Sega Dreamcast

Kamui2 Programmer's Guide

Sega Dreamcast

Kamui 2 Programmer's Guide
Table of Contents

Sega Dreamcast

1. The Kamui 2 Graphics API

Kamui™ 2 is a low-level triangle primitive graphics API that interfaces directly with the Dreamcast hardware. Kamui 2 provides Sega Dreamcase applications with all the graphics-related software they need to manage the operations of the SH-4 CPU, the Holly graphics chip, the CLX texture bus, and all the other components built into the Dreamcast system.

Kamui 2 is the successor to the Kamui 1 API, which shipped with earlier Sega Dreamcast development systems.

This document explains how to write applications using the Kamui 2 API, and provides some basic examples. Many other code samples can be found on the Kamui 2 distribution CD-ROM.

Special Features and Special Characteristics of Kamui 2

Kamui 2 has a number of special features, as well as a number of special characteristics that developers should be aware of. These include:

- Hardware register abstraction: Kamui 2 relies on a built-in memory management unit (MMU) to deal with the Sega Dreamcast system's hardware registers. Consequently, Kamui 2 applications do not access hardware registers directly; instead, they must access memory locations indirectly, via the MMU.
- Pipelined process/data flow: The Kamui 2 pipeline handles low-level graphics-related operations such as page flips, triangle-rendering commands, and DMA (direct memory access) timing.
- A triangle-strip primitive interface: Kamui 2 does not deal with triangles individually; it deals only with triangle strips. You can force the system to render a single triangle, however, by creating a list and then making your triangle the sole element in that list.
- List-based texture rendering: To create a triangle list, you build a list of triangle-processing commands and then place it in a queue. Kamui 2 defers the actual rendering of your list until your application specifically requests rendering by issueng a `kmRender` command.
- Interrupt callbacks: Kamui 2 provides interrupt driven callbacks that are executed during vertical blank interrupts, providing real-time feedback from the Sega Dreamcast rendering system.
- Special effects: The Sega Dreamcast graphics chip provides a number of hardware-driven special effects, including adjustable-density fog, bump mapping, and modifier volume (a mechanism that can create very impressive lighting and shadowing effects).

Later chapters provide more details about each of these special characteristics of Kamui 2.

Important: Because Kamui 2 is a low-level API, it performs only minimal error checking. If your application passes invalid parameters to a Kamui 2 function, the results are undefined.

2. Introducing Kamui 2

This chapter shows how the Kamui 2 API fits into the overall Sega Dreamcast development environment, and illustrates the processing flow of Kamui 2 operations. These are the main topics covered in this chapter:

- Sega Dreamcast Graphics Components.
- Stages of the Kamui 2 Process Flow.
- Kamui 2 Process Flow: A Closer Examination.
- Generating and Using a Strip Head Structure.
- The 2V and 3V Latency Models.
- Source Code of the QuikTest2.c Program.

The chapter concludes with a source-code listing of a sample program that illustrates the flow of a Kamui 2 application. The application displays four quads: one opaque untextured, one opaque textured, and two translucent textured quads. The routines in the code that produce the display are referred to throughout the chapter.

Sega Dreamcast Graphics Components

Figure 2.1 illustrates the architecture of the graphics-related components in the Sega Dreamcast system.

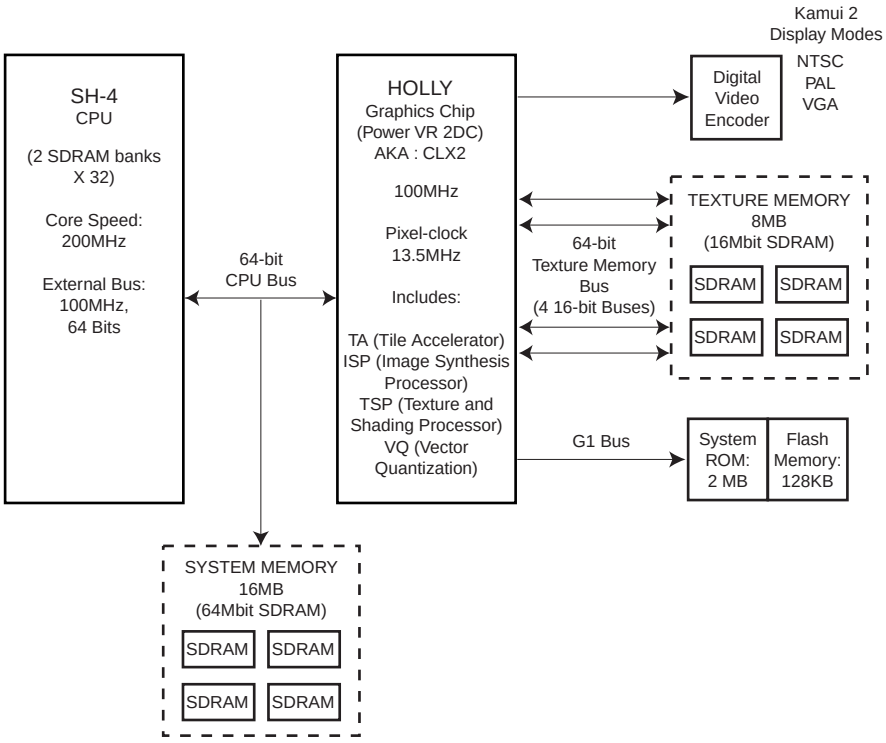


Figure 2.1 Sega Dreamcast Graphics Components

Explanation of the Diagram

The components shown in Figure 2.1 are described under the following subheadings.

DMA and the Store Queue

The CPU used in the Sega Dreamcast system is the Hitachi SH-4, a 32-bit RISC (reduced instruction set) microprocessor with an operating frequency of 200MHz. The SH-4 has an 8-kbyte instruction cache, a 16-kbyte operand cache, and a memory management unit (MMU) with a 64-entry fully associated unified translation lookaside buffer (TLB). The SH-4 also has an on-chip bus state controller (BSC) that allows direct connection to DRAM and and synchronous DRAM without external circuitry. Its 16-bit fixed-length instruction set enables program code size to be reduced by almost 50 per cent compared with 32-bit instructions.

The Sega Dreamcast system has 16MB of system memory, 2MB of system ROM, and a 2MB flash ROM. The SH-4 CPU passes data to and from system memory and to and from the system's PowerVR 2DC graphics chip (next item) via a 100MHz, 64-bit bus.

In the Sega Dreamcast system, a four-channel physical address DMA controller handles most major transfers of data. Another mechanism, called the store queue (described in the Sega Dreamcast SH4 Compiler User's Manual), is another efficient mechanism for writing data. But you will probably find that DMA is the fastest way to move large amounts of data; it operates using a double-buffered, 32-byte transfer mechanism that can move data at a peak transfer rate of 800 MB per second.

Transferring data using DMA is a straightforward process. Your application provides a starting point, a destination point, and the number of bytes to be transferred, and the system's DMA controller handles the rest. But it is important to remember that the Sega Dreamcast DMA controller (as well as the store queue mentioned in the preceding paragraph) moves data in 32-byte-aligned blocks, so applications must always make sure that data to be transferred is correctly 32-byte-aligned.

The PowerVR2DC and the Tile Accelerator

The graphics microprocessor used in the Sega Dreamcast system is the PowerVR 2DC microprocessor, which is sometimes referred to as the PowerVR 2DC chip. Sometimes the PowerVR 2DC is also referred to by its project codename, CLX2.

The PowerVR 2DC performs graphics processing at extremely high speeds by dividing the screen into 32-pixel by 32-pixel squares called tiles, which are then rendered with the help of a special hardware component called a tile accelerator, or TA. The tile accelerator does its job by analyzing the X, Y, and Z coordinates of each polygon on the screen to determine which tiles will potentially be needed to display each polygon. It then builds linked lists of the polygons in each tile, using a bounding-box algorithm to determine which polygons have the potential of appearing inside each tile. Finally, the TAcomposes each tile on the screen additively, discarding any obscured pixels inside each tile and ignoring portions of polygons that will never be rendered because they fall outside screen boundaries. The TA also performs whatever actions may be needed to ensure that translucent polygons are handled properly.

The PowerVR 2DC is designed to work precisely in parallel with the tile accelerator; it does not write out the contents of any tile on the screen until the final stage of rendering, after the tile has bee n completely written to. Then, because the PowerVR 2DC can concurrently process 32 pixels at a time, it can handle a complete scan line in parallel, making extremely fast rendering possible.

Furthermore, because the PowerVR 2DC has knowledge of the entire contents of each individual tile on the screen, it can discard any pixel that is obscured by another pixel in front of it. This strategy eliminates the need for a Z-buffer, so the PowerVR 2DC has no need to write out to a Z-buffer stored in memory.

Note: Although the TA mechanism used in the Sega Dreamcast system can dramatically increase rendering speeds, efficiency can suffer when it is called on to render certain kinds of unusually shaped polygons — for example, long, thin triangles that extend across the screen diagonally. For a description of this kind of potential problem, see section 4.3.2, "A Point to Remember about Tiling," in the Sega Dreamcase Roadmap and Overview manual.

(For more information about tile partitioning, see Section 4.3 of the Dreamcast Roadmap and Overview manual.)

Important Features of the PowerVR2DC

The PowerVR 2DC is equipped with onboard hardware that can perform a number of unique graphics operations, and can also perform many commonly used graphics operations with uncommon speed and efficiency. By learning about these special features of the PowerVR 2DC and mastering them, you can add some impressive effects to your Sega Dreamcast titles while speeding up graphics processing. Here are some of the special features of the PowerVR 2DC that can help you write faster and more impressive Sega Dreamcast applications:

- Modifier volume: A mechanism that can create dramatic effects, such as shafts of light and columns of shadow — by allowing your application to create two different models of a graphics object and then to control how the object represented by the two models is rendered in a scene. The modifier volume mechanism can modify any parameters you specify for a vertex — include the RGB, alpha, U, and V channels — and can then display a pair of models in which any or all of these parameters are modified in different ways. To achieve its effects, the modifier volume mechanism uses an algorithm that treats different areas of the screen in different ways, depending upon where the textures and materials used

in your models intersect and where they don't. (A light version of the modifier volume mechanism, called cheap shadow mode, is also available; for details, see Using Cheap Shadow Mode on page 16 . For more information on modifier volume, see Section 4.4 of the Dreamcast Roadmap and Overview manual.)

- Bump mapping: A technique your application can use to add a realistic 3D look to surfaces that would otherwise look flat. The PowerVR 2DC can apply realistic shadowing to a bump-mapped surface, making it appear that parts of the surface rise above other parts. For more details and an illustration, see Section 4.9 of the Dreamcast Roadmap and Overview manual.
- Free bi-linear filtering: The PowerVR 2DC supports four kinds of destination blending: point sampling, bi-linear filtering, tri-linear filtering, and anisotropic filtering (which can improve the looks of long straight lines, such as long flat horizons in driving games). But the kind of destination blending you'll probably be most interested in is bi-linear filtering, which you can use at practically no cost in the Kamui 2/Sega Dreamcast environment. Why? Because the PowerVR 2DC transfers data to and from texture memory using a 100MHz 64-bit bus that is divided into four banks of 16 bits each (see Figure 2.1). Because Sega Dreamcast textures always consume 16 bits of memory or less, this bus configuration means that the PowerVR 2DC can fetch 4 texels from texture memory in a single pass — and that means, in turn, that the PowerVR 2DC can acquire enough data in a single fetch operation to perform bi-linear filtering. So you can use as much bi-linear filtering as you like, virtually free of charge, in your Sega Dreamcast titles. (For more information about how Kamui performs bi-linear filtering and other kinds of destination blending, see Section 4.6 of the Dreamcast Roadmap and Overview manual.)
- Twiddled textures: The Sega Dreamcast development environment supports the use of two formats for texture storage: an ordinary scan-line format and a "twiddled" format, in which texture data is stored in a special reverse "N" format. This arrangement groups pixels together in a pattern that permits more efficient filtering and generally improves cache coherency over what it would be if pixels were fetched on a straight row-by-row basis. (To learn more details and to take a look at some diagrams that show how twiddling works, see Section 4.7 of the Dreamcast Roadmap and Overview manual.)
- VQ texture compression: To permit the storage and retrieval of twiddled texture data, the data must be compressed using a technique called vector quantization, or VQ. (For more information on VQ texture compression, see Section 4.7 of the Dreamcast Roadmap and Overview manual.)
- Fog: The PowerVR 2DC provides some interesting tools for creating and handling fog. When you want to create fog effects in a scene, you can let the PowerVR 2DC perform all calculations for you using a fog lookup table combined with some vertex calculations. Alternatively, you can perform all the necessary calculations yourself by setting your own vertex data. When you let the PowerVR 2DC's hardware do the calculating, it sets the fog's vertex data in the Z direction using its lookup table, and then calculates all the other positions needed to fill out the fog's appearance. When you set the vertex data yourself, the creation of the data takes more time because it is performed by the CPU. For more details on this topic, see Section 4.8 of the Dreamcast Roadmap and Overview manual.
- Mipmapping: Mipmapping is a common technique for using different-sized bitmaps to represent the same texture viewed from different distances; destination blending is used to change as smoothly as possible from one bitmap to another as the object comes closer or recedes from view. Mipmapping is important because the use of the correct mipmap can make an object look more realistic when it is viewed from different distances. Without mipmapping, the pixels that make up the object tend to lose coherency and get out of sync with each other as the distance between the viewer and the object changes. On the other hand, the proper use of mipmapping increases pixel coherency and yields better performance in fetching textures. When you provide a set of mipmaps for an object in a scene, the Sega Dreamcast system helps you perform the operations that are needed to implement mipmapping smoothly and efficiently. In the Sega Dreamcast system, the hardware selects which mipmap to use depending upon the distance between the

observer and the object for which mipmaps have been provided. (For more about mipmapping, and to view a picture of some mipmapped ladybugs, see Section 4.5 of the Dreamcast Roadmap and Overview manual.)

Sega Dreamcast Display Modes

The Sega Dreamcast Digital Video Encoder (shown in the upper right-hand corner in Figure 2.1) supports the VGA, NTSC, and PAL video formats in four display modes: 320 by 240, 320 by 480, 640 by 240, and 640 by 480 pixels. The most commonly used format is 640 by 480 pixels. An extended PAL mode stretches the display to fill a PAL TV screen.

For more details about the Sega Dreamcast display modes, see Kamui 2 Display Modes on page 12 .

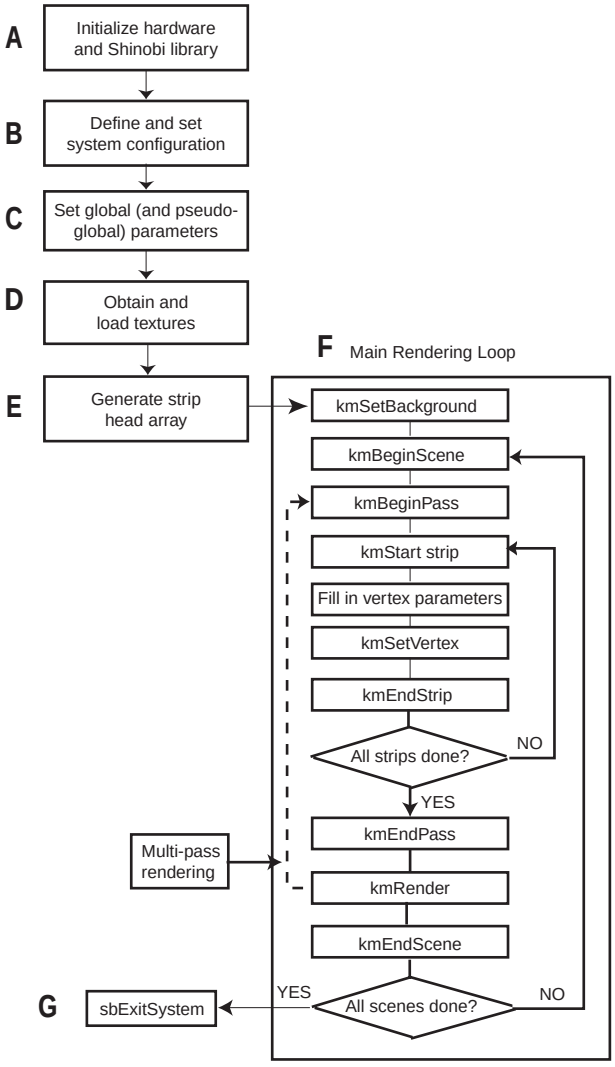


Figure 2.2 Kamui 2 Process Flow

Stages of the Kamui 2 Process Flow

Figure 2.2 illustrates the process flow of a typical Kamui 2 application. These are the steps shown in the diagram:

First, the application calls several API functions that intialize the Sega Dreamcast hardware and the Shinobi library (A). Next, the application provides Kamui 2 with some important details about the system configuration it will be using (B). Then, on a scene-by-scene basis, the application initializes a series of parameter settings called global settings that will be needed to render each scene (C). Global settings used in Kamui 2 vary from application to application. They can include fog settings, pixel-clipping settings, culling-register settings, and sometimes much more, depending on the needs of the executing application.

