SGI® OpenGL Multipipe™ SDK User's Guide

007-4239-002 Version 2.1

CONTRIBUTORS

Written by Ken Jones

Illustrated by Chrystie Danzer and Jenn Byrnes

Edited by Susan Wilkening

Production by Glen Traefald

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	May 2002 Original publication. Supports the 2.0 release of OpenGL Multipipe SDK.
002	October 2002 Updated to support the 2.1 release of OpenGL Multipipe SDK.

Contents

Figures

Tables

About This Guide

SGI OpenGL Multipipe SDK (MPK) 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. This guide describes how to use and configure an MPK application.

Audience

This guides targets Reality Center administrators. As such, you can configure graphics applications to run in multipipe environments. Using MPK, you can describe the physical display area (walls), the hardware resources, and the rendering options.

What This Guide Contains

This guide is divided into the following chapters:

- Chapter 1, “Overview,” describes the features of MPK and its components.
- Chapter 2, “Framebuffer Resources,” describes the function and hierarchy of the framebuffer data structures for pipes, windows, and channels.
- Chapter 3, “Frustum Descriptions,”describes a frustum in a virtual reality environment and two methods for computing a frustum for a projection system.
- Chapter 4, “Compounds,” describes the various schemes of decomposition available under MPK.
- Chapter 5, “Using Scalable Graphics Hardware,”describes the use of compounds with scalable graphics hardware.
- Chapter 6, “Configuration File Format,” describes the format of an MPK configuration file.

Related Publications

The following books contain additional information that may be helpful:

- *SGI OpenGL Multipipe SDK Programmer's Guide*
- *SGI InfinitePerformance: Scalable Graphics Compositor User's Guide*
- *Onyx2 DPLEX Option Hardware User's Guide*
- *IRIX Admin: Software Installation and Licensing*
- *OpenGL Multipipe User's Guide*

Obtaining Publications

To obtain SGI documentation, go to the SGI Technical Publications Library at the following URL:

http://docs.sgi.com

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://docs.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.

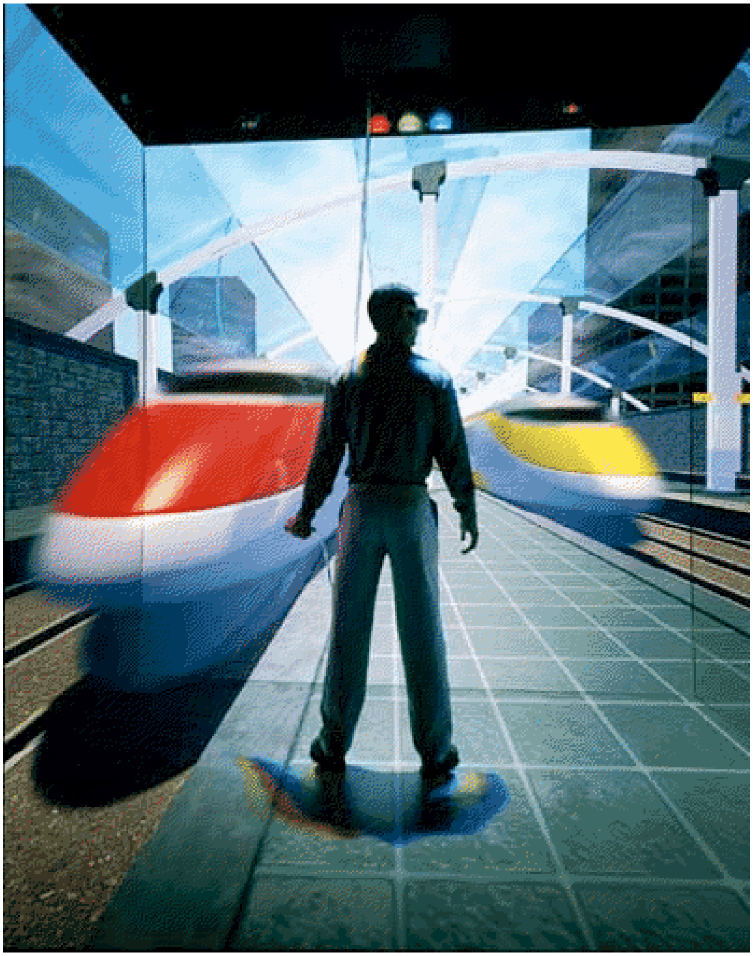


Figure 1-1 SGI Reality Center

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.

