SGI® OpenGL Multipipe™ SDK Programmer's Guide

007-4527-001

Version 1.0

CONTRIBUTORS
Written by Ken Jones
Illustrated by Chrystie Danzer
Edited by Susan Wilkening
Production by Karen Jacobson
Engineering contributions by Patrick Bouchaud, Davy Courvoisier, Stefan Eilemann, and Philippe Robert

New Features in This Release

OpenGL Multipipe SDK 2.1 includes the following features:

- Automatic load balancing
- A full-scene antialiasing (FSAA) compound
- Transparent scalability for Xinerama windows
- Programmatic access (MPKFrame and MPKImage) to the RGBA, depth, and stencil data during compound assembly
- Tighter integration with the following SGI graphics products:

  - OpenGL Multipipe
  - OpenGL Volumizer
  - SGI Scalable Graphics Compositor

Record of Revision

Version	Description
001	October 2002 Original publication. Supports the 2.1 release of OpenGL Multipipe SDK.

Contents

Figures

Tables

About This Guide

This guide describes OpenGL Multipipe SDK (MPK), which is a software development toolkit that allows you to adapt your graphics applications to run in immersive environments and to take advantage of the scalability provided by multiple pipes and other scalable graphics hardware.

Audience

This guide targets application programmers. It describes how application programmers can adapt OpenGL graphics applications to fit the MPK programming model. The manual *SGI OpenGL Multipipe SDK User's Guide* targets Reality Center administrators, who configure graphics applications to run in multipipe environments.

Related Publications

The following books might be helpful:

- *SGI OpenGL Multipipe SDK User's Guide*
- Neider, Jackie,Tom Davis, and Mason Woo, *OpenGL Programming Guide*. Reading, Mass.: Addison-Wesley Publishing Company, Inc., 1993. A comprehensive guide to learning OpenGL.
- Nye, Adrian, *Volume One: Xlib Programming Manual*. Sebastopol, California: O'Reilly & Associates, Inc., 1991.

Conventions

The following conventions are used throughout this publication:

Convention	Meaning
`command`	This fixed-space font denotes literal items such as commands, files, routines, path names, signals, messages, and programming language structures.
variable	Italic typeface denotes variable entries and words or concepts being defined.
`user input`	This fixed-space font denotes literal items that the user enters in interactive sessions. (Output is shown in nonbold, fixed-space font.)
function	Functions are denoted in bold with following parentheses.
`manpage`(*x*)	Man page section identifiers appear in parentheses after man page names.

Reader Comments

If you have comments about the technical accuracy, content, or organization of this document, please send them to SGI. Be sure to include the title and document number of the manual with your comments. (Online, the document number is located in the front matter of the manual. In printed manuals, you can find the document number on the back cover.)

You can contact us in any of the following ways:

- Send e-mail to the following address:

  `techpubs@sgi.com`

- Use the Feedback option on the Technical Publications Library World Wide Web page:

  http://techpubs.sgi.com

- Contact your customer service representative and ask that an incident be filed in the SGI incident tracking system.

- Send mail to the following address:

  Technical Publications
  SGI
  1600 Amphitheatre Pkwy., M/S 535
  Mountain View, California 94043-1351

- Send a fax to the attention of “Technical Publications” at +1 650 932 0801.

SGI values your comments and will respond to them promptly.

Overview

This overview of OpenGL Multipipe SDK (MPK) consists of the following sections:

- "A Reality Center Facility"
- "What MPK Provides"
- "Components of MPK"
- "Application Structure"
- "A Sample Configuration File"

A Reality Center Facility

Throughout this document, we shall use the term Reality Center facility to convey the following meaning: an SGI computer environment with extended visualization capabilities. Note that this definition not only applies to the traditional three-pipe theater (historically set up for flight simulation) but covers as well all kinds of immersive environments (such as a Cave, TANORAMA POWERWALL, or TAN HOLOBENCH facility) and also extends to encompass graphics clusters. Figure 1-1 on page 2 illustrates an SGI Reality Center facility.

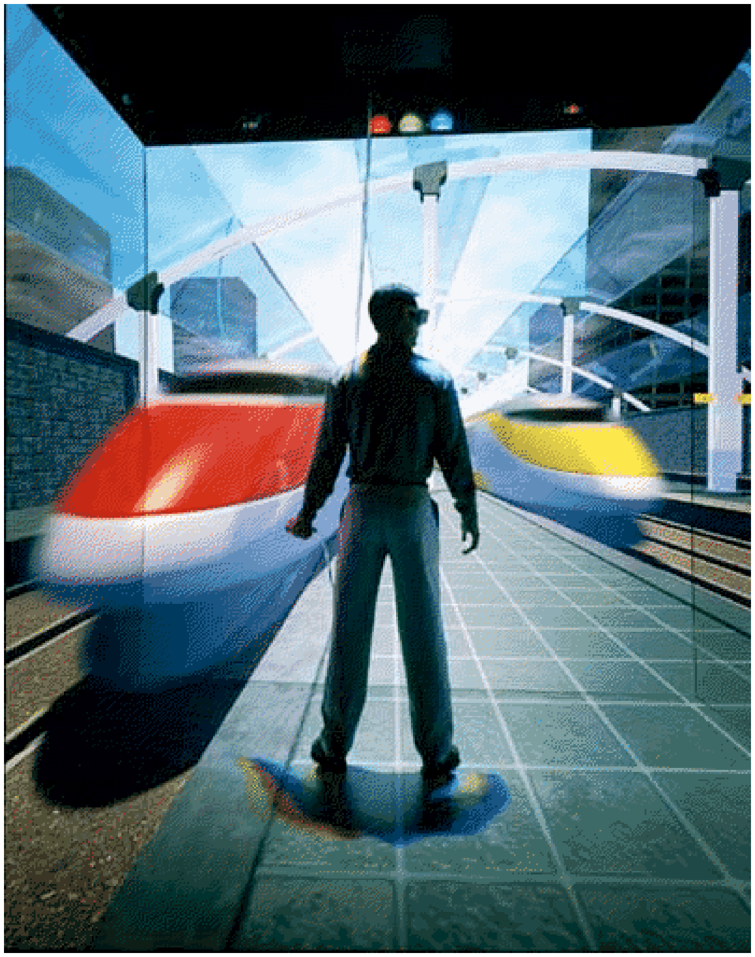

Figure 1-1 SGI Reality Center

What MPK Provides

As more and more graphics applications come into the virtual reality arena as a piece of immersive solutions, application developers face new requirements. Not only do developers need to take into account high frame rates and low latencies needed for temporal realism, but also better image quality for visual realism. OpenGL applications must improve their performances and must be able to run in increasingly complex environments that include various input peripherals and projection systems. For applications initially designed to run on a visual workstation in non-real time and with keyboard-mouse input, new releases now need to be time-accurate and should be able to integrate a moving frustum tied to head-tracking peripherals and several rendering engines (graphics pipes) that provide multiple and wider fields of view. Because these types of evolving environments have numerous parameters, the applications must be sufficiently flexible and robust to accommodate their demands.

MPK is an application programming interface (API) designed to help software developers meet the demands of these new immersive environments. This product enables the application to take advantage of the scalability provided by additional pipes and other scalable graphics hardware, as well as to support immersive environments. MPK provides the following specific features:

- Run-time configurability
- Run-time scalability
- Integrated support for scalable graphics hardware
- Integrated support for stereo and immersive environments

Run-Time Configurability

MPK allows developers to create applications that run on multiple platforms ranging from simple visual workstations to large and complex visualization environments, often based on several pipes for parallel rendering purposes. It implements a design that largely isolates the application from the graphics resources and the physical environment. Providing run-time configurability, an application written in the MPK programming model can run on a simple desktop platform or, without any modification or recompilation, in highly complex visualization environments like an SGI Reality Center facility.

Run-Time Scalability

Graphics-intensive applications often require several pipes in order to achieve a desired performance. Each pipe contributes to a part of the final rendering. This introduces the need for a decomposition paradigm and the issue of how the rendering performance scales with the number of pipes. Rendering in parallel requires the developer to manage several graphic contexts and then to create tasks or threads, each managing their own graphic context and sharing the scene to be rendered. MPK allows a multipipe applications developer to avoid dealing with such parallel programming paradigms and offers compound algorithms based on several decomposition types.

Integrated Support for Scalable Graphics Hardware

Scalable graphics hardware such as the SGI Scalable Graphics Compositor and the SGI Video Digital Multiplexer (DPLEX) can perform some of the compositing functions that MPK now provides in software. MPK supports such hardware as well as conventional graphics hardware.

Integrated Support for Stereo and Immersive Environments

Along with its scalability features, MPK has integrated the ability to exploit the stereo features of your application-display environment without recompilation. Having the related display characteristics of your environment described in a configuration file, you can specify at run time whether to run in stereo or mono.

In addition, MPK provides the application with the ability to support truly immersive environments by using a simple programming interface: the application only needs to provide real-world information about the position and orientation of the viewer. MPK then transparently adapts its left- and right-eye frustum computations to the actual user’s location.

The ease of configuring your application to accomodate different hardware resources (graphics pipes and head-tracking devices) and different display areas makes MPK ideal for use in immersive environments.

Components of MPK

MPK has two components:

- Application programming interface

  Designed for the applications programmer to adapt OpenGL graphics applications to fit the MPK programming model in order to support multipipe environments.

- Configuration file interface

  Designed for Reality Center administrators to configure MPK graphics applications to run in their environments. This ASCII file interface allows you to specify how the framebuffer resources (pipes, windows, and channels) are mapped onto the physical projection areas (walls) and the parallel decomposition schemes (compounds) to be used by your applications.

MPK is available on IRIX through C language function calls. It is designed as a thin layer on top of the operating system, X11, OpenGL, and GLX.

Application Structure

As an application will have to run in different configurations, MPK externalizes the configuration management by implementing an ASCII file that is separate from the other application code. The scene management and data workflow is separate from scene rendering (management of the graphics resources). Figure 1-2 illustrates the structure of an application based on MPK.

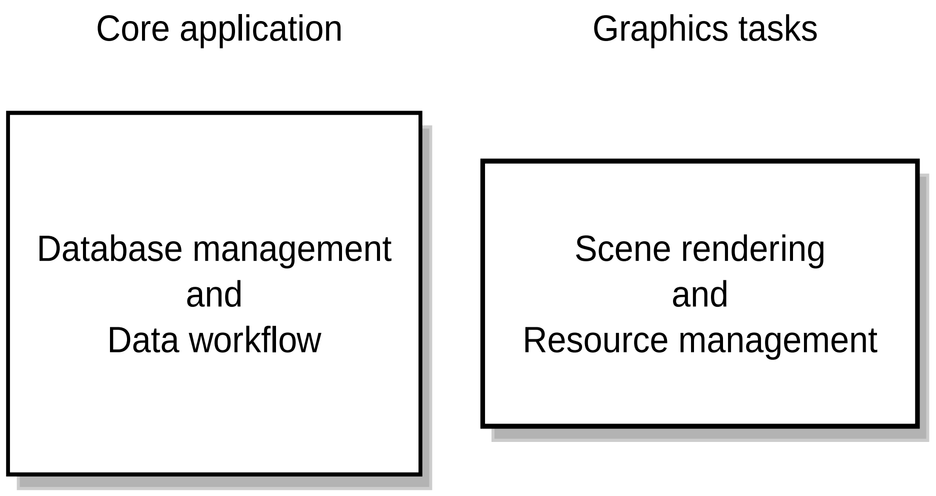
Core application
Graphics tasks
Database management and Data workflow
Scene rendering and Resource management

Figure 1-2 MPK Application Structure

A Sample Configuration File

Example 1-1 shows a one-pipe, one-window configuration file that can be used in conjunction with a MPK-structured program—for instance, `volview`, a scalable volume-viewer application packaged as part of the OpenGL Volumizer 2 product.

Example 1-1 Sample Configuration File

```
global {
    MPK_WATTR_PLANES_ALPHA  1
    MPK_DEFAULT_EYE_OFFSET 0.01
}
config {
    name    “Volview: 1-pipe”
    mode    mono

    mono    “/usr/gfx/setmon -n 1280x1024_76”
    stereo  “/usr/gfx/setmon -n str_top”
    pipe {
        window {
            viewport        [ 0, 0, 1.0, 1.0 ]
            channel {
                name                    “center”
                viewport                [ 0., 0., 1., 1. ]
                wall {
                    bottom_left     [ -.5, -.5, -1 ]
                    bottom_right    [  .5, -.5, -1 ]
                    top_left        [ -.5,  .5, -1 ]
                }
            }
        }
    }
}
```

The MPK Programming Model

This chapter describes the program structure and execution model of an OpenGL Mulitipipe SDK (MPK) program and compares this model with that of a conventional OpenGL program. It includes a walkthrough of a simple MPK application. This chapter has the following sections:

- "MPK Naming Conventions"
- "MPK Data Structures"
- "A Non-MPK Application Versus an MPK Application"
- "A Simple MPK Application"

MPK Naming Conventions

In large part, MPK follows the OpenGL naming conventions for its programming constructs. The primary MPK constructs are its data structures. Most of these data structures correspond to the graphics elements you will manage:

- Configuration (usually abbreviated to "Config" in names)
- Pipe
- Window
- Channel
- Compound (a decomposition mode)

This section describes how the data structures are named as well as the naming conventions for related functions and data constants.

MPK Data Structure Names

MPK uses a composite form for its data structure names: the MPK prefix plus the data structure type. MPK has the following user-accessible data structures:

- MPKConfig
- MPKPipe
- MPKWindow
- MPKChannel
- MPKCompound
- MPKEvent
- MPKFrame
- MPKGlobal (used to specify global default attributes)
- MPKImage

MPK Function Names

Generally, MPK function names have three parts:

1. mpk prefix
2. Data structure type
3. Action

The following are examples:

```
mpkWindowDelete()
mpkChannelApplyBuffer()
mpkCompoundGetRange()
```

There are special function names associated with MPKGlobal data structures. The section "MPKGlobal Attributes" in Appendix A describes the naming of these functions.

MPK Attribute Names

Like MPK data structure names and function names, MPK attribute names are composite names with an MPK prefix, but unlike the data structure and function names, attribute names use the underscore character (_) to separate the parts and the attribute names use all capital letters.

MPK attribute names have three or more parts, the first two of which are the following:

1. MPK prefix
2. Attribute type

`CATTR`	Specifies a channel attribute.
`PATTR`	Specifies a pipe attribute.
`WATTR`	Specifies a window attribute.
`DEFAULT`	Used only in the case of the stereo-related attribute `MPK_DEFAULT_EYE_OFFSET`.

The remaining parts contain additional attribute descriptors. The following are examples of attribute names:

```
MPK_CATTR_FAR
MPK_PATTR_STEREO_WIDTH
MPK_WATTR_HINTS_RGBA
```

MPK Data Structures

MPK encapsulates the graphics resources and rendering options in data structures. This section describes the hierarchy and function of these data structures as well as the special data structures MPKEvent, MPKFrame, MPKGlobal, and MPKImage and the interface MPKArena. The following subsections comprise this section:

- “The MPK Configuration Hierarchy”
- “The MPKConfig Data Structure”
- “The MPKPipe Data Structure”
- “The MPKWindow Data Structure”
- “The MPKChannel Data Structure”

- "The MPKCompound Data Structure"
- "The MPKEvent Data Structure"
- "The MPKGlobal Data Structure"
- "The MPKFrame and MPKImage Data Structures"
- "The MPKArena Interface"

The MPK Configuration Hierarchy

MPK uses a hierarchical tree to describe the configuration. The top-level data structure, the MPKConfig data structure, holds children of type MPKPipe. The MPKPipe data structure holds children of type MPKWindow, which hold children of type MPKChannel. As such, you can take advantage of the attendant inheritance. For instance, you can specify the screen dimensions at the MPKPipe level and they will be inherited by the child windows and child channels. This inheritance is made possible because MPK uses no absolute pixel dimensions but fractional viewport descriptions for its window and channels.

Figure 2-1 illustrates one possible configuration using two pipes, two windows, and four channels to render four different views. Example 2-1 illustrates the skeleton of the corresponding configuration file, which can be read using **mpkConfigLoad()**.

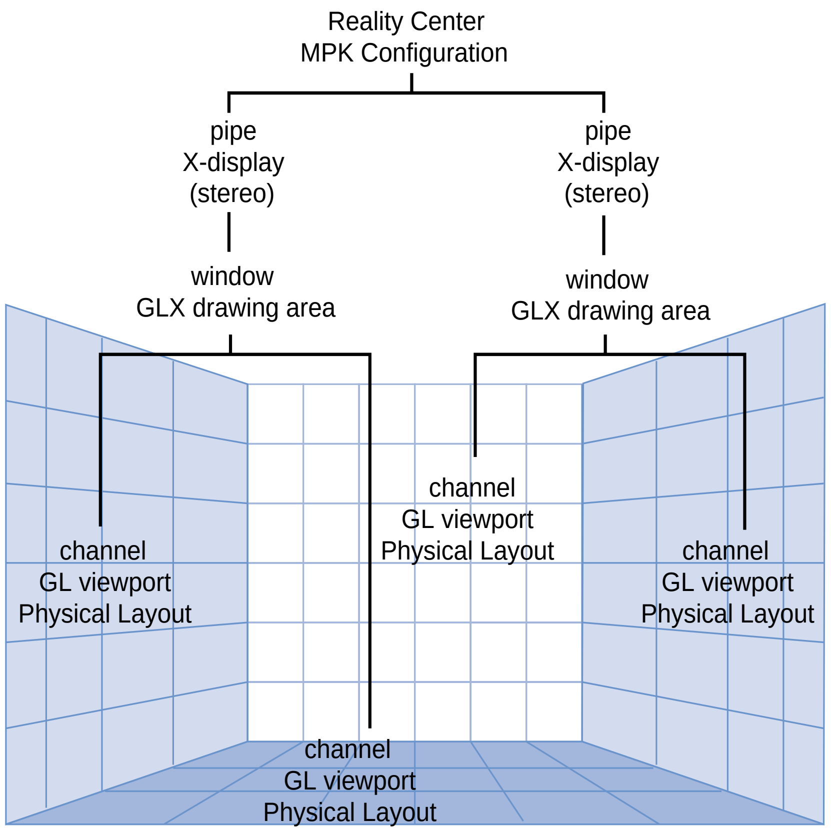

Figure 2-1 MPK Configuration Hierarchy

Example 2-1 MPK Data Structures in a Configuration File

```
config {
    pipe {
        window {
            viewport [ parameters1 ]
            channel {
                viewport [ parameters2 ]
                  .
                  .
                  .
            }
            channel {
                viewport [ parameters3 ]
                  .
                  .
                  .
             }
        }
    }
    pipe {
        window {
            viewport [ parameters4 ]
            channel {
                viewport [ parameters5 ]
                  .
                  .
                  .
            }
            channel {
                viewport [ parameters6 ]
                  .
                  .
                  .
             }
        }
    }
}
```

Reading this configuration file, MPK determines the following:

- What physical pipes it must allocate
- What parallel tasks it must create
- How to synchronize the rendering tasks
- The final rendering framebuffer area

The following sections describe the function of each data structure.

The MPKConfig Data Structure

The MPKConfig data structure primarily describes the rendering resources of an MPK application as a hierarchy of the following:

- Hardware rendering pipelines (MPKPipes)
- GLX software rendering threads (MPKWindows)
- OpenGL framebuffer rendering areas (MPKChannels)

It may also describe MPKCompounds, various parallelization schemes of the rendering across channels in order to scale performances.

You can read the MPKConfig data structure from an ASCII file using **mpkConfigLoad()** and launch the MPKConfig using **mpkConfigInit()**. Rendering threads are then spawned and the MPKConfig initialization callbacks invoked. These should in turn specify the rendering callbacks that will be triggered by **mpkConfigFrame()**.

The MPKPipe Data Structure

The MPKPipe data structure describes the rendering resources within an MPKConfig that are assigned to a given hardware rendering pipe. The pipe itself is characterized by the name of its corresponding X11 display as well as the expected mono and stereo mechanisms (full-screen, quad-buffer, and so on) to be applied by its rendering threads (MPKWindows).

You can specify the display sizes corresponding to the various stereo modes using MPKGlobal attributes; otherwise, MPK uses the values returned by

`DisplayWidth`(3X11) and `DisplayHeight`(3X11). Appendix A, “MPK Attributes” describes the MPKGlobal attributes.

The MPKWindow Data Structure

An MPKWindow data structure corresponds to a single GLX unit. A GLX unit is a single X window, pixel buffer (pbuffer), or X pixmap with its associated OpenGL visual and context. Essential in the MPK programming model is that each MPKWindow spawns its own rendering thread. MPK also supports nonthreaded windows, which are updated sequentially from the application thread.

The MPKChannel Data Structure

An MPKChannel data structure is essentially a view onto a scene and corresponds to a single viewport inside its parent MPKWindow. See the man page `glViewport`(3G) for information on viewports. In addition to the viewport description, an MPKChannel also contains the modeling coordinates for the projection rectangle in the real world.

The MPKCompound Data Structure

To achieve greater application performance, the MPKCompound data structure is used to describe a decomposition scheme as well as the recomposition by distributing the rendering workload across several graphics pipes.

It is essentially a container for children of type `MPKCompound`, each associated with an existing MPKChannel data structure. The rendering of the topmost MPKChannel in the hierarchy will be parallelized among the child channels by one or more of the following decomposition schemes:

- Portions of the destination viewport (mode `2D`)
- Portions of the frame data (mode `DB`)
- Stereo eye pass (mode `EYE` or `HMD`)
- Pipelined rendering cycles (mode `DPLEX` or `3D`)

Chapter 3, “Using Compounds” describes in detail the decomposition schemes.

The MPKEvent Data Structure

The MPKEvent data structure encapsulates an X11 event. It provides convenience functions to decode the data in the corresponding XEvent. Note that the MPKEvent is freed automatically by MPK. Hence, the pointer to an MPKEvent should not be stored within the application.

The MPKGlobal Data Structure

The MPKGlobal data structure specifies MPK default attribute values. It handles general default values—such as the execution mode, shared arena attributes, and the timer signal—as well as default attributes for the MPKPipe, MPKWindow, and MPKChannel data structures. These entities retrieve their default values during creation.

Appendix A, "MPK Attributes" describes the MPKGlobal attributes.

The MPKFrame and MPKImage Data Structures

The MPKFrame and MPKImage data structures provide access to the transported RGBA, depth, and stencil values during compound assembly. The section "Advanced Compositing" in Chapter 4 explains this API.