When the application has set up all the global parameters it will require, it creates a texture surface and then obtains and loads the textures that it will be using in each scene (D). When that operation is finished, the application creates the data structures, called Vertex Context structures, that describe the rendering properties required for each material. Using a series of API calls named `kmGenerateStripHead`*`xx`*, this information is compressed into an efficient strip header data chunk which can then be passed to the hardware at the start of each strip. (The `kmGenerateStripHead` functions replace the `ProcessVertexRenderState` function that was used in Kamui 1. They operate more efficiently and much faster than the `ProcessVertexRenderState` function did, so `ProcessVertexRenderState` has been deprecated in Kamui 2. For details, see Generating and Using a Strip Head Structure on page 17 .)

When the executing application has called `kmGenerateStripHead`, it enters its main rendering loop (F). During each iteration of this loop, the executing application calls the `kmStartStrip` function, which copies the strip head information to the tile accelerator.

The `kmStartStrip` function performs two actions. It

- Applies the strip render state to a new strip and
- Inserts control words into the vertex buffer. (To render polygons, Kamui 2 passes a series of control words to the hardware, and the hardware then uses these control words to create a display list. For more information about how control words are generated and used in Kamui 2 programming, see Generating a Strip Head Structure on page 9 .)

Your application does not need to call `kmStartStrip` for every strip; the only time a `kmStartStrip` is required is when a render statechanges.

To modify a strip head that has already been generated, Kamui 2 supplies a family of functions named `kmChangeStrip`*`xx`* (for example, `kmChangeStripBlendingMode`). To learn more about this group of functions, see the appropriate entries in the Kamui 2 Reference Manual.

The main rendering loop of a Kamui 2 application also includes calls to `kmSetBackground`, `kmBeginScene`, and (optionally) `kmBeginPass`, which starts each rendering pass in applications that make use of multi-pass rendering. You should invoke these three calls at the beginning of your application's main rendering loop, before you call `kmStartStrip` to start your first strip-rendering operation. When all the strip heads that your application will use have been rendered, call `kmEndPass` (if you use multi-pass rendering), `kmRender`, and `kmEndScene` to bring your main rendering loop to an end.

When your application's main rendering loop ends, so does the Kamui process flow illustrated in Figure 2.2. The processing flow concludes with a Shinobi Library call, `sbExitSystem` (G).

To study a Kamui 2 application that performs all of the actions described in this overview, see the source code for the `QuikTst2.c` program at the end of this chapter.

Note: Developers who have used Kamui 1 may recall that the Kamui 1 processing flow ended with a call to the `Flip` command, which performed a page-flipping operation that performed a fast exchange of the data in a pair of frame buffers (the onscreen frame buffer and an offscreen frame buffer). In Kamui 2, the `Flip` command is controlled within the `kmRender` call, in which you can use the syntax `kmFilter(FLIP)` or `kmFilter(NO FLIP)`.

Kamui 2 Process Flow: A Closer Examination

The preceding section presented a very brief overview of the process flow of a Kamui 2 application. This section provides a much more detailed examination of each of the topics touched upon in the previous section.

Initializing the Dreamcast Hardware and the Shinobi Library

Referring back to Step A Figure 2.2, notice that the Kamui 2 process flow begins with initialization of the Sega Dreamcast hardware and the Shinobi library. Eaxmine the `QuikTst` sample program presented at the end of this chapter, and you will see that the application calls two API functions during this initial stage of processing:

- `sCblCheckCable`: Checks to determine what kind of vide capability your DC is set up for (NTSC, PAL, or VGA).
- `sbInitSystem`: Initializes the Shinobi library.
- `kmInitDevice`: Initializes the hardware. This function outputs video signals to produce a blank screen.
- `kmSetDisplayMode`: Sets the display mode of the frame buffer.

Initializing the System Configuration

When the Sega Dreamcast hardware and the Shinobi library have been initialized, the application that is being executed initializes the Kamui 2 rendering system by taking a number of actions, including these:

- The application calls an API function named `kmGetVersionInfo`, which provides a structure containing library version information.
- The program sets the size of a structure named `KMSYSTEMCONFIGSTRUCT` by calling `kmSetSystemConfiguration`. This function sets a number of flags in the `KMSYSTEMCONFIGSTRUCT` to perform a number of important operations, including clearing the frame buffer and specifying whether the application will use 2V or 3V latency mode — that is, whether it will divide the rendering of each scene into two Vblank periods or into three Vblank periods. (For more information about these two latency modes, see The 2V and 3V Latency Models on page 21 .)

Other `KMSYSTEMCONFIGSTRUCT` fields that are set by the `kmSetSystemConfiguration` function include:

flags that provide Kamui 2 with data about the frame buffer surfaces that the application will use, about the texture handler setup that it will use, and the vertex buffers that will be used when rendering begins.

`kmSetSystemConfiguration` also sets flags in the `KMSYSTEMCONFIGSTRUCT` that tell Kamui 2 whether it intends to use multi-pass rendering — and, if so, what kinds of operations it plans to carry out in each rendering pass.

For a closer examination of how the Sega Dreamcast system is configured to run an application, see Setting the System Configuration on page 27 .)

Generating a Strip Head Structure

When the system configuration has been set, your application should call `kmGenerateStripHead` for every polygon type you need to render — for example, polygons used to construct opaque textured objects, translucent textured objects, objects with Gouraud shading, or opaque or translucent untextured objects.

to build a structure containing the rendering parameters that will be needed to perform all necessary rendering operations. To provide these parameters to the Sega Dreamcast graphics hardware in a form that the hardware can use for rendering, the`kmGenerateStripHead` function builds a structure called a `KMSTRIPHEAD` using

information that has been provided to it in a KMSTRIPCONTEXT structure. (For more details about how the kmGenerateStripHead function constructs a KMSTRIPHEAD structure, see Generating and Using a Strip Head Structure on page 17 .)

Setting Global Scene Parameters

When the Dreamcast hardware, the Shinobi library, and the Kamui 2 rendering system have been initialized, the application that is being executed initializes a set of global scene parameters. All the parameters in this category apply to a single scene and remain set throughout the life of that scene. (In contrast, there is a set of pseudo-global parameters that do not apply to an entire scene but do apply to multiple triangle strips, and therefore do not have to be changed every time a triangle strip changes.)

Some of the most important global scene parameters used in Kamui 2 are described under the subheadings that follow. For more information about the setting of global parameters, see Setting Global Scene Parameters on page 10 .

Initializing Fog Settings

Set fog and background color for each scene. In this step, the application sets parameters (such as fog table data and background color) related to an entire scene. The parameters that are set in this step are global parameters that will not change during rendering.

The Sega Dreamcast system provides support for fog effects in its graphics hardware. Two methods for producing fog effects are supported:

- A Lookup Table Mode, in which you let the Sega Dreamcast hardware perform the calculations needed to create fog effects, and
- A Per Vertex mode, in which you perform the calculations.

For more details about how these two modes work, see Section 4.8, "Fog," in the Sega Dreamcast Roadmap and Overview manual.

The following functions are used to initialize fog settings:

- kmSetFogTableColor: Specifies the fog color for the table fog when a table is used. (If you want to change the fog color when rendering, do so within a callback function for rendering termination.)

Texture Loading

In Kamui 2, you cannot transfer texture data to texture memory directly from files or I/O devices. To register a texture, an application simply transfers data from an area allocated in system memory into a different area of memory: specifically, texture memory (see Figure 2.1, "Sega Dreamcast Graphics Components.") It is important to note that you do not have to register textures at the exact point shown in Figure 2.2; actually, you can perform texture registration at any point shown in the diagram. Just be careful not to rewrite texture data during the rendering process.

- kmLoadTexture:
- kmQueryFinishLastTextureDMA:

Initializing Settings for Multi-Pass Rendering

In the Sega Dreamcast system, multi-pass rendering is a mechanism that lets you split a rendering operation into multiple passes, rather than having to start a new rendering operation every time you need to write a new list of vertex data and rendering parameters to the vertex buffer. By using multipass rendering, you can separate the polygons in a scene into different groups, and then draw each group of polygons in a separate pass of the same rendering operation, instead of using a different rendering operation for each set of polygons. This technique can be useful in Sega Dreamcast applications because certain parameters used

in rendering (such as the Z clipping plane) can be set to different values during the course of a rendering operation. Then another set of vertex data can be written to the vertex buffer immediately, during a subsequent rendering pass, without the necessity of starting a whole new rendering operation.

You can use multipass rendering in several different ways in your Sega Dreamcast applications. For more details on when and how to use multipass rendering, see Multipass Rendering on page 20 .

To start rendering a scene and to start using multi-pass rendering within a scene, use these two API functions:

- `kmBeginScene`: Starts the scene.
- `kmBeginPass`: Begins a pass when multi-pass rendering is used.

Vertex/Control Registration

The following functions handle the writing of vertex data and rendering parameters to the vertex buffer:

- `kmStartStrip`: Performs start of Strip(direct transfer). (This function writes the rendering parameters that were constructed in `KMSTRIPHEAD` to the vertex buffer indicated by `ListType` in the internal data. You need not call `kmStartStrip` for every strip; call it only when you want the render state to change.)
- `kmSetVertex`: Writes the vertex data referenced by `pVertex` to the vertex buffer.

Rendering and Page Flipping

These functions are used in connection with rendering and page-flipping:

- `kmEndPass`: Ends a pass when multi-pass rendering is used. (You can continue a pass by calling `kmContinuePass`.)
- `kmRender`: Notifies Kamui 2 that registration of one page of vertex data is complete in relation to the tile accelerator. When vertex data rendering is complete, Kamui 2 starts applying rendering to the back buffer.
- `kmRenderTexture`: Notifies the tile accelerator that all vertex data of a single scene has been written. The renderer begins rendering for a texture surface specified after vertex data expansion is completed.
- `kmEndScene`: Ends the scene. (This call is executed after all passes that are to be used have been registered and kmRender has been executed.)

Initializing Settings Used by Callbacks

Kamui 2 and Shinobi are equipped with a number of callbacks that can give you insight into what your application is doing and how it is performing. Available callbacks include:

- `kmSetEORCallback`: Specifies the callback function to be called at the end of rendering.
- `kmSetHSyncCallback`: Specifies the callback function to be called at an entry into the horizontal flyback segment (Hsync).

`syChainAddHandler`: Adds your callback to the chain of interrupt handlers.

Kamui 2 Display Modes

The Sega Dreamcast videogame system is compatible with the NTSC, PAL, and VGA display mode. Although the 640-pixel by 480-pixel screen mode is the one most commonly used, the Dreamcast can also create 320x24, 320x480, and 640x240 displays.

The Digital Video Encoder used in the Dreamcast system can create a screen display using four modes: interlaced (at 30Hz), non-interlaced (at 60Hz), pseudo-NI (in 60Hz fields), and a flicker-free mode (60Hz averaged). The encoder's flicker-free mode averages the current field with the previous one, creating a non-flicker image without interlacing. Try it; you might like it.

The Dreamcast graphics system is equipped with three sizes of frame buffers; a 16-bit size, a 24-bit size, and a 32-bit size. All three frame buffers use the RGB mode for color composition, and the 16-bit buffer can also use the ARGB 1555 color depth. The 24-bit frame buffer requires more memory, of course, but not too much for the Sega Dreamcast hardware to handle. Nonetheless, most developers use the system's 16-bit frame buffer.

The graphics hardware supports a 16-bit dithering mechanism that costs nothing and improves color banding. It also supports an antialiasing filter that renders each screen tile at four times its actual size, and then reduces the tile to its actual size in order to implement anti-aliasing. The result is a soft-edged image that you might try experimenting with to see if you find it useful in your application.

To set the display mode, and to obtain information about the display mode currently being used, call the API functions `kmGetDisplayInfo` and `kmSetDisplayMode`.

- `kmGetDisplayInfo`: Returns information about the display (specifically, pointers to `KMVERTEXBUFFDESC` and `KMSURFACEDESC`).
- `kmSetDisplayMode`: Sets the display mode of the frame buffer.

For more information on these functions, see the "Display Control" section of the Kamui 2 Reference Manual.

Setting Scene Parameters

Some of the most important global parameters used in Kamui applications are described in more detail under the subheadings that follow. This section discusses the initialization of:

- The Vsync count.
- Global clipping settings.
- Color clamping settings.
- Culling-register settings.
- Settings for color clamping.
- The punchthrough threshold.
- The texture stride width.
- The border color.
- Global palette settings.

Setting the VSync Count

As Figure 2.2 illustrates, an application terminates vertex data registration ends with a call to the `kmRender` function. When you start setting global scene parameters, a call to `KmFlipFrameBuffer` will direct Kamui 2 to wait for the next Vsync to start rendering a scene.

Initializing Global Clipping Settings

Global clipping settings such as `kmSetPixelClipping`: Sets pixel-unit clipping for rendering output to the frame buffer. Parameters passed to this function specify the coordinates of the upper-left and lower-right corners of a clipping area in pixel units.

Setting Color Clamping

In the Sega Dreamcast system, color clamping is used to avoid oversaturation of colors. The following functions are used to set color clamping values:

- `kmSetColorClampValue`: Specifies the color clamp value. (Color clamping is applied ahead of fogging. If you want to change the clamp color when rendering, do so within a callback function for rendering termination. If an attempt is made to change the clamp color at any other timing, a screen image may become invalid.)
- `kmSetColorClampMax`: Specifies the color clamp maximum value.
- `kmSetColorClampMin`: Specifies the minimum color clamp value.

For more information about these functions, see the “Global Settings” section of the Kamui 2 Reference Manual.

Setting the Culling Register

In the Sega Dreamcast system, a culling register holds a value that specifies how large a polygon must be in order to avoid being culled, or dropped. The `kmSetCullingRegister` function controls the setting of the culling register:

For more information on the Kamui 2 culling register , see the “Global Settings” section of the Kamui 2 Reference Manual.

Setting the Punchthrough Threshold

A punch-through polygon is a polygon that is completely transparent in some places and completely opaque in others. The Sega Deramcast graphics system can draw punchthrough polygons much faster than it can draw translucent polygons, so it’s a good idea to consider using punchthrough polygons rather than translucent polygons when you want part of a scene that lies behind a given graphics layer to show through the layer in front of it — for example, you could use punch-through polygons to make the background of a scene show through a line of trees. (For a more detailed explanation of how punchthrough polygons and translucent polygons work, see Section 4.2.1 of the Sega Dreamcast Roadmap and Overview manual. For information on how to set up the Sega Dreamcast system to sort and display translucent polygons, see Sorting Translucent Polygons on page 15 .)

To determine whether a polygon should be displayed as a punch-through polygon, place it in a PunchThrough list and then call the `kmSetPunchThroughThreshold` function. If the alpha value of data loaded to the PunchThrough list exceeds the threshold passed to `kmSetPunchThroughThreshold`, the polygon is treated as a punchthrough polygon.

Setting the Texture Stride Width

The `kmSetStrideWidth` function sets the stride size when the stride texture is used. The stride size must be a multiple of 32. The value that can be set is a multiple of 32 in the range of 32 to 992.

Setting the Border Color

In Kamui 2, the border color — that is, the area of the video screen that surrounds the display area — is black by default. You can set the border area to a different color by calling `kmSetBorderColor.`

Initializing Global Palette Settings

The Sega Dreamcast palette has 1,024 entries, which can be set to produce either 16-bit or 32-bit colors. Whether the bit depth is set to 16 bits or 32 bits, the number of entries in the palette is the same, as shown in Figure 2.3.

As you can see, palettized textures can be created using either a 4bpp mode or a 8bpp mode. When 4bpp mode is used, the 1,024 entries in the palette are are divided into 64 banks (1,024 entries/16 colors = 64 banks). When 8bpp mode is used, the 1,024 entries are divided into four banks (1,024 entries/256 colors = 4 banks). The banks are not separated physically; each bank is created by calculating pointers to the 1,024 entries in the palette.