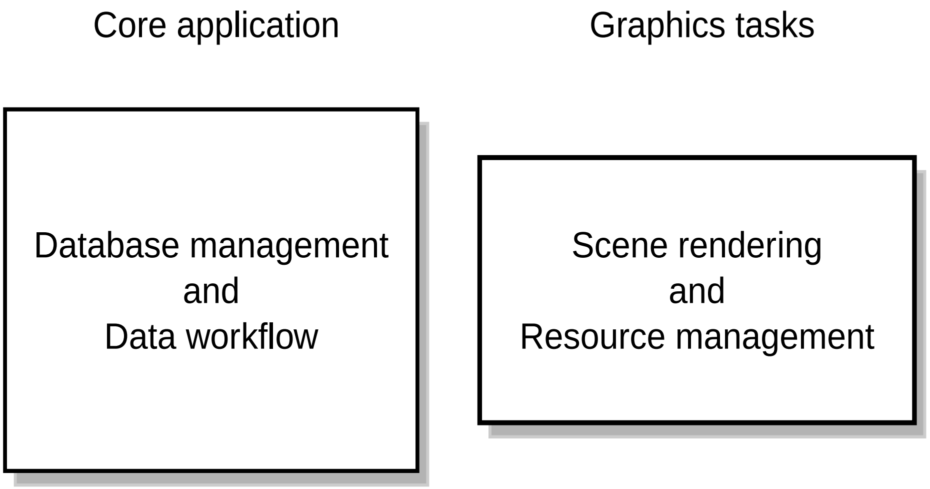


Figure 1-2 MPK Application Structure

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 ]
                }
            }
        }
    }
}
```

Chapter 2

Framebuffer Resources

As noted in the overview, MPK allows you to describe the framebuffer resources (pipes, windows, and channels), the physical display area (walls), and the rendering options. This chapter describes how you configure the framebuffer resources and contains the following sections:

- "The MPK Configuration Hierarchy"
- "The config Data Structure"
- "The pipe Data Structure"
- "The window Data Structure"
- "The channel Data Structure"
- "Stereo Description"

The MPK Configuration Hierarchy

The MPK configuration file uses a tree data structure to describe the physical graphics resources. The root of the data structure is the whole visualization facility and the leaves are the physical rendering layouts. Figure 2-1 shows the configuration of an application running on a two-pipe platform, two windows handling the GLX context, and four channels.

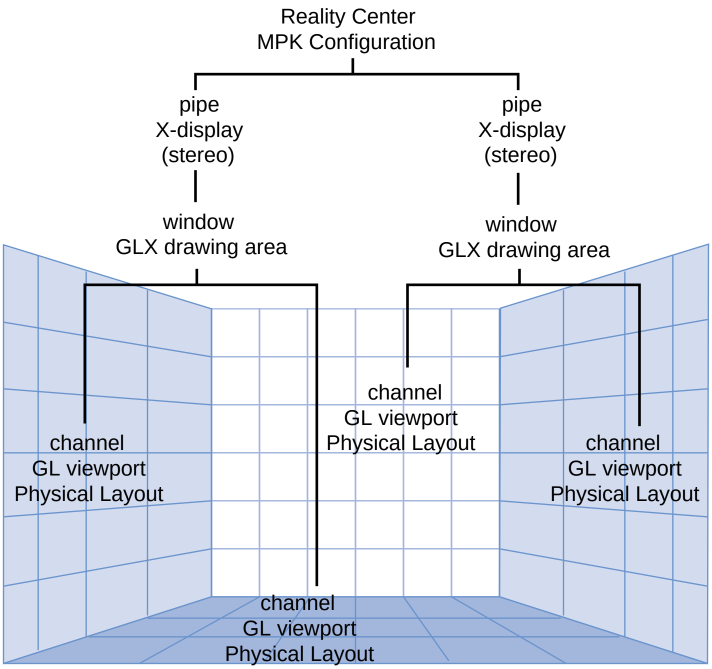


Figure 2-1 MPK Configuration Hierarchy

Example 2-1 shows a skeletal configuration file that describes Figure 2-1.

Example 2-1 Skeletal 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 the framebuffer data structures and Chapter 6, “Configuration File Format” describes their syntax for the configuration file.

The `config` Data Structure

The configuration level of the hierarchy, denoted in the configuration file by the `config` data structure, primarily describes the rendering resources of an MPK application as a hierarchy of the following:

- Hardware rendering pipelines (pipes)
- GLX software rendering threads (windows)
- OpenGL framebuffer rendering areas (channels)

It may also describe compounds, various parallelization schemes of the rendering across channels in order to scale performances. Chapter 4, “Compounds” describes the use of compounds in MPK.

As shown in Figure 2-1, pipes are children of the root configuration, windows are children of pipes, and channels are children of windows. As such, you can take advantage of the attendant inheritance. For instance, you can specify the screen dimensions at the pipe 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.

The `pipe` Data Structure

A `pipe` data structure describes the rendering resources within a configuration that are assigned to a given hardware rendering pipe. You must have one `pipe` entry for every pipe you want to use in your configuration. 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 the like) to be applied by its rendering threads (windows).

Example 2-2 shows a minimal pipe specification that is superior to that of the windows and channels.

Example 2-2 Sample Pipe Specification

```
pipe {
    display     ":0.0"
    window {
        .
        .
        .
        channel {
            .
            .
            .
        }
    }
}
```

You can specify the display sizes corresponding to the various stereo modes using global attributes or pipe attributes; otherwise, MPK uses the values returned by the X11 **DisplayWidth()** and **DisplayHeight()** functions. Chapter 6, "Configuration File Format" describes the pipe and global attributes.

The window Data Structure

A window data structure corresponds to a single GLX unit (that is, a single X window with its associated OpenGL visual and context). Essential in the MPK programming model is that each window spawns its own rendering thread.

In the configuration file, the window specification is subordinate to the pipe specification. Example 2-3 shows a minimal specification. Other optional specifications include the processor where the rendering thread is to run and an extensive set of window attributes. Chapter 6, "Configuration File Format" describes the complete set of fields and their syntax.

Example 2-3 Sample Window Specification

```
window {
    viewport    [ 0., 0., 1., 1. ]
    channel {
        .
        .
        .
    }
}
```

The `channel` Data Structure

A channel, denoted by a `channel` data structure in the configuration file, is essentially a view onto a scene and corresponds to a single viewport inside its parent window. In addition to the viewport description, a channel also contains the modeling coordinates for the projection rectangle in the real world. Chapter 3, “Frustum Descriptions” describes how you describe these coordinates.

Note: MPK allows applications to run in mirrored projection systems, which usually use mirrors to invert the projected image up and down (with respect to left and right) . To do this, you must specify a negative value for the height or width of the corresponding channel’s viewport.

Example 2-4 shows a sample channel specification.

Example 2-4 Sample Channel Specification

```
channel {
    viewport        [ 0., 0., 1., 1. ]
    projection {
        origin      [ 0., 0., 0. ]
        distance    3.
        fov         [ 54., 47. ]
        hpr         [ 0., 0., 0. ]
    }
}
```

Chapter 6, “Configuration File Format” describes the complete set of fields and their syntax.

Stereo Description

When running an MPK application in stereo mode, you can specify two elements from the configuration file:

- Stereo type

  This indicates how the framebuffer resources are configured for each eye pass. The stereo type is defined as a pipe attribute either in the global attributes section or per pipe in the `pipe` data structure. Chapter 6, “Configuration File Format” contains a detailed list of all attributes and describes how to specify global attributes.

- Stereo command

  This command will be executed by MPK when switching to mono or stereo rendering. The command is specified for mono and stereo in the `config` data structure.

Note: If you want to run in quad-buffered stereo mode, your windows must allocate stereo-capable visuals. This can be specified using the `stereo` window attribute hint.

The following is an example of stereo specifications in the `config` data structure:

```
mono "/usr/gfx/setmon -n 60HZ"
stereo "/usr/gfx/setmon -n 1024x768_96s"
```

Controlling the stereo specifications at the configuration or pipe levels requires you to kill windows and restart them. However, you can switch stereo on and off without killing them if all windows on all quad-stereo pipes have a quad-buffer-capable GLX visual and you use the window `hint` attribute, as shown in the following:

```
attributes {
    hints {
        stereo 1
    }
}
```

See Chapter 6, “Configuration File Format” for a complete description of the options and their syntax.

Frustum Descriptions

To allow graphical applications to behave properly in immersive environments, you must specify the physical layout of the display area and the relative position of the observer. MPK allows you to do this by extending the notion of the viewing frustum in a graphical application. This chapter uses the following sections to describe this task:

- "Orthographic Versus Perspective Frusta"
- "A Frustum in Immersive Environments"
- "Two Modeling Methods"
- "Examples of Common Reality Center Settings"

As described in Chapter 6, "Configuration File Format", MPK expects a frustum description as part of the `channel` data structure.

Orthographic Versus Perspective Frusta

Figure 3-1 depicts an orthographic frustum.

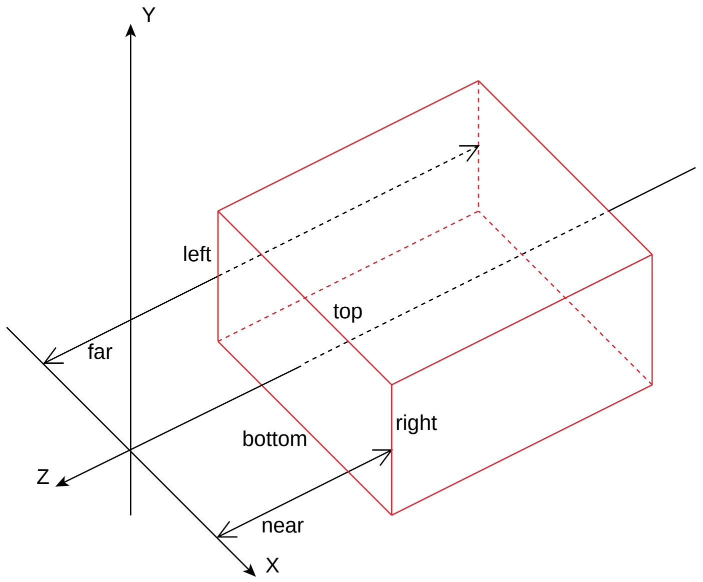


Figure 3-1 An Orthographic Frustum

Figure 3-2 depicts a perspective frustum.

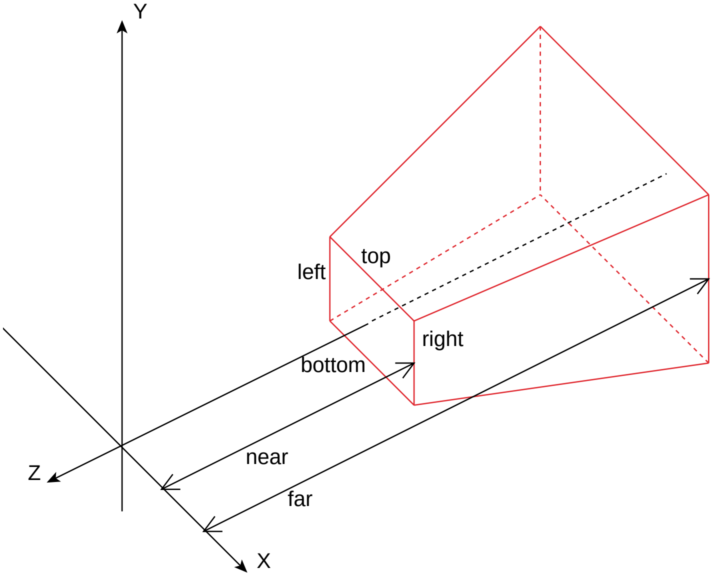


Figure 3-2 A Perspective Frustum

Note the following about the two frusta:

- Both are totally defined by the *near*, *far*, *left*, *right*, *top*, and *bottom* distances and the assumption of having the eyepoint at the origin and the *near* and *far* distances for each defined with respect to the Z axis. The OpenGL near and far planes are always parallel to each other and perpendicular to the Z axis. An off-axis OpenGL frustum corresponds to the case where the near-plane rectangle is not centered around the Z axis. This is generally the case in immersive environments. See Figure 3-4 on page 22 for an example of an off-axis frustum.
- Both final images (pixmaps) correspond to the near plane image.
- In the case of a perspective frustum and as illustrated in Figure 3-3, the near plane intersection with a line from the eyepoint to the scene graph object defines the pixel color. Distant objects appear smaller in the pixmap.
- In the case of an orthographic frustum and as illustrated in Figure 3-3 also, the near plane intersection with a line parallel to the Z axis and extending to the scene graph

object defines the pixel color. The size of objects is preserved in the orthographic view.

Figure 3-3 Projections through the Near Plane

Given these definitions of a pixmap, you control the appearance of the pixmap by selecting the type of frustum and setting the frustum parameters (*near*, *far*, *left*, *right*, *top*, and *bottom).*

In non-immersive environments, the location of the eyepoint and monitor screen in the real world is arbitrary and you do not need to account for it.

A Frustum in Immersive Environments

In non-immersive environments, you have arbitrary (usually symmetric) *left*, *right*, *top*, and *bottom* parameters.

In an immersive environment, you need to establish the location of the eyepoint and the monitor in real-world coordinates; they are no longer arbitrary. The monitor screen is now a see-through glass window into the scene graph. Figure 3-4 on page 22 illustrates the new effective frustum, which is completely determined by the following specifications:

- *near* and *far* distances
- Real-world eyepoint location
- Real-world screen position

MPK expects the dimensions (position and size) of the screen to be specified as part of the associated channel's data structure—that is, as if the screen itself was a 3D object in your database. You must specify the same units and coordinate system as will be used by the application when drawing the database.

MPK allows the eyepoint location to be specified using a simple programming interface. The application needs only to provide real-world information about the position and attitude of the viewer. MPK then transparently adapts its left- and right-eye frustum computation.

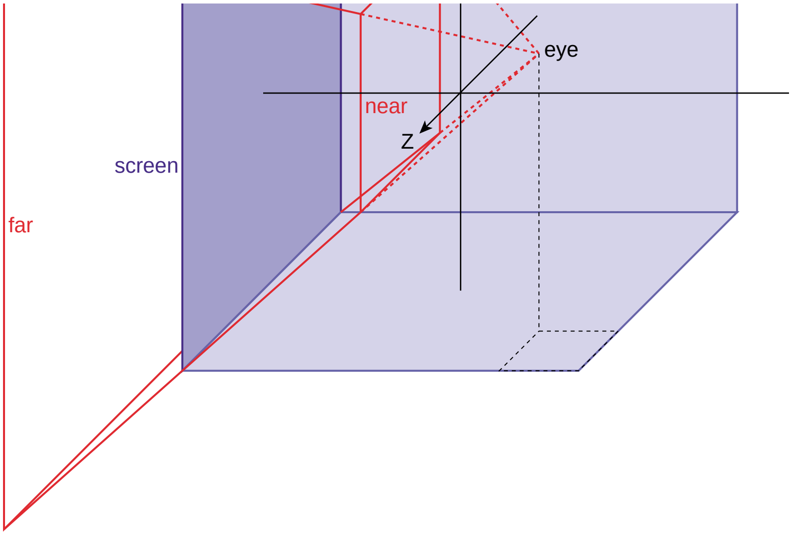


Figure 3-4 The Effective Frustum

Two Modeling Methods

MPK provides two modeling methods to describe your projection system:

- Wall model (Cartesian coordinates)
- Projection model (polar/cylindrical/spherical coordinates)

The most appropriate modeling method usually corresponds to the one requiring fewer manual computations in order to describe the geometry of your environment.

Specifying Wall Model Coordinates

Wall model coordinates are well-suited to describe projection screens that are arranged as flat screens, such as in a TANORAMA POWERWALL or TAN HOLOBENCH facility. To describe a projection screen using the wall model, you need to provide the Cartesian coordinates of three of its vertices using object data coordinates (*x′*,*y′*,*z′*):

- Bottom-left vertex
- Bottom-right vertex
- Top-left vertex

Example 3-1 shows an example of using wall model coordinates in a configuration file for a wall that is 3 meters by 3 meters and 1.5 meter in front of the viewer. The example assumes that the application expects the coordinates to be specified in meters:

Example 3-1 Specifying Wall Model Coordinates