The MPKArena Interface

MPK provides a simple memory allocation interface that enables applications to allocate data regardless of their current execution mode. Any data that is shared between the rendering threads and the application thread should be allocated using **mpkMalloc()** or **mpkRealloc()**.

Internally, MPK may use a shared arena (see the `usinit`(3P) man page) to allocate memory. The default parameters of this arena can be changed using the MPKGlobal interface.

A Non-MPK Application Versus an MPK Application

The typical OpenGL application, as shown in Figure 2-2, is a single-threaded application with a main rendering loop. Within that loop it updates the scene database, draws a new frame, and processes user input. This application is constrained to single-window output, as it is unable to scale the rendering across multiple pipes. In theory, it is still possible to update several windows sequentially. However, this leads to no scalability since the application is single-threaded with each update adding time to the total frame time.

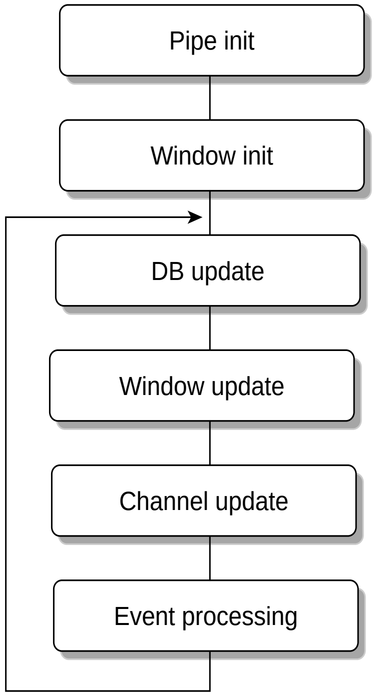

Figure 2-2 A Typical OpenGL Program

The evolution of an existing OpenGL application to a multithreaded, multipipe implementation requires application changes that are independent of the framework being used. The following steps describe the actions to be taken for this conversion:

1. Isolate scene graph manipulation and drawing.

   This first step isolates the application's rendering operation from its data-manipulation operation. Then, multiple rendering operations can execute concurrently on the application data in a manner that the rendering and data-manipulation operations do not modify each other′s variables. As a result, the rendering operation accesses the application data in a read-only mode and is limited to feeding the graphics pipeline. This separation requires a re-evaluation of the application's data structures for the channel-specific component. Therefore, you must consider the existing culling mechanisms. Although such mechanisms comprise an interface layer between channels and their associated views, they are typically specific to the application data or scene graph API. MPK ensures that its programming model does not infringe on any possible culling implementation.

2. Centralize events and data processing.

   Data access can be controlled by protecting thread-sensitive data (using a mutex or locks) , by engineering an entire application around a central data server (APP), or by integrating both of these methods. Although event processing is just one aspect of this issue, it impacts the entire design process.

Introducing a thread-safe implementation, this approach reflects OpenGL's "natural" application framework.

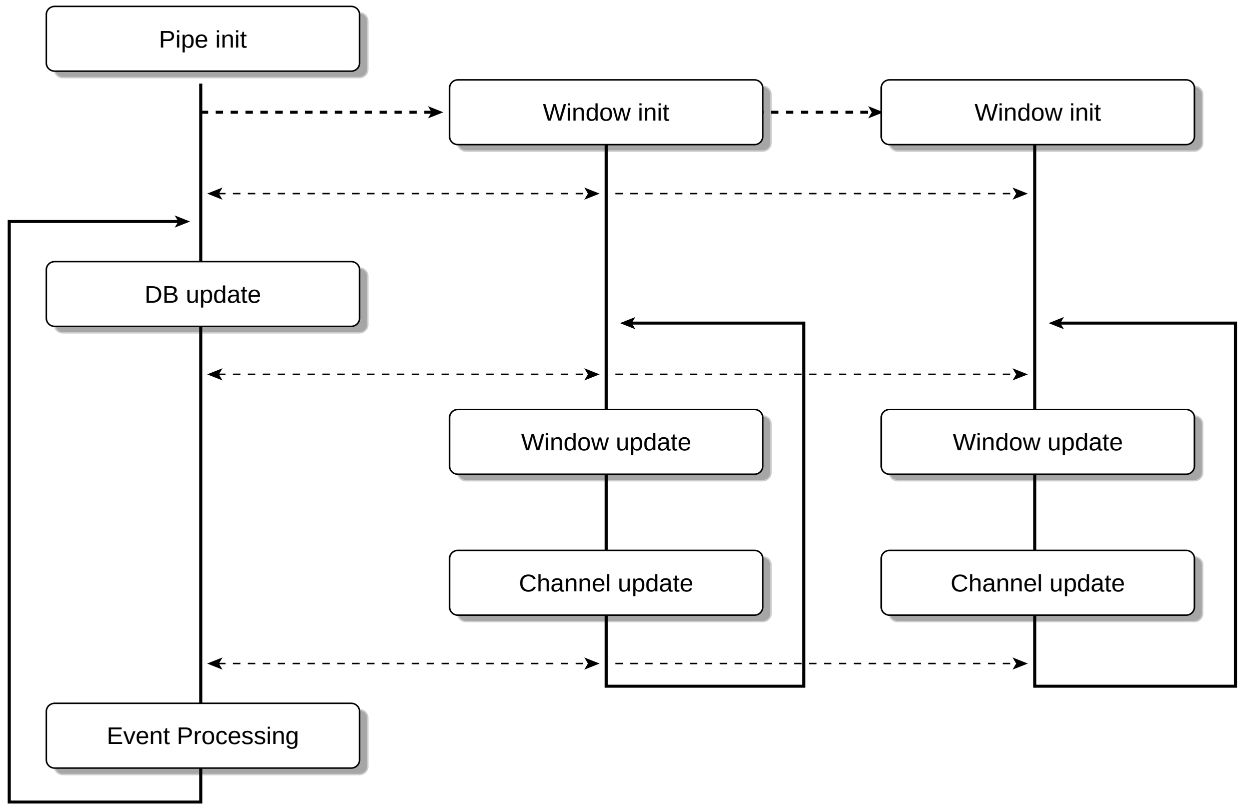

Figure 2-3 MPK Execution Framework

Once you complete the first two conversion steps, it is just a small step to use MPK as an application framework. You only need to transpose the drawing-related functions into their MPK counterparts. Figure 2-3 illustrates the MPK execution model.

The restructured application is now capable of parallel execution to scale its performance, as described in the section "Run-Time Scalability" in Chapter 1. MPK takes care of all the necessary synchronization.

A Simple MPK Application

This section describes the components of a very simple MPK program in the following subsections:

- "Creating and Initializing a Configuration"
- "The Main Loop"
- "The Rendering Callbacks"
- "Event Processing"
- "A Graceful Exit"
- "Example Code"

All the code fragments used in the various subsections to explain the parts of an MPK program are put together in the last subsection "Example Code" to compose an executable example.

Creating and Initializing a Configuration

Example 2-2 shows the typical initialization sequence for a simple MPK program. Each of the MPK calls are described after the example.

Example 2-2 A Simple MPK Initialization Sequence

```
MPKConfig *config;

mpkGlobalSetExecutionMode( MPK_EXECUTION_PTHREAD );

mpkInit();

// shared must be allocated after mpkInit().
shared = mpkMalloc( sizeof( Shared ));
initSharedData( shared );

if (argc < 2)
    config = mpkConfigLoad(“../configs/1-window”);
else
    config = mpkConfigLoad(argv[1]);

if ( config == NULL )
```

```
{
    fprintf(stderr, “Can’t load config file.\n”);
    exit (0);
}

mpkConfigOutput( config, 0 );

shared->stereo = ( mpkConfigGetMode(config) == MPK_STEREO ? 1 : 0 );

mpkConfigSetWindowInitCB(config,initWindow);
mpkConfigSetWindowExitCB(config,NULL);
mpkConfigSetChannelInitCB(config,initChannel);
mpkConfigSetDataFreeCB( config, freeFrameData );

mpkConfigInit( config, 0 );
```

The function **mpkGlobalSetExecutionMode()** selects the threading model used by this application. MPK supports `pthread`, `sproc`, or `fork` execution modes. Since the execution mode has to be known before initialization, it is set before **mpkInit()**. The default execution mode is `pthread`.

Calling **mpkInit()** initializes internal MPK data structures and the shared arena if necessary. This call has to be the first MPK call in an application, except for the following:

- **mpkGetString()**
- **mpkGlobalSetExecutionMode()**
- **mpkGlobalSetArenaAttribute()**
- **mpkGlobalSetArenaPath()**

The shared arena is used to allocate shared memory in `fork` execution mode as well as to create synchronization primitives in `fork` and `sproc` execution mode.

The next step for the application is to initialize its shared global data. Note that global data should always be located in a block of memory allocated by **mpkMalloc()** or **mpkCalloc()** to ensure the data is accessible by all threads. See “Data Handling” in Chapter 4 for more details.

After MPK and the application is initialized, an MPKConfig data structure is created. In this case, **mpkConfigLoad()** is used to read an ASCII configuration file into an MPKConfig structure. This could be done by other means—for example, by constructing an MPKConfig structure programmatically or by writing an alternate parser. See Chapter 4, “Advanced MPK Programming” for a description of the alternative approaches. The function **mpkConfigOutput()** can be used to print the given MPKConfig in a format compatible with **mpkConfigLoad()**.

The stereo mode, as specified in the configuration file, is obtained using **mpkConfigGetMode()**. This value is later needed during the main loop and in the event callbacks.

Some MPKConfig callbacks have to be set in order to run this configuration properly. These are the window initialization and channel initialization callbacks as well as the data deallocation callback. The initialization callbacks are explained in the following paragraphs. The purpose of the free-data callback is explained in the section “The Main Loop” later in this chapter.

Finally, the configuration is initialized by calling **mpkConfigInit()**. For each data structure in the configuration hierarchy, the MPKConfig initialization callback is invoked.

From the application thread perspective, the initialization of the threaded windows consists of simply launching the window thread. The first thing that is invoked from the newly created window rendering thread is the MPKConfig’s window initialization callback and the initialization callbacks for all channels of this window.

Nonthreaded windows are initialized from the application thread sequentially. All threaded window and channel initialization callbacks are called from their respective window threads. Therefore, the window initialization and channel initialization happen in parallel. Any critical data access in the window or channel initialization callbacks has to be protected using mutual exclusion.

The default window initialization callback is **mpkWindowCreate()**. Most applications overwrite the default callback in order to do per-window initialization. Example 2-3 shows a simple window initialization callback.

Example 2-3 Simple Window Initialization Callbacks

```
void initWindow( MPKWindow *w )
{
    // MPKWindow initialization

    mpkWindowSetDrawCB(w,MPK_WINDOW_DRAWCB_INIT_X,initWindowX);
    mpkWindowSetDrawCB(w,MPK_WINDOW_DRAWCB_INIT_GL,initWindowGL);
    mpkWindowSetDrawCB(w,MPK_WINDOW_DRAWCB_EXIT_X,exitWindowX);
    mpkWindowSetDrawCB(w,MPK_WINDOW_DRAWCB_EXIT_GL,exitWindowGL);

    mpkWindowSetEventCB(w,MPK_WINDOW_EVENTCB_MOUSE,windowMouse);
    mpkWindowSetEventCB(w,MPK_WINDOW_EVENTCB_BUTTON,windowMouse);
    mpkWindowSetEventCB(w,MPK_WINDOW_EVENTCB_EXIT,windowExit);
    mpkWindowSetEventCB(w,MPK_WINDOW_EVENTCB_KEYBOARD,windowKeyboard);
}

void initWindowX( MPKWindow *w )
{
    // X11 initialization
    mpkWindowCreate( w );
}

void initWindowGL( MPKWindow *w )
{
    // create ctx
    mpkWindowCreateContext( w );
    mpkWindowMakeCurrent( w );

    // GL initialization
    mpkWindowApplyViewport( w );

    glDepthFunc( GL_LEQUAL );
    glEnable( GL_DEPTH_TEST );

    glEnable( GL_LIGHTING );
    glEnable( GL_LIGHT0 );
    glLightfv( GL_LIGHT0, GL_POSITION, lightpos );

    glColorMaterial( GL_FRONT_AND_BACK, GL_DIFFUSE );
    glEnable( GL_COLOR_MATERIAL );

    glClearDepth( 1. );
    glClearColor( 0., 0., 0., 1. );
    // clear both buffers
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
    mpkWindowSwapBuffers(w);
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
}
```

For window initialization, there are three stages:

- MPKWindow initialization
- X window initialization
- OpenGL initialization

During the MPKWindow initialization, all necessary event callbacks are set. The event callbacks are described in the later section "Event Processing". The X initialization creates the window. The function **mpkWindowCreate()** performs the following operations:

```
mpkWindowOpenDisplay( window );

GLXFBConfig *fbconfig = mpkWindowChooseFBConfig( window, &n );
mpkWindowSetFBConfig( window, fbconfig[0] );

mpkWindowCreateDrawable( window );
mpkWindowMapDrawable( window );
```

The **initGL()** function creates an OpenGL context and initializes OpenGL.At the end of the OpenGL initialization callback, both draw buffers are cleared in order to avoid visual artifacts during the initial frames. The function **mpkWindowSwapBuffers()** should always be used instead of **glXSwapBuffers()**.

The channel initialization code, as shown in Example 2-4, is quite simple: it just sets this channel's clear and update draw callbacks. The purpose of these callbacks is explained in the later section "The Rendering Callbacks".

Example 2-4 A Sample Channel Initialization Callback

```
void initChannel( MPKChannel *c )
{
    mpkChannelSetDrawCB( c, MPK_CHANNEL_DRAWCB_CLEAR, clearChannel );
    mpkChannelSetDrawCB( c, MPK_CHANNEL_DRAWCB_UPDATE, updateChannel );
}
```

The Main Loop

The main rendering loop, as shown in Example 2-5, performs two basic operations: it updates the application's database based on user input and draws a new frame in an endless loop.

Example 2-5 The Main Rendering Loop

```
while (!shared->exit )
{
    // update DB
    incrementRotation(  shared->rotation,
                        shared->xangle,
                        shared->yangle );

    shared->translation[0] += shared->dx;
    shared->translation[1] += shared->dy;
    shared->translation[2] += shared->dz;

    shared->dx = 0.;
    shared->dy = 0.;
    shared->dz = 0.;


    // new frame
    mpkConfigChangeMode( config, shared->stereo );
    framedata = newFrameData( shared );
    mpkConfigFrame( config, framedata );
}
```

The function **mpkConfigChangeMode()** changes the configuration stereo mode to the passed mode. If the new mode is the same as the old, then the change is ignored. Switching the stereo mode involves exiting and restarting the configuration if any of the windows that will use quad-buffered stereo does not have a stereo-capable visual. For that reason, if a configuration file uses quad-buffered stereo, the MPKGlobal attribute `MPK_WATTR_HINTS_STEREO` should be set to `1` in the global section of the configuration file.

Finally, a new frame is drawn by calling **mpkConfigFrame()**. MPK provides a transport mechanism for frame data to render latency-correct frames. For certain decompositions, as discussed in Chapter 3, "Using Compounds", a channel draws an older frame than the current one. For that reason, frame-specific data—for example, the current translation of the scene—has to be managed through the MPK frame data mechanism. MPK keeps track of older frame data and always passes the appropriate frame data to the rendering callbacks. In order to free old frame data, MPK calls the free-data callback for the MPKConfig structure. The MPK internal data structures are latency-aware. For example, a channel's viewport in a rendering callback will be computed according to the current latency.

Example 2-6 uses the functions **newFrameData()** and **freeFrameData()** to manage the application's frame data. A simple linked list is used to recycle old frame data structures, in order to avoid subsequent **mpkMalloc()** and **mpkFree()** calls for each frame. To generate a new frame data structure, **newFrameData()** gets a structure allocated using **mpkMalloc()** and fills in frame-dependent data from the application's database. In this example, this is the translation and rotation of the scene. The free-data callback inserts the old structure into a linked list to recycle it for new use by **newFrameData()**.

Example 2-6 The Frame Data-Handling Functions

```
FrameData *newFrameData( Shared *shared )
{
    FrameData *framedata;
    if ( frameDataBuffer == NULL )
    {
        framedata = (FrameData *) mpkMalloc( sizeof(FrameData) );
    }
    else
    {
        framedata = frameDataBuffer;
        frameDataBuffer = framedata->next;
    }

    framedata->next = NULL;

    memcpy( framedata->translation, shared->translation,
     3*sizeof(float) );
    memcpy( framedata->rotation, shared->rotation, 16*sizeof(float) );

    return framedata;
}

void freeFrameData( MPKConfig *cfg, void *data )
{
    FrameData *framedata = (FrameData *)data;
    framedata->next = frameDataBuffer;
    frameDataBuffer = framedata;
}
```

The Rendering Callbacks

This section explains what actually happens during a **mpkConfigFrame()** call: what callbacks are invoked, their invocation order, and how the rendering threads are synchronized. Figure 2-4, not taking compounds into account, shows a simplified diagram of the execution.

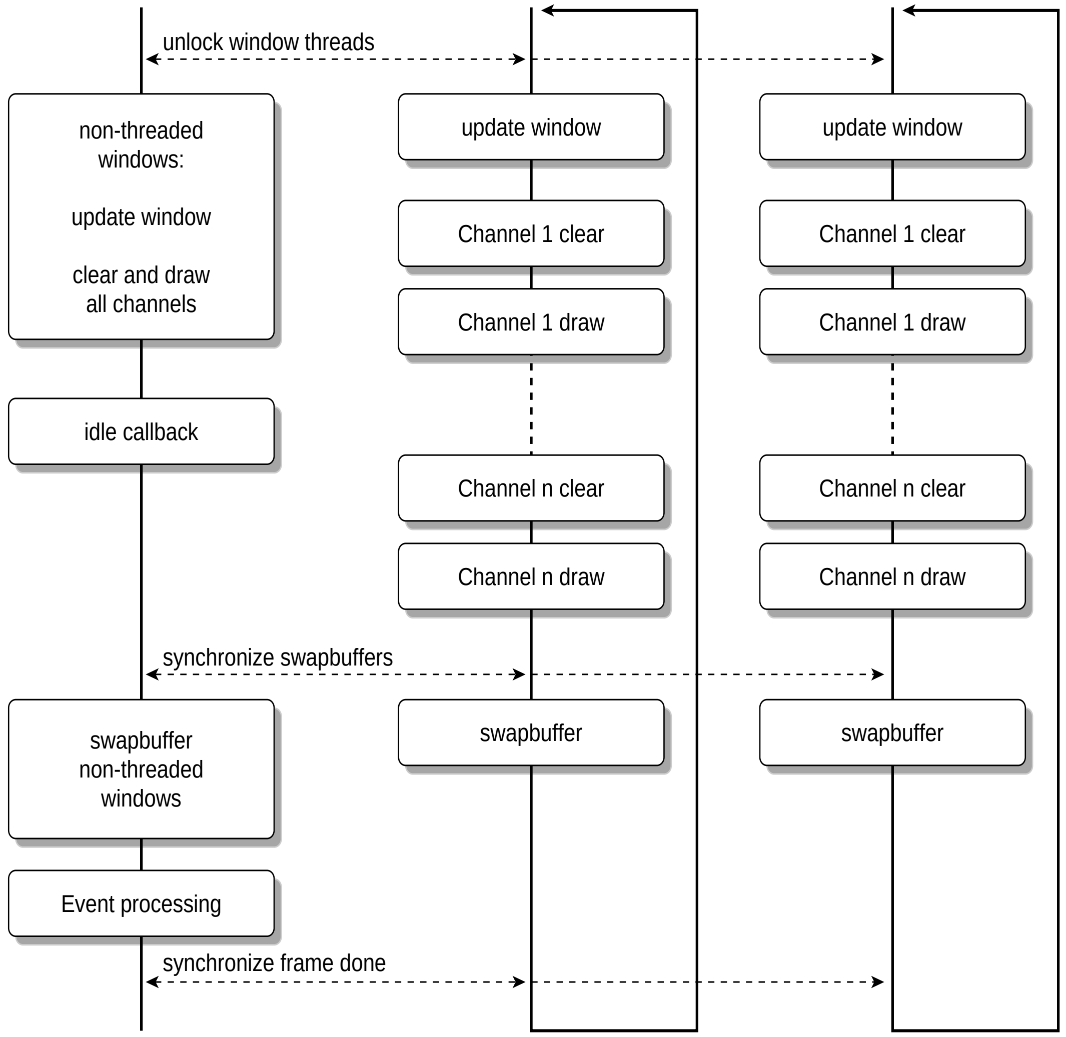

Figure 2-4 A Simplified **mpkConfigFrame()** Call

First, the function **mpkConfigFrame()** unlocks the rendering threads. This action invokes the update-window draw callback and then calls the update callbacks for each channel.

Usually, the application thread is idle while the window threads are drawing. Some applications may perform an intermittent task during this time—for example, to pipeline their culling— but the application must avoid modifying the data currently being used by the rendering processes. The MPKConfig idle callback serves this purpose.

The abstraction of the rendering from the main application enables an MPK application to transparently use stereo rendering. When running in stereo mode, MPK calls the update callbacks twice for each channel: once for the left eye and once for the right eye.

The update-window draw callback is called once at the beginning of each frame. Typically, this is the place to update the OpenGL context, for example, by creating new texture objects. This callback is not used in the example code in this section.

MPK, in contrast to other multipipe programming models, separates the update of a channel into two callbacks, clear and draw. The separation is necessary to do the recomposition during compound processing, as explained in Chapter 3, "Using Compounds".

A simple clear callback is shown in Example 2-7.

Example 2-7 A Channel Clear Callback

```
void clearChannel( MPKChannel *c, void *data )
   {
       mpkChannelApplyBuffer( c );
       mpkChannelApplyViewport( c );

       glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
   }
```

The purpose of the clear callback is to set up the current channel and clear its OpenGL framebuffer as needed. The function **mpkChannelApplyBuffer()** sets the correct OpenGL read and draw buffers (see the `glReadBuffer`(3G) and `glDrawBuffer`(3G) man pages) according to the current stereo mode and eye pass.

