the appearance of a texture when the camera moves in the opposite direction, pulling back to view the texture from a longer distance away.

This time, the top half of the picture shows what the texture looks like when the viewer pulls back from an un-mipmapped texture that was designed to be viewed from a midrange distance. Again, the texture becomes distorted.

The bottom half of the picture shows how the appearance of the texture improves when a mipmap is created for distant viewing.

4.6 Texture Filtering

Dreamcast supports three filtering modes for texture mapping: *point sampling*, *bi-linear filtering*, and *tri-linear filtering*. Figure 4.9, “Point Sampling,” shows how the point-sampling technique is used in Dreamcast.

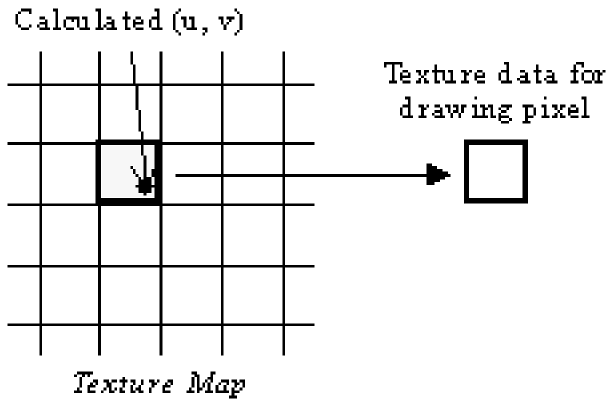


Figure 4.9 *Point Sampling*

4.6.1 Point Sampling

As shown in Figure 4.9, “Point Sampling,”, the point sampling technique uses the data from a pair of texture coordinates (u, v) that are derived from a sampling point (x, y) as texture data for the drawing pixel. Of the three kinds of texture-mapping that are available in Dreamcast, point-sampling imposes the lightest processing load on the graphics hardware. But the quality of the image deteriorates if it is enlarged or compressed.

4.6.2 Bi-Linear Filtering

Figure 4.10, “Bi-Linear Filtering,” on page 19 shows how *bi-linear filtering* can be used instead of point sampling in Dreamcast programming.

Bi-linear filtering takes the weighted average of the data from a pair of texture coordinates (*u, v*) that are derived from a sampling point (*x, y*), and the data from three adjacent texels (for a total of four texels), and uses the result as texture data for the drawing pixel.

Because the weighted average is taken from data for four texels, the quality of the image when it is expanded or compressed is better than images produced by point sampling (although in some cases, the image may appear to be out of focus).

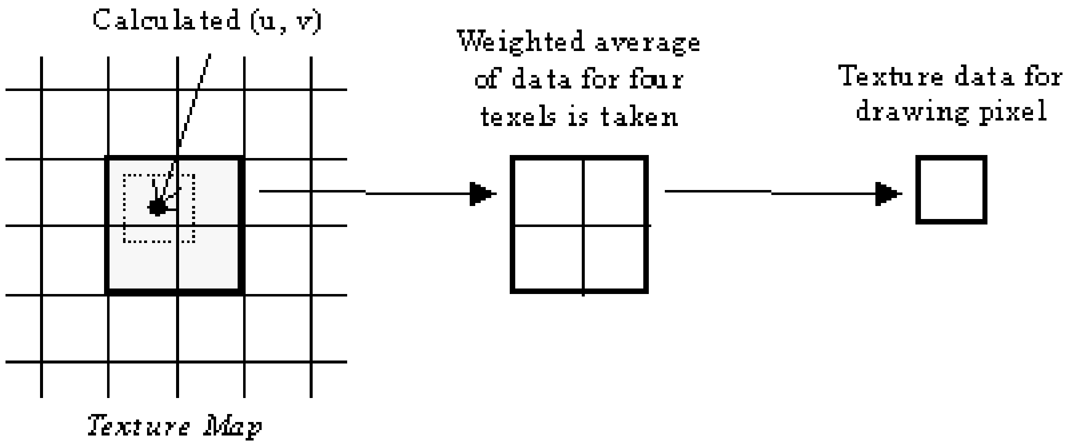

Calculated (u, v)
Weighted average of data for four texels is taken
Texture data for drawing pixel
Texture Map

Figure 4.10 *Bi-Linear Filtering*

When a Dreamcast program is working with twiddled-format textures (see “The Twiddled Texture Data Format” on page 21), processing time for bi-linear filtering is almost the same as that required by point sampling. But when a program is working with non-twiddled format textures, processing time can double in a worst-case instance.

4.6.3 Tri-linear Filtering

We have seen that when a texture uses a single texture map, using bi-linear filtering instead of point sampling can improve the quality of the texture image. We've also seen that mipmap processing can further improve the quality of the texture image when the compression factor is large (that is, if the image has moved some distance away from the viewer along the Z axis). However, even when bi-linear filtering and mipmap processing are used together, the viewer can clearly see where switchovers occur between mipmap textures of different sizes.

Tri-linear filtering is one technique for solving this problem Figure 4.11, "Tri-linear Filtering," shows how tri-linear filtering works in Dreamcast programs.

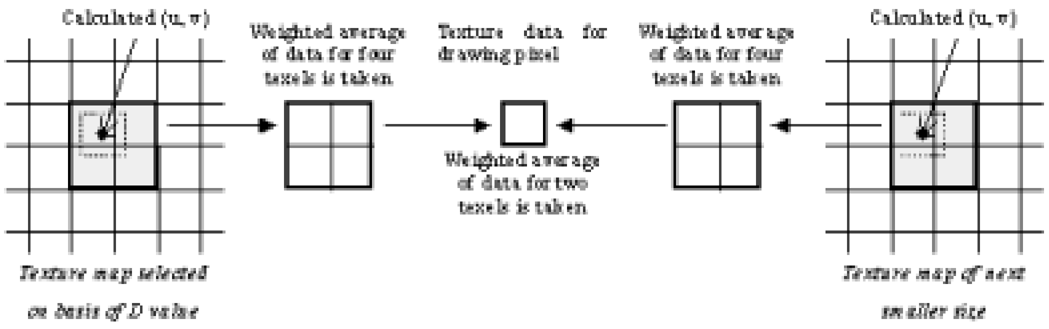

Calculated (u, v)
Weighted average of data for four texels is taken
Texture data for drawing pixel
Weighted average of data for four texels is taken
Calculated (u, v)
Weighted average of data for two texels is taken
Texture map selected on basis of D value
Texture map of next smaller size

Figure 4.11 *Tri-linear Filtering*

Tri-linear filtering takes the weighted average of the results produced when bi-linear filtering is used with mipmap textures of two different sizes, and uses the result as texture data for the drawing pixel. Because the weighted average is taken from data for a total eight texels in all, this approach offers maximum quality in enlarged and compressed images, and the switchovers between mipmap textures appear smooth. However, because this tri-linear filtering requires the most processing time of the three kinds of texture filtering that are available, it is not recommended for use in all situations.