```
#   a 3m x 3m screen located 1.5 m in front of the viewer
#
wall {
    bottom_left      [ -1.5, -1.5, -1.5 ]
    bottom_right     [  1.5, -1.5, -1.5 ]
    top_left         [ -1.5,  1.5, -1.5 ]
}
```

Specifying Projection Model Coordinates

Projection model coordinates are well-suited to describe projection screens that are in a curved or tilted arrangement, such as in the traditional three-pipe SGI Reality Center, the V-Dome, and the Responsive Workbench facilities. To describe a projection screen using the projection model, you need to provide the following information:

- Origin in Cartesian coordinates,using object data coordinates (x',y',z')
- Euler angles of the projection plane—that is, the counter-clockwise rotation around the Y axis (heading), X axis (pitch), and Z axis (roll) viewed from the positive side of the axis
- Distance of the projection plane from the origin in the application's measurement units
- Horizontal and vertical fields of view (FOV)

Example 3-2 shows an example of using projection model coordinates in a configuration file. The configuration is a three-channel, hemi-cylindrical Reality Center with the following dimensions:

Dimension	**Value**
Radius	3 meters
Edge blending	8 percent
Horizontal FOV	54 degrees (total FOV of 150 degrees)
Vertical FOV	47 degrees

Example 3-2 Specifying Projection Model Coordinates