The function **mpkChannelApplyViewport()** applies the current OpenGL viewport and scissor area (see the `glViewport`(3G) and `glScissor`(3G) man pages). The channel's pixel viewport is computed from the parent window's pixel viewport (that is, its width and height) and the channel's fractional viewport using the following formula:

```
#define IRND(a) ((int)((a)+.5))

// compute first pixel position of the channel
channel.pvp[0] = IRND(channel.vp[0] * window.pvp[2]);
channel.pvp[1] = IRND(channel.vp[1] * window.pvp[3]);

// compute last pixel position of the channel
channel.pvp[2] = IRND((channel.vp[0]+channel.vp[2]) * window.pvp[2]);
channel.pvp[3] = IRND((channel.vp[1]+channel.vp[3]) * window.pvp[3]);

// compute channel’s dimension
channel.pvp[2] -= channel.pvp[0];
channel.pvp[3] -= channel.pvp[1];
```

This method honors positions over dimensions in order to ensure adjacency whenever possible—for example, in a 1280x1024 window:

```
vp(1): [0.     0. 0.3333 1. ]    pvp(1): [0   0 427 1024]
vp(2): [0.3333 0. 0.3333 1. ]    pvp(2): [427 0 426 1024]
```

Note that in full-screen stereo mode (type `rect`) during the left eye pass, the value of the MPKGlobal variable `MPK_DATTR_FULLSTEREO_OFFSET` will be added to `channel.pvp[1]`.

Example 2-8 shows a simple update-channel draw callback. The purpose of the draw callback is to render a new frame based on the current frustum and frame data for this channel.

Example 2-8 A Channel Draw Callback

```
void updateChannel( MPKChannel *c, void *data )
{
    FrameData *framedata = (FrameData *)data;

    glMatrixMode( GL_PROJECTION );
    glLoadIdentity();
    mpkChannelApplyFrustum( c );

    glMatrixMode( GL_MODELVIEW );
    glLoadIdentity();
    mpkChannelApplyTransformation( c );

    glTranslatef(  framedata->translation[0],
                   framedata->translation[1],
                   framedata->translation[2] );
    glMultMatrixf( framedata->rotation );

    drawcube();
}
```

Since MPK manages the frustum, the draw callback should always use MPK functions to set up the frustum. The function **mpkChannelApplyFrustum()** applies an OpenGL frustum matrix for the passed MPKChannel with respect to the current eye pass, eye position, and the channel's physical layout specification. For orthographic views, MPK provides the call **mpkChannelApplyOrtho()** as an alternative to **mpkChannelApplyFrustum()**. MPK uses the mode `MPK_ORTHO_STILL` or `MPK_ORTHO_TRACKED` to apply an OpenGL orthographic matrix. If the orthographic mode is `MPK_ORTHO_STILL`, then **mpkChannelApplyOrtho()** simply uses the half-width and half-height dimensions of the channel layout to produce the distances used in **glOrtho()**. Otherwise, if orthographic mode is `MPK_ORTHO_TRACKED`, then **mpkChannelApplyOrtho()** uses the current view direction (for example, from **mpkConfigSetHeadOrientation()**) to produce consistent viewing across all channels in the configuration. Figure 2-5 illustrates an example of `MPK_ORTHO_STILL` (left side) and `MPK_ORTHO_TRACKED` (right side). The argument `zoom` specifies two-dimensional scaling on the X and Y screen coordinates.

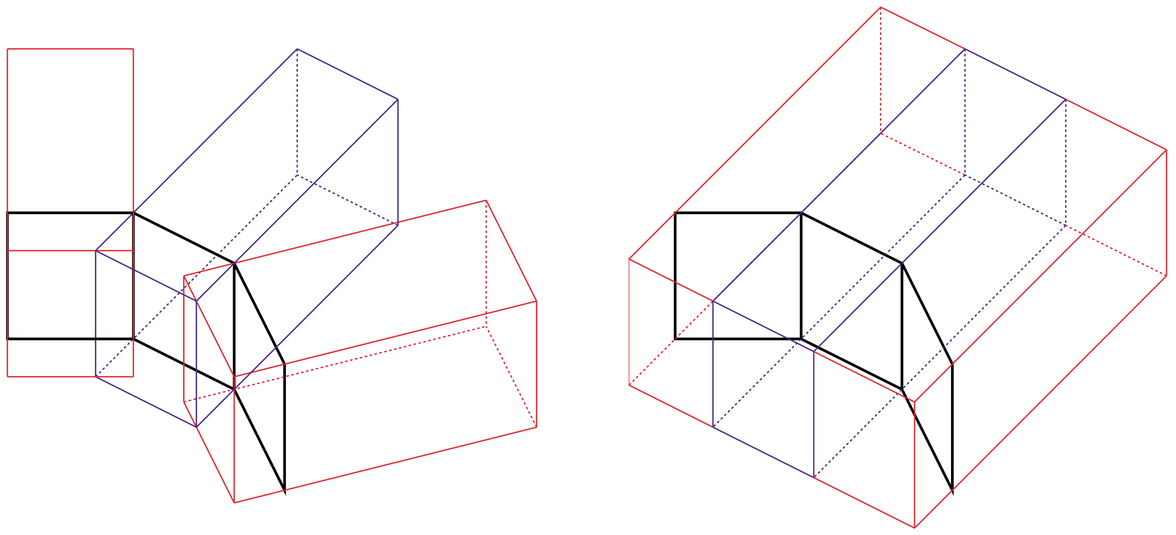

Figure 2-5 `MPK_ORTHO_STILL` and `MPK_ORTHO_TRACKED` Frusta

To position and orient the frustum specified by **mpkChannelApplyFrustum()** or **mpkChannelApplyOrtho()**, the function **mpkChannelApplyTransformation()** applies the necessary modeling transformation.

Once the frustum is set up correctly, the scene is positioned according to the rotation and translation of the current frame data. The function **drawcube()** draws the database, a colored cube.

Event Processing

MPK provides a flexible solution to process events. The event gathering and processing is centralized in the application thread.

In order to receive events, each window has an input display, which is created during **mpkWindowOpenDisplay()**. MPK uses **XSelectInput()** to request events on a given window. The functions **mpkConfigSelectInput()**, **mpkPipeSelectInput()**, and **mpkWindowSelectInput()** can be used to change the event mask for all windows in the given hierarchy.

The event processing is done at the end of the frame by calling an event processing callback. By default, this callback is set to **mpkConfigHandleEvents()**. The callback polls for XEvents on all input displays and encapsulates the XEvents in MPKEvents. This MPKEvent is then processed by the any-event callback of the matching window. The default any-event callback, **mpkWindowProcessEvent()**, calls the other window event callbacks based on the event type. The pseudo code for the default event processing in **mpkConfigHandleEvents()** is shown in Example 2-9.

Example 2-9 Pseudo Code for **mpkConfigHandleEvents()**

```
foreach input display connection
  while XEvent pending
    receive XEvent
    find matching MPKWindow
    encapsulate XEvent in MPKEvent
    update MPKEventXData
    call window any-event callback
      | default:
      | mpkWindowProcessEvent
      |   call exit, expose, configure, mouse, button or keyboard
      |   event callback based on event type
  end while
end for
```

This architecture enables MPK to provide support for the various event processing scenarios:

- Event-driven applications

  An application that is event-driven redraws only when user input or other events require a redraw. Usually, these applications set the configuration's event callback to `NULL` and process the events themselves by using **mpkConfigNextEvent()** and **mpkConfigCheckEvent().** This enables the application to issue a new **mpkConfigFrame()** call whenever necessary. The example `flip.eventDriven` shows such an application.

- Applications that are not event-driven

  For some applications it is necessary to continously draw new frames, for example, to display animations. MPK's default event processing does not block for new events; thus, it returns as soon as all pending events are processed. This leads naturally to the desired behavior.

- No MPK event processing

  Some applications already have their own event processing model. By setting the window's input display to `NULL` during window initialization and setting the configuration's event callback to `NULL`, MPK event processing is disabled.

As cited in an earlier section, the MPKEvent data structure encapsulates an X11 event. It provides convenience functions to decode the data in the corresponding XEvent. Note that the MPKEvent is freed automatically by MPK. Hence, the pointer to an MPKEvent should not be stored within the application.

The event callbacks of the example program in this section, as shown in Example 2-10, provide some mouse interaction as well as the possibility to switch the stereo mode by pressing `s`.

Example 2-10 Window Event Callbacks for Mouse, Keyboard, and Exit

```
void windowMouse( MPKWindow *w, MPKEvent *event )
{
    MPKEventXData *data = (MPKEventXData *) mpkEventGetData( event );

    if ( data->button.left )
    {
        if ( data->button.middle )
        {
            shared->dz += (float) data->mouse.dy/200.;
        }
        else
        {
            shared->dx += (float) data->mouse.dx/500.;
            shared->dy -= (float) data->mouse.dy/500.;
        }
    }
    else if ( data->button.middle )
    {
        shared->xangle = (float) data->mouse.dy*.5;
        shared->yangle  = (float) data->mouse.dx*.5;
    }
}

void windowExit( MPKWindow *w, MPKEvent *event )
{
    shared->exit = GL_TRUE;
}

void windowKeyboard( MPKWindow *w, MPKEvent *event )
{
    MPKEventXData *data = (MPKEventXData *)mpkEventGetData( event );

    if ( data->keyboard.state != MPK_PRESS )
        return;

    switch( data->keyboard.key )
    {
        case XK_S:
        case XK_s:
            shared->stereo = !shared->stereo;
            break;
    }
}
```

A Graceful Exit

Eventually, application processing leaves the main loop. In the sample program in this section, the exit-window callback, which is called whenever `Esc` is pressed, sets the exit flag to true. This causes the main loop shown in Example 2-5 in section "The Main Loop" to terminate.

Example 2-11 shows a simple exit sequence.

Example 2-11 A Simple Exit Sequence

```
//---------- restore MONO

mpkConfigSetWindowInitCB(config,NULL);
mpkConfigSetChannelInitCB(config,NULL);
mpkConfigChangeMode( config, MPK_MONO );

//---------- exit & delete config

mpkConfigExit( config );
mpkConfigDelete( config );

exitSharedData( shared );
mpkFree( shared );

mpkExit();
```

The first part of Example 2-11 restores the mono mode upon exiting. This may not be desired for other applications, but in this case it is provided for convenience. The initialization callbacks are set to `NULL` to avoid unnecessary window creation in case the configuration is restarted.

The second part contains the actual cleanup. It exits the configuration; this action will cause the exit callbacks, shown in Example 2-12, to be called in the reverse order of the initialization callbacks.

In Example 2-11, nothing has to be done in order to exit the MPKWindow. Therefore, the MPKConfig's exit-window callback is set to NULL. The X exit callback calls **mpkWindowDestroy()**, which performs the following operations:

```
mpkWindowDestroyDrawable( window );
mpkWindowDestroyFBConfig( window );

mpkWindowCloseDisplay( window );
```

The OpenGL exit callback destroys the OpenGL context.

Example 2-12 The Exit Callbacks

```
//----------------------------------------------------------------------
// exitWindowX
//----------------------------------------------------------------------
void exitWindowX( MPKWindow *w )
{
    mpkWindowDestroy( w );
}

//----------------------------------------------------------------------
// exitWindowGL
//----------------------------------------------------------------------
void exitWindowGL( MPKWindow *w )
{
    // destroy ctx
    mpkWindowMakeCurrentNone( w );
    mpkWindowDestroyContext( w );
}
```

After **mpkConfigExit()** is called, all window threads are terminated. The MPKConfig data structure and all of its children are freed by calling **mpkConfigDelete()**. Next, before the final **mpkExit()**, the shared data is deinitialized and freed using **mpkFree()**, the **mpkMalloc()** counterpart.

Example Code

The source code in Example 2-13 is the full example for the simple MPK application described part by part in the preceding subsections.

Example 2-13 A Simple MPK Application

```
/* compile using ‘cc -o example example.c -lm -lmpk -lGL -lpthread’ */

#include <mpk/mpk.h>

#include <X11/keysym.h>
#include <math.h>
#include <stdio.h>

#ifndef M_PI
#define M_PI 3.1415926535
#endif

#define DEG2RAD(a)  ((a)*M_PI/180.)

typedef struct
{
    int    stereo;
    int    exit;

    float   xangle, yangle,
            dx, dy, dz,
            translation[3],
            rotation[16];

} Shared;

typedef struct _FrameData
{
    float   translation[3],
            rotation[16];

    struct _FrameData *next;

} FrameData;

Shared    *shared;
FrameData *frameDataBuffer = NULL;
FrameData *newFrameData( Shared *shared );
```

```
void       freeFrameData( MPKConfig *, void * );

void    initSharedData( Shared *shared );
void    exitSharedData( Shared *shared );

void    initWindow( MPKWindow * );
void    initWindowX( MPKWindow * );
void    initWindowGL( MPKWindow * );
void    exitWindowX( MPKWindow * );
void    exitWindowGL( MPKWindow * );

void    initChannel( MPKChannel *c );
void    clearChannel( MPKChannel *, void * );
void    updateChannel( MPKChannel *, void * );

void    windowMouse( MPKWindow *, MPKEvent * );
void    windowExit( MPKWindow *, MPKEvent * );
void    windowKeyboard( MPKWindow *, MPKEvent * );

void    incrementRotation( float *, float, float );
void    drawcube();

static float lightpos[] = { 0., 0., 1., 0. };

//----------------------------------------------------------------------
//  main
//----------------------------------------------------------------------
main( int argc, char *argv[] )
{
    MPKConfig *config;
    FrameData *framedata;

    mpkGlobalSetExecutionMode( MPK_EXECUTION_PTHREAD );

    mpkInit();

    // shared must be allocated after mpkInit().
    shared = mpkMalloc( sizeof( Shared ));
    initSharedData( shared );

    if (argc < 2)
        config = mpkConfigLoad("../configs/1-window");
    else
        config = mpkConfigLoad(argv[1]);
```

```
if ( config == NULL )
{
    fprintf(stderr, “Can’t load config file.\n”);
    exit (0);
}

mpkConfigOutput( config, 0 );

shared->stereo = ( mpkConfigGetMode(config)==MPK_STEREO ? 1 : 0 );

mpkConfigSetWindowInitCB(config,initWindow);
mpkConfigSetWindowExitCB(config,NULL);
mpkConfigSetChannelInitCB(config,initChannel);
mpkConfigSetDataFreeCB( config, freeFrameData );

mpkConfigInit( config, 0 );

while (!shared->exit )
{
    // update DB
    incrementRotation(  shared->rotation,
        shared->xangle,
        shared->yangle );

    shared->translation[0] += shared->dx;
    shared->translation[1] += shared->dy;
    shared->translation[2] += shared->dz;

    shared->dx = 0.;
    shared->dy = 0.;
    shared->dz = 0.;

    // new frame
    mpkConfigChangeMode( config, shared->stereo );
    framedata = newFrameData( shared );
    mpkConfigFrame( config, framedata );
}

//---------- restore MONO

mpkConfigSetWindowInitCB(config,NULL);
mpkConfigSetChannelInitCB(config,NULL);
mpkConfigChangeMode( config, MPK_MONO );

//---------- exit & delete config
```

```
    mpkConfigExit( config );
    mpkConfigDelete( config );

    exitSharedData( shared );
    mpkFree( shared );

    mpkExit();
}

//-----------------------------------------------------------------------
//  initSharedData
//-----------------------------------------------------------------------
void initSharedData( Shared *shared )
{
    int i, j;

    shared->exit = 0;
    shared->stereo = MPK_MONO;
    shared->yangle = 0.;
    shared->xangle = 0.;

    for (i=0; i<4; i++)
        for (j=0; j<4; j++)
            shared->rotation[4*i+j] = (i==j) ? 1. : 0.;

    shared->dx = 0.;
    shared->dy = 0.;
    shared->dz = 0.;
    shared->translation[0] = 0.;
    shared->translation[1] = 0.;
    shared->translation[2] = -2.;
}

//-----------------------------------------------------------------------
//  exitSharedData
//-----------------------------------------------------------------------
void exitSharedData( Shared *shared )
{
    while ( frameDataBuffer != NULL )
    {
        FrameData *framedata = frameDataBuffer;

        frameDataBuffer = framedata->next;
        mpkFree( framedata );
    }
}

//----------------------------------------------------------------------
//  initWindow
//----------------------------------------------------------------------
void initWindow( MPKWindow *w )
{
    mpkWindowSetDrawCB( w, MPK_WINDOW_DRAWCB_INIT_X, initWindowX );
    mpkWindowSetDrawCB( w, MPK_WINDOW_DRAWCB_INIT_GL, initWindowGL );
    mpkWindowSetDrawCB( w, MPK_WINDOW_DRAWCB_EXIT_X, exitWindowX );
    mpkWindowSetDrawCB( w, MPK_WINDOW_DRAWCB_EXIT_GL, exitWindowGL );

    mpkWindowSetEventCB(w,MPK_WINDOW_EVENTCB_MOUSE,windowMouse);
    mpkWindowSetEventCB(w,MPK_WINDOW_EVENTCB_BUTTON,windowMouse);
    mpkWindowSetEventCB(w,MPK_WINDOW_EVENTCB_EXIT,windowExit);
    mpkWindowSetEventCB(w,MPK_WINDOW_EVENTCB_KEYBOARD,windowKeyboard);
}

//----------------------------------------------------------------------
// initWindowX
//----------------------------------------------------------------------
void initWindowX( MPKWindow *w )
{
    mpkWindowCreate( w );
}

//----------------------------------------------------------------------
// initWindowGL
//----------------------------------------------------------------------
void initWindowGL( MPKWindow *w )
{
    // create ctx
    mpkWindowCreateContext( w );
    mpkWindowMakeCurrent( w );

    // GL initialization
    mpkWindowApplyViewport( w );

    glEnable( GL_DEPTH_TEST );
    glDepthFunc (GL_LESS);

    glEnable( GL_LIGHTING );
    glEnable( GL_LIGHT0 );
    glLightfv( GL_LIGHT0, GL_POSITION, lightpos );

    glColorMaterial( GL_FRONT_AND_BACK, GL_DIFFUSE );
    glEnable( GL_COLOR_MATERIAL );

    glClearDepth( 1. );
    glClearColor( 0., 0., 0., 1. );

    // clear both buffers
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
    mpkWindowSwapBuffers(w);
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
}

//----------------------------------------------------------------------
// exitWindowX
//----------------------------------------------------------------------
void exitWindowX( MPKWindow *w )
{
    mpkWindowDestroy( w );
}

//----------------------------------------------------------------------
// exitWindowGL
//----------------------------------------------------------------------
void exitWindowGL( MPKWindow *w )
{
    // destroy ctx
    mpkWindowMakeCurrentNone( w );
    mpkWindowDestroyContext( w );
}

//----------------------------------------------------------------------
//  initChannel
//----------------------------------------------------------------------
void initChannel( MPKChannel *c )
{
    mpkChannelSetDrawCB( c, MPK_CHANNEL_DRAWCB_CLEAR, clearChannel );
    mpkChannelSetDrawCB( c, MPK_CHANNEL_DRAWCB_UPDATE, updateChannel );
}

//----------------------------------------------------------------------
//  newFrameData
//----------------------------------------------------------------------
FrameData *newFrameData( Shared *shared )
```

```
{
    FrameData *framedata;
    if ( frameDataBuffer == NULL )
    {
        framedata = (FrameData *) mpkMalloc( sizeof(FrameData) );
    }
    else
    {
        framedata = frameDataBuffer;
        frameDataBuffer = framedata->next;
    }

    framedata->next = NULL;

    memcpy( framedata->translation,
            shared->translation, 3*sizeof(float) );
    memcpy( framedata->rotation, shared->rotation, 16*sizeof(float) );

    return framedata;
}

//----------------------------------------------------------------------
//  freeFrameData
//----------------------------------------------------------------------
void freeFrameData( MPKConfig *cfg, void *data )
{
    FrameData *framedata = (FrameData *)data;

    framedata->next = frameDataBuffer;
    frameDataBuffer = framedata;
}

//----------------------------------------------------------------------
//  clearChannel
//----------------------------------------------------------------------
void clearChannel( MPKChannel *c, void *data )
{
    mpkChannelApplyBuffer( c );
    mpkChannelApplyViewport( c );

    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
}

//----------------------------------------------------------------------
//  updateChannel
```

```
//----------------------------------------------------------------------
void updateChannel( MPKChannel *c, void *data )
{
    FrameData *framedata = (FrameData *)data;

    glMatrixMode( GL_PROJECTION );
    glLoadIdentity();

    mpkChannelApplyFrustum( c );

    glMatrixMode( GL_MODELVIEW );
    glLoadIdentity();
    mpkChannelApplyTransformation( c );

    glTranslatef(  framedata->translation[0],
        framedata->translation[1],
        framedata->translation[2] );

    glMultMatrixf( framedata->rotation );

    drawcube();
}

//----------------------------------------------------------------------
//  windowMouse
//----------------------------------------------------------------------
void windowMouse( MPKWindow *w, MPKEvent *event )
{
    MPKEventXData *data = (MPKEventXData *) mpkEventGetData( event );

    if ( data->button.left )
    {
        if ( data->button.middle )
        {
            shared->dz += (float) data->mouse.dy/200.;
        }
        else
        {
            shared->dx += (float) data->mouse.dx/500.;
            shared->dy -= (float) data->mouse.dy/500.;
        }
    }
    else if ( data->button.middle )
    {
        shared->xangle = (float) data->mouse.dy*.5;
        shared->yangle  = (float) data->mouse.dx*.5;
    }
}

//-----------------------------------------------------------------------
//  windowExit
//-----------------------------------------------------------------------
void windowExit( MPKWindow *w, MPKEvent *event )
{
    shared->exit = GL_TRUE;
}

//-----------------------------------------------------------------------
//  windowKeyboard
//-----------------------------------------------------------------------
void windowKeyboard( MPKWindow *w, MPKEvent *event )
{
    MPKEventXData *data = (MPKEventXData *)mpkEventGetData( event );

    if ( data->keyboard.state != MPK_PRESS )
        return;

    switch( data->keyboard.key )
    {
        case XK_S:
        case XK_s:
            shared->stereo = !shared->stereo;
            break;
    }
}

//-----------------------------------------------------------------------
//  incrementRotation
//-----------------------------------------------------------------------
static void xformColumn(    float *m, int i, int j, int k,
                            float cosX, float sinX,
                            float cosY, float sinY )
{
    float aux = sinX*m[j] + cosX*m[k];
    float x = sinY*aux + cosY*m[i];
    float y = cosX*m[j] - sinX*m[k];
    float z = cosY*aux - sinY*m[i];
    m[i] = x;
    m[j] = y;
    m[k] = z;
```

```
}

void incrementRotation( float *matrix, float xangle, float yangle )
{
    float cosX, sinX, cosY, sinY;

    cosX = cos( DEG2RAD(xangle) );
    sinX = sin( DEG2RAD(xangle) );
    cosY = cos( DEG2RAD(yangle) );
    sinY = sin( DEG2RAD(yangle) );
    xformColumn( matrix, 0, 1, 2, cosX, sinX, cosY, sinY );
    xformColumn( matrix, 4, 5, 6, cosX, sinX, cosY, sinY );
    xformColumn( matrix, 8, 9, 10, cosX, sinX, cosY, sinY );
}

//----------------------------------------------------------------------
//  drawcube
//----------------------------------------------------------------------
#define CUBE_SIZE   .25
void drawcube()
{
    glColor3f( 0., 0., 1. );
    glNormal3f( 0., 0., -1. );
    glBegin( GL_TRIANGLE_STRIP );
    glVertex3f(-CUBE_SIZE,-CUBE_SIZE,-CUBE_SIZE);
    glVertex3f(-CUBE_SIZE, CUBE_SIZE,-CUBE_SIZE);
    glVertex3f( CUBE_SIZE,-CUBE_SIZE,-CUBE_SIZE);
    glVertex3f( CUBE_SIZE, CUBE_SIZE,-CUBE_SIZE);
    glEnd();

    glColor3f( 0., 1., 0. );
    glNormal3f( 0., -1., 0. );
    glBegin( GL_TRIANGLE_STRIP );
    glVertex3f(-CUBE_SIZE,-CUBE_SIZE,-CUBE_SIZE);
    glVertex3f(-CUBE_SIZE,-CUBE_SIZE, CUBE_SIZE);
    glVertex3f( CUBE_SIZE,-CUBE_SIZE,-CUBE_SIZE);
    glVertex3f( CUBE_SIZE,-CUBE_SIZE, CUBE_SIZE);
    glEnd();

    glColor3f( 1., 1., 1. );
    glNormal3f( -1., 0., 0. );
    glBegin( GL_TRIANGLE_STRIP );
    glVertex3f(-CUBE_SIZE,-CUBE_SIZE,-CUBE_SIZE);
    glVertex3f(-CUBE_SIZE, CUBE_SIZE,-CUBE_SIZE);
    glVertex3f(-CUBE_SIZE,-CUBE_SIZE, CUBE_SIZE);
    glVertex3f(-CUBE_SIZE, CUBE_SIZE, CUBE_SIZE);
    glEnd();

    glColor3f( 1., 0., 0. );
    glNormal3f( 0., 1., 0. );
    glBegin( GL_TRIANGLE_STRIP );
    glVertex3f(-CUBE_SIZE, CUBE_SIZE,-CUBE_SIZE);
    glVertex3f(-CUBE_SIZE, CUBE_SIZE, CUBE_SIZE);
    glVertex3f( CUBE_SIZE, CUBE_SIZE,-CUBE_SIZE);
    glVertex3f( CUBE_SIZE, CUBE_SIZE, CUBE_SIZE);
    glEnd();

    glColor3f( 1., 0., 1. );
    glNormal3f( 1., 0., 0. );
    glBegin( GL_TRIANGLE_STRIP );
    glVertex3f( CUBE_SIZE,-CUBE_SIZE,-CUBE_SIZE);
    glVertex3f( CUBE_SIZE,-CUBE_SIZE, CUBE_SIZE);
    glVertex3f( CUBE_SIZE, CUBE_SIZE,-CUBE_SIZE);
    glVertex3f( CUBE_SIZE, CUBE_SIZE, CUBE_SIZE);
    glEnd();

    glColor3f( 1., 1., 0. );
    glNormal3f( 0., 0., 1. );
    glBegin( GL_TRIANGLE_STRIP );
    glVertex3f(-CUBE_SIZE,-CUBE_SIZE, CUBE_SIZE);
    glVertex3f( CUBE_SIZE,-CUBE_SIZE, CUBE_SIZE);
    glVertex3f(-CUBE_SIZE, CUBE_SIZE, CUBE_SIZE);
    glVertex3f( CUBE_SIZE, CUBE_SIZE, CUBE_SIZE);
    glEnd();
}
```