In the Sega Dreamcast system, 4-bpp palettized textures and 8-bpp palettized textures can exist together in the same scene, but when this option is chosen, the overlapping portion of the 1,024 entries is shared, so changing the contents of a palette affects both the 4-bpp and 8-bpp textures.

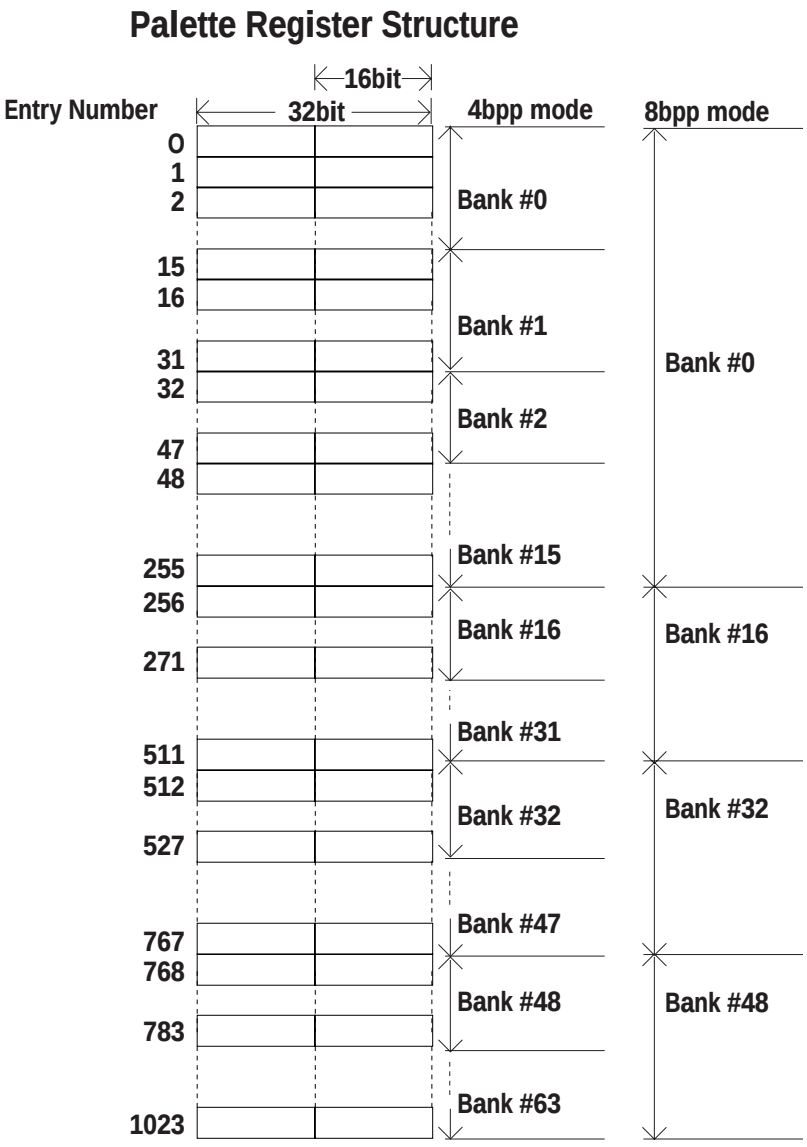


Figure 2.3 Stucture of the Kamui 2 Palette Register

The following functions can be used during the setting of global parameters to specify Sega Dreamcast palette settings:

- `kmSetPaletteData`: Sets the on-chip palette data. This function takes one parameter: a `PKMPALETTEDATA` (palette table) structure.
- `kmSetPaletteMode`: Specifies a mode of the on-chip palette used by the palettized texture.
- `kmSetPaletteBank` and `kmSetPaletteBankData`:Each of these functions rewrites a specified portion of the on-chip palette used by a palettized texture.

For a much more detailed explanation of the Sega Dreamcast color palette, see the entries for these three functions in the Kamui 2 Reference Manual.

Setting Pseudo-Global Scene Parameters

Some scene parameters are called pseudo-global because they do not affect entire scenes but can affect multiple strips, rather than just the strip currently being initialized. Unlike ordinary triangle-strip parameters — which must be reset each time you start a new strip — pseudo-global scene parameters remain in effect from one strip to the next. Pseudo-global scene parameters that can be set in Kamui 2 include:

- Background parameters. (In Kamui 2, the function that controls the background is `kmSetBackground`, which replaces the `kmSetBackgroundPlane` function used in Kamui 1).
- The mechanism used for sorting translucent polygons.
- Cheap shadow mode. (In Kamui 2, the `kmSetCheapShadowMode` function configures the Sega Dreamcast hardware for cheap shadow mode.
- User tile clipping, which is controlled by the `kmSetUserClipping` command.

Setting Background Parameters

To set up the background of a scene, call `kmSetBackground`, which specifies the vertex and `CONTEXT` for use in the background. The `kmSetBackground` function replaces the `kmSetBackgroundPlane` API function used in Kamui 1.

Sorting Translucent Polygons

Dreamcast provides a `kmSetAutoSortMode` function that can automatically sort translucent polygons on a pixel-by-pixel basis, but it should be used with caution. When you call `kmSetAutoSortMode`, the algorithm that it uses to differentiate levels of translucent triangles drawn in front of each other can execute quite slowly in complex scenes, so it is sometimes better to sort translucent polygons beforehand in your own code, and then pass your strip to the hardware for rendering, than it is to let the Dreamcast hardware sort your translucent polygons automatically.

Another option you may sometimes be able to get away with is to construct your scene using punch-through polygons instead of translucent polygons when that is possible. (For more information about translucent polygons and punch-through polygons, see Section 4.2 of the Sega Dreamcast Roadmap and Overview Manual . Also see the subsection titled Setting the Punchthrough Threshold on page 13 .)

When you have decided whether you want to perform automatic sorting of translucent polygons or not, call the `kmSetAutoSortMode` function, which switches between the auto-sort mode and presort modes for drawing translucent polygons. For a complete understanding of how to use the auto-sort mode and the presort mode, you should also be familiar with multipass rendering, a technique for sending polygon lists to the vertex buffer during multiple passes of the same rendering operation. For an explanation of how multipass rendering is used in Sega Dreamcast applications, see the section on Multipass Rendering on page 20 .

Using Cheap Shadow Mode

One of the most useful features built into the Sega Dreamcast graphics hardware is support for modifier volume — a mechanism that can yield remarkable effects when used to display shafts of light or the casting of dramatic shadows. To determine an area that lies inside a modifier volume, an application must divide the polygon into two sections — one that lies inside the modifier volume, and one that lies outside it. A polygon divided in this way is sometimes called a two-parameter polygon. To render the two sections of the polygon, the application calls `kmSetVertexRenderState` for one area of the polygon, and `kmSetModifierRenderState` for the other. (For more information about the modifier volume mechanism and how it works, see the Kamui 2 Reference Manual and Section 4.4 of the Sega Dreamcast Roadmap and Overview manual.)

Cheap shadow mode is a mechanism that can use a subset of the features provided by modifier volume to produce shadows that can still look quite realistic but require less overhead. Cheap shadow mode avoids the use of two-parameter polygons by creates shadows in a different way. Instead of using two-parameter polygons, cheap shadow mode uses ordinary one-parameter polygons but simply lowers their luminance when they are inside the modifier volume.

To use cheap shadow mode, call `kmSetCheapShadowMode`. The `kmSetCheapShadowMode` function takes one parameter — a value ranging from 0 to 255 — that sets the luminance of polygons lying inside the modifier volume.

Controlling User Tile Clipping

Because the Sega Dreamcast system segments the screen into tiles (as explained in Section 4.3 of the Sega Dreamcast Roadmap and Overview manual), the ordinary clipping techniques that developers are accustomed to have been supplemented by some additional kinds of clipping.

The Dreamcast system supports two kinds of clipping: tile clipping, which clips polygons displayed in adjacent tiles, and pixel clipping, the familiar kind of clipping that is applied when polygons extend past the edge 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 tiles and 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 screejn area outside the clip area intact but clipping off the edges of polygons that extend into it. (For more details, see Section 4.3.1 of the Sega Dreamcast Roadmap and Overview manual.)

The `kmSetUserClipping` function sets up the user clipping area. For more information on how to use the `kmSetUserClipping` function — and for a list of parameters expected by`kmSetUserClipping` — see the Kamui 2 Reference Manual.

Important: The number of parameters that you can pass to `kmSetUserClipping` has been expanded in Kamui 2. So before you call `kmSetUserClipping`, be sure to check the entry describing the function in the Kamui 2 Reference Manual.

Generating and Using a Strip Head Structure

In Kamui 2, an application initializes rendering parameters for triangle strips by calling the `kmGenerateStripHead` function. As you might guess from its name, `kmGenerateStripHead` generates a structure called a strip head structure (`KMSTRIPHEAD`)

The `kmGenerateStripHead` function describes the characteristics of a triangle strip by taking information contained in a strip-context structure (`KMVERTEXCONTEXT`) and placing it in a different kind of structure called a strip-head structure (`KMSTRIPHEAD`). As explained under the next subheading, Creating and Using the Display List, a `KMSTRIPHEAD` structure is a set of four highly compacted binary-readable "control words" that can be passed directly to the Sega Dreamcast graphics hardware for rendering.

When `kmGenerateStripHead` has finished converting your application's `KMVERTEXCONTEXT` structure into a `KMSTRIPHEAD`, the application calls `kmStartStrip` (for direct transfer) to start the strip: that is, to write the rendering parameters that have been constructed in `KMSTRIPHEAD` to the specified vertex buffer.

Creating and Using the Display List

To produce a strip head structure, the `kmGenerateStripHead` function generates four hardware-readable control words. These four words are then passed to the Sega Dreamcast hardware in the form of a display list, as shown in Figure 2.4. The four control words generated by `kmGenerateStripHead` are:

- A parameter control word: Supplies settings for global parameters and control parameters. Global parameters contain render state information, such as:
  - Data describing the render state,
  - Gouraud shading and other kinds of shading,
  - Mipmap information, and so on. Control parameters are settings for the user clip region and related objects.
- An ISP/TSP (Image Synthesis Processor/Texture and Shading Processor) instruction: Supplies ISP (Image Synthesis Processor) and TSP (Texture and Shading Processor) parameter information, and vertices that define an internal triangle strip.
- A TSP control word: Supplies additional information needed by the Texture and Shading Processor.
- A texture control word: A pointer to a texture parameter, which supplies a texture surface description.

Figure 2.4 shows how the control words generated by `kmGenerateStripHead` are passed to the Sega Dreamcast hardware as a display list, and how the display list is then processed by the hardware.

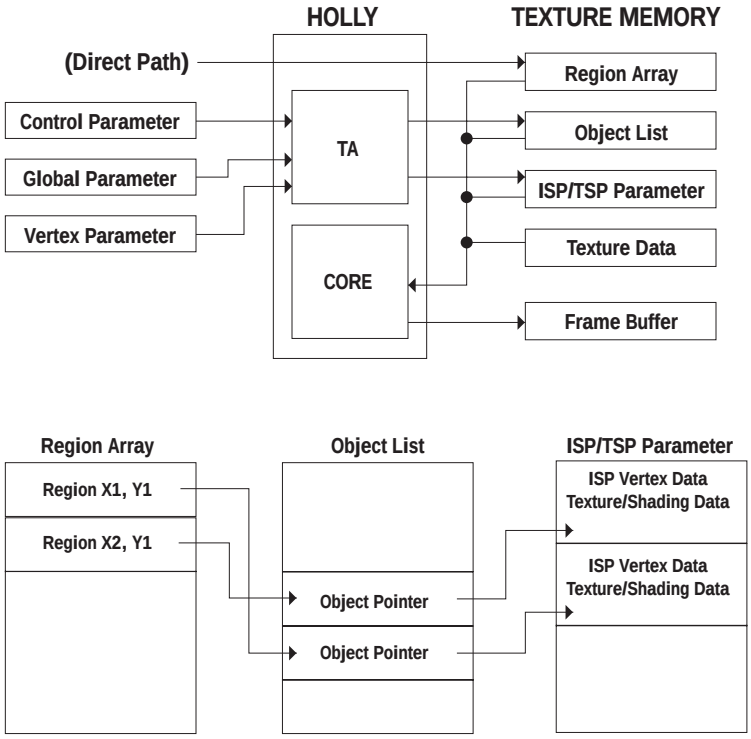


Figure 2.4 The Sega Dreamcast display list.

Components Shown in the Diagram

Here are explanations of the major components shown in Figure 2.4:

- The region array: An array of 32x32-pixel tiles containing head Object pointers for each of the five list types mentioned under the heading, The Vertex Data Buffer on page 18 .
- The object pointer list: A linked list of pointers to objects that have the potential of being included in each tile. This list is generated by the TA bounding box algorithm.
- Objects in the object list: ISP/TSP parameter words and vertices that define an internal triangle strip.

The Vertex Data Buffer

When a strip-head structure has been generated, a Kamui 2 application allocates a vertex data buffer area in system memory, enters vertex data in the vertex data buffer area, and issues a rendering command. The vertex data buffer area is divided into five buffers:

- Opaque polygon: Buffer for opaque polygons.
- Opaque modifier: Buffer for opaque modifier volumes.
- Translucent polygon: Buffer for translucent/transparent polygons.
- Translucent modifier: Buffer for translucent/transparent modifier volumes.
- Punchthrough polygon: Buffer for punchthrough polygons.

An application can store vertex data in any of these five buffers. Alternatively, an application can use the tile accelerator to write vertex data directly to the hardware without allocating any of the five available vertex data buffers.

When an application stores data in a vertex data buffer and writes it to the hardware later, the application is using what is known as buffer mode. Figure 2.5 illustrates the use of buffer mode.

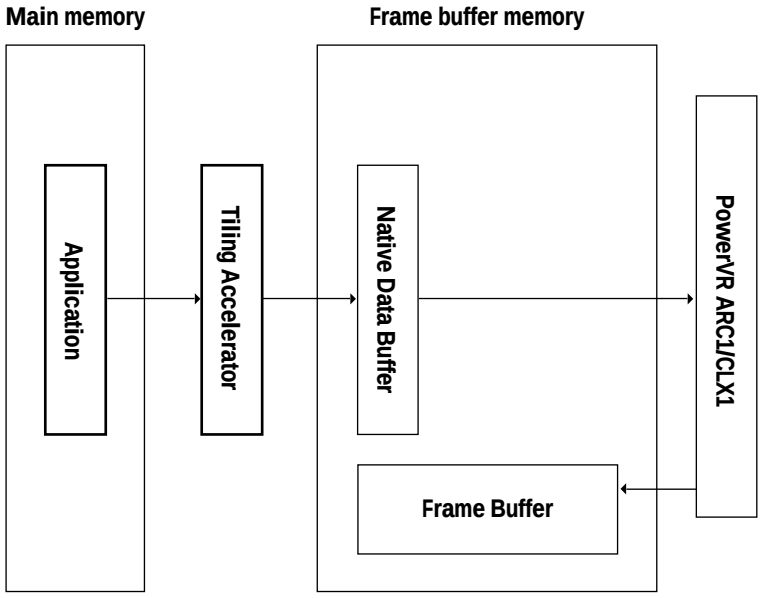


Figure 2.5 Rendering using buffer mode.

The method of writing vertex data directly to the hardware is referred to as direct mode. Figure 2.6 illustrates the use of direct mode.

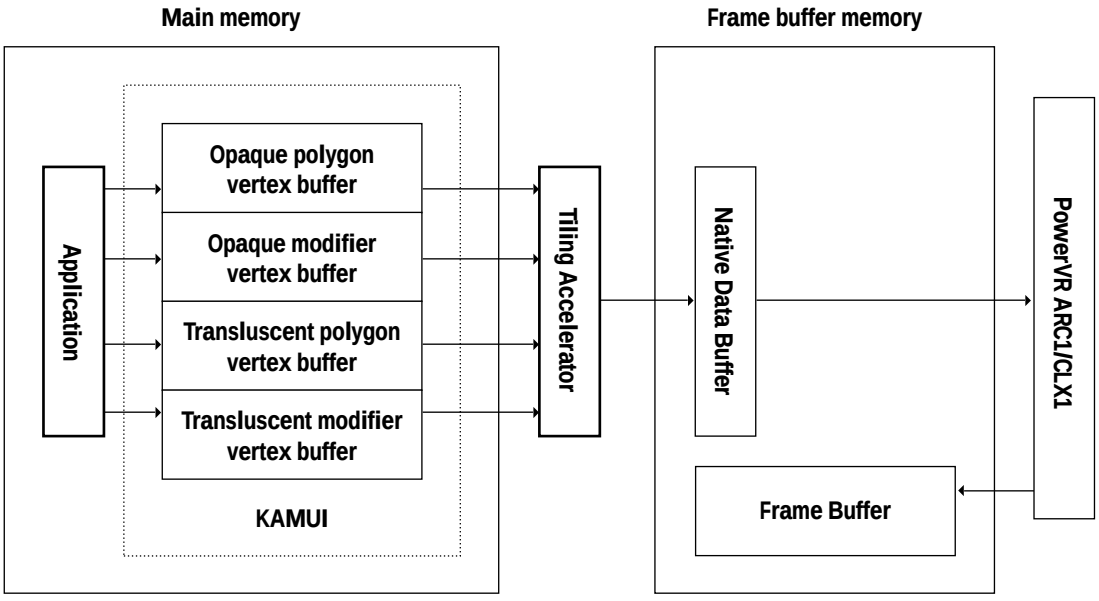


Figure 2.6 Rendering using direct mode.