```
channel {
    name “front-view”
    projection {
        origin      [ 0., 0., 0. ]
        distance    3.
        fov         [ 54., 47. ]
        hpr         [ 0., 0., 0. ]
    }
}
channel {
        name “left-view”
        projection {
            origin      [ 0., 0., 0. ]
            distance    3.
            fov         [ 54., 47. ]
            hpr         [ 50., 0., 0. ]
        }
}
channel {
        name “right-view”
        projection {
            origin      [ 0., 0., 0. ]
            distance    3.
            fov         [ 54., 47. ]
            hpr         [ -50., 0., 0. ]
        }
}
```

For each specified channel, the resulting projection area corresponds to the rectangle that would be produced by a hypothetical projection system located at `origin` with the orientation characterized by the `hpr` angles and projecting orthogonally onto a wall situated at `distance`.

Examples of Common Reality Center Settings

There are a variety of commercial products that illustrate the most appropriate choice for the two modeling methods. Figure 3-5 and Figure 3-6 illustrate flat-screen arrangements that are ideal for the use of the wall model.

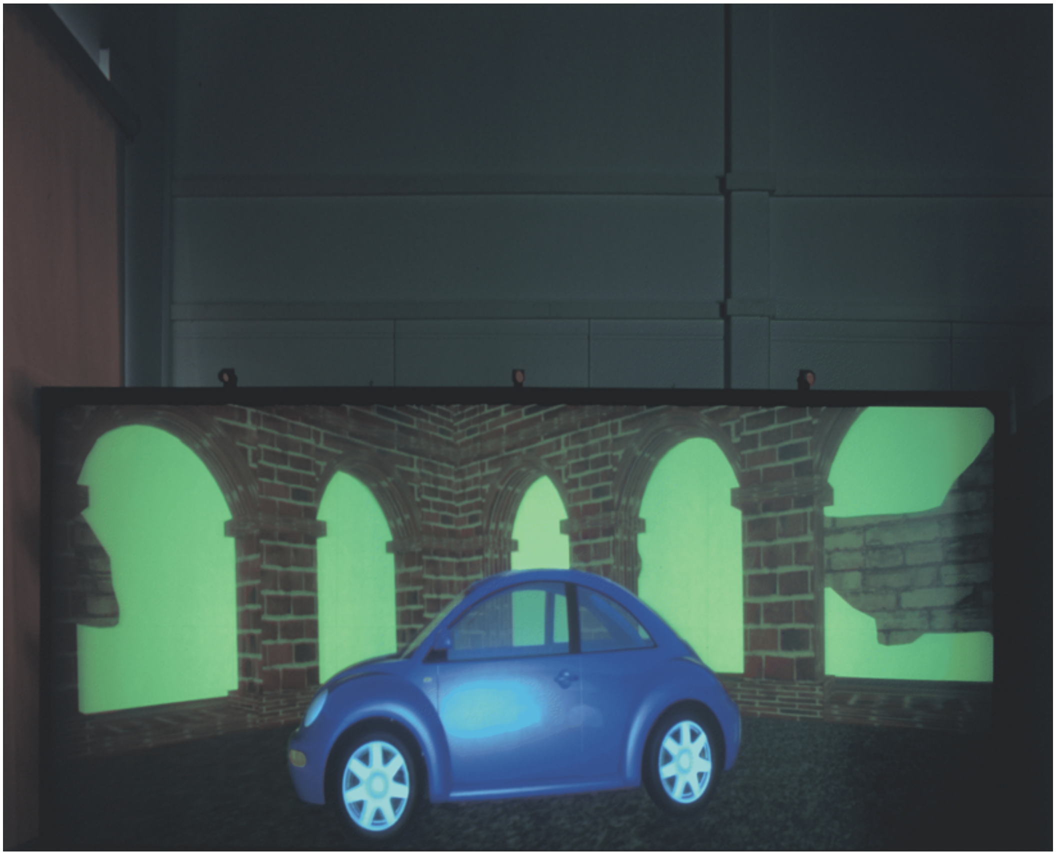


Figure 3-5 TANORAMA® POWERWALL (TAN/SGI Democenter)

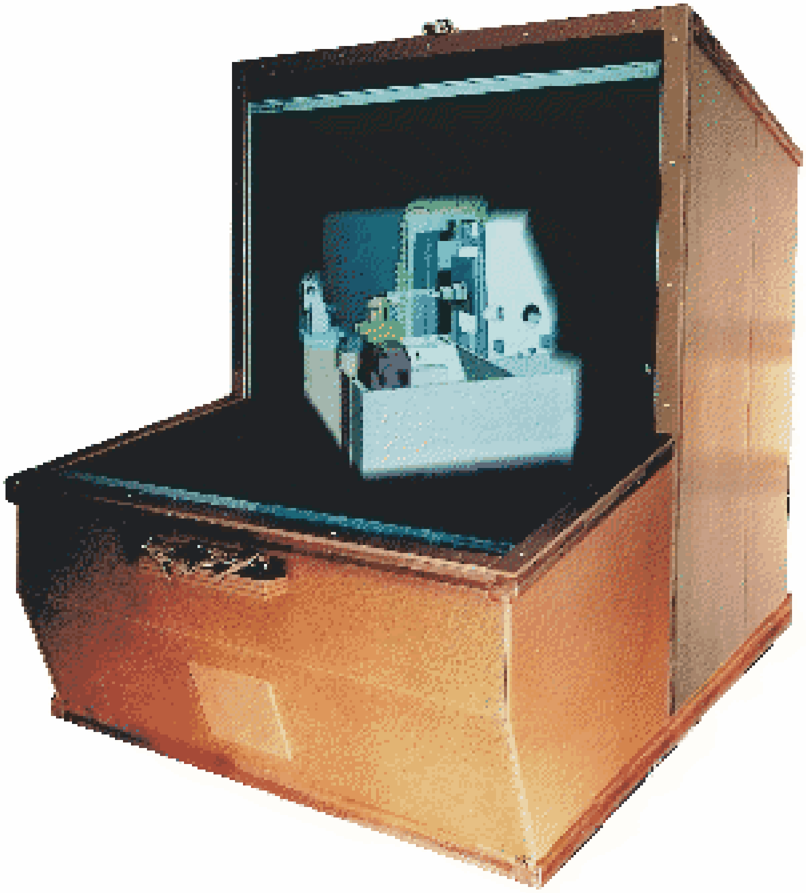


Figure 3-6 TAN HOLOBENCH® (photo courtesy of GMD)

Figure 3-7, Figure 3-8, and Figure 3-9 illustrate curved or tilted arrangements of projection screens and are ideal for the use of the projection model.

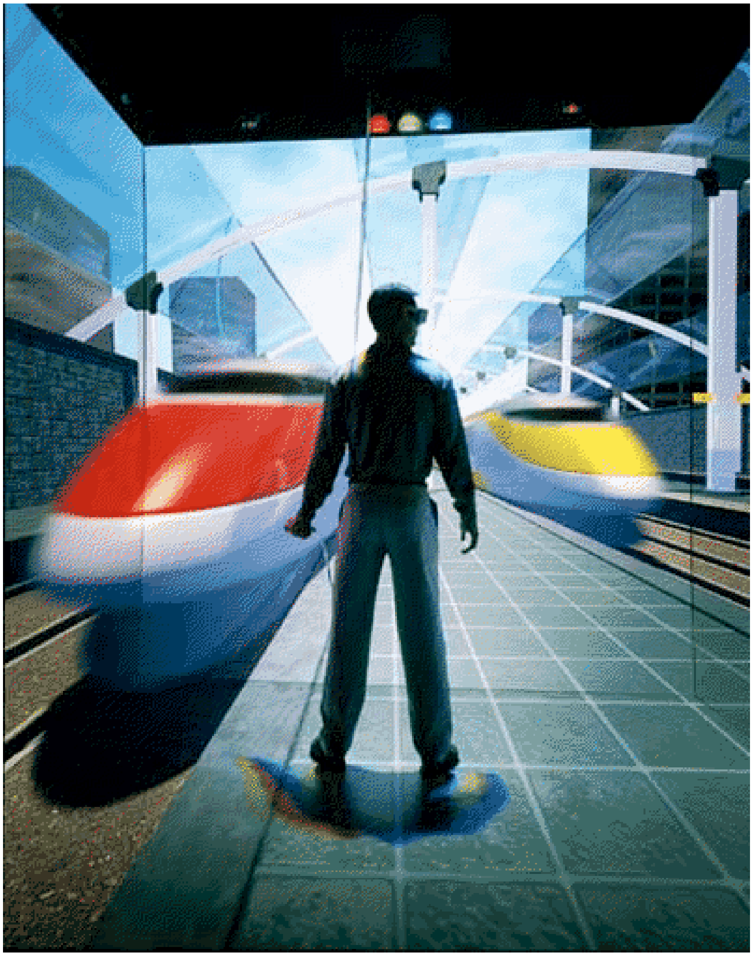