Using Compounds

This chapter describes how you can use compounds (or conversely, decomposition) to scale the performance of your graphics application. Decomposition allows you to use multiple pipes to render frames that would normally be rendered by a single pipe.

This chapter has the following sections:

- "Scalable Rendering"
- "Building Compounds"
- "Stereo-Selective Compounds"
- "Automatic Load Balancing for Compounds"
- "Choosing the Right Decomposition Mode"
- "Compound-Specific Callbacks"
- "Traversing Compounds"

Scalable Rendering

To achieve greater application performance, MPK allows you to decompose a global rendering task into smaller tasks and to assign the smaller tasks to individual pipes. The task division requires a decomposition scheme. In general, a decomposition scheme sends a scene to render to each pipe, gets back rendered images from each pipe for further composition, and then renders the final image. Figure 3-1 illustrates the role of *source* and *destination* channels in scalable rendering.

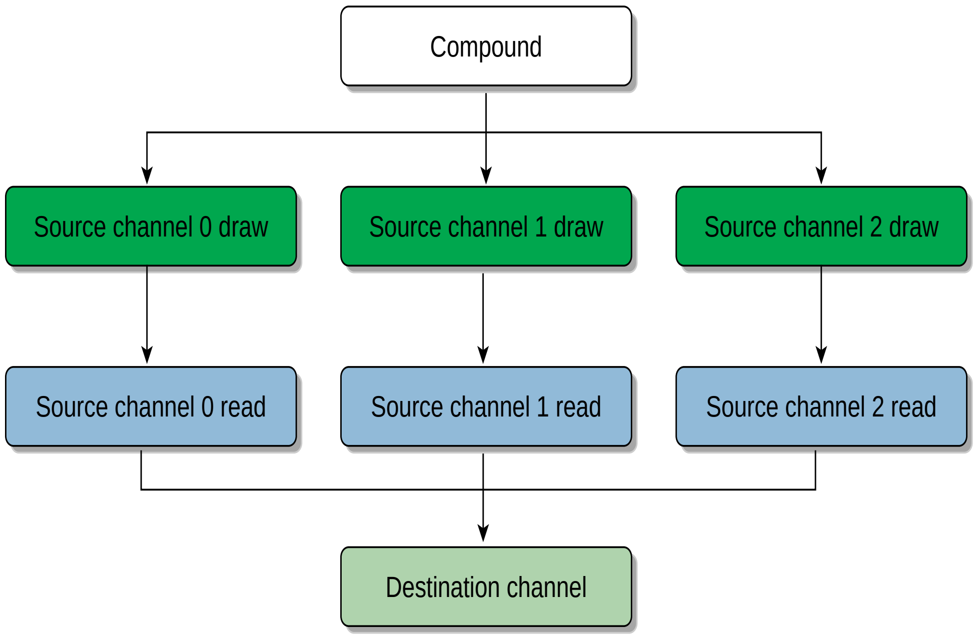

Figure 3-1 Source and Destination Channels

Building Compounds

To build a compound, you must create a MPKCompound data structure. The manual *SGI OpenGL Multipipe SDK User's Guide* describes the syntax of the MPKCompound data structures for your configuration file. This section describes how you build them logically.

Generally, to create a compound, you need to do the following:

1. Choose a decomposition scheme, which divides the global rendering task into smaller tasks.
2. Distribute the rendering of the smaller tasks to the source pipes for parallel processing.
3. Designate a destination channel for the reassembly of the final, coherent image.

The destination channel is usually one of the source channels. To achieve optimal performances, you would usually have one channel per pipe.

This chapter focuses on the three tasks just cited. Optionally, you can also do the following:

- Indicate whether your compound is used in only stereo or mono mode.
- Indicate controls for the pixel data transfers between the compound and its regions.
- Indicate whether to use scalable graphics hardware.
- Indicate whether to use automatic load balancing.

The section “Stereo-Selective Compounds” on page 79 describes how you control whether your compound is used depending on the stereo mode of the application. For more information on the first two optional tasks, see the *SGI OpenGL Multipipe SDK User's Guide* for the description of the `mode` and `format` fields of the MPKCompound data structure. “Hardware Compositing” in Chapter 4 describes the integration of scalable graphics with MPK. The section “Automatic Load Balancing for Compounds” on page 80 describes how MPK balances the rendering for certain compound modes.

MPK provides several decomposition schemes and the following subsections describe these schemes:

- “Frame Decomposition”
- “Temporal Decomposition”
- “Pixel-Based Decomposition”
- “Multilevel Decomposition”

Each decomposition mode improves performance or graphics quality, but the performance gain depends on the application type and the nature of the performance bottleneck. Four factors are important in choosing the decomposition scheme judiciously:

Factor	**Description**
Load balancing	For a given decomposition, each pipe should execute roughly the same amount of work since the slowest pipe dictates the overall performance. Unbalanced decomposition can seriously affect the scalability.

Scalability of scheme	Scalability is the degree to which the performance grows as the number of graphics resources increases. To optimize performance, you only add resources to address the source of the bottleneck. For example, adding more geometry power to an application limited by pixel fill will not improve performance.
Latency added	Depending on the decomposition scheme, the frame delay between a user input and the associated frame output may be greater than one frame. Minimizing this latency may be critical for some event-driven applications.
Graphics I/O consumption	A typical decomposition involves the reading and writing of images from the source channels (contributing channels) to a destination channel. This transfer might stress the graphics I/O and memory capabilities of the system.

Frame Decomposition

In frame decomposition, a frame or view is divided into regions, which are, in turn, assigned to individual source pipes for rendering. Based on the following perspectives, there are several approaches to dividing the frame into regions:

- Screen topology (screen decomposition)
- Scene graph primitives (database decomposition)
- Eye view (eye decomposition)

Each approach yields a different flavor of frame decomposition.

Screen Decomposition

In screen decomposition (also referred to as 2D decomposition), each pipe renders a part of the screen area. Assembling side-to-side each image part constitutes the final rendering. This type of decomposition is used when the intrinsic pixel fill or geometry capacity of each pipe slows down the application. The scalability depends on the balancing of the workloads. The model to display needs to be uniformly distributed across the screen to accommodate a good balancing and, thus, scalability. The graphics I/O is relatively low, because the traveling source images are small.

Figure 3-2 illustrates screen decomposition.

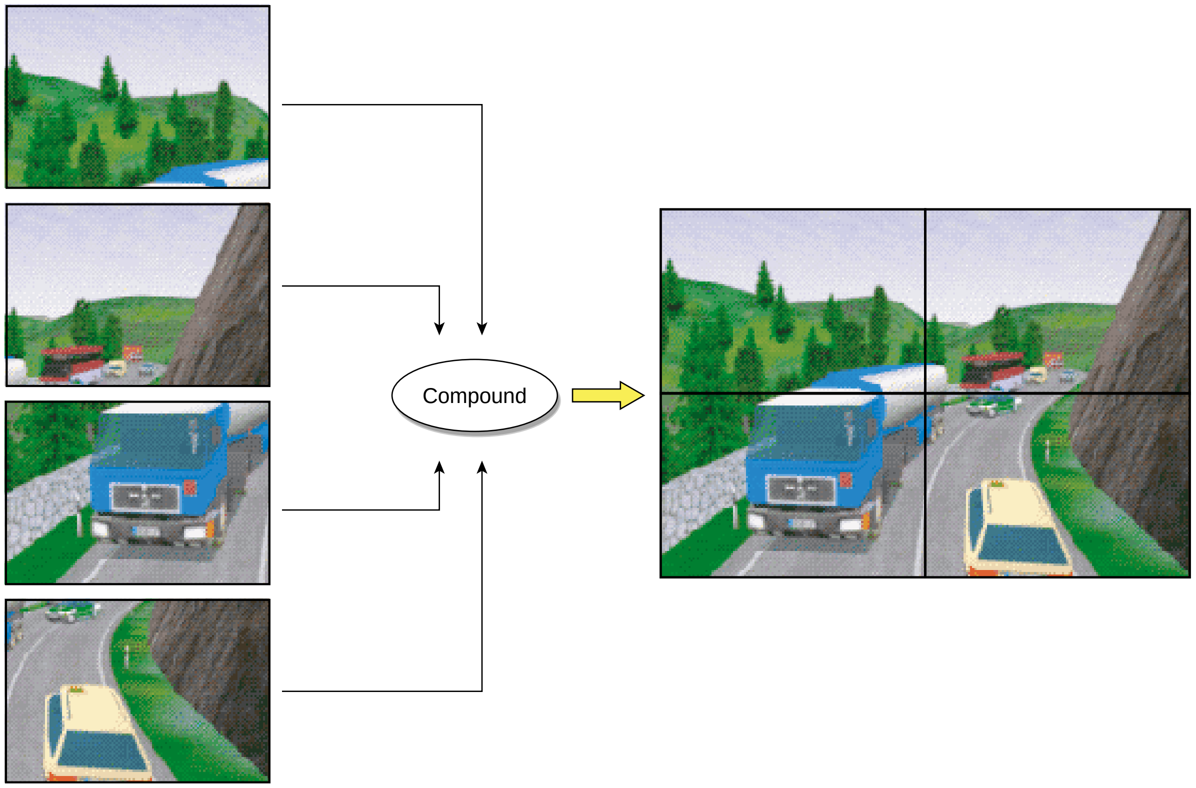

Figure 3-2 Screen Decomposition

Example 3-1 shows the configuration file specifications for the screen decomposition illustrated in Figure 3-2.

Example 3-1 2D Compound in a Configuration File

```
compound {
    mode [2D]
    channel "destination"

# The top left of "destination" image will be
# rendered on "source0"...
    region {
        viewport [ 0., .5, .5, .5 ]
        channel "source0"
    }
# The top right of "destination" image will be
# rendered on "source1"...
    region {
        viewport [ .5, .5, .5, .5]
        channel "source1"
    }
# The bottom left of "destination" image will be
# rendered on "source2"...
    region {
        viewport [ 0., 0., .5, .5 ]
        channel "source2"
    }
# ... while "destination" itself takes care of
# the bottom right
    region {
        viewport [ .5, 0., .5, .5 ]
        channel "destination"
    }
}
```

Figure 3-3 shows how the configuration in Example 3-1 executes.

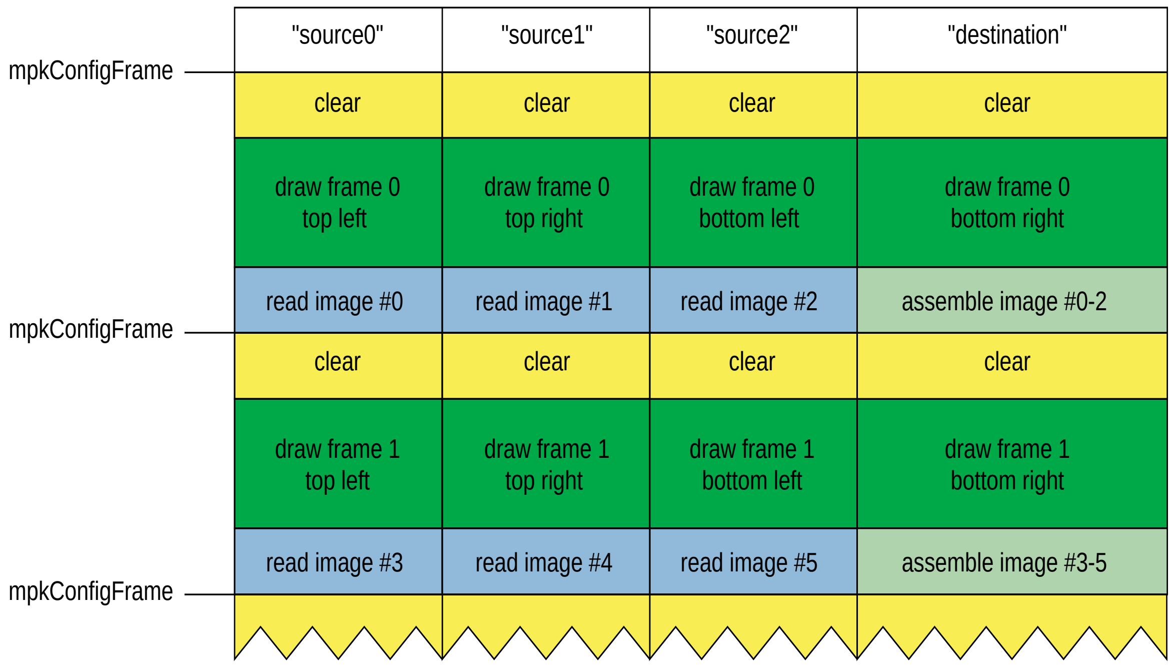

Figure 3-3 The Execution of a `2D` Compound

By adding the mode flag `ASYNC`, the frame rate can be improved because of the lower consumption of graphics I/O, as shown in Figure 3-4. Note that the `ASYNC` configuration has a latency of one.

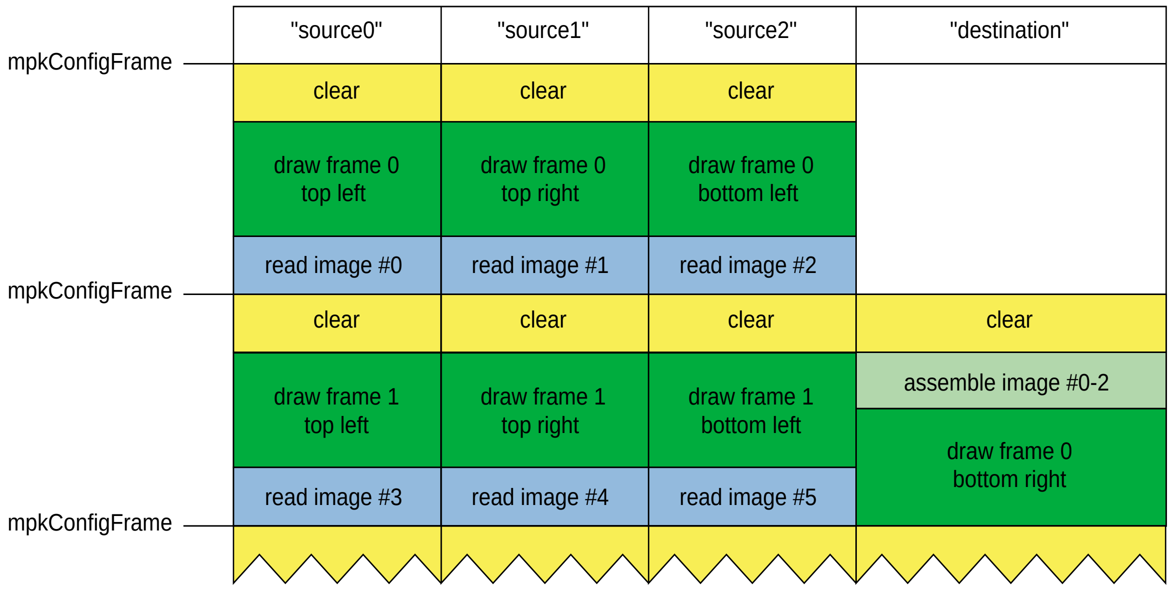

Figure 3-4 The Execution of a `2D ASYNC` Compound

Database Decomposition

In database (DB) decomposition, the scene is rendered in parallel by dividing it among the different graphics pipes. Each pipe renders its share of the scene to generate partial images. These images are then composited by MPK to generate the final image in the destination channel. During composition, the application can use depth testing and/or alpha blending to achieve the desired effect. Database decomposition allows you to scale both the geometry and the pixel fill performance of the system. For some applications, such as volume rendering, it also scales the texture memory capacity of the system by the number of pipes.

Figure 3-5 demonstrates the use of database decomposition in volume rendering. The volume data is divided equally among the four pipes and the partial images are composited on the destination channel. In this case, the destination channel (top left portion of the figure) is also contributing to the rendering as a source channel.

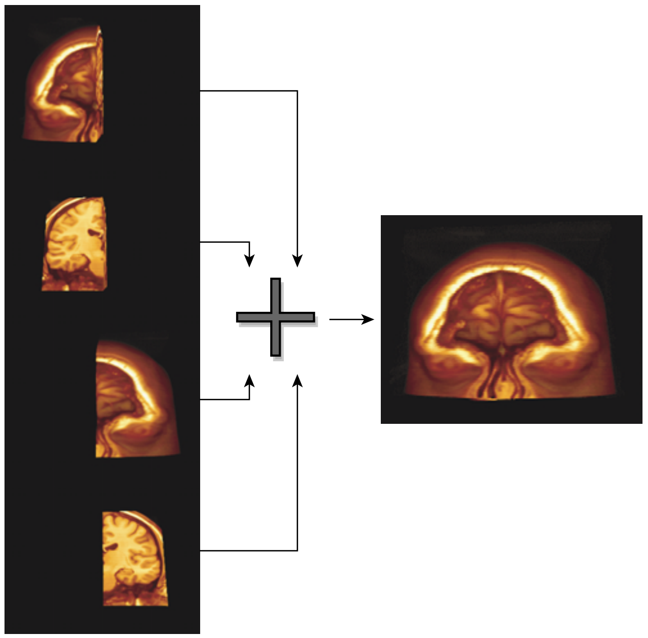

Figure 3-5 Database Decomposition

Example 3-2 shows the configuration file specifications for the database decomposition illustrated in Figure 3-5.

Example 3-2 DB Compound in a Configuration File

```
compound {
    mode    [ DB ]
    format  [ RGBA DEPTH ]
    channel “channel”

    region {
        range       [ 0., .25 ]
        channel     “buffer0”
    }

    region {
        range       [ .25, .5 ]
        channel     “buffer1”
    }

    region {
        range       [ .5, .75 ]
        channel     “buffer3”
    }

    region {
        range       [ .75, 1. ]
        channel     “channel”
    }
}
```

Figure 3-6 shows how the confiuration in Example 3-2 executes.

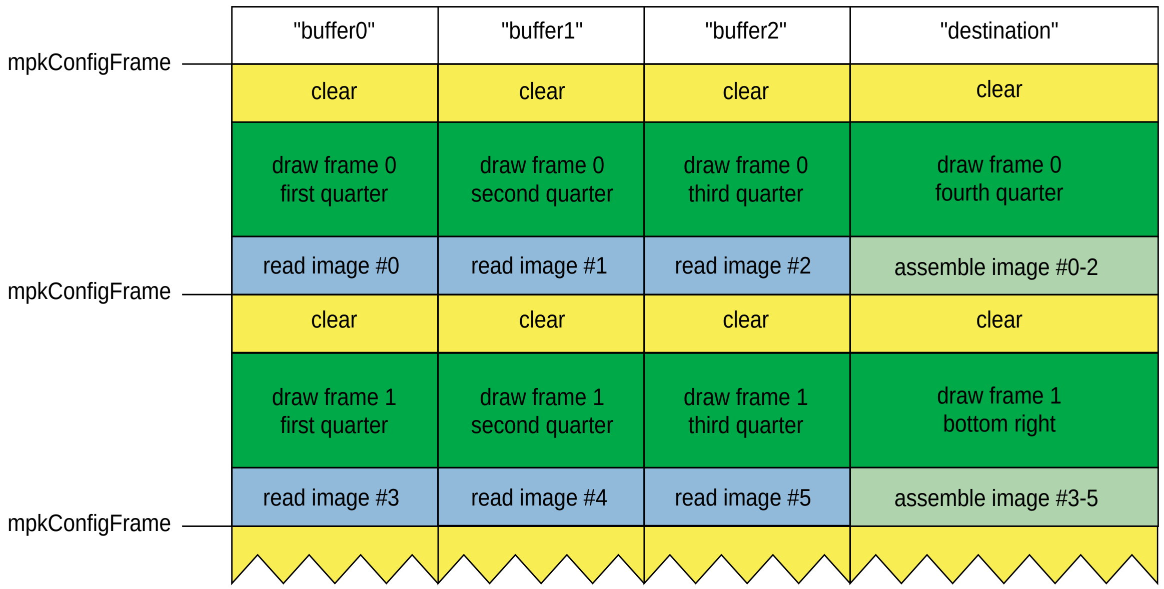

Figure 3-6 The Execution of a `DB` Compound

In order to support this decomposition scheme, the application has to use the `range` field to draw the correct part of the database in the MPKChannel's update draw callback. The range can be acquired using **mpkChannelGetRange()**.