The direct mode method uses a smaller memory area than the other method. However, the application program must supply each type of vertex data (opaque polygon, opaque modifier, translucent polygon, translucent modifier, and punchthrough polygon) in sorted order for the same scene to the hardware (tile accelerator).

Multipass Rendering

As noted earlier, in the subsection titled Initializing Settings for Multi-Pass Rendering on page 10 , multipass rendering is a technique for writing multiple polygon lists to the vertex buffer during multiple passes of a rendering operation, rather than having to start a new rendering operation. Three examples of using multipass rendering in a Sega Dreamcast application are:

- Switching between auto-sorting and pre-sorting.
- Grouping translucent objects for rendering.
- Changing the setting of the Z clipping plane during a rendering operation.

Switching between Auto-sorting and Presorting

As mentioned previously, in the subsection titled Initializing Global Clipping Settings on page 13 , you can switch between an auto-sort mode and a presort mode for drawing translucent polygons by calling the `kmSetAutoSortMode` function. When you want to display an object using translucent polygons, and you do not want your object to overlap objects that surround it, calculating the best Z value to use can be very difficult. This difficulty arises because all translucent polygons are drawn in auto-sort mode in a Sega Dreamcast application, and when auto-sort is specified, translucent ZCompareMode is fixed at the GE (Greater or Equal) setting.

By using multipass filtering, you can solve this probem quite easily. Simply do a rendering pass in which you draw your translucent polygons with ZCompareMode in the Always setting. Then set up the rest of your polygons for drawing in a separate pass, and make that pass in presort mode.

Grouping Translucent Objects for Rendering

When drawing a large number translucent polygons in auto-sort mode, a bottleneck may occur not with the fill rate, but with rendering. This phenomenon often occurs when small polygons from many viewpoints overlap in the Z direction. You can eliminate this bottleneck by grouping translucent polygons together and then drawing them in separate passes of the same rendering operation using multipass rendering. This technique works with objects that are far from your point of view because it yields almost the same drawing effect that you would get if you sorted all of the translucent polygons. This occurs because the synthesis of the nth pass and the n + 1 pass is carried out by alpha blending of registration. Consequently, when objects between passes wait at a cross-section (an intersected area), the drawing results are essentially invalid.

Changing the Z Clipping Plane Setting During a Rendering Operation

There may be times during your Sega Dreamcast application when you want complete separation of 3D models of distant views — for example, between the inside of a cockpit and faraway objects seen through the cockpit window. In such cases, you can use multipass rendering to reset the Z clipping value during a rendering operation without starting a new render. You can perform this operation using three passes:

- Pass 1: Draw the distant-view model.
- Pass 2: Perform another drawing passafter adjusting the transparent Z clipping polygon in Always to the value of the Z value the you want to clip.
- Pass 3: Draw close-up model.

In this kind of operation, the Z value is reset in the second pass. Thereafter, the close-up Z value is drawn based on the Z value of the resultant clipping plane.

The 2V and 3V Latency Models

To help your application perform vertex registration in a way that is best suited to its particular requirements, the Sega Dreamcast system provides two different latency models: a 3V latency model and a 2V latency model. By default, the Dreamcast operates in 3V latency mode. If you decide that 2V latency mode is a better match for your application's particular behavior and requirements, you can switch to the 2V latency models by calling a changeover function.

Note: In Kamui 2, the terms "2V latency model" and "3V latency model" were accurate and easily understood terms because the operations used in rendering and DMA operations were directly related to vertical blank interrupts. In Kamui 2, rendering operations are not so closely tied to vertical blank interrupts — in fact, you can specifically instruct the Sega Dreamcast system not to wait for vertical blanks by setting the render control flag `KM_CONFIGFLAG_NOWAITSYNC` flag. Because of this change, the terms "2V latency model" and "3V latency model" no longer not describe Sega Dreamcast rendering and DMA operations as accurately as they once did. However, for the sake of consistency — and because these two terms still do indicate, in a rough fashion, how vertex-registration operations work — the terms "2V latency model" and "3V latency model" continue to be used by Sega Dreamcast developers.

When you use the default 3V latency model for your application, Kamui 2 splits each rendering operation into three vertical blank periods. These three Vblank periods are assigned as follows:

- The first Vblank period is used for vertex data registration.
- The next Vblank is used for a DMA transfer operation (that is, transferring vertex data to the tile accelerator).
- A third Vblank is used for scene rendering.

The 2V latency model uses just two Vblank periods, instead of three, for each rendering operation. To achieve this saving, it assigns two Vblank periods as follows:

- One Vblank period is used both for vertex data registration and for a DMA transfer operation.
- A second Vblank is used for scene rendering.

Because the 3V latency model uses only two V-blank periods for each registration and rendering operation instead of three, response to game controllers can be faster. Also, the 2V latency model uses less system memory than the 3V model.

Comparing the 2V and 3V Latency Models

This section compares the advantages and disadvantages of using the 2V latency model and the 3V latency model.

Advantages of Using the 3V Latency Model

The designers of the Sega Dreamcast system made the 3V latency model the default model because they felt it would yield good performance under most conditions. These are some of the advantages it offers:

- Because each rendering operation takes exactly three Vblank periods to complete, the 3V latency model ensures that appliction performance will be smooth and consistent.
- When you use the 3V latency model, Kamui 2 ensures that vertex data is registered properly; that is, it makes sure that the five different kinds of polygons mentioned earlier (in The Vertex Data Buffer on page 18) are passed to the graphics hardware in the correct order. This registration of vertex data takes place during the first V-blank period used by the 3V latency model. (In applications written for

the Sega Dreamcast system, each kind of polygon must be passed to the hardware in a specific order, and when you don't use the 3V latency model, it is your responsibility to ensure that this order is followed so that all polygons are registered properly.)

- The 3V latency model does not require your application to make an explicit assignment of a vertex buffer for the kind of polygons (typically, opaque polygons) that the application uses most frequently. But it does require the use of vertex buffers, so it reserves space for those buffers in system memory.
- When you use the 3V latency model, the transfer of vertex data to the tile accelerator's native buffer takes advantage of the high-speed burst mode offered by the Sega Dreamcast system's DMA transfer mechanism. This DMA transfer takes place in during the second V-blank period used by the 3V latency model. (During the third and last V-blank period, scene rendering takes place.

Advantages of Using the 3V Latency Model

These are some of the main benefits of choosing the 2V latency model:

- Because the 2V latency model takes one-third less time for each rendering operation than the 3V latency model, it offers a quicker response to control-pad events. In games that require very fast control-panel response, this difference can be very noticeable. In the 3V latency model, as many as four Vblank periods can pass before the system recognizes a control-panel action by the user. By switching to 2V latency mode, you can ensure that the video display does not lag behind the user's control-panel manipulations by any more than two frames.
- The 2V latency model uses less system memory than the 3V latency model. In fact, if you optimize your code in such a way that rendering never takes more than two Vblank periods, your application's vertex-rendering operations won't use any system memory at all!

Points to Consider when Using the 2V Latency Model

Although the 2V latency model may offers certain advantages, it also places more responsibility on the programmer. If you decide to use the 2V latency model in your application, these are some points you should be aware of:

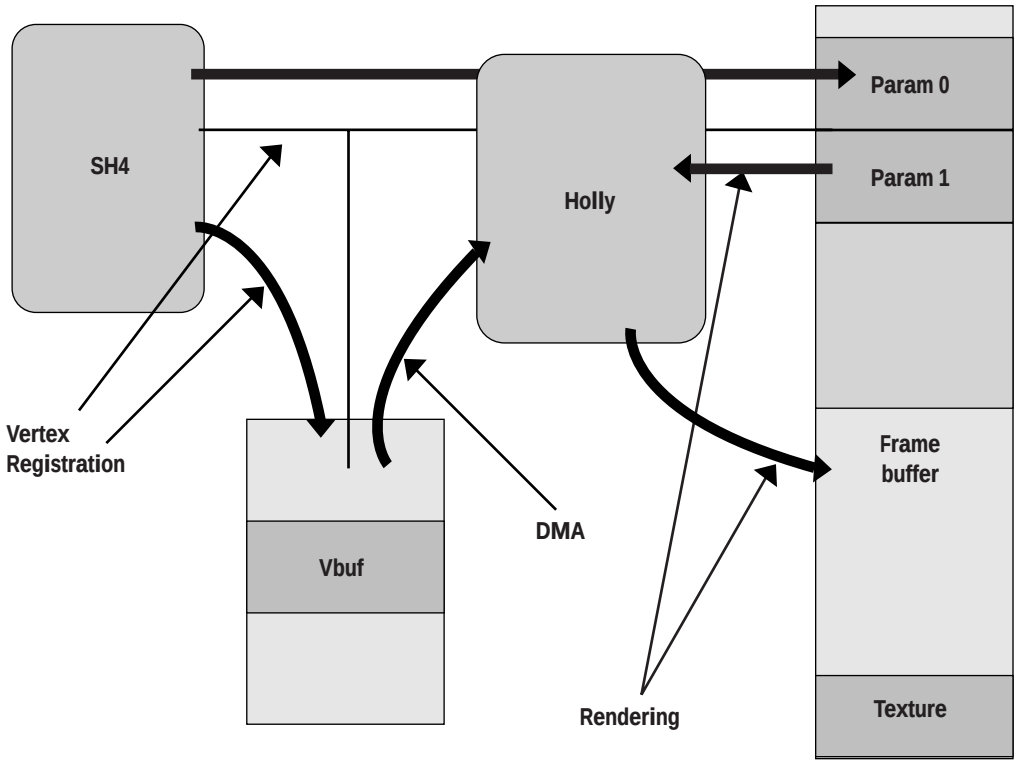
- First, because Kamui 2 uses the 3V latency model by default, you will have to start up the 2V model yourself if you want to use it. (To initialize the 2V latency model, you set the `CONFIGFLAG_ENABLE_2V_LATENCY` flag before calling `kmSetSystemConfiguration`; for an example of how to do that, see The Structure QuikTest2.c Program on page 26 .)
- When you use the 2V latency model is the fact that you must pre-sort your polygons in such a way that polygons are sent to the hardware in the correct order (that is, in the order shown in the bulleted list under the heading The Vertex Data Buffer on page 18). In contrast, when you use the 3V latency model, all triangle sorting is performed for you automatically.
- When using the 2V latency model, you can use choose one vertex data type to be sent directly to the tile accelerator using the Store Queue mechanism. (Because most scenes typically use far more opaque polygons than any other kind, the data type chosen for direct transfer is usually opaque polygons.) The remaining vertex buffers are then DMA'd to the tile accelerator's native buffer. All these operations take place during just one V-blank period used by the 2V latency model, so you must make sure that your application does not overload the DMA pipeline. One way you can stall the pipeline is by expecting it to DMA a scene that contains a large number of translucent polygons.
- Alternatively, you can batch up a vertex data type and periodically call `kmFlushVertexBuffer` to transfer that data type to the tile accelerator using the Dreamcast system's DMA transfer mechanism. All other data types are then buffered in system memory.

Program Pipelining of the 2V and 3V Latency Models

This section presents diagrams that compare the program pipelining of the 2V and 3V latency models.

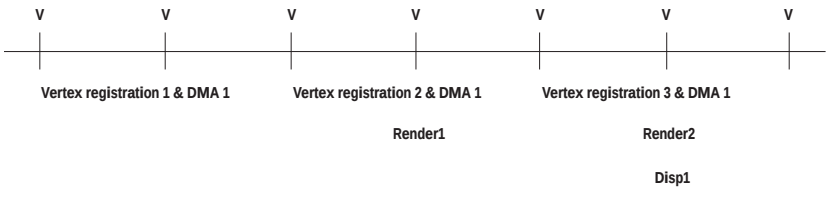
The 2V Latency Model Programming Pipeline

The following diagram shows how pipelining in the 2V latency model compresses vertex data registration and DMA processing within one V period:



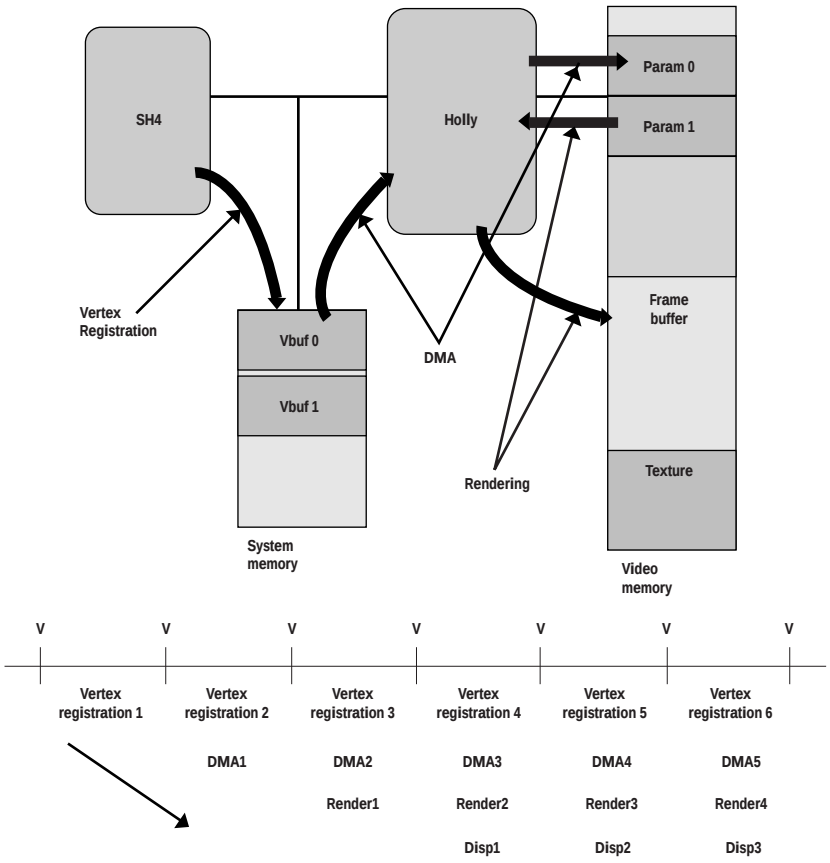
Vertex data registration in the 2V latency programming model can be performed using the three methods explained below.

The first method supplies a list of specific vertex types directly to the TA. Assuming that the data type most frequently used throughout a scene is the opaque polygon, the opaque data is sent directly to the TA. Data registered in other lists (opaque modifier, translucent, translucent modifier) is temporarily stored in memory. When kmRender is executed, vertices in these lists are transferred by DMA. If a DMA transfer does not end within a frame where it begins, a prolonged process occurs.



The 3V Latency Model Programming Pipeline

The following diagram shows how the programming pipeline behaves when an application uses the 3V latency model:



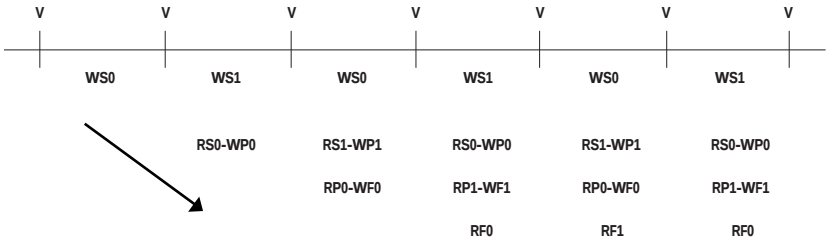
Vertex registration:	Represents the registration of the vertex data that constitutes a scene. It includes a game sequence, geometry, and the creation of data to be sent to the TA.
DMA:	Data is sent to texture memory via the TA using a DMA mode.
Render:	Rendering is carried out as directed by parameters generated by the TA.
Disp:	The results of rendering is displayed.