Figure 3-7 SGI Reality Center

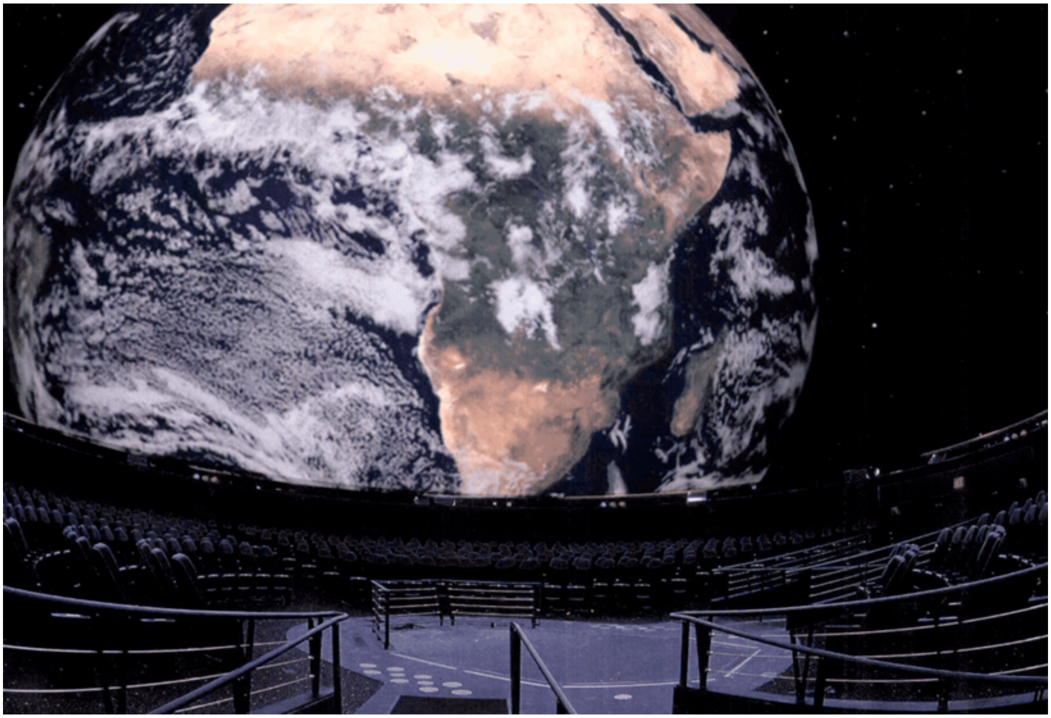


Figure 3-8 V-Dome™ (designed and installed by Trimension Systems, Inc.)

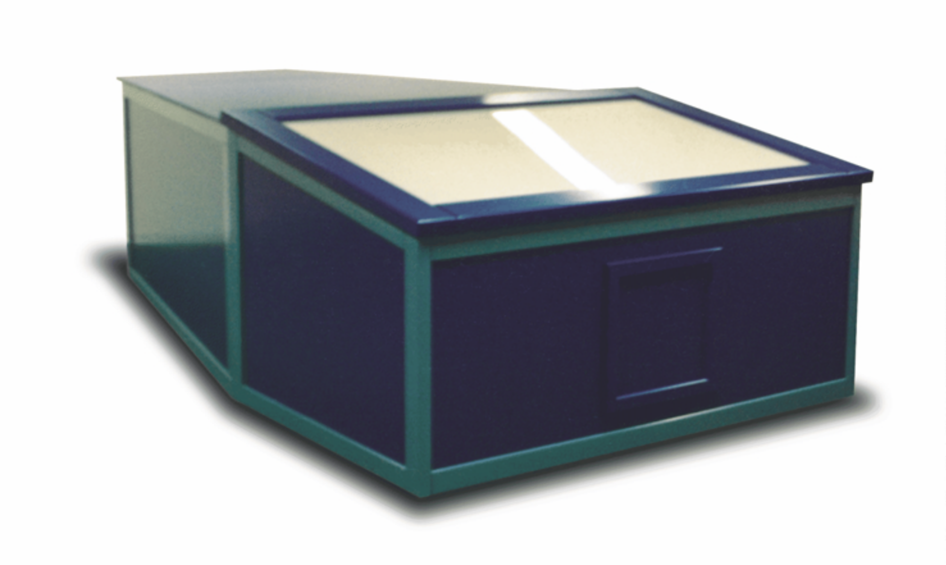


Figure 3-9 Responsive Workbench® (photo courtesy of DaimlerChrysler AG)

Chapter 4

Compounds

This chapter describes how you can use compounds (or conversely, decomposition) to scale the performance of your graphics system. 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"

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 4-1 illustrates the role of *source* and *destination* channels in scalable rendering.

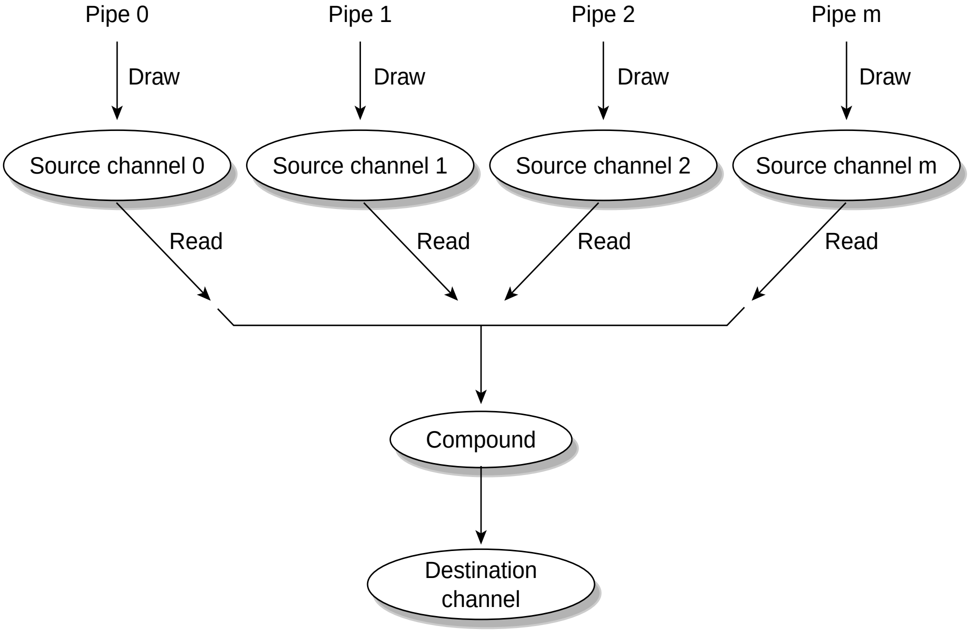


Figure 4-1 Source and Destination Channels

Building Compounds

To build a compound, you must create a `compound` data structure. Chapter 6, "Configuration File Format" describes the syntax of `compound` 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 50 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 descriptions of the `mode` and `format` fields in section Chapter 6, “Configuration File Format”. Chapter 5, “Using Scalable Graphics Hardware” describes the integration of scalable graphics with MPK. “Automatic Load Balancing for Compounds” on page 51 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 4-2 illustrates screen decomposition.

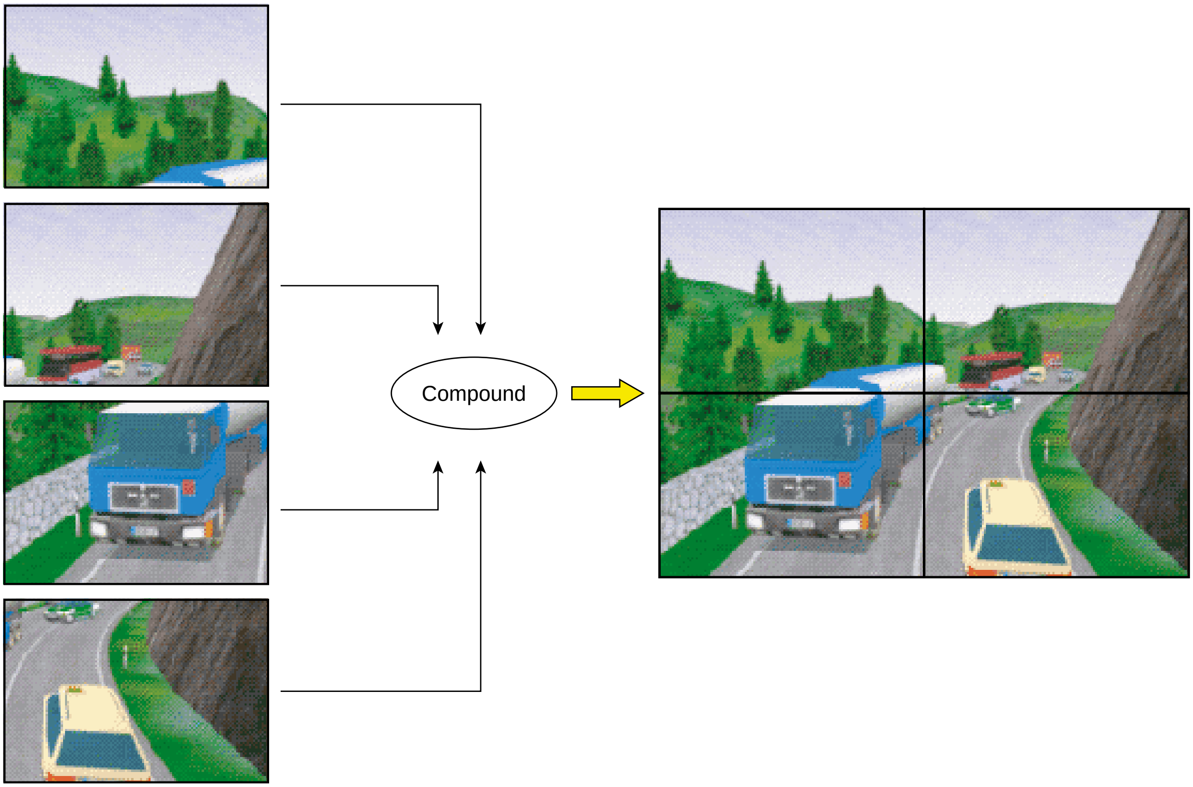


Figure 4-2 Screen Decomposition

Example 4-1 shows the configuration file specifications for the screen decomposition illustrated in Figure 4-2.

Example 4-1 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"
    }
}
```