4.7 Storing Texture Data

Dreamcast supports both an ordinary scan-line format and a special "twiddled" format for storage of texture data. The twiddled format is an irregular storage format that is generally preferred in Dreamcast programs because it permits the compression of texture data -- which, in turn provides very fast access to the data used to store textures.

Figure 4.12, "The Twiddled Texture Data Format," shows how texture data is stored in the twiddled format. As you can see, twiddled-format texture data is stored in a special order (a reverse "N") shown in the diagram below in order to minimize performance loss when reading texture data for drawing.

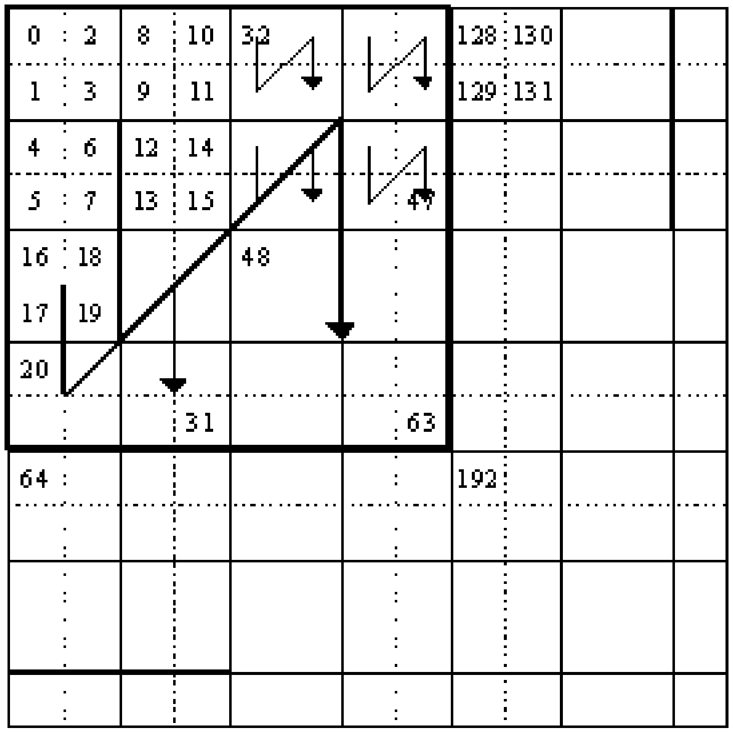


Figure 4.12 *The Twiddled Texture Data Format*

In Dreamcast, non-twiddled format texture data is stored in sequence, similar to bitmapped data. The non-twiddled format is used in Dreamcast when drawing results are to be used as texture data. The non-twiddled format results in reduced drawing performance compared to the twiddled format.

4.7.1 VQ Texture Compression

One useful feature of the twiddled format is that it permits the use of compressed texture data. More specifically, twiddling permits the use of a texture-compression technique called "VQ (Vector Quantization)."

To store a texture using the VQ compression format, you use two types of data: a *index* and a *code book*. The relationship between the index and the code book is similar to the usual relationship between palette texture data and palette data; that is, the index specifies a matrix of four texels (two horizontal texels by two vertical texels) of the texture prior to compression, using a code book number.

Figure 4.13, "The VQ Compression Format," shows how you use the VQ technique to compress texture data.

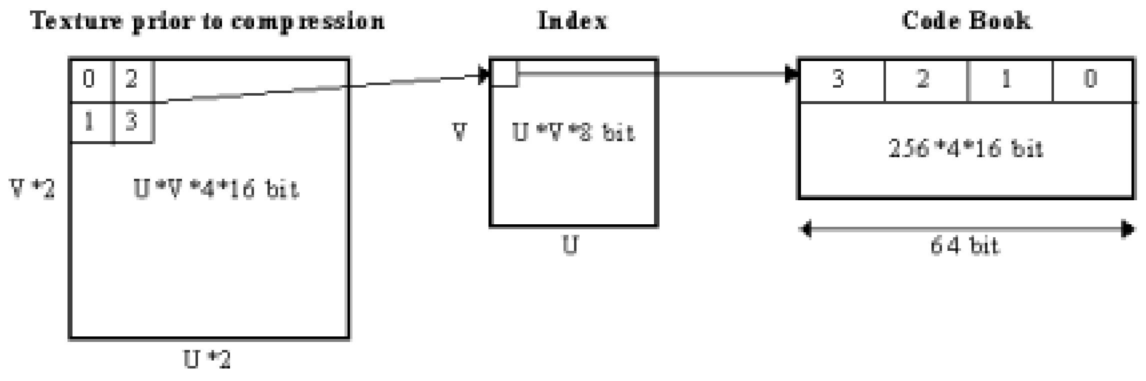


Figure 4.13 *The VQ Compression Format*

The VQ code book is a grouping of units of data for storing textures. Each unit stores data for four texels (64 bits), and usually consists of a matrix measuring 256 by 64 bits. The four-texel data in the code book is expanded in a reverse "N" shape, similar to the storage format used by the twiddled format. The texture size that is actually drawn is about twice the specified size in both the horizontal and vertical directions.

4.8 Fog

One interesting feature of Dreamcast is that support for fog effects is provided in the graphics hardware. When you want to create fog effects in a scene, you can let the hardware can perform the calculations for you, or you can make your calculations yourself and set the vertex data.

The programming mode in which you let the hardware do the calculations is called *Lookup Table Mode*. The mode in which you perform fog calculations is called *Per Vertex Mode*. Figure 4.14, "Lookup Table Mode," on page 23 illustrates this process.

When you use Lookup Table Mode and let the Dreamcast hardware perform fog calculations, it can apply fog only in the Z or depth direction, but if you do the calculations yourself, you can create fog in the up and down directions as well. When the hardware does the calculating, the fog's shape at specified units in the Z direction is placed in a table, and the hardware calculates other positions to fill out the fog's appearance.

When you use Per Vertex Mode and perform calculations yourself, you calculate the fog's shape for each vertex, but the hardware still handles the rendering. You can then apply fog in the up and down directions as well as in the Z direction, but because you have to calculate all of the data yourself, the processing takes more time.