Frame buffer double buffering requires the following hardware resources:

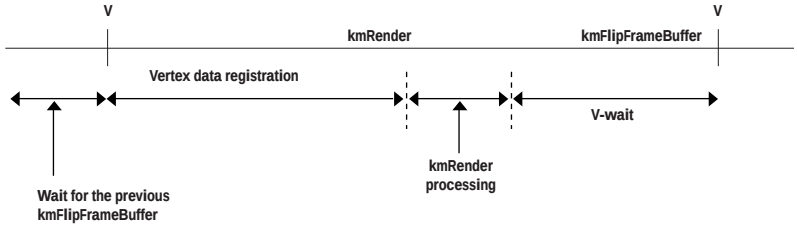
1) SH4 system memory side
   Two vertex data buffers (one buffer for one scene)
2) CLX (ARC1) side memory
   Two native data buffers that match CLX's internal parameters (one buffer for one scene)
   Two display frame buffers
   Texture memory

The pipeline mentioned above uses memory as follows:

WS:	Writing to system memory (0 or 1) by SH4
RS:	Reading from system memory (0 or 1) by DMA
WP:	Writing to CLX parameter memory (0 or 1) by DMA
RP:	Reading from CLX parameter memory (0 or 1) by renderer
WF:	Writing to CLX frame buffer (0 or 1) by renderer
RF:	Reading from CLX frame buffer (0 or 1) for display by renderer



Use of the pipeline does not cause contention for hardware resources and can minimize arbitration for 60 fps.

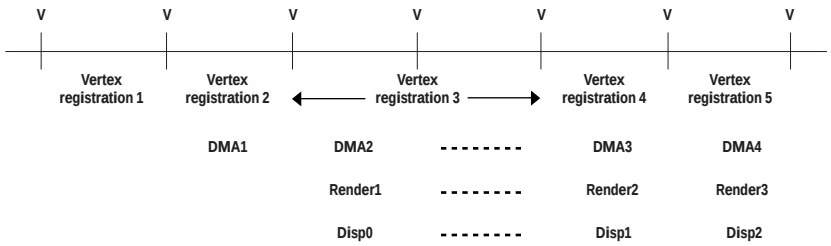


3V Latency Model Processing Overflow

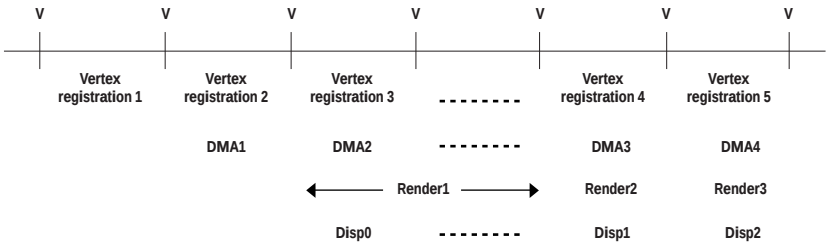
If a pipeline process does not fit within one V blank (that is, if processing overflows), pipelining is continued by allowing the process to be prolonged.

If rendering does not end within one V blank, it is possible to reset the renderer to start rendering another scene (except for COSMOS+ARC1). If rendering is discontinued forcibly during a scene, the scene cannot be completed, resulting in an invalid scene.

An example of a prolonged vertex processing period is shown below.



In this example, vertex registration 3 is prolonged, thus deferring DMA 3, render 2, and display 1.



In the example shown above, render 1 takes additional time, causing other processes to wait.

The Structure QuikTest2.c Program

This section examines the structure of a very basic Kamui 2 program: the `QuikTest2.c` sample program, which ships with Kamui 2. The `QuikTest2.c` application was designed to demonstrate the architecture and implementation of a bare-bones Kamui 2 program.

This section highlights three parts of the `QuickTest2.c` program:

- Setting the System Configuration on page 27 .'
- Loading Textures on page 27 .
- Strip Rendering on page 28 .
- The Main Render Loop on page 29 .

The source code of the `QuikTest2.c` program is presented in its entirety under the heading Source Code of the QuikTest2.c Program on page 34 .

Setting the System Configuration

The first section of the QuikTest2.c program shows to set up the system configuration for a Kamui 2 program using the KMSYSTEMCONFIGSTRUCT structure:

```
KMSYSTEMCONFIGSTRUCT kmsc;      /* Kamui2 system configuration structure. */

/* Set size of system configuration struct, required. */
    kmsc.dwSize = sizeof(KMSYSTEMCONFIGSTRUCT);

    /* Render control flags. */
    kmsc.flags  = KM_CONFIGFLAG_ENABLE_CLEAR_FRAMEBUFFER    /* Clear FB at initialization. */
    //            | KM_CONFIGFLAG_NOWAITVSYNC                /* Don't wait for the VBlank. */
                | KM_CONused FIGFLAG_ENABLE_2V_LATENCY;         /* Use 2V latency render model. */

/* Frame buffer surfaces information. */
    FBSurfaces[0] = &PrimarySurfaceDesc;
    FBSurfaces[1] = &BackSurfaceDesc;
    kmsc.ppSurfaceDescArray = FBSurfaces;   /* Set frame buffer surface description array. */
    kmsc.fb.nNumOfFrameBuffer = 2;          /* Number of frame buffers (double buffered). */
    kmsc.fb.nStripBufferHeight = 32;        /* Strip buffer height = 32xN pixels, if used. */

    /* Texture handler setup. */
    kmsc.nTextureMemorySize = TEXTUREMEMORYSIZE;    /* Texture VRAM, divisible by 32 bytes. */
    kmsc.nNumOfTextureStruct = MAXTEXTURES;         /* Maximum number of textures tracked. */
    kmsc.nNumOfSmallVQStruct = MAXSMALLVQ;          /* Max. number of small VQ textures. */
    kmsc.pTextureWork = TextureWorkArea;            /* Texture work area in system memory. */

    /* Generate 32-byte aligned vertex buffer (double-buffered). */
    VertexBufferPtr = (PKMDWORD) Align32Malloc (2 * VERTEXBUFFERSIZE * sizeof (KMDWORD));

    /* Kamui2 requires the vertex buffer region to be 32-byte aligned and non-cacheable. */
    kmsc.pVertexBuffer = (PKMDWORD) SH4_P2NonCachedMem (VertexBufferPtr);

    /* Define vertex buffer. */
    kmsc.nVertexBufferSize = VERTEXBUFFERSIZE * 2;  /* Size of vertex buffer X2, in bytes. */
    kmsc.pBufferDesc = &VertexBufferDesc;   /* Struct used to maintain vertex buffer info. */
    kmsc.nNumOfVertexBank = 2;              /* Number of vertex buffers (double buffered). */

    /* Set multi-pass rendering information. */
    kmsc.nPassDepth = 1;                                        /* Number of passes. */
    kmsc.Pass[0].dwRegionArrayFlag = KM_PASSINFO_AUTOSORT   /* Autosort translucent polys. */
        | KM_PASSINFO_ENABLE_Z_CLEAR;                           /* Clear Z buffer. */
    kmsc.Pass[0].nDirectTransferList = KM_OPAQUE_POLYGON;   /* Type sent direct in 2V mode. */

    /* Percent of vertex buffer used for each vertex type in a pass (must total 100%). */
    kmsc.Pass[0].fBufferSize[0] =  0.0f;        /* Opaque polygons (0% if sent direct). */
    kmsc.Pass[0].fBufferSize[1] =  0.0f;        /* Opaque modifier. */
    kmsc.Pass[0].fBufferSize[2] = 50.0f;        /* Translucent. */
    kmsc.Pass[0].fBufferSize[3] =  0.0f;        /* Translucent modifier. */
    kmsc.Pass[0].fBufferSize[4] = 50.0f;        /* Punchthrough. */

    /* Set system configuration. */
    kmSetSystemConfiguration (&kmsc);
```

Loading Textures

The next section of the QuikTest2.c program shows how you can create a texture surface and load textures in a Kamui 2 program.

```
  /* Create a texture surface. */
    kmCreateTextureSurface (&TexSurfaceDesc, 256, 256,
        (KM_TEXTURE_TWIDDLED | KM_TEXTURE_RGB565));

    /* Load a texture file from disk. */
```

```
    TexturePtr = LoadTextureFile ("FBI.pvr");

    /* The next few calls demonstrate how to algorithmically generate a texture.  In this   */
    /* case, a rectangle format texture is loaded from disk and twiddled at run-time.       */

    /* Allocate memory for twiddled texture (256x256 16-bit). */
    TwiddledPtr = (PKMDWORD) Align32Malloc (256 * 256 * 2);
                                                          s
    /* Run-time twiddle source rectangle texture + 16 bytes to skip PVRT chunk header. */
    Rectangle2Twiddled ((char *) ((long) TexturePtr + 16), (char *) TwiddledPtr, 256, 16);

    /* Transfer twiddled texture from system mem to video mem using a DMA transfer. */
    kmLoadTexture (&TexSurfaceDesc, TwiddledPtr);
```

Strip Rendering

The next section of `QuikTest2.c` demonstrates the generation and modification of a strip head structure.

```
/* Generate a strip head array for opaque color (no texture) polygons. */
    memset (&StripHead_01, 0, sizeof(StripHead_01));
    kmGenerateStripHead01 (&StripHead_01, &DefaultContext);

    /* Set the texture surface of the opaque polygon context to the newly loaded texture. */
    DefaultContext.ImageControl[KM_IMAGE_PARAM1].pTextureSurfaceDesc = &TexSurfaceDesc;

    /* Generate a strip head array for opaque textured polygons. */
    memset (&StripHead_05_Opaque, 0, sizeof(StripHead_05_Opaque));
    kmGenerateStripHead05 (&StripHead_05_Opaque, &DefaultContext);

    /* Modify the standard polygon context to support translucency. */
    DefaultContext.StripControl.nListType = KM_TRANS_POLYGON;
    DefaultContext.ImageControl[KM_IMAGE_PARAM1].bUseAlpha = KM_TRUE;
    DefaultContext.ImageControl[KM_IMAGE_PARAM1].nTextureShadingMode = KM_MODULATE_ALPHA;

    /* Generate a strip head array for translucent textured polygons. */
    memset (&StripHead_05_Trans, 0, sizeof(StripHead_05_Trans));
    kmGenerateStripHead05 (&StripHead_05_Trans, &DefaultContext);

    /* Just to demonstrate how a strip head can be modified once generated.. */
    kmChangeStripBlendingMode (&StripHead_05_Trans, KM_IMAGE_PARAM1,
        KM_SRCALPHA, KM_INVSRCALPHA);
```