A 2D compound has no frame latency, unless the `mode` flag `ASYNC` has been set, in which case the latency is one frame but you get better overall performance.

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 4-3 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.

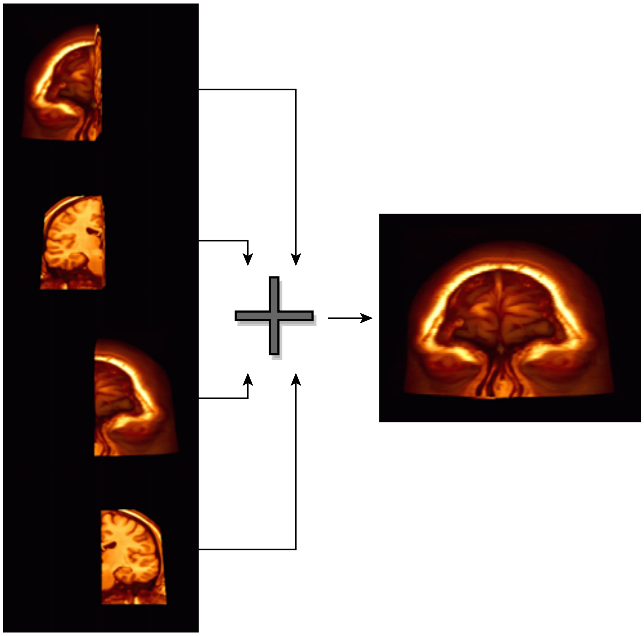


Figure 4-3 Database Decomposition

Example 4-2 shows the configuration file specifications for the database decomposition illustrated in Figure 4-3.

Example 4-2 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”
    }
}
```

The application must support the DB 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 4-4, 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.

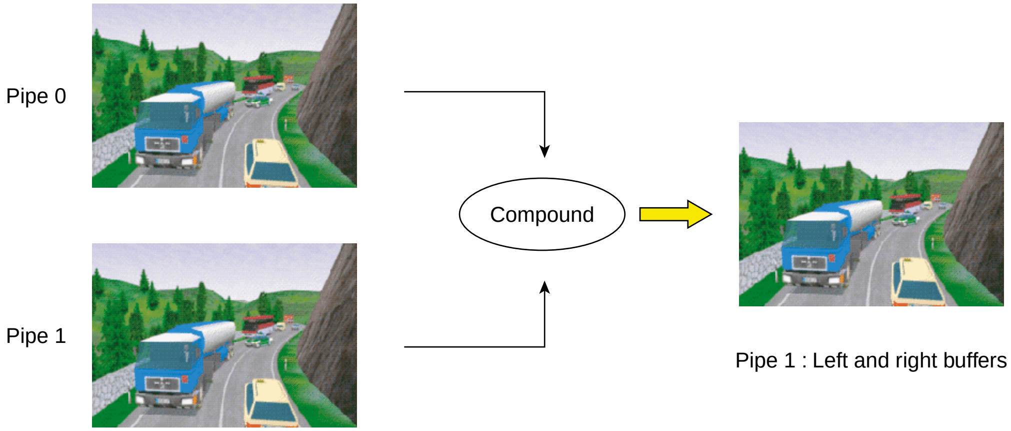


Figure 4-4 Eye Decomposition

Example 4-3 shows the configuration file specifications for the eye decomposition illustrated in Figure 4-4.

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

Example 4-4 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.

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 4-5 illustrates frame multiplexing on a four-pipe system.

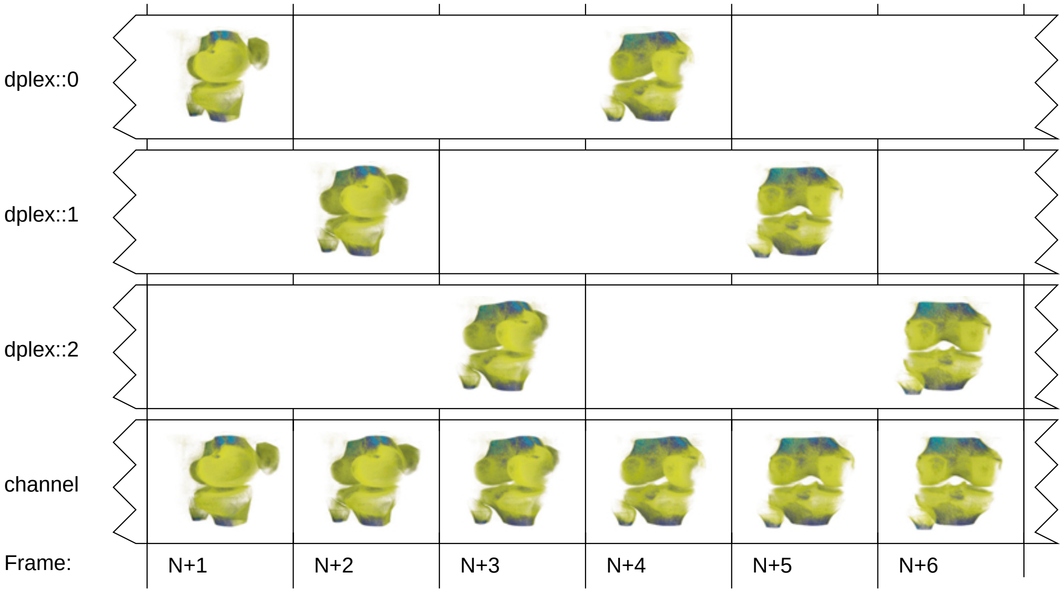


Figure 4-5 Frame Multiplexing Decomposition

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 Chapter 5, "Using Scalable Graphics Hardware".

Example 4-5 shows the configuration file specifications for the screen decomposition illustrated in Figure 4-5. The application must support the DPLEX compound.

Example 4-5 DPLEX Compound in a Configuration File

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

    region {
        channel “dplex::0”
    }

    region {
        channel “dplex::1”
    }

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

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 and the application must support this feature. Example 4-6 shows a configuration file structured for full scalability using the DPLEX compound.

Example 4-6 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 4-6. 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 4-6, 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.

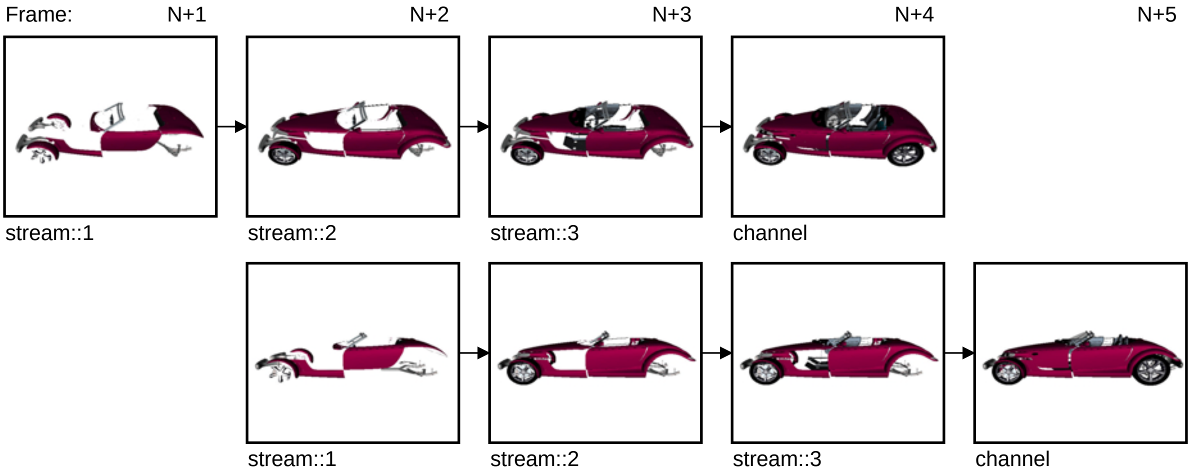


Figure 4-6 Data Streaming Decomposition

As shown in Example 4-7, the configuration file specification for a data streaming decomposition is similar to that for database decomposition.

Example 4-7 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”
    }
}
```