The default DB assembly uses the following pseudocode to assemble the frames using Z-based recomposition:

```
if ASYNC flag is set
    draw depth buffer of first input frame
    draw color buffer of first input frame
end if

enable stencil test

for all remaining input frames
    enable depth test
    draw depth buffer and set stencil bit where depth-test passes
    disable depth test
    draw color buffer where stencil bit set
end for
```

Some applications, such as volume rendering, may want to use a different recomposition technique than MPK's default procedure. The later section "Compound-Specific Callbacks"explains how to use custom assemble-compound callbacks.

By adding the mode flag `ASYNC`, the frame rate can be improved because of the lower consumption of graphics I/O, as shown in Figure 3-7. Note that the `ASYNC` configuration has a latency of one.

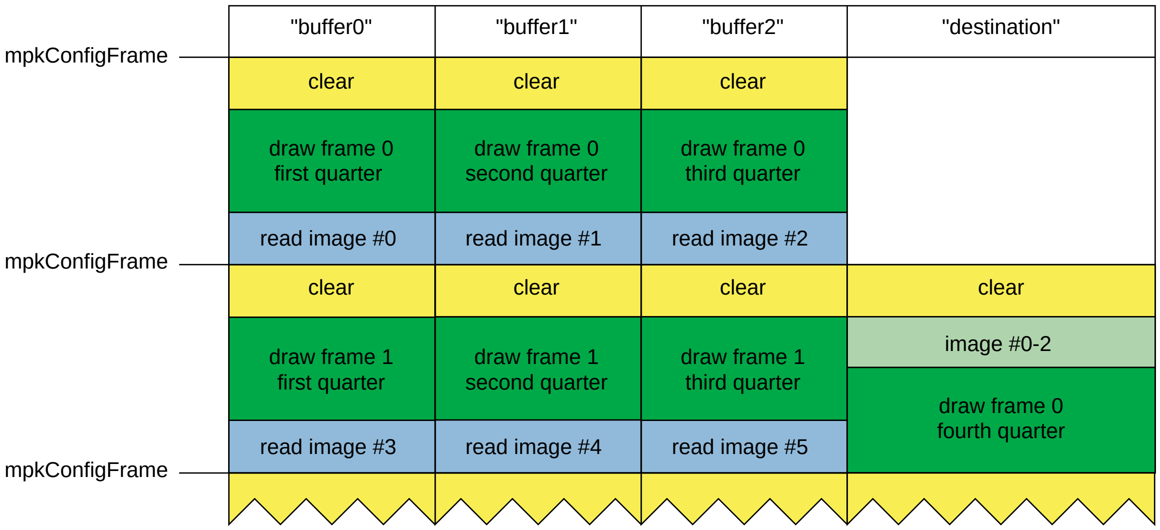
"buffer0"
"buffer1"
"buffer2"
"destination"
mpkConfigFrame
clear
draw frame 0 first quarter
draw frame 0 second quarter
draw frame 0 third quarter
read image #0
read image #1
read image #2
mpkConfigFrame
clear
draw frame 1 first quarter
draw frame 1 second quarter
draw frame 1 third quarter
image #0-2
draw frame 0 fourth quarter
read image #3
read image #4
read image #5
mpkConfigFrame

Figure 3-7 The Execution of a `DB ASYNC` Compound

Eye Decomposition

Eye decomposition is well-suited for stereo or multiple-view rendering. Each pipe renders a particular view (left, right, mono). The final rendering depends on the type of display. As illustrated in Figure 3-8, if stereo is active, then each pipe view fills in the right or left buffer of the final rendering. This provides good load balancing and scalability, especially for stereo-view rendering, because the scene content remains similar during run time.

An `EYE` compound has no frame latency, unless the `mode` qualifier `ASYNC` has been specified and pixel transfer needs to occur, in which case the latency is 1.

The number of regions of an eye compound is not limited. If more than one region correspond to the same eye view, MPK uses the first specified region (for this eye) as source for the pixel transfer, if needed.

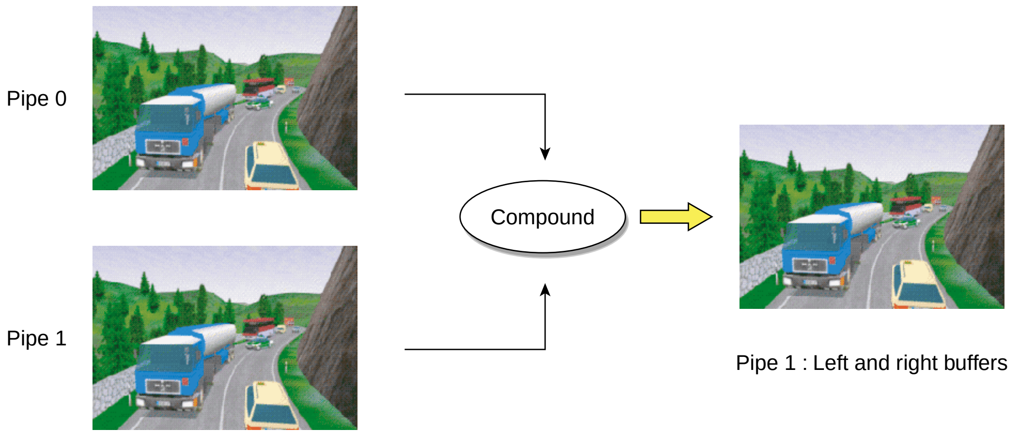

Figure 3-8 Eye Decomposition

Example 3-3 shows the configuration file specifications for the eye decomposition illustrated in Figure 3-8.

Example 3-3 Eye Compound in a Configuration File

```
compound {
        mode    [ EYE STEREO ]
        channel “channel”

        region {
            eye         LEFT
            channel     “buffer”
        }

        region {
            eye         RIGHT
            channel     “channel”
        }
}
```

Head-Mounted-Device (HMD) decomposition is very similar to that of eye decomposition, except that the head position actually specifies a new origin for the physical layout of the channels.

Example 3-4 shows a configuration file specification for an HMD decomposition:

Example 3-4 HMD Compound in a Configuration File

```
compound {
    mode [HMD]
    channel “destination”

    region {
        eye     left
        channel “source::left”
    }

    region {
        eye     right
        channel “source::right”
    }
}
```

If a destination channel is specified, then the frustum is inherited from the destination channel’s `wall` or `projection` frustum specification; otherwise, the source channel’s frustum specification will be used.

Figure 3-9 shows how the configurations for Example 3-3 and Example 3-4 execute.

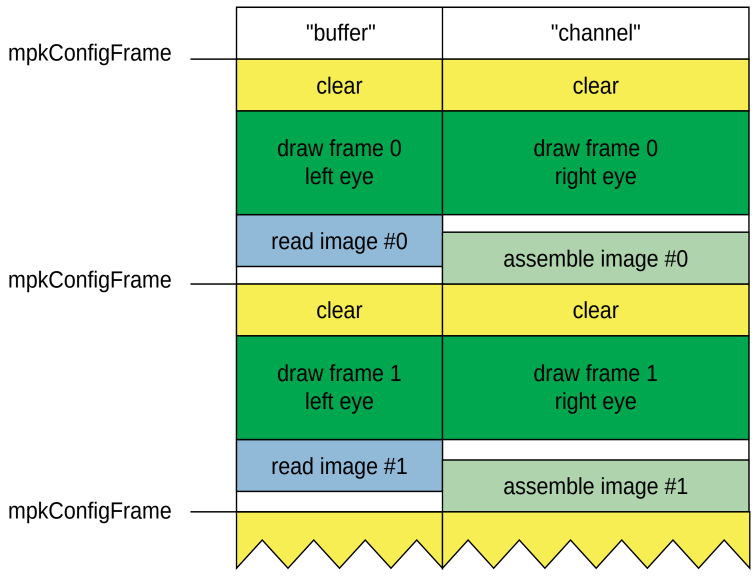

Figure 3-9 The Execution of an `EYE` or `HMD` Compound

By adding the mode flag `ASYNC`, the frame rate can be improved because of the lower consumption of graphics I/O, as shown in Figure 3-10. Note that the `ASYNC` configuration has a latency of one.

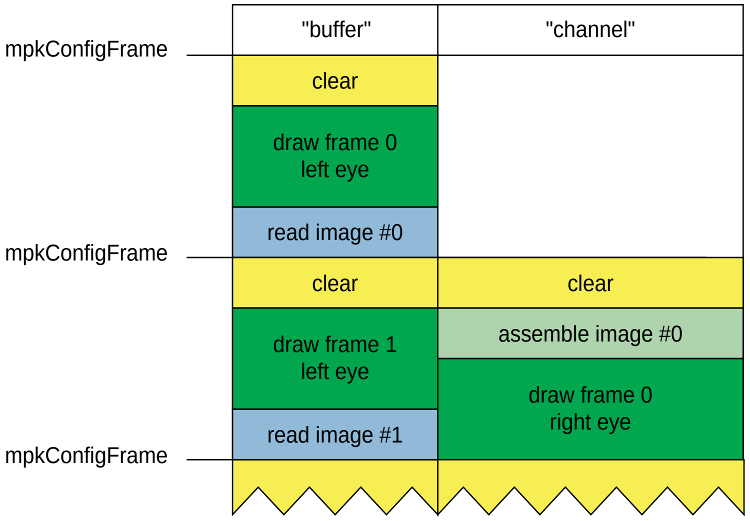

Figure 3-10 The Execution of an `EYE` or `HMD ASYNC` Compound

Temporal Decomposition

In contrast to frame decomposition, where the focus of load balancing is on dividing the frame into regions, temporal decomposition balances the workload by scheduling the work on each pipe in sync with that of the other pipes to produce a steady stream of rendered frames. The time scheduling rather than the frame division is the focus. There are two types of temporal decomposition: frame multiplexing and data streaming. The work done by each pipe largely distinguishes the two.

Frame Multiplexing

Frame multiplexing (also referred to as DPLEX decomposition) distributes entire frames to the source pipes over time for parallel processing. The first pipe begins rendering frame 1; a specified fraction of a frame later the second pipe begins rendering frame 2; another fraction of a frame later the third pipe begins rendering frame 3; and so on for all of the pipes.

Figure 3-11 illustrates frame multiplexing on a four-pipe system.

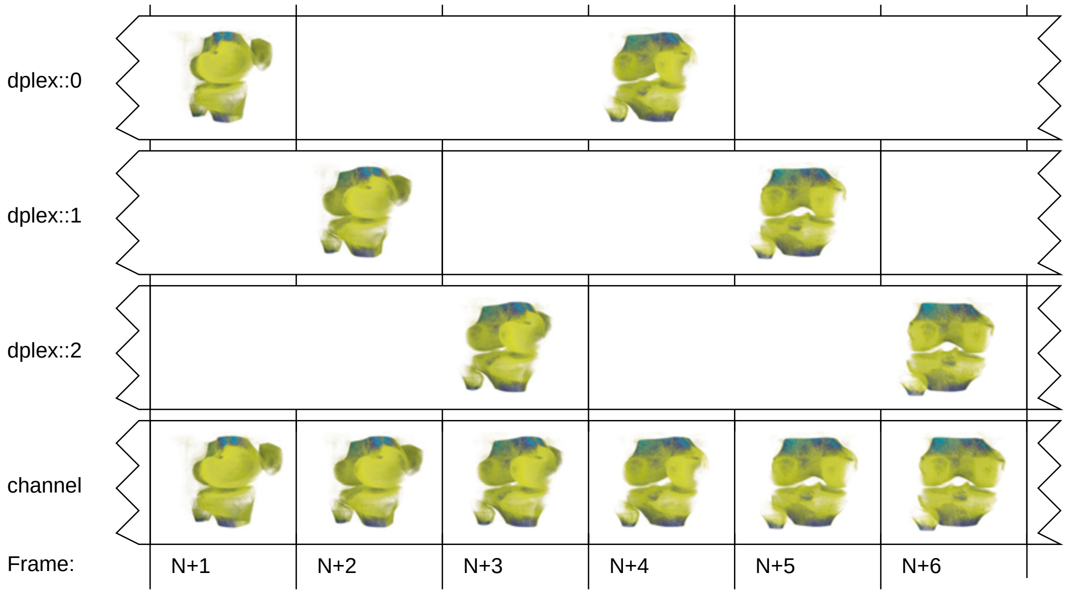

Figure 3-11 Frame Multiplexing Decomposition

Frame multiplexing globally scales geometry and pixel fill performance, as the workload balance between pipes is intrinsically maintained. This scheme has an increased transport delay inherent to frame synchronization required across the pipes. It produces a latency of (*pipes* – 1) frames—that is, there will be a (*pipes* – 1) frames delay between a user input and the corresponding output frame.

Frame multiplexing can also be accelerated in hardware using the SGI Video Digital Multiplexer (DPLEX), which connects pipes together with a bus, thereby avoiding the image readbacks from the contributing pipes. The pipes are daisy-chained to achieve reduced latency. For more details, see "Hardware Compositing" in Chapter 4.

Example 3-5 shows the configuration file specifications for the screen decomposition illustrated in Figure 3-11.

Example 3-5 DPLEX Compound in a Configuration File

```
compound {
    mode    [ DPLEX ]
    channel “channel”

    region {
        channel “dplex::0”
    }

    region {
        channel “dplex::1”
    }

    region {
        channel “dplex::2”
    }
}
```

Figure 3-12 shows how the configuration in Example 3-5 executes.

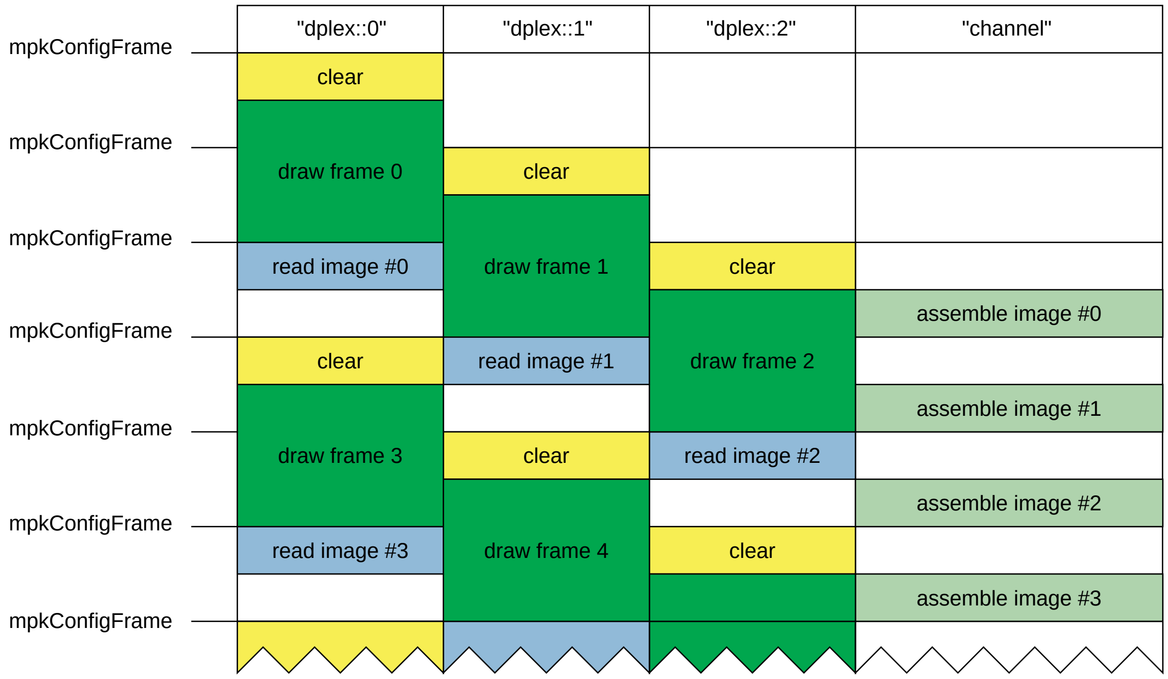

Figure 3-12 The Execution of a DPLEX Compound

By adding the mode flag `ASYNC`, the frame rate can be improved because of the lower consumption of graphics I/O, as shown in Figure 3-13. Note that the `DPLEX ASYNC` configuration has a latency of *pipes* frames while the DPLEX configuration has a lency of (*pipes* – *1*) frames.

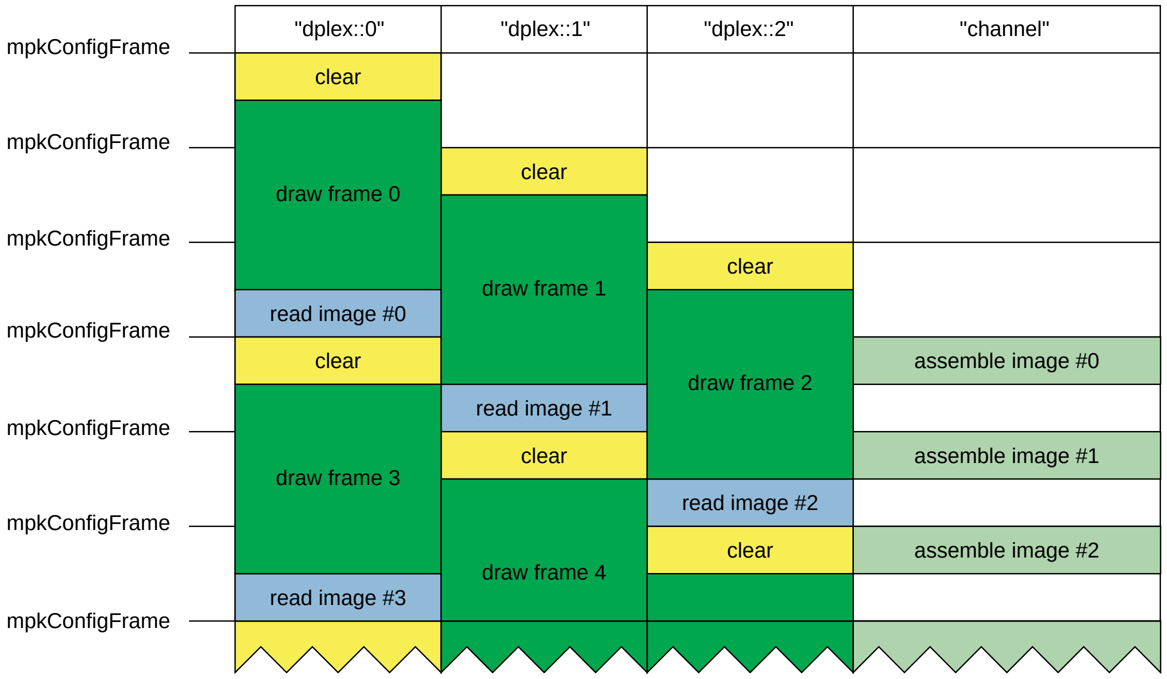

Figure 3-13 The Execution of a DPLEX `ASYNC` Compound

You can achieve full scalability—that is, scale by the number of pipes rather than by (pipes – *1*)—using a DPLEX compound. To do so, you must specify the destination channel as a source channel also. To support this feature, the application has to be modified as explained in the following paragraph.

DPLEX operation implies that while one frame is being drawn, another one is copied and displayed on the destination window. In full-scale DPLEX operation, one of the source channels draws a frame while it has to simultaneously display another frame from a different source channel. Drawing into a pbuffer enables this channel to draw its frame into the pbuffer and to display the frame of another source channel in the X window at the same time. MPK needs to know when the drawing into the pbuffer is interruptible in order to composite into the normal source window. Therefore, the application should call **mpkChannelSyncDPlex()** whenever possible to switch the OpenGL context from the pbuffer to the X window—that is, outside of a **glBegin()** ... **glEnd()** sequence. Figure 3-14 shows this decomposition for a two-pipe full-scale DPLEX compound.

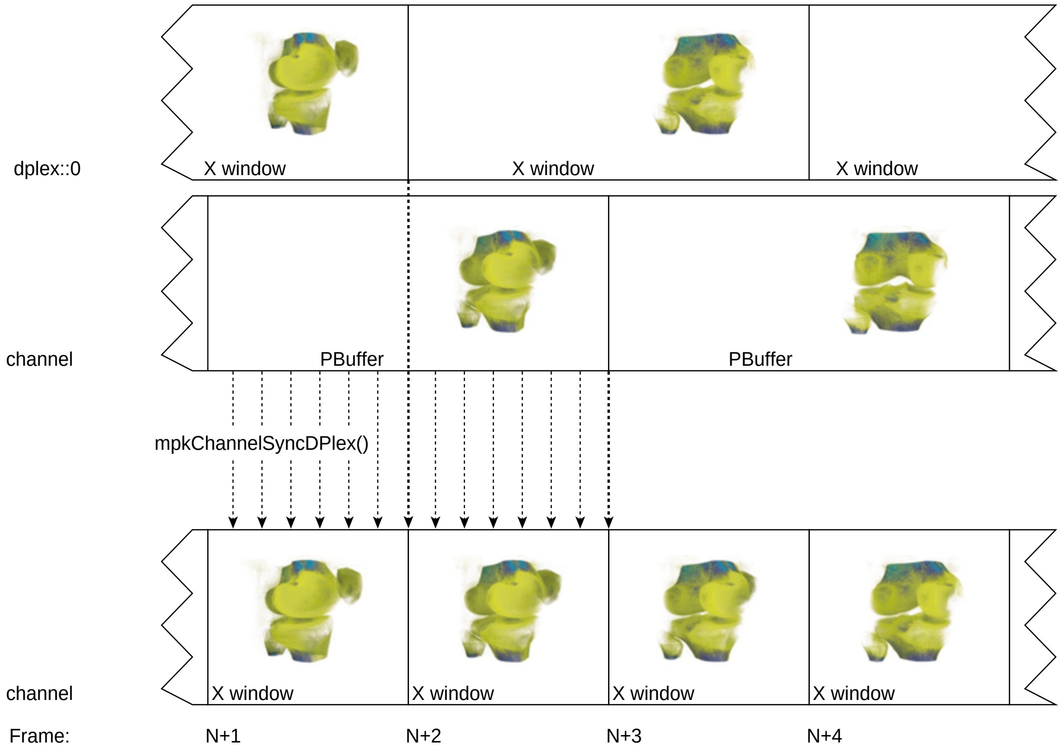

Figure 3-14 Two-Pipe Full-Scale DPLEX Compound

Example 3-6 shows a configuration file structured for full scalability using the DPLEX compound.