The Main Render Loop

Here is an example of the main render loop of a Kamui 2 program:

```
/* The main render loop. */
    while (TRUE)
          {
        /* Get control pad info filled out by the 'PeripheralServer'. */
        per = (PDS_PERIPHERAL *) pdGetPeripheral (PDD_PORT_A0);

        /* Set the vertices of the background plane for a nice color gradient. */
        /* NOTE: Yes, you can change the background plane on-the-fly. */
        CurrentStrip_01[0].ParamControlWord     = KM_VERTEXPARAM_NORMAL;
        CurrentStrip_01[0].fX                   =   0.0f;
        CurrentStrip_01[0].fY                   = 479.0f;
        CurrentStrip_01[0].u.fZ                 =   0.2f;
        CurrentStrip_01[0].fBaseAlpha           =   1.0f;
        CurrentStrip_01[0].fBaseRed             =   0.0f;
        CurrentStrip_01[0].fBaseGreen           =   0.0f;
        CurrentStrip_01[0].fBaseBlue            =   0.0f;

        CurrentStrip_01[1].ParamControlWord     = KM_VERTEXPARAM_NORMAL;
        CurrentStrip_01[1].fX                   =   0.0f;
        CurrentStrip_01[1].fY                   =   0.0f;
        CurrentStrip_01[1].u.fZ                 =   0.2f;
        CurrentStrip_01[1].fBaseAlpha           =   1.0f;
        CurrentStrip_01[1].fBaseRed             =   Red;
        CurrentStrip_01[1].fBaseGreen           =   Green;
        CurrentStrip_01[1].fBaseBlue            =   Blue;

        CurrentStrip_01[2].ParamControlWord     = KM_VERTEXPARAM_ENDOFSTRIP;
        CurrentStrip_01[2].fX                   = 639.0f;
        CurrentStrip_01[2].fY                   = 479.0f;
        CurrentStrip_01[2].u.fZ                 =   0.2f;
        CurrentStrip_01[2].fBaseAlpha           =   1.0f;
        CurrentStrip_01[2].fBaseRed             =   0.0f;
        CurrentStrip_01[2].fBaseGreen           =   0.0f;
        CurrentStrip_01[2].fBaseBlue            =   0.0f;

        /* Set the background plane to the vertices filled in above. */
        kmSetBackGround (&StripHead_01, KM_VERTEXTYPE_01,
            &CurrentStrip_01[0], &CurrentStrip_01[1], &CurrentStrip_01[2]);

        /* Update colors - just to show we're doing something. */
        if (Blue < 0.99)
        {
            Blue += ColorDelta;
            if (ColorDelta < 0)
            {
                Red += ColorDelta;
                Green += ColorDelta;
            }
        }
        else
        {
            if (Red > 0.99)
            {
                ColorDelta = -ColorDelta;
                Blue += ColorDelta;
            }
            Red += ColorDelta;
            Green += ColorDelta;
        }

        if (Blue < 0.0)
        {
            Blue = 0.0;
            Red = 0.0;
            Green = 0.0;
            ColorDelta = 0.01f;
```

```
}

/* Tell Kamui2 to start accepting polygon data for a new scene. */
kmBeginScene (&kmsc);

/* Start the first pass (for multipass rendering). */
kmBeginPass (&VertexBufferDesc);

/* Start rendering opaque color strips. */
kmStartStrip (&VertexBufferDesc, &StripHead_01);

/* Set the vertices in the current strip to draw a colorful square. */
CurrentStrip_01[0].ParamControlWord     = KM_VERTEXPARAM_NORMAL;
CurrentStrip_01[0].fX                   = 170.0f;
CurrentStrip_01[0].fY                   = 305.0f;
CurrentStrip_01[0].u.fZ                 =  10.0f;
CurrentStrip_01[0].fBaseAlpha           =   1.0f;
CurrentStrip_01[0].fBaseRed             =   1.0f;
CurrentStrip_01[0].fBaseGreen           =   0.0f;
CurrentStrip_01[0].fBaseBlue            =   0.0f;

CurrentStrip_01[1].ParamControlWord     = KM_VERTEXPARAM_NORMAL;
CurrentStrip_01[1].fX                   = 170.0f;
CurrentStrip_01[1].fY                   = 175.0f;
CurrentStrip_01[1].u.fZ                 =  10.0f;
CurrentStrip_01[1].fBaseAlpha           =   1.0f;
CurrentStrip_01[1].fBaseRed             =   0.0f;
CurrentStrip_01[1].fBaseGreen           =   1.0f;
CurrentStrip_01[1].fBaseBlue            =   0.0f;

CurrentStrip_01[2].ParamControlWord     = KM_VERTEXPARAM_NORMAL;
CurrentStrip_01[2].fX                   = 310.0f;
CurrentStrip_01[2].fY                   = 305.0f;
CurrentStrip_01[2].u.fZ                 =  10.0f;
CurrentStrip_01[2].fBaseAlpha           =   1.0f;
CurrentStrip_01[2].fBaseRed             =   0.0f;
CurrentStrip_01[2].fBaseGreen           =   0.0f;
CurrentStrip_01[2].fBaseBlue            =   1.0f;

CurrentStrip_01[3].ParamControlWord     = KM_VERTEXPARAM_ENDOFSTRIP;
CurrentStrip_01[3].fX                   = 310.0f;
CurrentStrip_01[3].fY                   = 175.0f;
CurrentStrip_01[3].u.fZ                 =  10.0f;
CurrentStrip_01[3].fBaseAlpha           =   1.0f;
CurrentStrip_01[3].fBaseRed             =   0.0f;
CurrentStrip_01[3].fBaseGreen           =   0.0f;
CurrentStrip_01[3].fBaseBlue            =   0.0f;

for (i=0; i<=3; i++)
    kmSetVertex (&VertexBufferDesc, &CurrentStrip_01[i],
        KM_VERTEXTYPE_01, sizeof (KMVERTEX_01));

/* Done sending opaque color strip. */
kmEndStrip (&VertexBufferDesc);

/* Start rendering opaque textured strips. */
kmStartStrip (&VertexBufferDesc, &StripHead_05_Opaque);

/* Set the vertices in the current strip to draw a textured square. */
CurrentStrip_05[0].ParamControlWord     = KM_VERTEXPARAM_NORMAL;
CurrentStrip_05[0].fX                   = 250.0f;
CurrentStrip_05[0].fY                   = 240.0f;
CurrentStrip_05[0].u.fZ                 =  11.0f;
CurrentStrip_05[0].fU                   =   0.0f;
CurrentStrip_05[0].fV                   =   1.0f;
CurrentStrip_05[0].fBaseAlpha           =   1.0f;
CurrentStrip_05[0].fBaseRed             =   0.0f;
CurrentStrip_05[0].fBaseGreen           =   0.15f;
CurrentStrip_05[0].fBaseBlue            =   0.73f;

CurrentStrip_05[1].ParamControlWord     = KM_VERTEXPARAM_NORMAL;
CurrentStrip_05[1].fX                   = 250.0f;
```

```
CurrentStrip_05[1].fY                   = 110.0f;
CurrentStrip_05[1].u.fZ                 =  11.0f;
CurrentStrip_05[1].fU                   =   0.0f;
CurrentStrip_05[1].fV                   =   0.0f;
CurrentStrip_05[1].fBaseAlpha           =   1.0f;
CurrentStrip_05[1].fBaseRed             =   0.55f;
CurrentStrip_05[1].fBaseGreen           =   0.87f;
CurrentStrip_05[1].fBaseBlue            =   0.96f;

CurrentStrip_05[2].ParamControlWord     = KM_VERTEXPARAM_NORMAL;
CurrentStrip_05[2].fX                   = 390.0f;
CurrentStrip_05[2].fY                   = 240.0f;
CurrentStrip_05[2].u.fZ                 =  11.0f;
CurrentStrip_05[2].fU                   =   1.0f;
CurrentStrip_05[2].fV                   =   1.0f;
CurrentStrip_05[2].fBaseAlpha           =   1.0f;
CurrentStrip_05[2].fBaseRed             =   0.55f;
CurrentStrip_05[2].fBaseGreen           =   0.87f;
CurrentStrip_05[2].fBaseBlue            =   0.96f;

CurrentStrip_05[3].ParamControlWord     = KM_VERTEXPARAM_ENDOFSTRIP;
CurrentStrip_05[3].fX                   = 390.0f;
CurrentStrip_05[3].fY                   = 110.0f;
CurrentStrip_05[3].u.fZ                 =  11.0f;
CurrentStrip_05[3].fU                   =   1.0f;
CurrentStrip_05[3].fV                   =   0.0f;
CurrentStrip_05[3].fBaseAlpha           =   1.0f;
CurrentStrip_05[3].fBaseRed             =   0.0f;
CurrentStrip_05[3].fBaseGreen           =   0.15f;
CurrentStrip_05[3].fBaseBlue            =   0.73f;

for (i=0; i <= 3; i++)
    kmSetVertex (&VertexBufferDesc, &CurrentStrip_05[i],
        KM_VERTEXTYPE_05, sizeof (KMVERTEX_05));

/* Done sending opaque texture strip. */
kmEndStrip (&VertexBufferDesc);

/* Start translucent textured strips. */
kmStartStrip (&VertexBufferDesc, &StripHead_05_Trans);

/* Set the vertices in the current strip to draw a translucent textured square. */
CurrentStrip_05[0].ParamControlWord     = KM_VERTEXPARAM_NORMAL;
CurrentStrip_05[0].fX                   = 250.0f;
CurrentStrip_05[0].fY                   = 370.0f;
CurrentStrip_05[0].u.fZ                 =  21.0f;
CurrentStrip_05[0].fU                   =   0.0f;
CurrentStrip_05[0].fV                   =   1.0f;
CurrentStrip_05[0].fBaseAlpha           =   0.25f;
CurrentStrip_05[0].fBaseRed             =   0.0f;
CurrentStrip_05[0].fBaseGreen           =   0.15f;
CurrentStrip_05[0].fBaseBlue            =   0.73f;

CurrentStrip_05[1].ParamControlWord     = KM_VERTEXPARAM_NORMAL;
CurrentStrip_05[1].fX                   = 250.0f;
CurrentStrip_05[1].fY                   = 240.0f;
CurrentStrip_05[1].u.fZ                 =  21.0f;
CurrentStrip_05[1].fU                   =   0.0f;
CurrentStrip_05[1].fV                   =   0.0f;
CurrentStrip_05[1].fBaseAlpha           =   0.25f;
CurrentStrip_05[1].fBaseRed             =   0.55f;
CurrentStrip_05[1].fBaseGreen           =   0.87f;
CurrentStrip_05[1].fBaseBlue            =   0.96f;

CurrentStrip_05[2].ParamControlWord     = KM_VERTEXPARAM_NORMAL;
CurrentStrip_05[2].fX                   = 390.0f;
CurrentStrip_05[2].fY                   = 370.0f;
CurrentStrip_05[2].u.fZ                 =  21.0f;
CurrentStrip_05[2].fU                   =   1.0f;
CurrentStrip_05[2].fV                   =   1.0f;
CurrentStrip_05[2].fBaseAlpha           =   0.25f;
CurrentStrip_05[2].fBaseRed             =   0.55f;
```

```
CurrentStrip_05[2].fBaseGreen          =   0.87f;
CurrentStrip_05[2].fBaseBlue           =   0.96f;

CurrentStrip_05[3].ParamControlWord    = KM_VERTEXPARAM_ENDOFSTRIP;
CurrentStrip_05[3].fX                  = 390.0f;
CurrentStrip_05[3].fY                  = 240.0f;
CurrentStrip_05[3].u.fZ                =  21.0f;
CurrentStrip_05[3].fU                  =   1.0f;
CurrentStrip_05[3].fV                  =   0.0f;
CurrentStrip_05[3].fBaseAlpha          =   0.25f;
CurrentStrip_05[3].fBaseRed            =   0.0f;
CurrentStrip_05[3].fBaseGreen          =   0.15f;
CurrentStrip_05[3].fBaseBlue           =   0.73f;

/* NOTE: We can keep adding vertices, even past an ENDOFSTRIP flag - see below. */
CurrentStrip_05[4].ParamControlWord    = KM_VERTEXPARAM_NORMAL;
CurrentStrip_05[4].fX                  = 330.0f;
CurrentStrip_05[4].fY                  = 305.0f;
CurrentStrip_05[4].u.fZ                =  20.0f;
CurrentStrip_05[4].fU                  =   0.0f;
CurrentStrip_05[4].fV                  =   1.0f;
CurrentStrip_05[4].fBaseAlpha          =   0.75f;
CurrentStrip_05[4].fBaseRed            =   0.0f;
CurrentStrip_05[4].fBaseGreen          =   0.15f;
CurrentStrip_05[4].fBaseBlue           =   0.73f;

CurrentStrip_05[5].ParamControlWord    = KM_VERTEXPARAM_NORMAL;
CurrentStrip_05[5].fX                  = 330.0f;
CurrentStrip_05[5].fY                  = 175.0f;
CurrentStrip_05[5].u.fZ                =  20.0f;
CurrentStrip_05[5].fU                  =   0.0f;
CurrentStrip_05[5].fV                  =   0.0f;
CurrentStrip_05[5].fBaseAlpha          =   0.75f;
CurrentStrip_05[5].fBaseRed            =   0.55f;
CurrentStrip_05[5].fBaseGreen          =   0.87f;
CurrentStrip_05[5].fBaseBlue           =   0.96f;

CurrentStrip_05[6].ParamControlWord    = KM_VERTEXPARAM_NORMAL;
CurrentStrip_05[6].fX                  = 470.0f;
CurrentStrip_05[6].fY                  = 305.0f;
CurrentStrip_05[6].u.fZ                =  20.0f;
CurrentStrip_05[6].fU                  =   1.0f;
CurrentStrip_05[6].fV                  =   1.0f;
CurrentStrip_05[6].fBaseAlpha          =   0.75f;
CurrentStrip_05[6].fBaseRed            =   0.55f;
CurrentStrip_05[6].fBaseGreen          =   0.87f;
CurrentStrip_05[6].fBaseBlue           =   0.96f;

CurrentStrip_05[7].ParamControlWord    = KM_VERTEXPARAM_ENDOFSTRIP;
CurrentStrip_05[7].fX                  = 470.0f;
CurrentStrip_05[7].fY                  = 175.0f;
CurrentStrip_05[7].u.fZ                =  20.0f;
CurrentStrip_05[7].fU                  =   1.0f;
CurrentStrip_05[7].fV                  =   0.0f;
CurrentStrip_05[7].fBaseAlpha          =   0.75f;
CurrentStrip_05[7].fBaseRed            =   0.0f;
CurrentStrip_05[7].fBaseGreen          =   0.15f;
CurrentStrip_05[7].fBaseBlue           =   0.73f;

/* NOTE: A lot of context switching and setup can be avoided by providing          */
/*       several strips of the same type in kmSetVertex() calls without starting   */
/*       a new strip.  Don't call kmStartVertexStrip() redundantly.                */
for (i = 0; i <= 7; i++)
    kmSetVertex (&VertexBufferDesc, &CurrentStrip_05[i],
    KM_VERTEXTYPE_05, sizeof (KMVERTEX_05));

/* Done sending translucent texture strips. */
kmEndStrip (&VertexBufferDesc);

/* End the first (and only) pass. */
kmEndPass (&VertexBufferDesc);
```

```
/* Tell Kamui2 to begin transferring vertices to the TA and enqueue a page flip. */
kmRender (KM_RENDER_FLIP);

/* Finish the scene. */
kmEndScene (&kmsc);

/* Ping the peripheral server. */
pdExecPeripheralServer();
   }
```