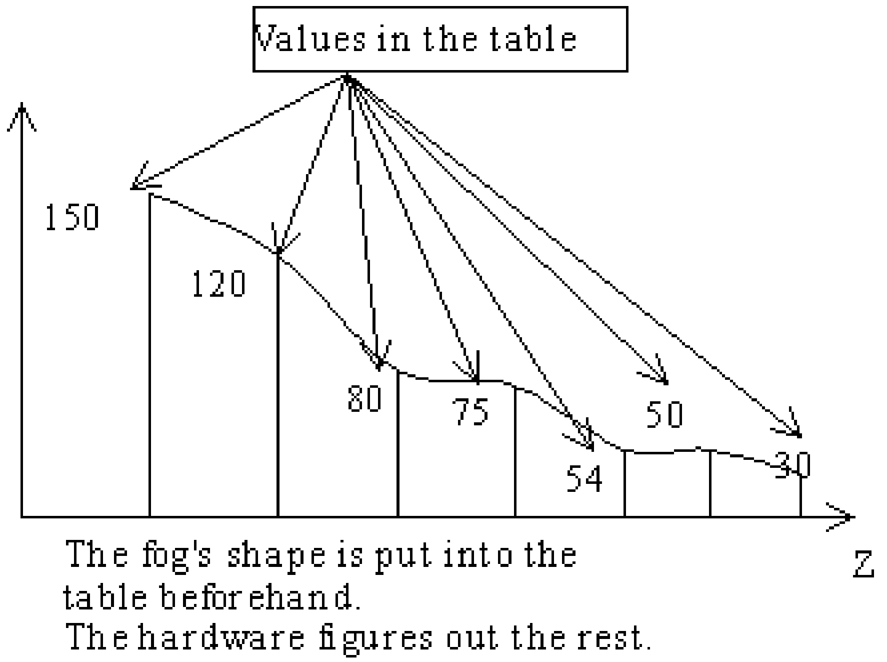


Figure 4.14 *Lookup Table Mode*

4.9 Bump Mapping

Bump mapping is a technique for creating the appearance of unevenness on a surface that does not originally have any unevenness. When you apply bump mapping to a surface, you replace the pixel data in texture data with information defining the unevenness of a surface.

Figure 4.15, "A Bump Map," is an overhead view of a bump map.

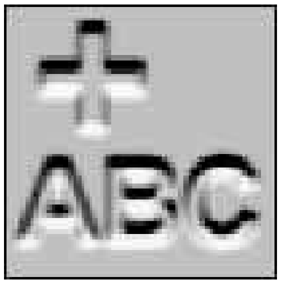


Figure 4.15 *A Bump Map*

To implement the bump mapping mechanism, you assume that a polygon shape with a virtual uneven surface exists on top of the original polygon. Then you express the normal line vectors of the virtual surface for each texel in numeric terms, and use those values as the texture data. The result is that the texel data for a bump-mapping texture is expressed not as color data, but as a pair of angles expressing the normal line vectors for the texel in question. Figure 4.16, "Bump Mapping," illustrates the concept of bump mapping. S and R are the angles representing a normal line vector. In a bump map, you can calculate illumination information using this normal line vector as a unit vector.

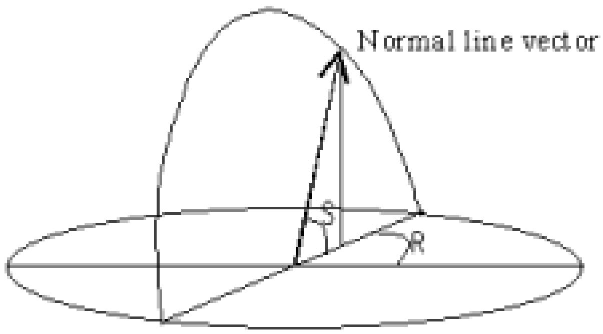
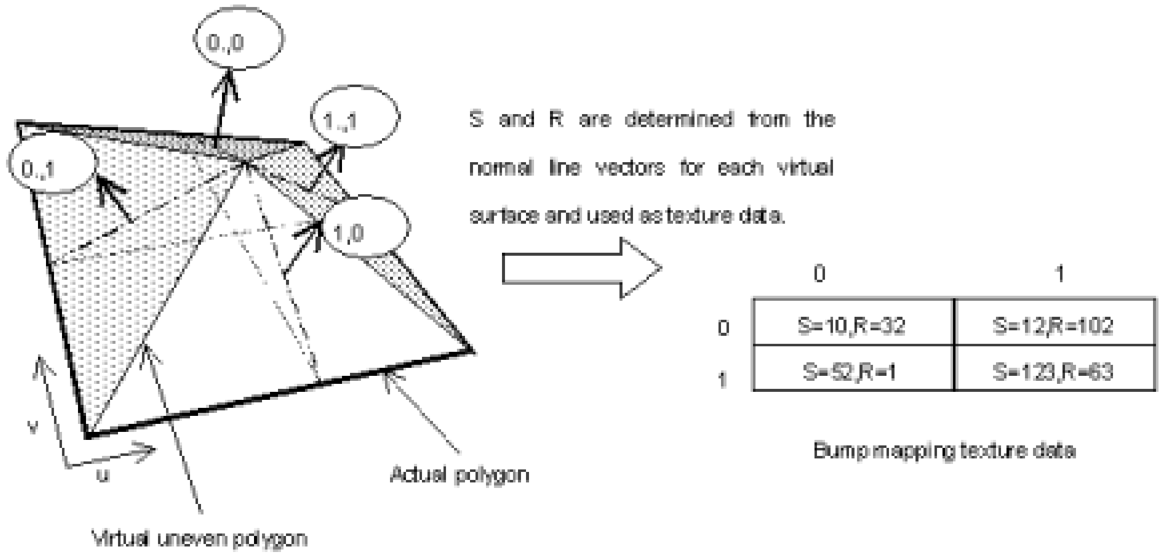


Figure 4.16 *Bump Mapping*

From an internal storage point of view, a bump map texture is a monochrome texture in which only the [alpha] value is changed by bump map processing. But often, this change alone will not be enough to give a surface the kind of uneven look you might be looking for. So a bump-map texture is usually used in combination with other textures. One way to create a really realistic-looking bump map to draw an opaque polygon that has a texture, and then to overlay another texture that has been subjected to bump-map processing on top of that. The result is a texture that gives an appearance of unevenness.

Figure 4.17, "Implementing Bump Mapping," shows how bump mapping works in a Dreamcast program.