Example 3-6 DPLEX Compound Structured for Full Scalability

```
compound {
    mode [ DPLEX ]
    channel “channel”

    region {
        channel “channel”
    }

    region {
        channel “buffer”
    }
}
```

Note: Full scalability using the DPLEX compound is supported only on InfiniteReality graphics systems.

Data Streaming

Data streaming (also referred to as 3D decomposition) is similar to database decomposition in that it allows the application to divide the scene among multiple pipes and then composite the partial results to give the final rendering. But, in this case, the composition is done using a series of successive compounds for each frame, as shown in Figure 3-15. For frame N+1, channel `stream::1` draws the first quarter of the database, which is copied to channel `stream::2` at the beginning of the next frame. During frame N+2, channel `stream::2` draws the second quarter of the database on top while channel `stream::1` starts a new frame. At frame N+4, the destination channel `channel` finishes drawing the last quarter and displays the frame started three time steps ago.

Like DPLEX decomposition, this scheme also has a latency of (*pipes* – 1) frames—that is, there will be a (*pipes* – 1) frames delay between a user input and the corresponding output frame. As shown in Figure 3-15, this latency is due to successive compounds at each frame. You must wait for (*pipes* – 1) frame computations before the final rendering is displayed. Each compound needs to read only one source image. Consequently, this keeps graphics I/O consumption low while performance scaling is achieved by pipelining the rendering in parallel across the pipes.

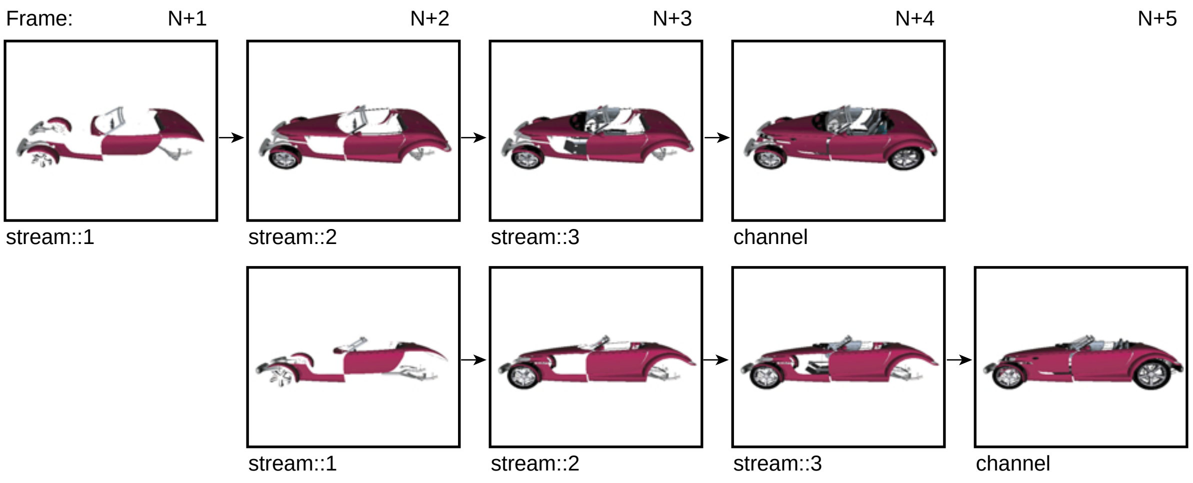

Figure 3-15 Data Streaming Decomposition

As shown in Example 3-7, the configuration file specification for a data streaming decomposition is similar to that for database decomposition.

Example 3-7 Data Streaming Compound (3D) in a Configuration File

```
compound {
    mode    [ 3D ]
    format  [ RGBA DEPTH ]
    channel “channel”

    region {
        range      [ .0 .25 ]
        channel    “stream::1”
    }

    region {
        range      [ .25 .5 ]
        channel    “stream::2”
    }

    region {
        range      [ .5 .75 ]
        channel    “stream::3”
    }
    region {
        range      [ .75 1. ]
        channel    “channel”
    }
}
```

In order to support this decomposition scheme, the application has to use the `range` field to draw the correct part of the database in the MPKChannel’s update draw callback. The range can be acquired using **mpkChannelGetRange()**. Some applications, such as volume rendering, may want to use a different recomposition technique than MPK’s default procedure. The later section “Compound-Specific Callbacks”explains how to use custom assemble-compound callbacks.

Figure 3-16 shows how the configuration for Example 3-7 executes. Data streaming compounds always assemble before drawing; therefore, the `ASYNC` flag is ignored.

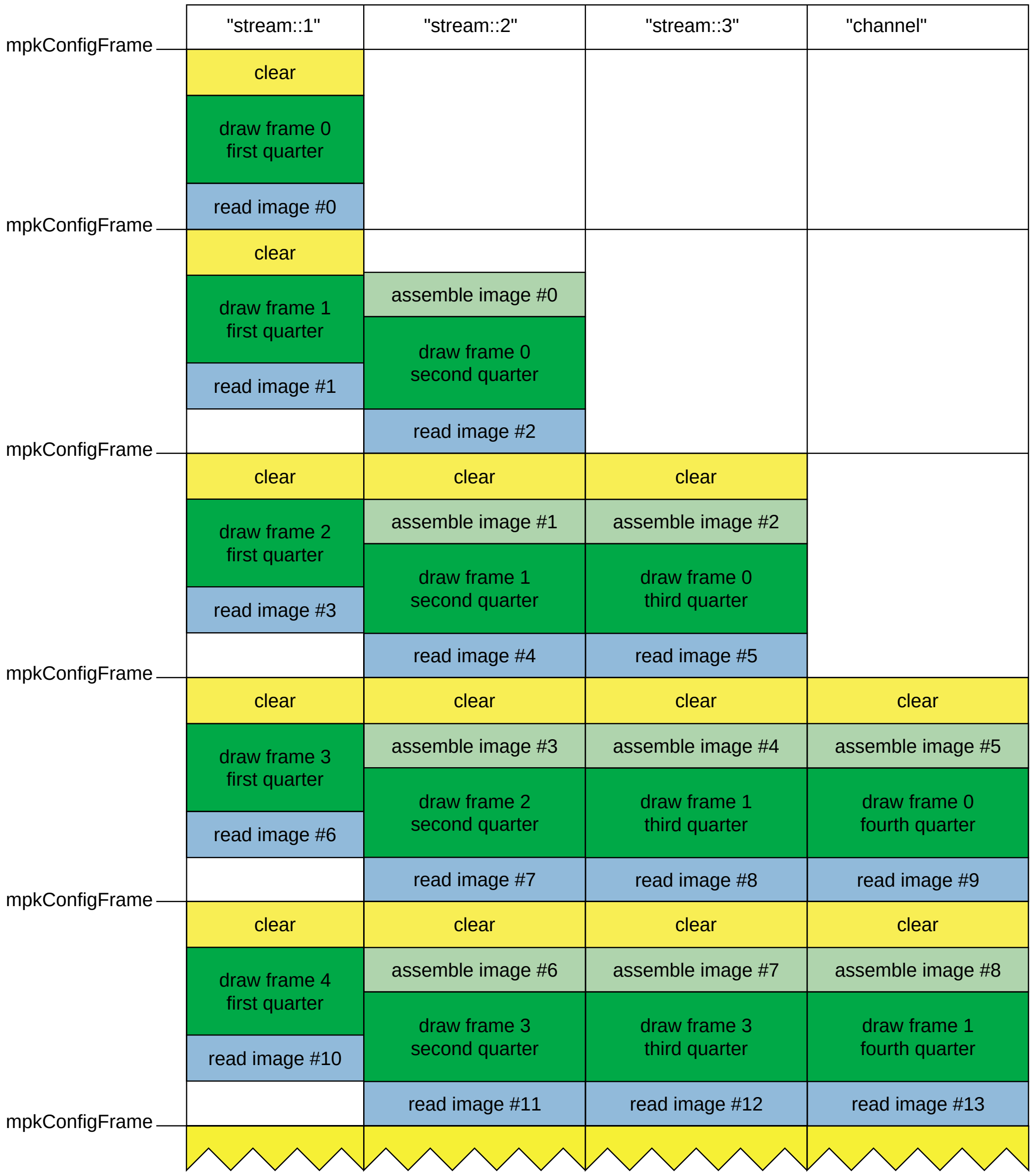

Figure 3-16 The Execution of a 3D Compound

Pixel-Based Decomposition

In pixel-based decomposition, a frame is rendered using a multipass approach where single passes are assigned to individual source pipes for rendering. Assembling each frame using accumulation techniques constitutes the final rendering. Accumulation of the frames can be achieved using one of the following techniques:

- The SGI Scalable Graphics Compositor
- OpenGL accumulation
- OpenGL blending

In order to use OpenGL accumulation, you must use an appropriate visual; otherwise, MPK uses blending.

Full-Scene Antialiasing (FSAA) Decomposition

MPK has implemented one scheme of pixel-based decomposition, a full-scene antialiasing (FSAA) compound. Each pipe renders the full scene from a slightly different viewpoint. The number of rendering passes of a FSAA compound is defined by its number of sources. Furthermore, every channel can thereby be used multiple times. This type of decomposition is used when the the resulting output quality has highest priority. The scalability and final rendering quality depends on the number of available pipes.

FSAA Compound Examples

Example 3-8 shows an FSAA compound using the SGI Scalable Graphics Compositor:

Example 3-8 Four-Pipe 4x FSAA Compound Using the SGI Scalable Graphics Compositor

```
compound {
    mode    [ FSAA HW NOCOPY ]
    channel “channel-0”

    # The number of sources defines the FSAA mode
    region {
        channel “channel-0”
    }
    region {
        channel “channel-1”
    }
    region {
        channel “channel-2”
    }
    region {
        channel “channel-3”
    }
}
```

Figure 3-17 illustrates the advantage of using a 4x FSAA solution.

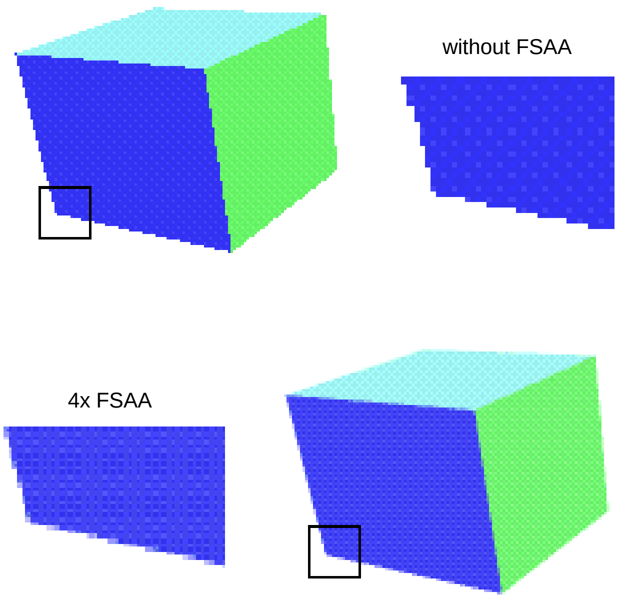

Figure 3-17 4x FSAA Decomposition

Example 3-9 shows how to use the same channel multiple times as a source channel to support multipass rendering in MPK on machines with only a few pipes.

Example 3-9 Multiple Use of a Single Channel in FSAA Decompostion

```
compound {
    mode    [ FSAA ]
    channel “channel”

    # The number of sources defines the FSAA mode
    region {
        channel “channel”
    }
    region {
        channel “channel”
    }
    region {
        channel “channel”
    }
    region {
        channel “channel”
    }
}
```

Multilevel Decomposition

MPK allows you to combine the various decomposition schemes to fix performance bottlenecks that differ in nature. For example, a combined solution can use a database and temporal decomposition scheme for optimizing performance (but it will have a limiting transport delay) or can use an eye and database decomposition scheme for stereo volume rendering.

Figure 3-18 shows a four-pipe solution using an eye and database decomposition scheme.

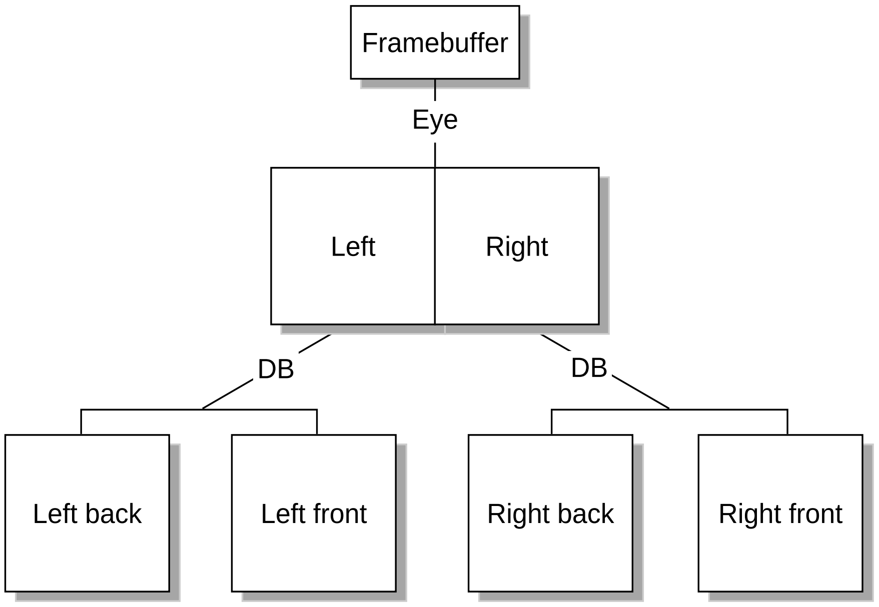

Figure 3-18 Eye-DB Multilevel Decomposition

Example 3-10 shows the configuration file specifications for the multilevel decomposition illustrated in Figure 3-18.

Example 3-10 Multilevel Compound in a Configuration File

```
compound {
    mode    [ EYE ]
    channel “right-front”

    region {
        eye    LEFT
        compound {
            mode    [ DB ]
            channel “left-front”

            region {
                range   [ 0., .5 ]
                channel “left-back”
            }

            region {
                range   [ .5, 1. ]
                channel “left-front”
            }
        }
    }

    region {
        eye    RIGHT
        compound {
            mode    [ DB ]
            channel “right-front”

            region {
                range   [ 0., .5 ]
                channel “right-back”
            }

            region {
                range   [ .5, 1. ]
                channel “right-front”
            }
        }
    }
}
```

Stereo-Selective Compounds

In many instances, it will be desirable to control which compounds will be used by the application based on whether the application is running in stereo mode. MPK provides a mode parameter for this purpose. For instance, if the application is to run in stereo mode, you may want to use eye decomposition and when in mono mode, to use another type of decomposition. Example 3-11 illustrates this conditional use of compounds.

Example 3-11 Stereo-Selective Compounds

```
compound {
    mode    [ EYE STEREO ]
    channel “channel”

    region {
        eye         LEFT
        channel     “buffer”
    }
    region {
        eye         RIGHT
        channel     “channel”
    }
}

compound {
    mode    [ 2D MONO ]
    channel “channel”

    region {
        viewport    [ 0., 0., 1., .5 ]
        channel     “buffer”
    }

    region {
        viewport    [ 0., .5, 1., .5 ]
        channel     “channel”
    }
}
```

The `MONO` and `STEREO` flags allow you to specify different channel decompositions depending on the current configuration mode. This is especially useful for eye decomposition. In this example, when the destination channel is in stereo mode, MPK uses the eye decomposition. When the destination channel is in mono mode, MPK uses the 2D decomposition.

Automatic Load Balancing for Compounds

Achieving an ideal decomposition among the children of a compound can be difficult, since the workload per child often changes on a per-frame basis. To address this problem, MPK provides automatic load balancing for 2D, DB, and 3D compounds.

Figure 3-19 contrasts dynamic and static load balancing for a 2D compound using `volview`. Volume rendering is bound by fill rate; therefore, the load balancing can adjust the compound's region so that each pipe has approximately the same amount of volume to rasterize. When using static tiling, one pipe may have to render the whole volume as it is moved around. Since the slowest child dictates overall performance, the frame rate is better, in this case, when using load balancing.

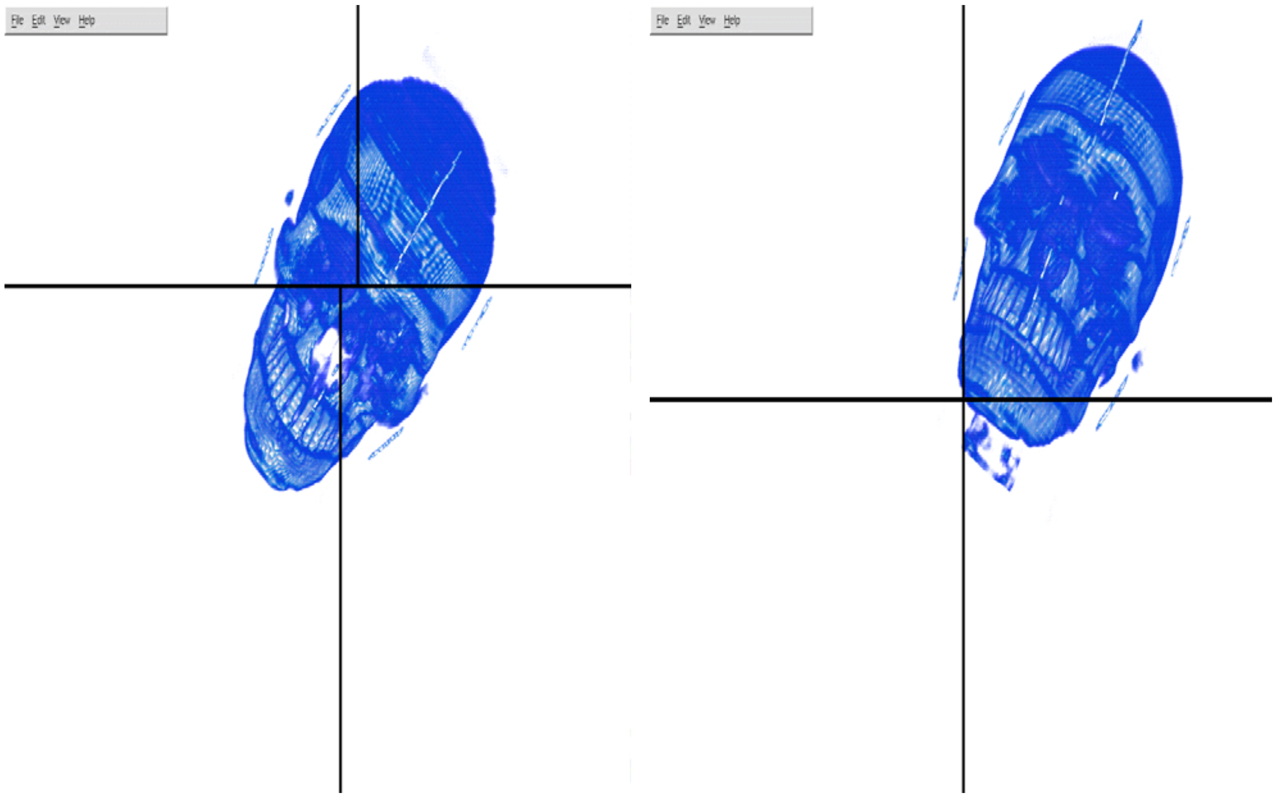

Figure 3-19 Dynamic Versus Static Load Balancing

Using the rendering times for each child, MPK computes a new viewport or range each frame. This approach needs the following conditions to work properly:

Condition	Description
Low latency	A new workload can only be computed after all children have drawn. Therefore, the higher the latency, the higher the difference will be between the frame which is used to compute the new balance and the frame for which the balance is computed. Logically, high latency is counterproductive in achieving proper load balancing.

Frame consistency	Since the new viewport or range is computed based on the last finished frame but applied to the next frame, the two frames should be similar. This is true for most applications.
Scalable compound mode	The chosen decomposition mode has to solve the application's bottleneck. For example, load balancing a 2D compound for a geometry-limited application will fail, unless this application uses view-frustum culling.
Imbalance in decomposition	If the decomposition is already well-balanced—for example, for a DB compound—the static compound may provide a better frame rate.

One of the following mode flags can be used to enable load balancing:

Mode Flags	**Description**
`ADAPTIVE`	Can be used for 2D, DB, and 3D compounds. 2D compounds will use tiles while DB and 3D compounds will adapt the range to decompose the rendering.
`ADAPTIVE_V`	Can only be used for 2D compounds, which will use vertical stripes to decompose the rendering.
`ADAPTIVE_H`	Can only be used for 2D compounds, which will use horizontal stripes to decompose the rendering.

Choosing the Right Decomposition Mode

There are no hard and fast rules for choosing the correct decomposition scheme, but the following are some general guidelines to aid you in selecting a reasonable scheme for your environment:

Mode	**Recommended Use**
`2D`	Use this scheme if your application is fill-limited.You can also scale geometry performance and texture memory if your application is using view-frustum culling techniques.
`3D`	Use this scheme where you would normally use the `DB` scheme but where you experience scalability problems caused by a graphics I/O bottleneck on the destination pipe. For `3D` decomposition, the graphics I/O per pipe is constant when changing the number of contributing pipes. Unlike the `DB` scheme, however, adding pipes to a `3D` compound increases latency.
`DB`	Use this scheme when your application's frame rendering can be sequenced into equally consuming phases. This requires the application to divide your scene into multiple components and then to composite them correctly. Scalability here can be either on fill, geometry, or graphics resources (texture) depending on the application.
`FSAA`	Use this scheme if graphics quality is a primary concern.
`EYE`	Use this scheme for stereo viewing.
`DPLEX`	Use this scheme for general load balancing where the application maintains a reasonably steady frame rate.