Source Code of the QuikTest2.c Program

This is the source of the `QuikTest2.c` program, a simple application that demonstrates basic Kamui 2 programming. You can also find the program in the examples directory of the Kamui 2 SDK.

```
/******************************************************************************************/
/* CONFIDENTIAL AND PROPRIETARY:                                                          */
/*      Copyright 1999, Sega of America.  All rights reserved.                            */
/*      This sample source code is provided for demonstration purposes only.              */
/*      Use and distribution of Sega proprietary material is governed by                  */
/*      developer licensing agreement.                                                    */
/******************************************************************************************/

/******************************************************************************************/
/* Name:     k2QTest.c                                                                    */
/* Title:    QuickTest Kamui2 Example                                                     */
/* Author:   Gary Lake                                                                    */
/* Created:  June 4, 1999                                                                 */
/*                                                                                        */
/* Version:  1.0                                                                          */
/* Platform: Dreamcast | Set5.24 | Shinobi | Kamui2                                       */
/*                                                                                        */
/* Description:                                                                           */
/*      The purpose of this example is to provide an extremely simple test program        */
/*      that also demonstrates some basic Kamui2 optimizations.                           */
/*                                                                                        */
/* History:                                                                               */
/*      06/04/99 - Ported QuikTest to Kamui2.                                             */
/*      02/24/99 - Modified memory allocation routines for 32-byte alignment.             */
/*      11/06/98 - Added extra goodies to demonstrate Set5.24 functionality (Gary Lake).  */
/*      11/04/98 - Fixes added for Set5.23 (Kerry Thompson).                              */
/*      10/11/98 - Ported to Set5.16 (Gary Lake).                                         */
/*      07/03/98 - Ported to Set4 (Gary Lake).                                            */
/*      04/08/98 - Program created (Gary Lake).                                           */
/******************************************************************************************/


/******************************************************************************************/
/* Includes                                                                               */
/******************************************************************************************/

#include <shinobi.h>    /* Shinobi system routines. */
#include <kamui2.h>     /* Kamui2 low-level graphics HAL. */
#include <sn_fcntl.h>   /* LibCross file types. */
#include <usrsnasm.h>   /* LibCross I/O routines. */
#include <sg_syCbl.h>   /* NTSC/RGB/VGA Cable check interface. */


/******************************************************************************************/
/* Definitions                                                                            */
/******************************************************************************************/

#define DIRECT  0       /* Direct mode has been removed from Kamui2.. use 2V, it's faster.  */

#define VERTEXBUFFERSIZE    0x40000     /* Size of vertex buffer, in bytes (to be doubled). */
#define TEXTUREMEMORYSIZE   0x400000    /* VRAM set aside for textures, in bytes (4 MB). */
#define MAXTEXTURES         4096        /* Maximum number of textures to be tracked. */
#define MAXSMALLVQ          0           /* Maximum number of small VQ textures. */

/* Override Shinobi's syMalloc() / syFree() macros. */
#ifdef syMalloc
#undef syMalloc
#define syMalloc    malloc
#endif /* syMalloc */

#ifdef syFree
#undef syFree
#define syFree      free
#endif /* syFree */
```

```
/*****************************************************************************************/
/* Global Variables                                                                      */
/*****************************************************************************************/

void *(*MallocPtr)(unsigned long) = syMalloc;   /* Default allocation is syMalloc(). */
void (*FreePtr)(void *) = syFree;               /* Default de-allocation is syFree(). */

extern long *kmiDeviceExtension;    /* External reference to Kamui2 device driver. */

KMSTRIPCONTEXT DefaultContext =     /* Standard opaque polygon. */
{
    0x90, {KM_OPAQUE_POLYGON, KM_USERCLIP_DISABLE, KM_NORMAL_POLYGON,
    KM_FLOATINGCOLOR, KM_FALSE, KM_TRUE}, {KM_GREATER, KM_CULLCCW, KM_FALSE,
    KM_FALSE, 0}, {0, 0}, {KM_ONE, KM_ZERO, KM_FALSE, KM_FALSE, KM_NOFOG,
    KM_FALSE, KM_FALSE, KM_FALSE, KM_NOFLIP, KM_NOCLAMP, KM_BILINEAR,
    KM_FALSE, KM_MIPMAP_D_ADJUST_1_00, KM_MODULATE, 0, 0}
};

KMPACKEDARGB    Border;         /* Border color packed ARGB value. */

/* Kamui2 texture work area, must be aligned to 32-byte boundary. */
#ifdef  __MWERKS__
#pragma align (32);
KMDWORD
#else
#ifdef __GNUC__
KMDWORD __attribute__ ((aligned (32)))
#else
#pragma alignsdata32(TextureWorkArea)
KMDWORD
#endif /* __GNUC__ */
#endif /* _MWERKS__ */
TextureWorkArea[MAXTEXTURES * 24 / 4 + MAXSMALLVQ * 76 / 4];

#ifdef  __MWERKS__
#pragma align (4);/* Return Metrowerks alignment to 4-bytes. */
#endif


/*****************************************************************************************/
/* Macros                                                                                */
/*****************************************************************************************/

/* Set up 32-byte aligned malloc() / free() memory management. */
#define Align32Malloc   (*MallocPtr)
#define Align32Free     (*FreePtr)

/* Align a pointer to the nearest 32-byte aligned memory address. */
#define Align32Byte(ADR)        ((((long) ADR) + 0x1F) & 0xFFFFFFE0)

/* Reference physical memory through a cached memory address (SH4 P1 memory region). */
#define SH4_P1CachedMem(ADR)    ((((long) ADR) & 0x0FFFFFFF) | 0x80000000)

/* Reference physical memory through a non-cached memory address (SH4 P2 memory region). */
#define SH4_P2NonCachedMem(ADR) ((((long) ADR) & 0x0FFFFFFF) | 0xA0000000)

/* Pack red, green, blue, and alpha values into a 32-bit RGBA color word. */
#define PackRGBA(R, G, B, A)  ((unsigned long) (((A) << 24) | ((R) << 16) | ((G) << 8) | (B)))


/*****************************************************************************************/
/* MACRO _TWIDDLEADR():                                                                  */
/*      Internal macro used by TwiddledOffset() to convert a rectangular coordinate into */
/*      a twiddled address.                                                              */
/*****************************************************************************************/

#define _TWIDDLEADR(COORD)      ((COORD & 1)            | ((COORD >> 1) & 1) << 2   |  \
                                ((COORD >> 2) & 1) << 4  | ((COORD >> 3) & 1) << 6   |  \
                                ((COORD >> 4) & 1) << 8  | ((COORD >> 5) & 1) << 10  |  \
                                ((COORD >> 6) & 1) << 12 | ((COORD >> 7) & 1) << 14  |  \
                                ((COORD >> 8) & 1) << 16 | ((COORD >> 9) & 1) << 18)
```

```
/*****************************************************************************************/
/* MACRO TwiddledOffset():                                                               */
/*      Convert a rectangular coordinate pair (x, y) into a twiddled offset.             */
/*                                                                                       */
/* Inputs:                                                                               */
/*      X   = Horizontal pixel coordinate.                                               */
/*      Y   = Vertical pixel coordinate.                                                 */
/*                                                                                       */
/* Outputs:                                                                              */
/*      Return = Pixel offset into twiddled texture.                                     */
/*****************************************************************************************/

#define TwiddledOffset(X, Y)    ((_TWIDDLEADR(X) << 1) | _TWIDDLEADR(Y))


/*****************************************************************************************/
/* Functions & Procedures                                                                */
/*****************************************************************************************/

/*****************************************************************************************/
/* _Align32Malloc():                                                                     */
/*      Allocate a memory region with a size padded to the nearest multiple of 32 bytes  */
/*      and an alignment shifted to the nearest 32-byte boundary.                        */
/*                                                                                       */
/* Inputs:                                                                               */
/*      Size    = Size of memory region to allocate.                                     */
/*                                                                                       */
/* Outputs:                                                                              */
/*      Return  = Pointer to 32-byte aligned memory region.                              */
/*****************************************************************************************/

void *_Align32Malloc (unsigned long Size)
{
    void *Ptr, *AlignedPtr;

    /* Adjust the requested size to a multiple of 32 bytes. */
    Size = (Size + 0x1F) & 0xFFFFFFE0;

    /* Allocate requested size plus 32 bytes padding. */
    Ptr = syMalloc (Size + 32);

    /* Align to 32-bytes (add 32 bytes if already aligned - the padding is used below). */
    AlignedPtr = (void *) ((((long) Ptr) + 0x20) & 0xFFFFFFE0);

    /* Place cookie one byte earlier for _Align32Free(). */
    *((char *) AlignedPtr - 1) = (char) ((long) AlignedPtr - (long) Ptr);

    return (AlignedPtr);
}


/*****************************************************************************************/
/* _Align32Free():                                                                       */
/*      Free memory allocated using _Align32Malloc().                                    */
/*                                                                                       */
/* Inputs:                                                                               */
/*      Ptr     = Pointer to memory region to free.                                      */
/*****************************************************************************************/

void _Align32Free (void *Ptr)
{
    char Diff;

    /* Read cookie and adjust pointer back to original unaligned address before freeing. */
    Diff = *((char *) Ptr - 1);
    Ptr = (void *) ((long) Ptr - Diff);

    syFree (Ptr);
}
```

```
/*****************************************************************************************/
/* Init32Malloc():                                                                       */
/*      Check the built-in malloc() / free() routines for 32-byte alignment.             */
/*      If allocations are not aligned, install alternate Align32Malloc() and            */
/*      Align32Free() functions.                                                         */
/*                                                                                       */
/* Outputs:                                                                              */
/*      Align32Malloc   = Global macro, adjusted to point to 32-byte aligned malloc().   */
/*      Align32Free     = Global macro, adjusted to point to valid 32-byte aligned free(). */
/*****************************************************************************************/

void Init32Malloc (void)
{
    char *Ptr1, *Ptr2;

    /* Use syMalloc() / syFree() by default. */
    MallocPtr = syMalloc;
    FreePtr = syFree;

    Ptr1 = syMalloc (1);
    Ptr2 = syMalloc (1);

    /* Test if either allocation was not 32-byte aligned. */
    if (((long) Ptr1 & 0x1F) || ((long) Ptr2 & 0x1F))
    {
        MallocPtr = _Align32Malloc;
        FreePtr = _Align32Free;
    }

    syFree (Ptr1);
    syFree (Ptr2);
}


/*****************************************************************************************/
/* Rectangle2Twiddled():                                                                 */
/*      Convert a rectangle texture (linear order) into a PVR twiddled format texture.   */
/*      This routine was coded for sample purposes and not optimized for run-time use.   */
/*                                                                                       */
/* Inputs:                                                                               */
/*      Rectangle    = Pointer to rectangle texture.                                     */
/*      Dimension    = Texture dimension (assumed to be square, i.e. width = height).    */
/*      BitsPerPixel = Bit depth of texture.                                             */
/*                                                                                       */
/* Outputs:                                                                              */
/*      Twiddled = Buffer for twiddled texture output (same size as rectangle).          */
/*****************************************************************************************/

void Rectangle2Twiddled (char *Rectangle, char *Twiddled, int Dimension, int BitsPerPixel)
{
    int x, y;
    char *In = Rectangle, *Out;

    /* Convert BitsPerPixel into a shift value.. 8-bit = shift by 0, 16-bit = shift by 1. */
    BitsPerPixel = BitsPerPixel >> 4;

    for (y = 0; y < Dimension; y++)
    {
        for (x = 0; x < Dimension; x++)
        {
            /* TwiddledOffset macro converts coordinate into twiddled texture offset. */
            Out = Twiddled + ((TwiddledOffset (x, y)) << BitsPerPixel);

            /* Transfer pixel from Rectangle texture to Twiddled. */
            *Out = *In;
            if (BitsPerPixel > 0)           /* Account for 16-bit data. */
                *(Out + 1) = *(In + 1);
            In += BitsPerPixel + 1;
        }
    }
}
```

```
/*****************************************************************************************/
/* LoadTextureFile():                                                                    */
/*      Allocate system memory for a PVR texture and load the specified texture file from   */
/*      the GD filesystem.  Returns a pointer to the allocated memory block.             */
/*                                                                                       */
/*      Note: The PVR texture data will be offset to the nearest 32-byte aligned address    */
/*      within the returned memory region.                                               */
/*                                                                                       */
/* Inputs:                                                                               */
/*      Filename    = Filename character string, must be valid for the current directory.   */
/*                                                                                       */
/* Outputs:                                                                              */
/*      Twiddled    = Buffer containing PVR texture data.                                */
/*****************************************************************************************/

PKMDWORD LoadTextureFile (char *Filename)
{
    PKMDWORD    TexturePtr;
    GDFS        gdfs;
    long        FileBlocks;

    /* Open input file. */
    if (!(gdfs = gdFsOpen (Filename, NULL)))
        return NULL;

    /* Get file size (in blocks/sectors). */
    gdFsGetFileSctSize (gdfs, &FileBlocks);

    /* Allocate memory to nearest block size (2048 bytes). */
    TexturePtr = Align32Malloc (FileBlocks * 2048);

    /* Read file to memory region (Destination address must be 32-byte aligned). */
    gdFsReqRd32 (gdfs, FileBlocks, TexturePtr);

    /* Wait for file access to finish. */
    while (gdFsGetStat (gdfs) != GDD_STAT_COMPLETE)
        ;

    /* Close file. */
    gdFsClose (gdfs);

    return TexturePtr;
}


/*****************************************************************************************/
/* main():                                                                               */
/*      Entry point for C code (Where it all happens).                                   */
/*****************************************************************************************/

int main (void)
{
    KMSYSTEMCONFIGSTRUCT kmsc;      /* Kamui2 system configuration structure. */

    KMVERSIONINFO       VersionInfo;            /* Kamui2 version information. */
    KMSURFACEDESC       PrimarySurfaceDesc;     /* Primary framebuffer. */
    KMSURFACEDESC       BackSurfaceDesc;        /* Back framebuffer surface. */
    KMSURFACEDESC       TexSurfaceDesc;         /* Texture surface. */
    KMVERTEXBUFFDESC    VertexBufferDesc;       /* Vertex buffer. */
    PKMSURFACEDESC      FBSurfaces[2];          /* Array of pointers to FB surfaces. */
    KMSTRIPHEAD         StripHead_01;           /* Render params for oqaque color polys. */
    KMSTRIPHEAD         StripHead_05_Opaque;    /* Render params for opaque textured polys. */
    KMSTRIPHEAD         StripHead_05_Trans;     /* Params for translucent textured polys. */
    KMVERTEX_01         CurrentStrip_01[16];    /* Strip of opaque color triangles. */
    KMVERTEX_05         CurrentStrip_05[16];    /* Strip of textured triangles. */
    PKMDWORD            VertexBufferPtr;        /* Pointer to user-allocated vertex buffer. */
    PKMDWORD            TexturePtr;             /* Pointer to texture. */
    PKMDWORD            TwiddledPtr;            /* Pointer to twiddled texture. */
    PDS_PERIPHERAL      *per;                   /* Structure used to read control pad. */

    int                 i;
    float               Red = 0.0f, Green = 0.0f, Blue = 0.0f;      /* Gradient colors. */
```

```
    float               ColorDelta = 0.01f;
    char                TmpString[80];

#ifdef __GNUC__
        shinobi_workaround();
#endif

    /* Tell the user about this app. */
    debug_write (SNASM_STDOUT, "QuickTest2: Demonstrates basic Kamui2 operation.", 48);

    /* Check the cable for NTSC/PAL or VGA */
    switch (syCblCheckCable())
          {
        /* Initialize the display device and set the frame buffer based on the video mode. */
        case SYE_CBL_CABLE_NTSC:    /* Note: SYE_CBL_CABLE_NTSC/PAL defined as same value. */