0.,0
1.,1
0.,1
1,0
v
u
Actual polygon
Virtual uneven polygon
S and R are determined from the normal line vectors for each virtual surface and used as texture data.
Bump mapping texture data

	0	1
0	S=10,R=32	S=12,R=102
1	S=52,R=1	S=123,R=63

Figure 4.17 *Implementing Bump Mapping*

4.10 Summary

This chapter introduced some of the most important built-in graphics hardware features of the Dreamcast system, and provided on some important information about them. It did not attempt to explain how to write source code using the special features provided by Dreamcast, or how to create or structure Dreamcast programs. To learn how to program the Dreamcast system at the source-code level, please refer to the various programming and user manuals that came with your Dreamcast system.

5. Introducing Dreamcast Audio

This chapter introduces the audio components built into the Dreamcast development system, and shows how they work together. It does not attempt to explain how to use the Dreamcast audio hardware in Dreamcast programs, and it does not tell how to write audio-related software routines. To learn how to use the features and capabilities of the Dreamcast audio system in your own programs, see the programming and user manuals that came with your Dreamcast development system.

The major topics covered in this chapter are:

- "Audio Overview" on page 27.
- "The ADPCM Digital Sound Format" on page 28.
- "What is ADPCM?" on page 30.
- "Summary" on page 31.

5.1 Audio Overview

Dreamcast system has a state-of-the-art audio system powered by the Yamaha AICA sound chip, which uses the ARM7DI -- manufactured by Advanced RISC Machines (ARM) as a sound controller. Dreamcast generates sound using a 64-channel PCM/ADPCM sound source that is built into the AICA microprocessor.

The Dreamcast sound system also includes:

- Separate LFOs (Low Frequency Oscillators) for each slot.
- 64-channel 4-segment EG (Envelope Generator) LPF with a cutoff frequency that varies over time according to the 4-segment EG Forward loop function.

The system can generate up to 64 sounds simultaneously ADPCM permits pitch changes of up to one octave.

5.2 The ADPCM Digital Sound Format

The AICA chip digitizes sound using the ADPCM format, a proprietary modification by Yamaha of the standard PCM digital audio format. Instead of expressing volume at any given moment as a measured value – as a standard PCM system does – the ADPCM method records as volume data the difference between the volume that is expected (based on previous results) and the actual measured value. Because the degree of change in the difference from the expected value is smaller than the change in the actual measured volume, less data needs to be recorded for a waveform, and processing efficiency increases accordingly.

To understand how the ADPCM digital sound format works, it helps to know that sound can be understood as a composite of waves of various wavelengths that propagate through the air. Figure 5.1, "Sound Waveform," is a diagram representing a sound wave.

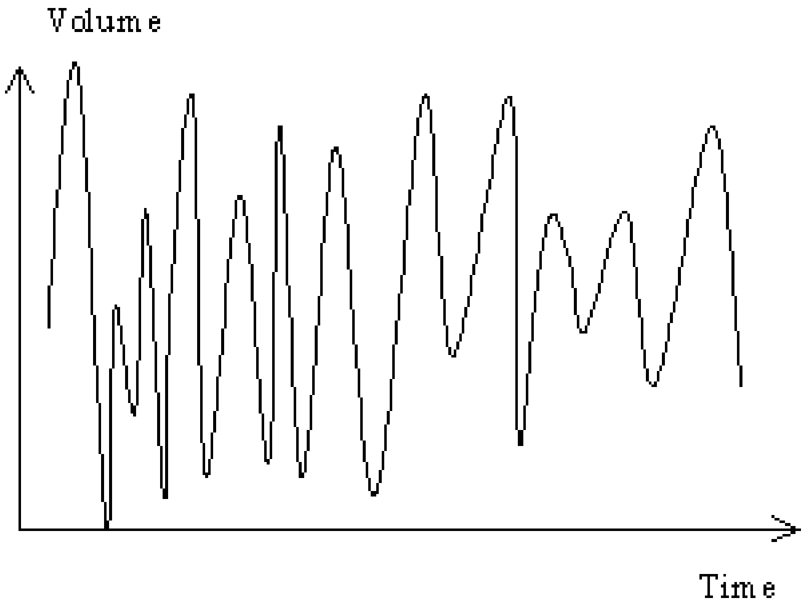


Figure 5.1 *Sound Waveform*

This diagram illustrates the relationship between time and volume. The PCM method records the volume at specific times; for example, 44.1KHz PCM divides one second into 44,100 equal segments and records what the volume was in the first 1/44,100-second segment, what the volume was in the second 1/44,100-second segment, and so on.

So, in one second of sound there are 44,100 points at which the volume is measured. Normally, the volume value that is measured here is an analog value such as "16dB." These values are then digitized into a range from 0 to 255, or from 0 to 65,535. In the example we are discussing, 44.1kHz is called the *sampling frequency,* and the range of digital values into which the volume is converted is expressed in terms of the number of *quantization bits* – for example, 8 bits in the case of a range from 0 to 255 (= 2[8])).

Figure 5.2, “PCM Data,” shows the details.

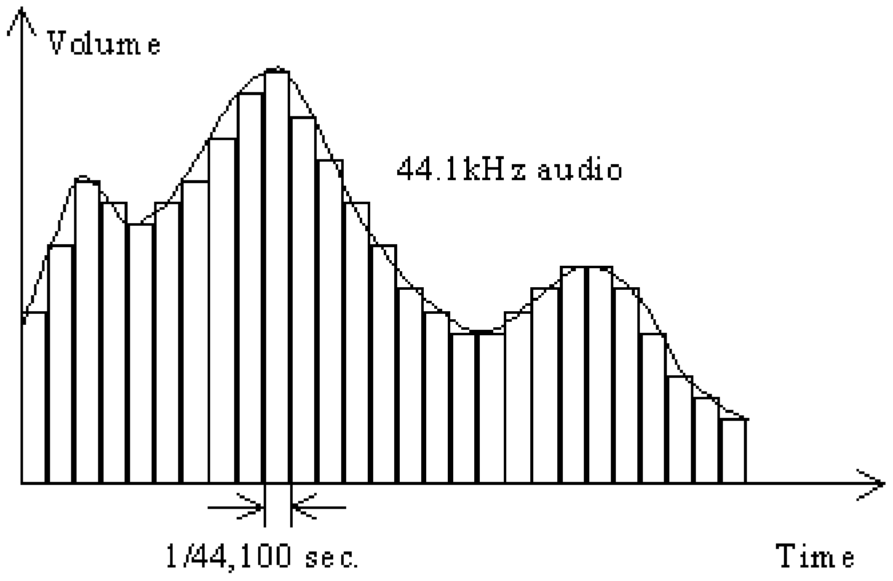


Figure 5.2 *PCM Data*

5.3 What is ADPCM?

In the ADPCM method, instead of expressing the volume at a given time as a measured value, it records as volume data the difference between the volume that is expected based on previous results and the actual measured value. Because the degree of change in the difference from the expected value is smaller than the change in the actual measured volume, less data needs to be recorded for a waveform than when recorded by the PCM method.