Note: With the `DB`, `3D`, and full-scale `DPLEX` modes, the application must support the feature.

These are very high-level guidelines that may very well overlap. As noted in the section "Multilevel Decomposition" on page 76, you can combine the various decomposition modes to fix different performance bottlenecks.

Compound-Specific Callbacks

This section describes how to customize the compound processing using the callbacks provided by the MPKCompound data structure.

This section has the following subsections:

- "Optimizing Frame Transport"
- "Custom Assembly"
- "Customizing the Clear Callback"

Optimizing Frame Transport

The adaptive readback interface enables the specification of a subviewport of a compound's input channel to be read back, transported, and assembled during compound operations.

This subviewport usually corresponds to the portion of the channel that was drawn during the last update-channel draw callback and, therefore, enables the optimization of pixel transfers for this channel.

The default **mpkCompoundReadOutputFrame()** callback basically reads the full channel's viewport by using **mpkChannelReadFrame()** on the handle returned by **mpkCompoundGetOutputFrame()**. A customized version of this callback, as shown in Example 3-12, typically uses only the channel subportion transported by user data.

Example 3-12 A Custom Read-Output Compound Callback

```
void readOutputFrame( MPKCompound *compound )
  {
      MPKChannel  *c;
      ChannelData *channelData;
      MPKFrame    *frame;

      c = mpkCompoundGetChannel( compound );

      frame = mpkCompoundGetOutputFrame( compound );

      // get the effectively drawn region computed in updateChannel
      channelData = ( ChannelData* )mpkChannelGetUserData( c );
```

```
        mpkChannelReadFrame( c, frame, channelData->region );
    }
```

The example `flip.adaptiveRB` computes the screen area covered by the scene in the update-channel draw callback in order to optimize the frame transport using the read-output callback. The read-output compound callback for the MPKConfig can be set using **mpkConfigSetCompoundReadOutputCB()**.

Custom Assembly

The purpose of the custom MPKCompound interface is to allow customization of the MPKCompound pre- and post-assembly passes.

Upon the **mpkConfigFrame()** invocation, each MPKWindow thread traverses the configuration's compound tree[s] and updates each compound whose channel belongs to the window. If the compound is a leaf node, then MPK invokes the user-specified clear and update callbacks on the associated [source] channel. Otherwise, if the compound is not a leaf node, then MPK assembles the frames output from the compound children into its associated [destination] channel.

This traversal actually occurs in several passes. Figure 3-20 shows the update of one window as in Figure 2-4 on page 29, but here Figure 3-20 shows the necessary callbacks for compound processing. A complete update for one channel consists of the following actions:

1. Invoke update clear callback if this channel is used as source.
2. Invoke pre-assemble callback if this channel is used as destination.
3. Invoke update draw callback if this channel is used as source.
4. Invoke post-assemble callback if this channel is used as destination.

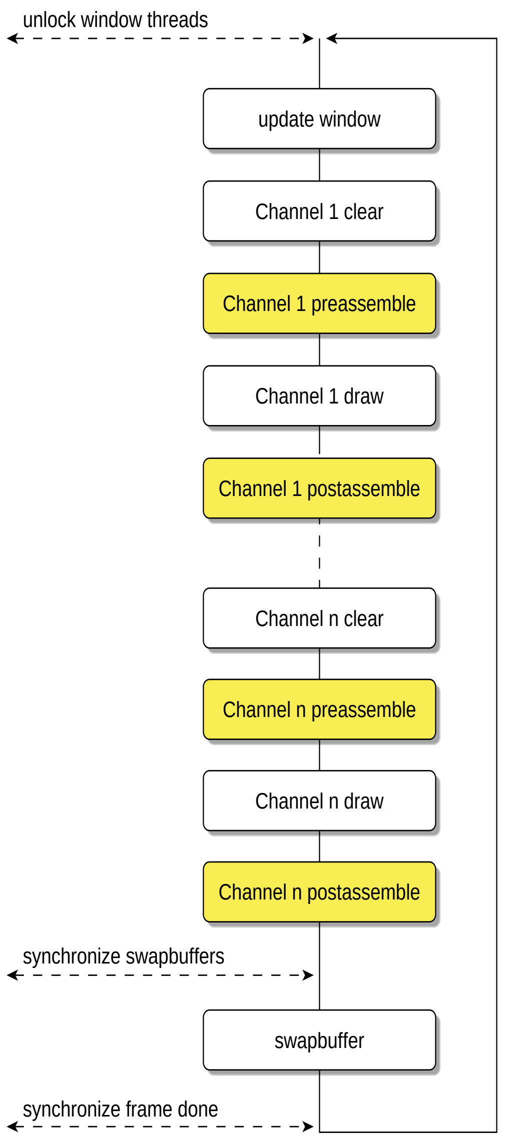

Figure 3-20 A Detailed Window Update

Neither the pre- nor post-assemble callback is invoked if any of the following conditions are true:

- The compound channel is `NULL`.
- The compound channel's window is frozen.
- The compound latency is greater than current frame number (initial countdown).

A channel can be used as source and destination at the same time, it may even be used more than once as a source channel. In that case, each callback may be invoked more than once during a frame.

The default pre- and post-assemble callback perform the following tests in order to determine if they should perform any assembly:

- The compound mode is not `NOCOPY`.
- The compound has input frames—that is, is a destination channel.

If one of these tests fail, they do not assemble. In addition, **mpkCompoundPreAssemble()** tests that the `ASYNC` flag is set. Then the images output from the source channels during last frame are assembled in the destination channel prior to any rendering. Likewise, **mpkCompoundPostAssemble()** tests that the `ASYNC` flag is not set. Then the images from the source channels are assembled in the destination channel during the same frame as soon as they have been produced.

By overriding the default assemble callbacks, the application can customize the way frames are assembled in the destination channel. This technique is used, for example, in volume rendering, where the input frames are assembled in a fixed order using special blending modes.

Note that if the compound channel is identical to its parent channel, then no output frame is generated for this compound.

Customizing the Clear Callback

The compound clear callback is invoked on all destination compounds. The default callback, **mpkCompoundClear()**, only clears the framebuffer in special cases—for example, when the adaptive readback callback is used. This callback should only be customized if really necessary since **mpkCompoundClear()** optimizes the clear as extensively as possible.

Traversing Compounds

The MPKCompound data structure is a container for children of MPKCompound, each associated with an existing MPKChannel. The resulting MPKCompound tree can be traversed using the following three functions:

Function	**Description**
mpkCompoundTraverseAll()	Traverses all children of the specified MPKCompound, regardless of any state information.
mpkCompoundTraverseActive()	Traverses the active children of the passed MPKCompound with respect to the current stereo mode (see the previous section "Stereo-Selective Compounds").
mpkCompoundTraverseCurrent()	Only traverses the active and current children of the passed MPKCompound with respect to the current stereo mode and current DPLEX cycle.

They provide a general mechanism for traversing a compound hierarchy in a top-to-bottom, left-to-right order. Figure 3-21 shows the traversal for the MPKCompound tree of the used in Example 3-10 in the previous section "Multilevel Decomposition".

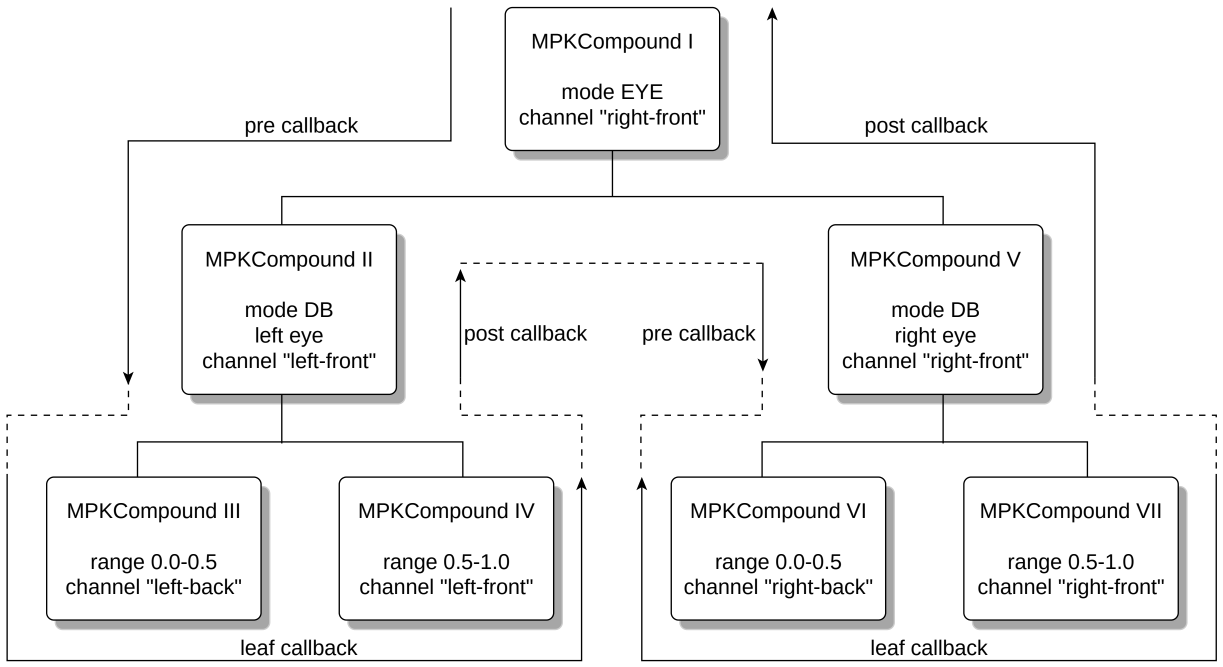

Figure 3-21 MPKCompound Traversal

The traversal functions are using three types of callbacks during traversal:

Callback Type	**Description**
Leaf callback function	Applied only on the leaf nodes of the compound tree, that is, on compounds without any children (in Figure 3-21, compounds III, IV, VI, and VII).
Pre-callback functions	Applied when traversing parent compounds downwards. In Figure 3-21, they apply to the compounds I, II, and V.
Post-callback functions	Applied when traversing parent compounds upwards. In Figure 3-21, they apply to the compounds I, II, and V.

The return values of these functions must be either `MPK_TRAV_CONT`, `MPK_TRAV_PRUNE`, or `MPK_TRAV_TERM` to indicate that the traversal should continue, skip this node, or terminate, respectively. The value `MPK_TRAV_PRUNE` is equivalent to `MPK_TRAV_CONT` for the post-callback function.

Advanced MPK Programming

This chapter describes the following advanced topics:

- "Using an Alternate Parser"
- "Creating Configurations without a Configuration File"
- "The Idle Callback"
- "Controlling the Frame Rate"
- "Data Handling"
- "MPK and Xinerama"
- "Hardware Compositing"
- "Advanced Compositing"
- "MPK and Other APIs"

Using an Alternate Parser

The MPK configuration file loader allows the specification of a preprocessor through the use of the `MPK_PARSER_CMD` environment variable. The preprocessor command will be invoked with the configuration file name as the first argument, and its output will be parsed by the normal MPK loader. Therefore, the output of this preprocessor has to be an MPK configuration file. For example, the C preprocessor can be used to put different configurations in one ASCII file and to select one using the `#define` functionality.

Creating Configurations without a Configuration File

The MPK configuration file loader is using only MPK's exposed C functional interface. Therefore, it is possible to create configurations programmatically or even write an alternate parser. The function in Example 4-1 creates a simple one-window configuration.

Example 4-1 Creating a Configuration Programmatically

```
MPKConfig *createSimpleConfig( void )
{
    // new config
    MPKConfig *config = mpkConfigNew();
    mpkConfigSetName( config, “1-window” );

    // one pipe
    MPKPipe *p = mpkPipeNew();
    mpkConfigAddPipe( config, p );
    mpkPipeSetName( p, “pipe” );

    // one window with viewport [0.25, 0.25,0.5,0.5]
    MPKWindow *w = mpkWindowNew();
    mpkPipeAddWindow( p, w );
    mpkWindowSetName( w, “MPK: simple” );
    float vp[4] = { 0.25, 0.25, 0.5, 0.5 };
    mpkWindowSetViewport( w, vp );

    // one channel
    MPKChannel *c = mpkChannelNew();
    mpkWindowAddChannel( w, c );
    mpkChannelSetName( c, “channel” );
    vp[0]=0.0; vp[1]=0.0; vp[2]=1.0; vp[3]=1.0;
    mpkChannelSetViewport( c, vp );
    float bl[] = { -.5, -.5, -1},
          br[] = {.5,  -.5, -1},
          tl[] = {-.5, .5, -1};
    mpkChannelSetWall( c, bl, br, tl );

    // mono and stereo monitor characteristics
    mpkPipeSetAttribute( p, MPK_PATTR_STEREO_TYPE, MPK_STEREO_RECT );
}
```

The Idle Callback

As described in the section “The Rendering Callbacks” in Chapter 2, the idle callback can be used to utilize the idle time in the application thread during the rendering of a frame. The exact idle time depends on a number of conditions—for example, the decomposition mode or the current viewpoint. The function **mpkConfigIsIdle()** can be used to optimize the usage of this idle time, as shown in Example 4-2.

Example 4-2 A Simple Idle Callback

```
void configIdle( MPKConfig *config )
{
    while( mpkConfigIsIdle( config ))
    {
        // do some processing
    }
}
```

Note that no data used in the rendering callbacks should be modified in the idle callback.

Controlling the Frame Rate

MPK currently supports two timers, `MPK_TIMER_AUTO` and `MPK_TIMER_FRAME`. By default, only the `MPK_TIMER_AUTO` timer is enabled. Both timers can be enabled and disabled using the functions **mpkConfigTimerEnable()** and **mpkConfigTimerDisable()**. MPK timer values are always expressed in milliseconds.

The `MPK_TIMER_AUTO` timer is used for automatically load-balancing DPLEX compounds. Its duration, determined by MPK, is based on the rendering time of the DPLEX children and cannot be set by the application.

The `MPK_TIMER_FRAME` timer is used to control the execution time of a MPKConfig frame. When the timer is enabled, **mpkConfigTimerGetTime()** returns the actual duration of the last MPKConfig frame. The desired minimal duration for the subsequent MPKConfig frames can be set using the function **mpkConfigTimerSetTime()**.

MPK uses a POSIX timer to fire alarms. By default, the signal `SIGALRM` is used to deliver this signal. The function **mpkGlobalSetTimerSignal()** may be used to change this signal. When using `pthread` execution mode, the delivery of the timer signal has to be blocked in all threads, except the application thread. Example 4-3 shows the code used by MPK to block the timer signal in all internally created threads.

Example 4-3 Blocking the Timer Signal

```
// block timer signal in this thread
if( mpkGlobalGetExecutionMode() == MPK_EXECUTION_PTHREAD )
{
    int signal = mpkGlobalGetTimerSignal();
    sigset_t set;

    sigemptyset ( &set );
    sigaddset( &set, signal );
    pthread_sigmask( SIG_BLOCK, &set, NULL );
}
```

Data Handling

MPK applications are multithreaded and may render several views and versions of the same database due to the latency imposed by some decomposition modes. Given this possibility, some types of application data have to be handled carefully. Within an MPK application, there are typically the following types of data:

- Application-only data
- Static shared data
- Dynamic shared data
- Frame data

Application-Only Data

Application-only data is only used by the application thread. Therefore, no special attention has to be paid when using this kind of data.

Static Shared Data

Static shared data is initialized before calling **mpkConfigInit()** and is used afterward only in a read-only fashion. Like application-only data, no special precautions have to be taken.

Dynamic Shared Data

Dynamic shared data is data modified during run time. Therefore, it has to be allocated using **mpkMalloc()** or **mpkCalloc()** to allocate the memory from a shared arena in `fork` execution mode. If this data is accessed for reading and writing concurrently from several threads, it has to be protected using a mutex.

Frame Data

Frame data is shared data that is characteristic to a specific frame. This can be the viewpoint, the position of dynamic parts in the scene, or lighting characteristics. This data, like dynamic shared data, has to be allocated using **mpkMalloc()** or **mpkCalloc()**. Frame data is passed from the application thread (the producer) to the rendering threads (the consumers) through the use of **mpkConfigFrame()**. MPK stores past frame data and passes it to the rendering callbacks according to the latency to be rendered by this callback. When a frame data pointer is not needed anymore, it is passed to the MPKConfig's free-data callback to be freed by the application. Due to the nature of frame data, mutual exclusion is not necessary. The application thread writes to frame data before passing it to **mpkConfigFrame()**. After it is passed to MPK, the frame data is only accessed by the rendering threads in a read-only fashion.

MPK and Xinerama

Xinerama is an X server extension that presents multiple physical screens managed by an X server as a single logical screen to X client applications. Xinerama does not support this abstraction for applications using OpenGL and GLX. A Xinerama-enabled X server will execute all OpenGL and GLX calls on the first physical pipe.

SGI Xinerama is an enhanced version of Xinerama. SGI Xinerama provides the same functionality but is optimized for use on SGI graphics systems. Subsequent references to Xinerama will refer to SGI Xinerama.

In order to meet the various needs of different applications, MPK provides flexible support for Xinerama-enabled X servers. MPK provides the following two features to integrate the Xinerama functionality:

- Support for Xinerama-aware windows
- Transparent scalability for Xinerama windows

Support for Xinerama-Aware Windows

MPK allows the creation of Xinerama-aware windows. These windows can be specifically created on one of the real pipes of the logical Xinerama screen. Thus, the feature allows OpenGL rendering on this pipe while the system is still running a Xinerama-enabled X server. Such windows are typically used for source channels contributing to a compound.

The window attribute hint `MPK_WATTR_HINTS_XINERAMA` is used to determine if a window should be created using the Xinerama extension or by bypassing the extension on a Xinerama-enabled X server. If set to true, the default value, the window is created using Xinerama. Otherwise, MPK bypasses Xinerama. This hint has no effect on X servers having the Xinerama extension disabled.

The use of Xinerama-aware windows has some caveats with respect to window handling and the X events that are received for these windows. The following two caveats are noteworthy:

- Xinerama-aware windows bypass the window manager—that is, they are not handled at all.
- Mouse events report their position with respect to the physical pipe as if no Xinerama is used. In contrast, mouse events for Xinerama windows are reported with respect to the logical pipe. Moving the mouse pointer over the window does not necessarly give you the keyboard focus on that window; the keyboard events may go to another window. Mouse events always go to the pointer-defined window.

Transparent Scalability for Xinerama Windows

MPK transparently parallels the rendering for Xinerama windows across the pipes virtualized by Xinerama.

In order to correctly handle the window, the MPKWindow, the X11 and the OpenGL initialization and exit have to be separated. The MPKConfig's window initialization callback initializes the MPKWindow in which the X11 and OpenGL initialization callbacks are set. Likewise, the MPKConfig's exit-window callback does clean up the MPKWindow, and the window X11 and OpenGL exit callbacks are responsible for de-initializing X11 and OpenGL.

Hardware Compositing

MPK offers native support for compositing hardware. Eliminating the need for pixel transfers and software recomposition, this specialized hardware performs the recomposition part of an MPKCompound.

Currently, MPK provides support for DPLEX option boards and the SGI Scalable Graphics Compositor. MPK handles the setup and control of the hardware by using the `GLX_SGIX_hyperpipe` API (see the `hyperpipe`(3) man page). An application written for MPK can immediatly take advantage of this hardware without having to take care of the `GLX_SGIX_hyperpipe` extension.

The mode flag `HW` is used to enable the support for DPLEX, EYE, FSAA, and 2D compounds. The mode flag `NOCOPY` has to be specified to disable the pixel transfer and software recomposition.

Advanced Compositing

MPK provides two data structures to give access to the transported RGBA, depth and stencil values during compound assembly:

- MPKFrame
  This data structure holds the RGBA, depth, and stencil MPKImage for one compound child along with the region it covers in the destination channel.
- MPKImage
  This data structure holds the pixel data along with information about the format, size, and position of this image.

MPKFrames for compound children can be retrieved using the function **mpkCompoundGetAssemblyFrame()** in the pre- and post-assemble callbacks (see section "Custom Assembly" on page 85).

Note: The function **mpkFrameDelete()** should not be called on MPKFrames acquired using **mpkCompoundGetAssemblyFrame()**, since these are managed by MPK.

The format field of an MPKFrame indicates which images it contains. The region defines the fractional viewport covered by this frame with respect to the destination channel.

The individual MPKImages for each MPKFrame can be retrieved using **mpkFrameGetImage()**. For assembly frames, each MPKImage used has a buffer allocated, which holds the pixel data for this MPKImage. The format and type specify the attributes of the pixel data, as described in the man page `glDrawPixels`(3G). The size of the MPKImage defines the width and height of the pixel data stored in the buffer. The offset specifies the pixel offset with respect to the region of the parent MPKFrame or **mpkChannelDrawImage()**. If you change one parameter, you must ensure that the other parameters (especially, the allocated buffer) are adjusted accordingly.

The final pixel viewport for one MPKImage is computed as follows:

```
width = ChannelPixelViewport[2];
height = ChannelPixelViewport[3];

// compute frame's pixel viewport
MPKImagePixelViewport[0] = MPKFrameRegion[0] * width;
MPKImagePixelViewport[1] = MPKFrameRegion[1] * height;

MPKImagePixelViewport[2] = (MPKFrameRegion[0]+MPKFrameRegion[2]) *
width;
MPKImagePixelViewport[3] = (MPKFrameRegion[1]+MPKFrameRegion[3]) *
height;

MPKImagePixelViewport[2] -= MPKImagePixelViewport[0];
MPKImagePixelViewport[3] -= MPKImagePixelViewport[1];

// clip to image size
if ( MPKImagePixelViewport[2]  > MPKImageSize[0] )
    MPKImagePixelViewport[2] = MPKImageSize[0];
if ( MPKImagePixelViewport[3] > MPKImageSize[1] )
    MPKImagePixelViewport[3] = MPKImageSize[1];
```

```
// add image offset
MPKImagePixelViewport[0] += MPKImageOffset[0];
MPKImagePixelViewport[1] += MPKImageOffset[1];
```

Figure 4-1 illustrates a frame and an image within one channel.