            switch (syCblCheckBroadcast())
            {
                case SYE_CBL_BROADCAST_NTSC:    /* U.S./North America NTSC (60Hz). */
                case SYE_CBL_BROADCAST_PALM:    /* Brazil PAL-M (60Hz). */
                    sbInitSystem (KM_DSPMODE_NTSCNI640x480, KM_DSPBPP_RGB565, 1);
                    break;
                case SYE_CBL_BROADCAST_PAL:     /* Europe PAL (50Hz). */
                case SYE_CBL_BROADCAST_PALN:    /* Argentina PAL-N (50Hz). */
                    sbInitSystem (KM_DSPMODE_PALNI640x480, KM_DSPBPP_RGB565, 1);
                    break;
            }
            break;
        case SYE_CBL_CABLE_VGA:
            sbInitSystem (KM_DSPMODE_VGA, KM_DSPBPP_RGB565, 1);
            break;
           }

    /* Check malloc alignment. */
    Init32Malloc();

    /* Check Kamui2 version and memory information. */
    debug_write (SNASM_STDOUT, "Kamui2 Version =", 16);

    /* Use kmGetVersionInfo() or peek around in memory like a real hacker. */
    #if 1
        kmGetVersionInfo (&VersionInfo);
    #else
        VersionInfo.kmMajorVersion = *kmiDeviceExtension;
        VersionInfo.kmLocalVersion = ((*(kmiDeviceExtension + 1) & 0xFF) << 24);
        VersionInfo.kmLocalVersion |= ((*(kmiDeviceExtension + 2) & 0xFF) << 16);
        VersionInfo.kmLocalVersion |= (*(kmiDeviceExtension + 3) & 0xFFFF);
        VersionInfo.kmFrameBufferSize = *((long *) ((long) &kmiDeviceExtension + 24));
    #endif

    /* Print version information to the Codescape log window. */
    sprintf (TmpString, "  Major Version: %d", VersionInfo.kmMajorVersion);
    debug_write (SNASM_STDOUT, TmpString, strlen (TmpString));
    sprintf (TmpString, "  Minor Version: %d", ((VersionInfo.kmLocalVersion >> 24) & 0xFF));
    debug_write (SNASM_STDOUT, TmpString, strlen (TmpString));
    sprintf (TmpString, "  Major Build:   %d", ((VersionInfo.kmLocalVersion >> 16) & 0xFF));
    debug_write (SNASM_STDOUT, TmpString, strlen (TmpString));
    sprintf (TmpString, "  Minor Build:   %d", (VersionInfo.kmLocalVersion & 0xFFFF));
    debug_write (SNASM_STDOUT, TmpString, strlen (TmpString));
    sprintf (TmpString, "VRAM size %d bytes.", VersionInfo.kmFrameBufferSize);
    debug_write (SNASM_STDOUT, TmpString, strlen (TmpString));

    /* Set size of system configuration struct, required. */
    kmsc.dwSize = sizeof(KMSYSTEMCONFIGSTRUCT);

    /* Render control flags. */
    kmsc.flags  = KM_CONFIGFLAG_ENABLE_CLEAR_FRAMEBUFFER    /* Clear FB at initialization. */
    //            | KM_CONFIGFLAG_NOWAITVSYNC                 /* Don't wait for the VBlank. */
                | KM_CONFIGFLAG_ENABLE_2V_LATENCY;          /* Use 2V latency render model. */

    /* Frame buffer surfaces information. */
    FBSurfaces[0] = &PrimarySurfaceDesc;
    FBSurfaces[1] = &BackSurfaceDesc;
```

```
kmsc.ppSurfaceDescArray = FBSurfaces;   /* Set frame buffer surface description array. */
kmsc.fb.nNumOfFrameBuffer = 2;          /* Number of frame buffers (double buffered). */
kmsc.fb.nStripBufferHeight = 32;        /* Strip buffer height = 32xN pixels, if used. */

/* Texture handler setup. */
kmsc.nTextureMemorySize = TEXTUREMEMORYSIZE;    /* Texture VRAM, divisible by 32 bytes. */
kmsc.nNumOfTextureStruct = MAXTEXTURES;         /* Maximum number of textures tracked. */
kmsc.nNumOfSmallVQStruct = MAXSMALLVQ;          /* Max. number of small VQ textures. */
kmsc.pTextureWork = TextureWorkArea;            /* Texture work area in system memory. */

/* Generate 32-byte aligned vertex buffer (double-buffered). */
VertexBufferPtr = (PKMDWORD) Align32Malloc (2 * VERTEXBUFFERSIZE * sizeof (KMDWORD));

/* Kamui2 requires the vertex buffer region to be 32-byte aligned and non-cacheable. */
kmsc.pVertexBuffer = (PKMDWORD) SH4_P2NonCachedMem (VertexBufferPtr);

/* Define vertex buffer. */
kmsc.nVertexBufferSize = VERTEXBUFFERSIZE * 2;  /* Size of vertex buffer X2, in bytes. */
kmsc.pBufferDesc = &VertexBufferDesc;   /* Struct used to maintain vertex buffer info. */
kmsc.nNumOfVertexBank = 2;              /* Number of vertex buffers (double buffered). */

/* Set multi-pass rendering information. */
kmsc.nPassDepth = 1;                                    /* Number of passes. */
kmsc.Pass[0].dwRegionArrayFlag = KM_PASSINFO_AUTOSORT   /* Autosort translucent polys. */
    | KM_PASSINFO_ENABLE_Z_CLEAR;                       /* Clear Z buffer. */
kmsc.Pass[0].nDirectTransferList = KM_OPAQUE_POLYGON;   /* Type sent direct in 2V mode. */

/* Percent of vertex buffer used for each vertex type in a pass (must total 100%). */
kmsc.Pass[0].fBufferSize[0] =  0.0f;        /* Opaque polygons (0% if sent direct). */
kmsc.Pass[0].fBufferSize[1] =  0.0f;        /* Opaque modifier. */
kmsc.Pass[0].fBufferSize[2] = 50.0f;        /* Translucent. */
kmsc.Pass[0].fBufferSize[3] =  0.0f;        /* Translucent modifier. */
kmsc.Pass[0].fBufferSize[4] = 50.0f;        /* Punchthrough. */

/* Set system configuration. */
kmSetSystemConfiguration (&kmsc);

/* Set screen border color. */
Border.dwPacked = 0x0;
kmSetBorderColor (Border);

/* Create a texture surface. */
kmCreateTextureSurface (&TexSurfaceDesc, 256, 256,
    (KM_TEXTURE_TWIDDLED | KM_TEXTURE_RGB565));

/* Load a texture file from disk. */
TexturePtr = LoadTextureFile ("FBI.pvr");

/* The next few calls demonstrate how to algorithmically generate a texture.  In this   */
/* case, a rectangle format texture is loaded from disk and twiddled at run-time.       */

/* Allocate memory for twiddled texture (256x256 16-bit). */
TwiddledPtr = (PKMDWORD) Align32Malloc (256 * 256 * 2);
                                                        s
/* Run-time twiddle source rectangle texture + 16 bytes to skip PVRT chunk header. */
Rectangle2Twiddled ((char *) ((long) TexturePtr + 16), (char *) TwiddledPtr, 256, 16);

/* Transfer twiddled texture from system mem to video mem using a DMA transfer. */
kmLoadTexture (&TexSurfaceDesc, TwiddledPtr);

/* Generate a strip head array for opaque color (no texture) polygons. */
memset (&StripHead_01, 0, sizeof(StripHead_01));
kmGenerateStripHead01 (&StripHead_01, &DefaultContext);

/* Set the texture surface of the opaque polygon context to the newly loaded texture. */
DefaultContext.ImageControl[KM_IMAGE_PARAM1].pTextureSurfaceDesc = &TexSurfaceDesc;

/* Generate a strip head array for opaque textured polygons. */
memset (&StripHead_05_Opaque, 0, sizeof(StripHead_05_Opaque));
kmGenerateStripHead05 (&StripHead_05_Opaque, &DefaultContext);

/* Modify the standard polygon context to support translucency. */
```

```
DefaultContext.StripControl.nListType = KM_TRANS_POLYGON;
DefaultContext.ImageControl[KM_IMAGE_PARAM1].bUseAlpha = KM_TRUE;
DefaultContext.ImageControl[KM_IMAGE_PARAM1].nTextureShadingMode = KM_MODULATE_ALPHA;

/* Generate a strip head array for translucent textured polygons. */
memset (&StripHead_05_Trans, 0, sizeof(StripHead_05_Trans));
kmGenerateStripHead05 (&StripHead_05_Trans, &DefaultContext);

/* Just to demonstrate how a strip head can be modified once generated.. */
kmChangeStripBlendingMode (&StripHead_05_Trans, KM_IMAGE_PARAM1,
    KM_SRCALPHA, KM_INVSRCALPHA);

/* The main render loop. */
while (TRUE)
      {
    /* Get control pad info filled out by the 'PeripheralServer'. */
    per = (PDS_PERIPHERAL *) pdGetPeripheral (PDD_PORT_A0);

    /* Set the vertices of the background plane for a nice color gradient. */
    /* NOTE: Yes, you can change the background plane on-the-fly. */
    CurrentStrip_01[0].ParamControlWord     = KM_VERTEXPARAM_NORMAL;
    CurrentStrip_01[0].fX                   =   0.0f;
    CurrentStrip_01[0].fY                   = 479.0f;
    CurrentStrip_01[0].u.fZ                 =   0.2f;
    CurrentStrip_01[0].fBaseAlpha           =   1.0f;
    CurrentStrip_01[0].fBaseRed             =   0.0f;
    CurrentStrip_01[0].fBaseGreen           =   0.0f;
    CurrentStrip_01[0].fBaseBlue            =   0.0f;

    CurrentStrip_01[1].ParamControlWord     = KM_VERTEXPARAM_NORMAL;
    CurrentStrip_01[1].fX                   =   0.0f;
    CurrentStrip_01[1].fY                   =   0.0f;
    CurrentStrip_01[1].u.fZ                 =   0.2f;
    CurrentStrip_01[1].fBaseAlpha           =   1.0f;
    CurrentStrip_01[1].fBaseRed             =   Red;
    CurrentStrip_01[1].fBaseGreen           =   Green;
    CurrentStrip_01[1].fBaseBlue            =   Blue;

    CurrentStrip_01[2].ParamControlWord     = KM_VERTEXPARAM_ENDOFSTRIP;
    CurrentStrip_01[2].fX                   = 639.0f;
    CurrentStrip_01[2].fY                   = 479.0f;
    CurrentStrip_01[2].u.fZ                 =   0.2f;
    CurrentStrip_01[2].fBaseAlpha           =   1.0f;
    CurrentStrip_01[2].fBaseRed             =   0.0f;
    CurrentStrip_01[2].fBaseGreen           =   0.0f;
    CurrentStrip_01[2].fBaseBlue            =   0.0f;

    /* Set the background plane to the vertices filled in above. */
    kmSetBackGround (&StripHead_01, KM_VERTEXTYPE_01,
        &CurrentStrip_01[0], &CurrentStrip_01[1], &CurrentStrip_01[2]);

    /* Update colors - just to show we're doing something. */
    if (Blue < 0.99)
    {
        Blue += ColorDelta;
        if (ColorDelta < 0)
        {
            Red += ColorDelta;
            Green += ColorDelta;
        }
    }
    else
    {
        if (Red > 0.99)
        {
            ColorDelta = -ColorDelta;
            Blue += ColorDelta;
        }
        Red += ColorDelta;
        Green += ColorDelta;
    }
```

```
if (Blue < 0.0)
{
    Blue = 0.0;
    Red = 0.0;
    Green = 0.0;
    ColorDelta = 0.01f;
}

/* Tell Kamui2 to start accepting polygon data for a new scene. */
kmBeginScene (&kmsc);

/* Start the first pass (for multipass rendering). */
kmBeginPass (&VertexBufferDesc);

/* Start rendering opaque color strips. */
kmStartStrip (&VertexBufferDesc, &StripHead_01);

/* Set the vertices in the current strip to draw a colorful square. */
CurrentStrip_01[0].ParamControlWord     = KM_VERTEXPARAM_NORMAL;
CurrentStrip_01[0].fX                   = 170.0f;
CurrentStrip_01[0].fY                   = 305.0f;
CurrentStrip_01[0].u.fZ                 =  10.0f;
CurrentStrip_01[0].fBaseAlpha           =   1.0f;
CurrentStrip_01[0].fBaseRed             =   1.0f;
CurrentStrip_01[0].fBaseGreen           =   0.0f;
CurrentStrip_01[0].fBaseBlue            =   0.0f;

CurrentStrip_01[1].ParamControlWord     = KM_VERTEXPARAM_NORMAL;
CurrentStrip_01[1].fX                   = 170.0f;
CurrentStrip_01[1].fY                   = 175.0f;
CurrentStrip_01[1].u.fZ                 =  10.0f;
CurrentStrip_01[1].fBaseAlpha           =   1.0f;
CurrentStrip_01[1].fBaseRed             =   0.0f;
CurrentStrip_01[1].fBaseGreen           =   1.0f;
CurrentStrip_01[1].fBaseBlue            =   0.0f;

CurrentStrip_01[2].ParamControlWord     = KM_VERTEXPARAM_NORMAL;
CurrentStrip_01[2].fX                   = 310.0f;
CurrentStrip_01[2].fY                   = 305.0f;
CurrentStrip_01[2].u.fZ                 =  10.0f;
CurrentStrip_01[2].fBaseAlpha           =   1.0f;
CurrentStrip_01[2].fBaseRed             =   0.0f;
CurrentStrip_01[2].fBaseGreen           =   0.0f;
CurrentStrip_01[2].fBaseBlue            =   1.0f;

CurrentStrip_01[3].ParamControlWord     = KM_VERTEXPARAM_ENDOFSTRIP;
CurrentStrip_01[3].fX                   = 310.0f;
CurrentStrip_01[3].fY                   = 175.0f;
CurrentStrip_01[3].u.fZ                 =  10.0f;
CurrentStrip_01[3].fBaseAlpha           =   1.0f;
CurrentStrip_01[3].fBaseRed             =   0.0f;
CurrentStrip_01[3].fBaseGreen           =   0.0f;
CurrentStrip_01[3].fBaseBlue            =   0.0f;

for (i=0; i<=3; i++)
    kmSetVertex (&VertexBufferDesc, &CurrentStrip_01[i],
        KM_VERTEXTYPE_01, sizeof (KMVERTEX_01));

/* Done sending opaque color strip. */
kmEndStrip (&VertexBufferDesc);

/* Start rendering opaque textured strips. */
kmStartStrip (&VertexBufferDesc, &StripHead_05_Opaque);

/* Set the vertices in the current strip to draw a textured square. */
CurrentStrip_05[0].ParamControlWord     = KM_VERTEXPARAM_NORMAL;
CurrentStrip_05[0].fX                   = 250.0f;
CurrentStrip_05[0].fY                   = 240.0f;
CurrentStrip_05[0].u.fZ                 =  11.0f;
CurrentStrip_05[0].fU                   =   0.0f;
CurrentStrip_05[0].fV                   =   1.0f;
CurrentStrip_05[0].fBaseAlpha           =   1.0f;
```

```
CurrentStrip_05[0].fBaseRed            =   0.0f;
CurrentStrip_05[0].fBaseGreen          =   0.15f;
CurrentStrip_05[0].fBaseBlue           =   0.73f;

CurrentStrip_05[1].ParamControlWord    = KM_VERTEXPARAM_NORMAL;
CurrentStrip_05[1].fX                  = 250.0f;
CurrentStrip_05[1].fY                  = 110.0f;
CurrentStrip_05[1].u.fZ                =  11.0f;
CurrentStrip_05[1].fU                  =   0.0f;
CurrentStrip_05[1].fV                  =   0.0f;
CurrentStrip_05[1].fBaseAlpha          =   1.0f;
CurrentStrip_05[1].fBaseRed            =   0.55f;
CurrentStrip_05[1].fBaseGreen          =   0.87f;
CurrentStrip_05[1].fBaseBlue           =   0.96f;

CurrentStrip_05[2].ParamControlWord    = KM_VERTEXPARAM_NORMAL;
CurrentStrip_05[2].fX                  = 390.0f;
CurrentStrip_05[2].fY                  = 240.0f;
CurrentStrip_05[2].u.fZ                =  11.0f;
CurrentStrip_05[2].fU                  =   1.0f;
CurrentStrip_05[2].fV                  =   1.0f;
CurrentStrip_05[2].fBaseAlpha          =   1.0f;
CurrentStrip_05[2].fBaseRed            =   0.55f;
CurrentStrip_05[2].fBaseGreen          =   0.87f;
CurrentStrip_05[2].fBaseBlue           =   0.96f;

CurrentStrip_05[3].ParamControlWord    = KM_VERTEXPARAM_ENDOFSTRIP;
CurrentStrip_05[3].fX                  = 390.0f;
CurrentStrip_05[3].fY                  = 110.0f;
CurrentStrip_05[3].u.fZ                =  11.0f;
CurrentStrip_05[3].fU                  =   1.0f;
CurrentStrip_05[3].fV                  =   0.0f;
CurrentStrip_05[3].fBaseAlpha          =   1.0f;
CurrentStrip_05[3].fBaseRed            =   0.0f;
CurrentStrip_05[3].fBaseGreen          =   0.15f;
CurrentStrip_05[3].fBaseBlue           =   0.73f;

for (i=0; i <= 3; i++)
    kmSetVertex (&VertexBufferDesc, &CurrentStrip_05[i],
        KM_VERTEXTYPE_05, sizeof (KMVERTEX_05));

/* Done sending opaque texture strip. */
kmEndStrip (&VertexBufferDesc);

/* Start translucent textured strips. */
kmStartStrip (&VertexBufferDesc, &StripHead_05_Trans);

/* Set the vertices in the current strip to draw a translucent textured square. */
CurrentStrip_05[0].ParamControlWord    = KM_VERTEXPARAM_NORMAL;
CurrentStrip_05[0].fX                  = 250.0f;
CurrentStrip_05[0].fY                  = 370.0f;
CurrentStrip_05[0].u.fZ                =  21.0f;
CurrentStrip_05[0].fU                  =   0.0f;
CurrentStrip_05[0].fV                  =   1.0f;
CurrentStrip_05[0].fBaseAlpha          =   0.25f;
CurrentStrip_05[0].fBaseRed            =   0.0f;
CurrentStrip_05[0].fBaseGreen          =   0.15f;
CurrentStrip_05[0].fBaseBlue           =   0.73f;

CurrentStrip_05[1].ParamControlWord    = KM_VERTEXPARAM_NORMAL;
CurrentStrip_05[1].fX                  = 250.0f;
CurrentStrip_05[1].fY                  = 240.0f;
CurrentStrip_05[1].u.fZ                =  21.0f;
CurrentStrip_05[1].fU                  =   0.0f;
CurrentStrip_05[1].fV                  =   0.0f;
CurrentStrip_05[1].fBaseAlpha          =   0.25f;
CurrentStrip_05[1].fBaseRed            =   0.55f;
CurrentStrip_05[1].fBaseGreen          =   0.87f;
CurrentStrip_05[1].fBaseBlue           =   0.96f;

CurrentStrip_05[2].ParamControlWord    = KM_VERTEXPARAM_NORMAL;
CurrentStrip_05[2].fX                  = 390.0f;
```

```
CurrentStrip_05[2].fY                  = 370.0f;
CurrentStrip_05[2].u.fZ                =  21.0f;
CurrentStrip_05[2].fU                  =   1.0f;
CurrentStrip_05[2].fV                  =   1.0f;
CurrentStrip_05[2].fBaseAlpha          =   0.25f;
CurrentStrip_05[2].fBaseRed            =   0.55f;
CurrentStrip_05[2].fBaseGreen          =   0.87f;
CurrentStrip_05[2].fBaseBlue           =   0.96f;

CurrentStrip_05[3].ParamControlWord    = KM_VERTEXPARAM_ENDOFSTRIP;
CurrentStrip_05[3].fX                  = 390.0f;
CurrentStrip_05[3].fY                  = 240.0f;
CurrentStrip_05[3].u.fZ                =  21.0f;
CurrentStrip_05[3].fU                  =   1.0f;
CurrentStrip_05[3].fV                  =   0.0f;
CurrentStrip_05[3].fBaseAlpha          =   0.25f;
CurrentStrip_05[3].fBaseRed            =   0.0f;
CurrentStrip_05[3].fBaseGreen          =   0.15f;
CurrentStrip_05[3].fBaseBlue           =   0.73f;

/* NOTE: We can keep adding vertices, even past an ENDOFSTRIP flag - see below. */
CurrentStrip_05[4].ParamControlWord    = KM_VERTEXPARAM_NORMAL;
CurrentStrip_05[4].fX                  = 330.0f;
CurrentStrip_05[4].fY                  = 305.0f;
CurrentStrip_05[4].u.fZ                =  20.0f;
CurrentStrip_05[4].fU                  =   0.0f;
CurrentStrip_05[4].fV                  =   1.0f;
CurrentStrip_05[4].fBaseAlpha          =   0.75f;
CurrentStrip_05[4].fBaseRed            =   0.0f;
CurrentStrip_05[4].fBaseGreen          =   0.15f;
CurrentStrip_05[4].fBaseBlue           =   0.73f;

CurrentStrip_05[5].ParamControlWord    = KM_VERTEXPARAM_NORMAL;
CurrentStrip_05[5].fX                  = 330.0f;
CurrentStrip_05[5].fY                  = 175.0f;
CurrentStrip_05[5].u.fZ                =  20.0f;
CurrentStrip_05[5].fU                  =   0.0f;
CurrentStrip_05[5].fV                  =   0.0f;
CurrentStrip_05[5].fBaseAlpha          =   0.75f;
CurrentStrip_05[5].fBaseRed            =   0.55f;
CurrentStrip_05[5].fBaseGreen          =   0.87f;
CurrentStrip_05[5].fBaseBlue           =   0.96f;

CurrentStrip_05[6].ParamControlWord    = KM_VERTEXPARAM_NORMAL;
CurrentStrip_05[6].fX                  = 470.0f;
CurrentStrip_05[6].fY                  = 305.0f;
CurrentStrip_05[6].u.fZ                =  20.0f;
CurrentStrip_05[6].fU                  =   1.0f;
CurrentStrip_05[6].fV                  =   1.0f;
CurrentStrip_05[6].fBaseAlpha          =   0.75f;
CurrentStrip_05[6].fBaseRed            =   0.55f;
CurrentStrip_05[6].fBaseGreen          =   0.87f;
CurrentStrip_05[6].fBaseBlue           =   0.96f;

CurrentStrip_05[7].ParamControlWord    = KM_VERTEXPARAM_ENDOFSTRIP;
CurrentStrip_05[7].fX                  = 470.0f;
CurrentStrip_05[7].fY                  = 175.0f;
CurrentStrip_05[7].u.fZ                =  20.0f;
CurrentStrip_05[7].fU                  =   1.0f;
CurrentStrip_05[7].fV                  =   0.0f;
CurrentStrip_05[7].fBaseAlpha          =   0.75f;
CurrentStrip_05[7].fBaseRed            =   0.0f;
CurrentStrip_05[7].fBaseGreen          =   0.15f;
CurrentStrip_05[7].fBaseBlue           =   0.73f;

/* NOTE: A lot of context switching and setup can be avoided by providing          */
/*       several strips of the same type in kmSetVertex() calls without starting   */
/*       a new strip.  Don't call kmStartVertexStrip() redundantly.                */
for (i = 0; i <= 7; i++)
    kmSetVertex (&VertexBufferDesc, &CurrentStrip_05[i],
    KM_VERTEXTYPE_05, sizeof (KMVERTEX_05));
```

```
            /* Done sending translucent texture strips. */
            kmEndStrip (&VertexBufferDesc);

            /* End the first (and only) pass. */
            kmEndPass (&VertexBufferDesc);

            /* Tell Kamui2 to begin transferring vertices to the TA and enqueue a page flip. */
            kmRender (KM_RENDER_FLIP);

            /* Finish the scene. */
            kmEndScene (&kmsc);

            /* Ping the peripheral server. */
            pdExecPeripheralServer();
               }

        sbExitSystem();
    }
```