As illustrated in Figure 5.3, "How ADPCM Works," the job of determining "expected values" can yield unexpected results. For instance, given a data history of 54 and 71, one might expect the next value not to be 80, but to be 78. Then again, one might expect a value 75. The truth is that no one value is correct; there's a variety of different forecasting methods.

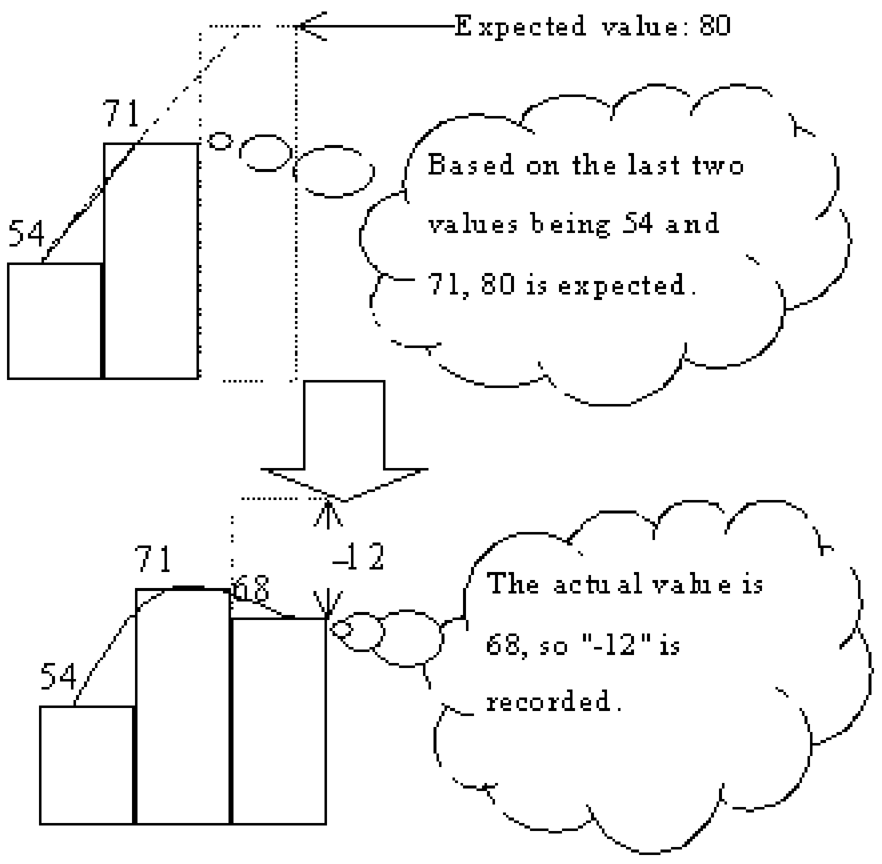

Expected value: 80
71
54
Based on the last two values being 54 and 71, 80 is expected.
71
68
-12
54
The actual value is 68, so "-12" is recorded.

Figure 5.3 *How ADPCM Works*

One well-known technique is the *Philips forecasting method*, which is generally known as part of the XA standard. If the same forecasting method is not used for both compression and decompression, then the decompressed waveform will differ from the original audio waveform. Therefore, it is not possible to simply use ADPCM just because you happen to have the ADPCM compression tool. Dreamcast uses the proprietary forecasting method developed by Yamaha, which also developed the sound chip that is used by Dreamcast. Sega supplies ADPCM compression tools that use the Yamaha forecasting method.

5.4 Summary

This chapter introduced the audio components built into the Dreamcast development system, and provided some information about how they work. It id not attempt to explain how to use the Dreamcast audio hardware in Dreamcast programs, and it does not tell how to write audio-related software routines. To learn how to use the Dreamcast audio system to in your own game titles, please refer to the programming and user manuals that came with your Dreamcast development system.

6. The SH4 and Memory Management

The Dreamcast CPU is the Hitachi SH4 microprocessor, customized for Dreamcast, with an internal processing speed of 200 MHz and a 100 MHz external bus. In the Dreamcast system, the SH4 handles processing involving the game sequence, artificial intelligence, 3D calculation, and issuing 3D graphics instructions. The SH4 also provides a general-purpose serial port with an FIFO buffer for use by external I/O devices.

The Dreamcast system is also equipped with an FPU (floating-point unit) module that can perform floating-point calculations at high speed. This module is dedicated to numeric processing that would take a great deal of time if unaided, such as the calculation of 3D coordinates.

Dreamcast has 16MB of main system memory, plus 8MB of texture memory, including memory used for drawing and display functions. There's also 2MB of memory used for sound. All memory uses synchronous DRAM, a type of RAM that allows high-speed access. The Dreamcast Visual Memory System (VMS) can provide the system with expansion memory.

This chapter introduces the Dreamcast SH4 microprocessor and some of its associated processors. It covers these main topics:

6.1 Dreamcast Memory

Dreamcast is equipped with a memory management unit (MMU) that assigns virtual addresses to addresses in physical memory, allowing the system to behave as if it has more memory than it actually has. The Dragon operating system (described in Chapter 1, ”The Sega Dreamcast Development System,”) uses this MMU module to conceal the physical memory space to a certain extent and make segmented memory appear to be continuous memory. The user does not need to be aware that memory is actually segmented.

The Dreamcast MMU provides the following for the address conversion buffer (TLB) for the MMU:

- Instructions (can also be used for data): 4 entries.
- Data: 64 entries.

Figure 6.1, “The Dreamcast MMU,” shows the layout and functionality of the MMU used in the Dreamcast system.