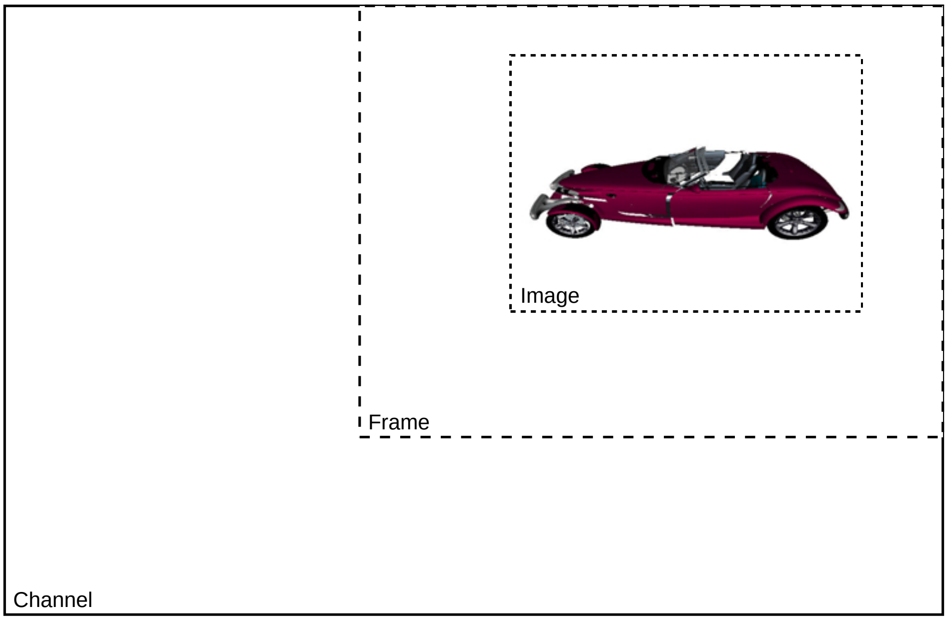

Figure 4-1 A Frame and an Image within one Channel

One possible use of this interface is to do software-based composition, as described by the following pseudo-code:

```
void assembleCB( MPKCompound *compound, void *userData )
{
    //====================
    // tile assembly frames :
    //
    // assembly images may have different positions and sizes.
    // Therefore we need to extract tiles in which the number of
    // contributing images is constant. For this we simply sort
    // all input image boundaries into xDim[] and yDim[] vectors,
    // so that
    //
```

```
// tile[i,j] = [ xDim[i], xDim[i+1]-1 ] x [ yDim[j], yDim[j+1]-1 ]

    for each assembly frame
        for each image (color, depth, stencil)

            compute image’s xStart, xEnd, yStart, yEnd

            sorted-add xStart, xEnd to xDim vector
            sorted-add yStart, yEnd to yDim vector

        end for
    end for

//====================
// allocate output frame and image buffers :
//
// the output frame dimensions (ie. frame origin and images size)
// corresponds to the bounding box of all tiles.
// Note that we could also allocate a vector of output tiles, and
// later perform tiled re-composition.
// Note also that the image buffers must be initialized.

    allocate output frame
    set origin to xDim[0], yDim[0]

    allocate image buffers for output frame
    set output frame images to image buffers
    set size of output images to xDim[n]-xDim[0], yDim[m]-yDim[0]

    initialize image buffer values wrt depth, color, stencil

//====================
// compose output frame :
//
// loop over all tiles (boundaries), and
//
// 1. first setup a vector of flags that is indexed by frame#
//    and imageType. Each element of the vector indicates whether
//    the corresponding image contributes to the tile.
//
//  2. then we loop over all pixel coordinates within the tile
//     and compose each assembly frame’s pixel value, depending
//     on the flag for this frame and image type.
```

```
        for each tile ( i, j )

            // flag image contributions for this tile

            for each assembly frame
                for each image (color, depth, stencil)

                    compare xStart, yStart with tile boundaries
                    set flag[frame#][imageType] accordingly

                end for
            end for

            if no image used in the tile
                continue
            end if

            // compose all contributing pixels

            for each x  ( xDim[i], ... xDim[i+1]-1 )
                for each y ( yDim[j], ... yDim[j+1]-1 )

                    for each assembly frame
                        for each image (color, depth, stencil)

                            if !flag[frame#][imageType]
                                continue
                            end if

                            compose image.pixel( x, y ) to output
                            frame’s corresponding
                            image.pixel( x-xDim[0], y-yDim[0] )

                    end for
                end for

            end for
        end for
}
```

MPK and Other APIs

This section describes items to consider when you use MPK in conjunction with the following libraries:

- "OpenGL Volumizer 2"
- "Open Inventor"
- "Motif"
- "Non-Thread-Safe Libraries"

OpenGL Volumizer 2

There are several topics to consider when using MPK with OpenGL Volumizer:

- "Execution Modes"
- "Rendering"
- "Scalability"

For sample code, see the `volview` application that ships with OpenGL Volumizer. The application is a full volume-viewer application that uses OpenGL Multipipe SDK as the underlying software layer to provide run-time configurability and scalability.

Execution Modes

MPK supports the `pthread`, `fork`, and `sproc` multiprocess mechanisms. The following table describes their use with OpenGL Volumizer:

Mode	**Prescribed Use with OpenGL Volumizer**
`pthread`	Works well with OpenGL Volumizer since OpenGL Volumizer is thread-safe.
`fork`	Works well with OpenGL Volumizer since OpenGL Volumizer is thread-safe. Use the vzMemory class to ensure allocation of OpenGL Volumizer objects from shared memory by using the code shown in Example 4-4.
`sproc`	Does not work with OpenGL Volumizer since OpenGL Volumizer links against the `pthread` library.

Example 4-4 Using vzMemory to Allocate Objects from Shared Memory

```
// Set the allocation and deallocation callback functions
vzMemory::setMemoryManagementCallbacks(allocate, deallocate, NULL);

// The allocator callback function
void *allocate(size_t size, void *userData) {
   return mpkMalloc(size);
}

// The de-allocator callback function
void deallocate(void *pointer, void *userData) {
        mpkFree(pointer);
}
```

Rendering

Ensure that OpenGL Volumizer nodes are stored as part of the shared data for the application. The data should not be replicated across multiple pipes unless there are special needs since volume data tends to be quite big in practice. OpenGL Volumizer render actions manage the graphics resources on a per-pipe basis—that is, they create one render action per MPKWindow. Do not use multiple MPKWindows per MPKPipe since this will lead to inefficient resource management and also force unnecessary context switches. However, this practice might be okay for testing purposes on single-pipe machines.

Scalability

The following notes are pertinent regarding scalability and stereo:

- 2D decomposition

  Scales fill rate in a straightforward manner. If you can divide the volume data into smaller bricks, you can use view-frustum culling to scale the texture memory size also by moving these bricks in and out of the texture memory of the pipes.
- DB decomposition

  Scales fill rate and texture memory size. Divide the volume data into multiple bricks and render equal number of bricks on each of the pipes. You need to pre-modulate the transfer function (lookup table) before rendering to compensate for the image blending operation used to composite the results together.

- 3D decomposition

  Data streaming (3D decomposition) is not supported by OpenGL Volumizer since the order in which the source channels are composited cannot be controlled in this mode. However, true volume decomposition can be achieved using DB decomposition, which enables the application to scale in texture memory by distributing the data across multiple pipes. This technique is illustrated in the `volview` example.
- DPLEX decomposition

  DPLEX rendering is supported by OpenGL Volumizer. Full-scale software recomposition (that is, where the destination channel is also contributing actively in the rendering) simply requires the application to set the OpenGL Volumizer slice callback appropriately so that **mpkChannelSyncDPlex()** is invoked after every slice in the rendering thread.
- Stereo

  Stereo is implemented in a straight-forward fashion.

Open Inventor

The critical concern in using Open Inventor with MPK is thread safety. There are currently two versions of Open Inventor, each differing in terms of thread safety:

- Open Inventor from SGI

  Open Inventor from SGI is not thread-safe. Therefore, applications using this version must use `fork` execution mode. The example `flip.iv` uses one Open Inventor scene graph and renderer per window process. This restricts the example to render static scenes but proper event handling could update the scene on each window thread to allow dynamically changing scenery.
- Open Inventor from TGS

  Starting with version 3.0, Open Inventor from TGS is thread-safe. This feature allows the rendering of the same scene graph by multiple threads concurrently. One renderer per window thread can then be used to render the scene graph, which may be transported through frame data, if necessary.

Motif

A Motif application is event-driven. Motif has an event loop, from which user-defined callbacks are called. A *WorkProc*, which is called whenever the application is idle, may be used to achieve a non-event-driven behavior. Instead of having a main loop, such as that in Example 2-13 on page 38, the application transfers this responsibility to Motif by calling **XtAppMainLoop()**. Therefore, a new MPKConfigFrame is either triggered directly from the event callbacks or from a WorkProc specified by the function **XtAppAddWorkProc()**. See the `XtAppAddWorkProc`(3Xt) man page for more details.

Non-Thread-Safe Libraries

Often libraries and functions that are not thread-safe are used during rendering. The ideal solution, if feasible, is to rework this software to be thread-safe. If that is not possible, the library has to be used in a way that prevents concurrent access of shared data. The following are possibile solutions:

- If the functionality is used sparsely, a global lock around all calls to the library may be used. Note that this lock prevents the window threads to use the library concurrently and may lead quickly to scalability problems. Additionally, any software that uses the OpenGL context to store certain information—for example, display lists—cannot be used using this approach.
- Depending on the task, it is often the best solution to initialize and use one instance of the given library per window. If the library is using global variables to store certain state information, this approach requires the `fork` execution mode to separate the address spaces between the window processes.

MPK Attributes

MPK uses attributes to specify certain properties for pipes, windows, and channels. In general, you can specify these attribute values in the following places:

- Individual data structures
- The MPKGlobal data structure (defaults only)
- The configuration file

The *SGI OpenGL Multipipe User's Guide* describes how you specify the attributes in the configuration file. This appendix describes the attributes and how you specify them in the individual data structures and in the special MPKGlobal data structure. The following sections are included:

- "MPK Attribute Names"
- "Managing Attributes"
- "MPKPipe Attributes"
- "MPKWindow Attributes"
- "MPKChannel Attributes"
- "MPKGlobal Attributes"

MPK Attribute Names

As described in the section "MPK Naming Conventions" in Chapter 2, MPK attribute names have three or more parts, the first two of which are the following:

1. MPK prefix
2. Attribute type

`CATTR`	Specifies a channel attribute.
`PATTR`	Specifies a pipe attribute.

`WATTR`	Specifies a window attribute.
`DEFAULT`	Used only in the case of the stereo-related attribute `MPK_DEFAULT_EYE_OFFSET`.

The remaining parts contain additional attribute descriptors. The following are examples of attribute names:

```
MPK_CATTR_FAR
MPK_PATTR_STEREO_WIDTH
MPK_WATTR_HINTS_RGBA
```

Managing Attributes

As cited in the introduction, you can programmatically manage the attributes in the individual data structures or in the MPKGlobal data structure. This section describes how you manage the attributes in the individual data structures, and the later section "MPKGlobal Attributes" describes how you do so from the MPKGlobal data structure.

The individual data structures of interest are the MPKPipe and MPKWindow data structures. The management of channel attributes is covered in the later section "MPKGlobal Attributes".

MPK provides the functions shown in Table A-1 to manage attributes in the MPKPipe and MPKWindow data structures.

Table A-1 Attribute-Managing Functions for Individual Data Structures

Function	Description

mpkPipeSetAttribute()
mpkWindowSetAttribute()
	Sets the specified attribute.

mpkPipeUnsetAttribute()
mpkWindowUnsetAttribute()
	Unsets the specified attribute. No value is assigned to that variable. The special enum value `MPK_PATTR_ALL` or `MPK_WATTR_ALL` can be used to unset all attributes of the respective data structure.

Table A-1 Attribute-Managing Functions for Individual Data Structures **(continued)**

Function	Description
mpkPipeResetAttribute() **mpkWindowResetAttribute()**	Sets the attribute to the default value obtained from the MPKGlobal structure. If the default value is `MPK_UNDEFINED`, the attribute is unset. The special enum value `MPK_PATTR_ALL` or `MPK_WATTR_ALL` can be used to unset all attributes of the respective data structure.
mpkPipeTestAttribute() **mpkWindowTestAttribute()**	Returns `1` if the attribute is set and, `0` if unset.
mpkPipeGetAttribute() **mpkWindowGetAttribute()**	Gets the value of the specified attribute if set and returns `1`. If the value is not set, the function returns `0`.

To set the attributes of a MPKPipe or MPKWindow data structure to their default values, MPK calls the function **mpkPipeResetAttribute(** *pipe*, `MPK_PATTR_ALL` **)** or **mpkWindowResetAttribute(** *window*, `MPK_WATTR_ALL` **)** upon creation of the data structure.

MPKPipe Attributes

Table A-2 describes the MPKPipe attributes.

Table A-2 MPKPipe Attributes

Attribute Name	Valid Values	Description
MPK_PATTR_MONO_HEIGHT	int	Specifies the height of the display to be used by the function **mpkWindowUpdatePixelViewport()** for mono mode instead of that returned by the X11 **DisplayHeight()** function. If no value is set, 492 is used for MPK_STEREO_RECT, MPK_STEREO_BOT, and MPK_STEREO_TOP.
MPK_PATTR_MONO_WIDTH	int	Specifies the width of the display to be used by the function **mpkWindowUpdatePixelViewport()** for mono mode instead of that returned by the X11 **DisplayWidth()** function.
MPK_PATTR_STEREO_HEIGHT	int	Specifies the height of the display to be used by the function **mpkWindowUpdatePixelViewport()** for stereo mode instead of that returned by the X11 **DisplayHeight()** function.
MPK_PATTR_STEREO_OFFSET	int	Specifies the offset of the display to be used by the function **mpkWindowUpdatePixelViewport()** for rect and bottom stereo modes. The default value is 532.

Table A-2 MPKPipe Attributes **(continued)**

Attribute Name	Valid Values	Description
`MPK_PATTR_STEREO_TYPE`	`MPK_STEREO_USER` `MPK_STEREO_QUAD` `MPK_STEREO_RECT` `MPK_STEREO_TOP` `MPK_STEREO_BOT`	Specifies one of the following stereo types: `none`, `user`, `quad`, `rect`, `top`, or `bottom`.
`MPK_PATTR_STEREO_WIDTH`	`int`	Specifies the width of the display to be used by the function **mpkWindowUpdatePixelViewport()** for stereo mode instead of that returned by `the` X11 **`DisplayWidth()`** function.

MPKWindow Attributes

Table A-3 describes the MPKWindow attributes.

Table A-3 MPKWindow Attributes

Attribute Name	Valid Values	Description
`MPK_WATTR_HINTS_CAVEAT`	`MPK_GLX_SLOW` `MPK_GLX_NOCAVEAT` `MPK_GLX_NON_CONFORMANT`	Specifies the caveats associated with the window framebuffer configuration.
`MPK_WATTR_HINTS_DECORATION`	`boolean`	Specifies whether the window should have window manager decorations.
`MPK_WATTR_HINTS_DIRECT`	`boolean`	Specifies whether the window GLX context should be direct.

Table A-3 MPKWindow Attributes **(continued)**

Attribute Name	Valid Values	Description
`MPK_WATTR_HINTS_DOUBLEBUFFER`	`boolean`	Specifies whether the window framebuffer configuration should be double-buffered. Note that setting this attribute on a window will affect the behavior of the function **mpkWindowSwapBuffers()**. The default is `1`.

`MPK_WATTR_HINTS_DRAWABLE`	`MPK_GLX_WINDOW`
	`MPK_GLX_PBUFFER`
	`MPK_GLX_PIXMAP`
		Specifies the window drawable type.

`MPK_WATTR_HINTS_LARGEST`	`boolean`	Specifies the MPKWindow pbuffer characteristics. This attribute will be ignored by windows for which the `DRAWABLE` hint is not set to `MPK_GLX_PBUFFER`.
`MPK_WATTR_HINTS_PRESERVED`	`boolean`	Specifies the MPKWindow pbuffer characteristics. This attribute will be ignored by windows for which the `DRAWABLE` hint is not set to `MPK_GLX_PBUFFER`.
`MPK_WATTR_HINTS_RGBA`	`boolean`	Specifies whether RGBA visuals are used. If the hint is not set, a color-index visual is used. The default is `1`.
`MPK_WATTR_HINTS_STEREO`	`boolean`	Specifies whether the window framebuffer configuration should support quad-buffer stereo.
`MPK_WATTR_HINTS_THREAD`	`boolean`	Specifies whether the window should be made a separate thread from the application.
`MPK_WATTR_HINTS_TRANSPARENT`	`boolean`	Specifies whether the window framebuffer configuration should be transparent.

Table A-3 MPKWindow Attributes **(continued)**

Attribute Name	Valid Values	Description
MPK_WATTR_HINTS_VISUAL	MPK_GLX_TRUE_COLOR MPK_GLX_PSEUDO_COLOR MPK_GLX_DIRECT_COLOR MPK_GLX_STATIC_COLOR MPK_GLX_GRAYSCALE MPK_GLX_STATIC_GRAY	Specifies the window visual type.
MPK_WATTR_HINTS_X_RENDERABLE	boolean	Specifies whether only framebuffer configuration that have associated X visuals (and can be used to render to windows and/or GLX pixmaps) should be considered.
MPK_WATTR_HINTS_XINERAMA	boolean	Determines if a window should be created using Xinerama (if enabled). Setting it to 1 causes the window to be created using Xinerama and setting it to 0 causes a Xinerama-aware window to be created. The default value is 1 if the XINERAMA_AWARE environment variable is not set. If XINERAMA_AWARE is set, the default value is the opposite value of XINERAMA_AWARE.
MPK_WATTR_PLANES_ACCUM_ALPHA	int	Specifies the minimum number of accumulation alpha bitplanes. This attribute is ignored if the RGBA hint of the window is not set.
MPK_WATTR_PLANES_ACCUM_BLUE	int	Specifies the minimum number of accumulation blue bitplanes. This attribute is ignored if the RGBA hint of the window is not set.
MPK_WATTR_PLANES_ACCUM_GREEN	int	Specifies the minimum number of accumulation green bitplanes. This attribute is ignored if the RGBA hint of the window is not set.

Table A-3 MPKWindow Attributes **(continued)**

Attribute Name	Valid Values	Description
MPK_WATTR_PLANES_ACCUM_RED	int	Specifies the minimum number of accumulation red bitplanes. This attribute is ignored if the RGBA hint of the window is not set.
MPK_WATTR_PLANES_ALPHA	int	Specifies the minimum number of alpha bitplanes. This attribute is ignored if the RGBA hint of the window is not set. The default is 0.
MPK_WATTR_PLANES_AUX	int	Specifies the number of auxiliary buffers.
MPK_WATTR_PLANES_BLUE	int	Specifies the minimum number of blue bitplanes. This attribute is ignored if the RGBA hint of the window is not set. The default is 1.
MPK_WATTR_PLANES_COLOR	int	Specifies the minimum color-index buffer size. This attribute is ignored if the RGBA hint of the window is set to 1.
MPK_WATTR_PLANES_DEPTH	int	Specifies the minimum size of the depth buffer. The default is 1.
MPK_WATTR_PLANES_GREEN	int	Specifies the minimum number of green bitplanes. This attribute is ignored if the RGBA hint of the window is not set. The default is 1.
MPK_WATTR_PLANES_LEVEL	int	Specifies the window buffer level. The default is 0.
MPK_WATTR_PLANES_RED	int	Specifies the minimum number of red bitplanes. This attribute is ignored if the RGBA hint of the window is not set. The default is 1.
MPK_WATTR_PLANES_SAMPLES	int	Specifies the minimum number of samples required in the multi-sample buffer.

Table A-3 MPKWindow Attributes **(continued)**

Attribute Name	Valid Values	Description
`MPK_WATTR_PLANES_STENCIL`	`int`	Specifies the minimum size of the stencil buffer.
`MPK_WATTR_TRANSPARENT_ALPHA`	`int`	Specifies the alpha component of the window transparent color. This attribute is ignored if the `RGBA` hint of the window is not set or if the `TRANSPARENT` hint of the window is not set.
`MPK_WATTR_TRANSPARENT_GREEN`	`int`	Specifies the green component of the window transparent color. This attribute is ignored if the `RGBA` hint of the window is not set or if the `TRANSPARENT` hint of the window is not set.
`MPK_WATTR_TRANSPARENT_BLUE`	`int`	Specifies the blue component of the window transparent color. This attribute is ignored if the `RGBA` hint of the window is not set or if the `TRANSPARENT` hint of the window is not set.
`MPK_WATTR_TRANSPARENT_INDEX`	`int`	Specifies the window transparent index. This attribute is ignored if the `RGBA` hint of the window is set or if the `TRANSPARENT` hint of the window is not set.
`MPK_WATTR_TRANSPARENT_RED`	`int`	Specifies the red component of the window transparent color. This attribute is ignored if the `RGBA` hint of the window is not set or if the `TRANSPARENT` hint of the window is not set.

MPKChannel Attributes

There are only two formal MPKChannel attributes: `MPK_CATTR_NEAR` and `MPK_CATTR_FAR`. Their descriptions are included in the next section.

MPKGlobal Attributes

The MPKGlobal data structure allows you to specify attribute defaults for pipes, windows, and channels. Some attributes—for example, the default eye offset—can only be specified at the MPKGlobal data structure level.

MPK provides MPKGlobal functions to manage the attributes for the various data structures. The function names have the following five parts:

1. The `mpkGlobal` prefix
2. The operation `Set` or `Get`
3. The data structure type (`Pipe`, `Window`, or `Channel`)
4. The word `Attribute`
5. A suffix to determine the data type of the attribute

The following are examples:

`void mpkGlobalSetPipeAttributei(int` *pattr*`, int` *val*`);`

`float mpkGlobalGetChannelAttributef(int` *cattr*`);`

Accessing an unset variable will either return `MPK_UNDEFINED` or the default value of this variable.

Table A-4 describes the MPKGlobal attributes not included in the prior sections "MPKPipe Attributes" and "MPKWindow Attributes".

Table A-4 MPK Global Attributes

Attribute Name	Valid Values	Description
`MPK_DEFAULT_EYE_OFFSET`	`float`	Specifies the default value of the eye offset used by the frustum computations for the channel. The function **mpkInit()** sets this value to 0.035.
`MPK_CATTR_NEAR`	`float`	Specifies the default near distance of the channel. This value is preempted by the function **mpkChannelSetNearFar()**.

Table A-4 MPK Global Attributes **(continued)**

Attribute Name	Valid Values	Description
`MPK_CATTR_FAR`	`float`	Specifies the default far distance of the channel. This value is preempted by the function **mpkChannelSetNearFar()**. The default is `100`.
`MPK_XINERAMA`	`boolean`	Controls window-intialization performance. This variable can be set to `0` if all windows are created using Xinerama, which is the default behavior. Setting it to `0` improves window-initialization performance but causes problems when creating Xinerama-aware windows. The default is `1`.

Index