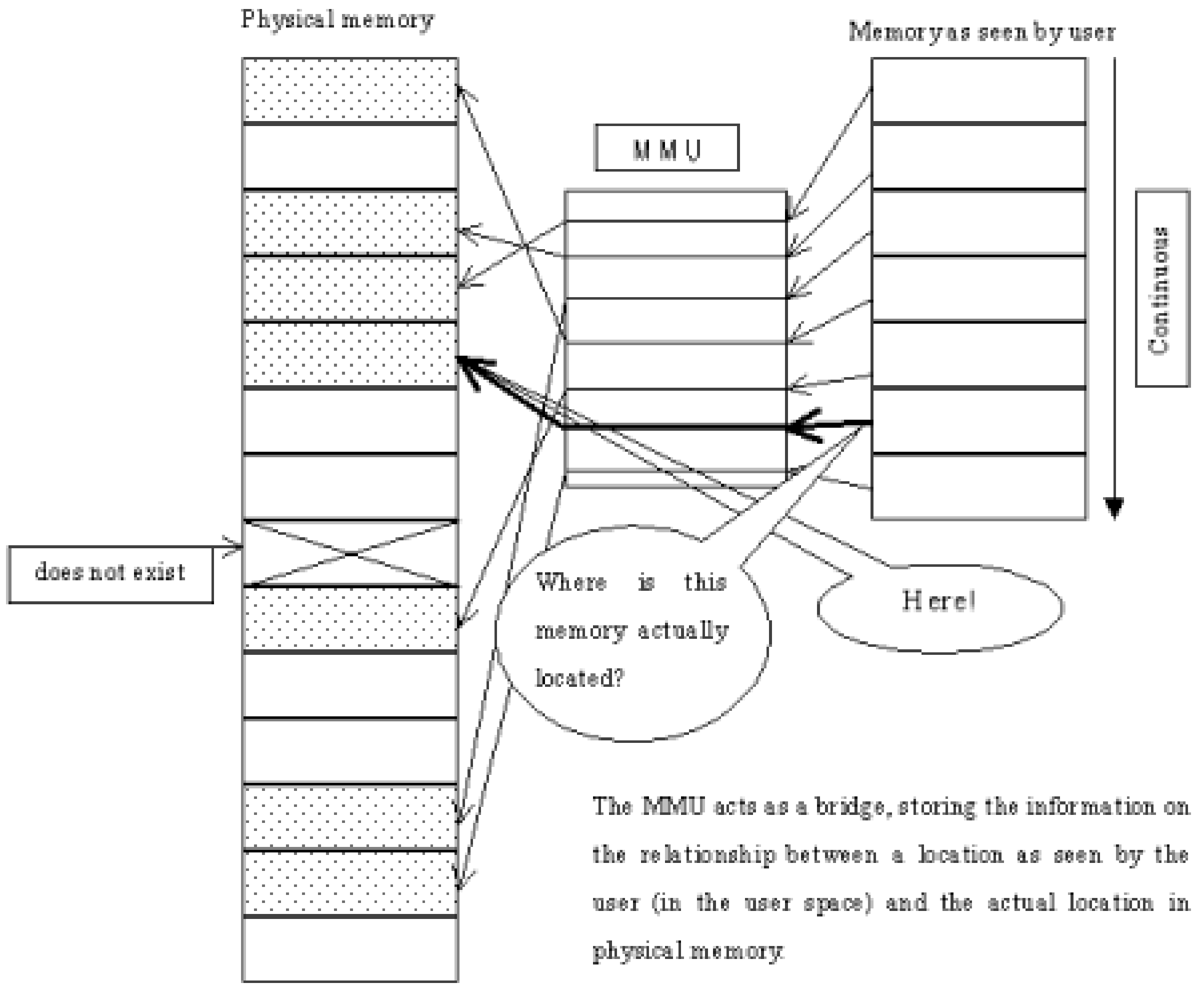

Physical memory
Memory as seen by user
MMU
Continuous
does not exist
Where is this memory actually located?
Here!
The MMU acts as a bridge, storing the information on the relationship between a location as seen by the user (in the user space) and the actual location in physical memory.

Figure 6.1 *The Dreamcast MMU*

Unlike other CPUs that have a built-in MMU, the SH4 supports a variable page size (the minimum memory block size handled by the MMU): 1K, 4K, 64K, and 1MB.

These page sizes can co-exist; for example, following arrangement is possible:

- 1K page size in 0x0000000 to 0x0FFFFFF
- 4K page size in 0x1000000 to 0x1FFFFFF
- 64K page size in 0x2000000 to 0x7FFFFFF
- 1M page size in 0x8000000 to 0xFFFFFFF

If the 1K page size is used for the entire memory space, then a conversion table 16 x 1024 = 16,384 entries in size is required in order to cover all of the Dreamcast's main memory.

However, the SH4 only has a maximum of 68 entries in its conversion table.

If the conversion information for the desired address does not exist in the internal table, the MMU searches the master conversion table that is allocated in main memory, and loads the data into the internal buffer. But this process of loading the data into the internal buffer consumes much more time than simply referencing the internal buffer. In a case such as this, where the master table must be referenced frequently, much processing time is wasted.

But what if the page size is, say, 1MB? With a page size of 1MB, only 16 table entries are required, which is well within the capacity of the internal buffer. But in this case, a different problem arises. When an instruction to allocate memory is received by the MMU, it allocates memory according to the page size. So it still allocates 1MB of physical memory even if it is instructed to allocate just 16 bytes, for example. Clearly, this wastes memory.

In short, in order to use the MMU's internal conversion table effectively and also to use memory effectively, a large page size should be designated for an area that is required to handle large amounts of data, such as a buffer area for a movie, while a smaller page size should be designated for program areas and temporary data areas (a data heap, for example). However, because modules that are responsible for such things as OS management usually handle this work, it is generally not necessary to be aware of these concerns.

6.2 DMA

In Dreamcast, the DMA transfers large amounts of data from one location to another. Because it is possible for the CPU's calculation functions, etc., to operate in parallel while the DMA is transferring data, processing is not interrupted until the transfer is completed. This makes it possible to stop time from being wasted.

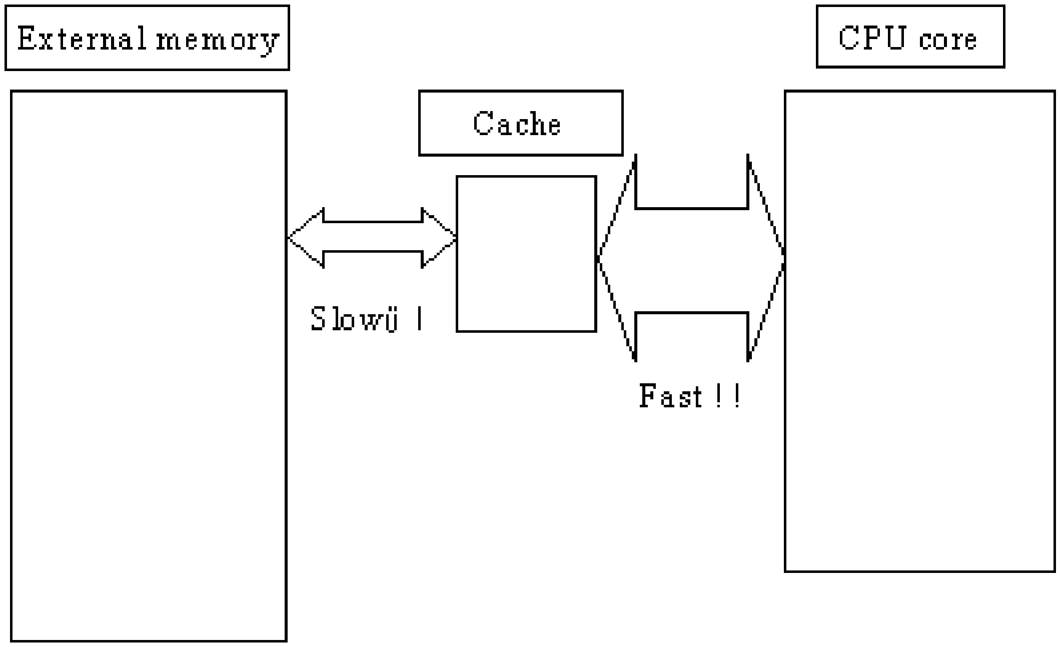
The SH4 has four DMAs, but each channel is used for a different purpose. Of the four, two are used by the system, and two are available for use at the user level. Of the two DMAs reserved for the system, one is used for high-speed access to texture memory, and can only be used for transfers in the direction from main memory to texture RAM. The other DMA reserved for the system is used for transfers between external devices and memory, including transfers from texture RAM to main memory.

6.3 The SH4Timer

The SH4 has an internal timer that operates independently.

6.4 The Cache

The SH4 has an 8K instruction cache and a 16K data cache that can be used to reduce the number of accesses made to external memory, which is slower. But accessing memory over the external bus is extremely slow, requiring 10 times longer that when accessing cache memory.



The cache stores both the content of information that is stored in external memory, and the address (location) of that information. (The more the better.)
Although initially the (slow) external memory is accessed, the next time that information is retrieved from the same location, the CPU will actually read the information stored in the cache, which is faster.

Figure 6.2 *Benefits of the Cache*

6.5 The Pipelines and Superscalar Units

The SH4 is equipped with a superscalar function. This feature gives the SH4 a tremendous advantage over earlier processors.

Simply put, once an older CPU received an instruction (the "order" shown making its way into a factory in Figure 6.3, "Early CPUs,") it was not able to accept any additional instructions until it completed processing of the first instruction ("delivery"). This is similar to a situation in which there is only one machine to perform all processes on a production line.

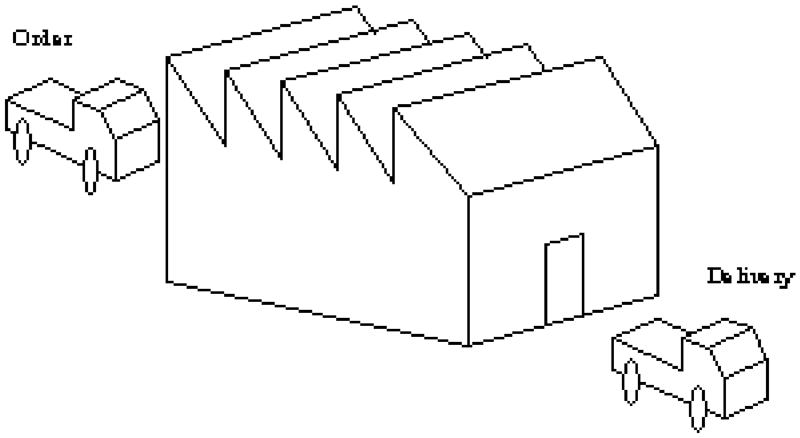


Figure 6.3 *Early CPUs*

Because this arrangement is inefficient, a review of the work process resulted in the work being divided into several independent processes.

In Figure 6.3, "Early CPUs," imagine that separate machines were introduced for each process, and different tasks had been assigned to each machine. This change would create the appearance of being able to handle multiple orders. Figure 6.4, "Pipeline Processing," shows the result.

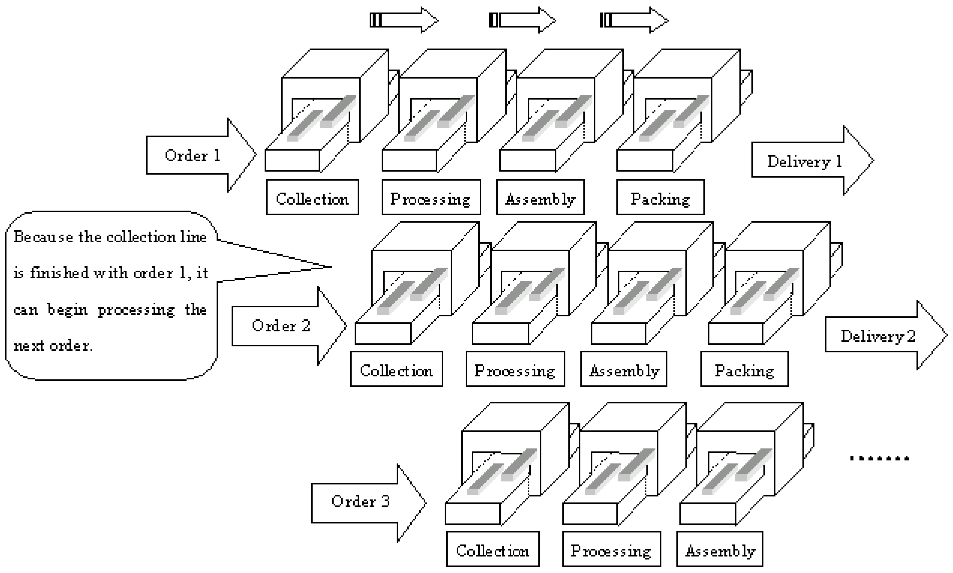


Figure 6.4 *Pipeline Processing*

You may notice that three pipelines are shown in this picture. But the SH4 has five pipelines, so it can process five instructions simultaneously, in the same way that three instructions are being processed in the diagram.

But it is important to note that if one order requires a product that results from the previous order for its own raw material, processing of the later order cannot begin until the previous order is completed. This results in a decrease in work efficiency. (In other words, such an arrangement should be avoided if at all possible in order to obtain the best performance.) Figure 6.5, "Efficient Pipeline Usage," shows an improved system.

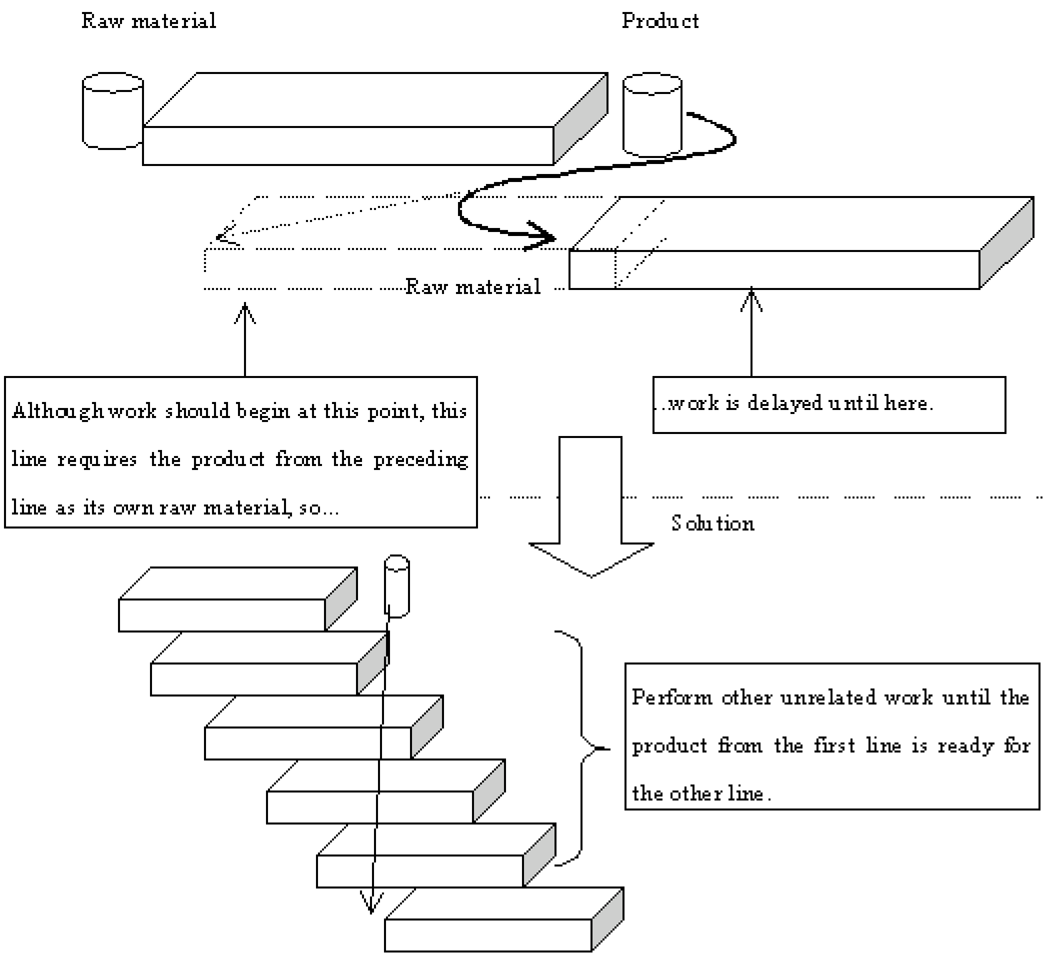

Raw material
Product
Raw material
Although work should begin at this point, this line requires the product from the preceding line as its own raw material, so...
..work is delayed until here.
Solution
Perform other unrelated work until the product from the first line is ready for the other line.

Figure 6.5 *Efficient Pipeline Usage*

How can we further increase efficiency? The simplest method is to construct another factory. The superscalar units in the SH4 – shown in Figure 6.6, "Superscalar Units," – are the CPU equivalent to this method of increasing the number of factories; they can handle two instructions completely simultaneously.

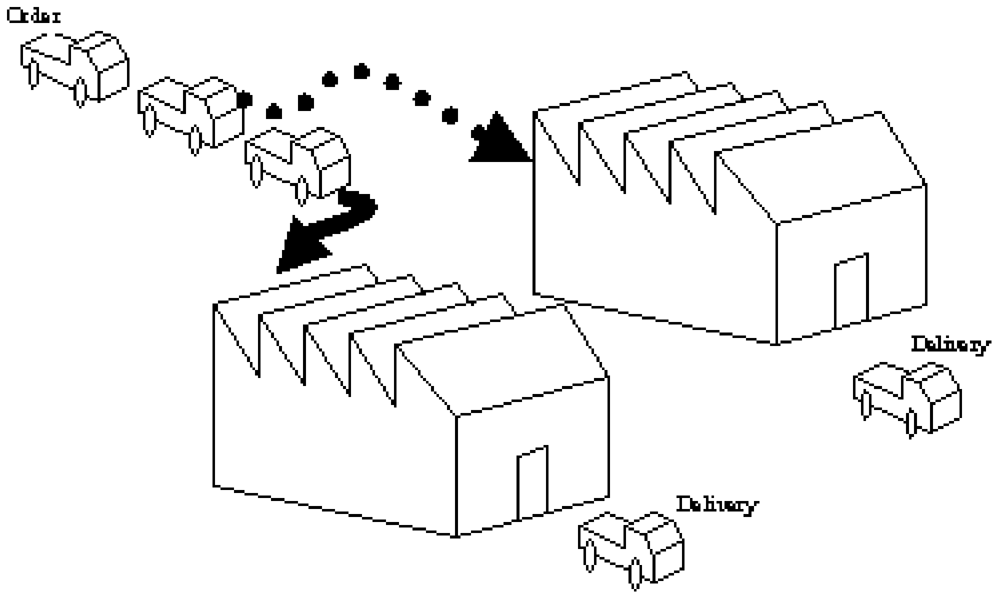


Figure 6.6 *Superscalar Units*

The process in each factory is segmented (into a pipeline) so that orders that are received can be handled efficiently. However, as in the case previously referred to, if one process requires the product produced for a previous order, work will come to a halt until that order is completed. Resolving this problem requires know-how about how to use a CPU's resources efficiently. Another technique used to speed up processing in the SH4 is hurry up the completion of slow tasks by boosting the clock speed. Considering that the SH2 used in the Saturn system ran at a 28MHz clock speed, the 200MHz clock speed of the SH4 increases the speed of processing by a factor of 7.

6.6 Summary

This chapter introduced and described some of the most important special features of the SH4 microprocessor and its associated processors.