The application must support the 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 4-8 shows an FSAA compound using the SGI Scalable Graphics Compositor:

Example 4-8 Four-Pipe 4x FSAA Compound Using the SGI 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 4-7 illustrates the advantage of using a 4x FSAA solution.

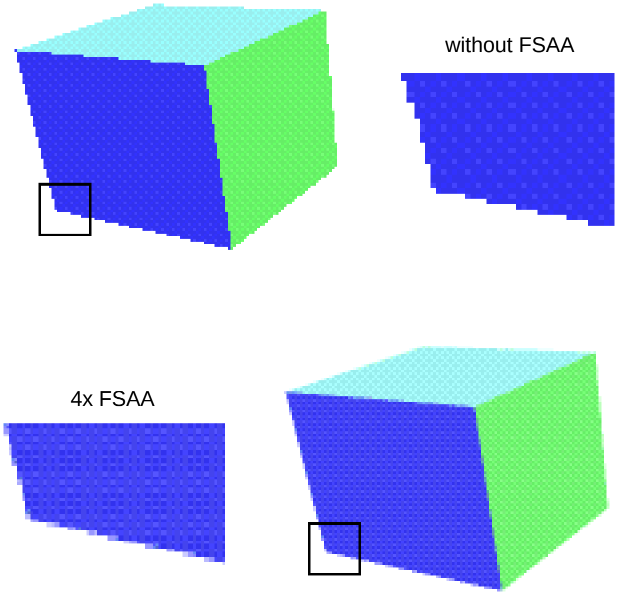


Figure 4-7 4x FSAA Decomposition

Example 4-9 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 4-9 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 4-8 shows a four-pipe solution using an eye and database decomposition scheme.

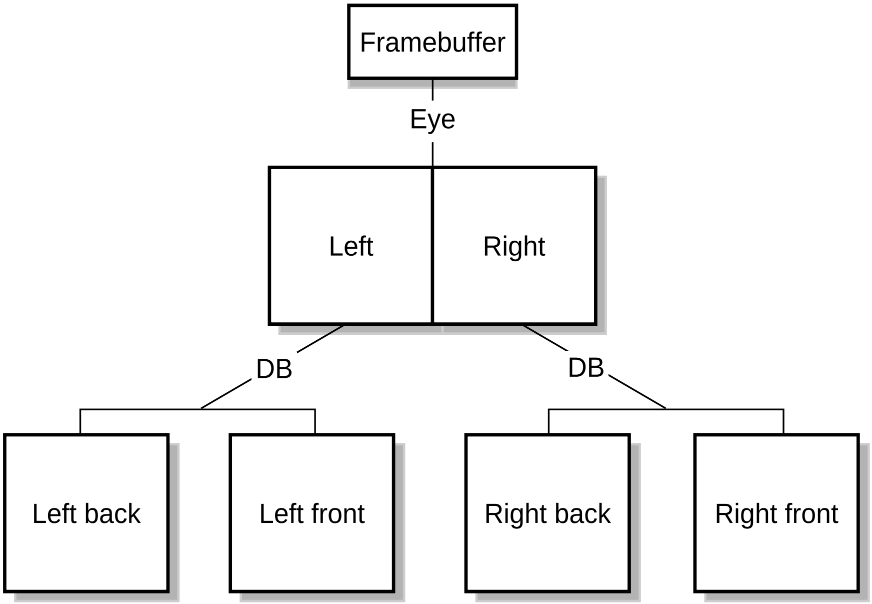


Figure 4-8 Eye-DB Multilevel Decomposition

Example 4-10 shows the configuration file specifications for the multilevel decomposition illustrated in Figure 4-8.

Example 4-10 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 4-11 illustrates this conditional use of compounds.

Example 4-11 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 4-9 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.

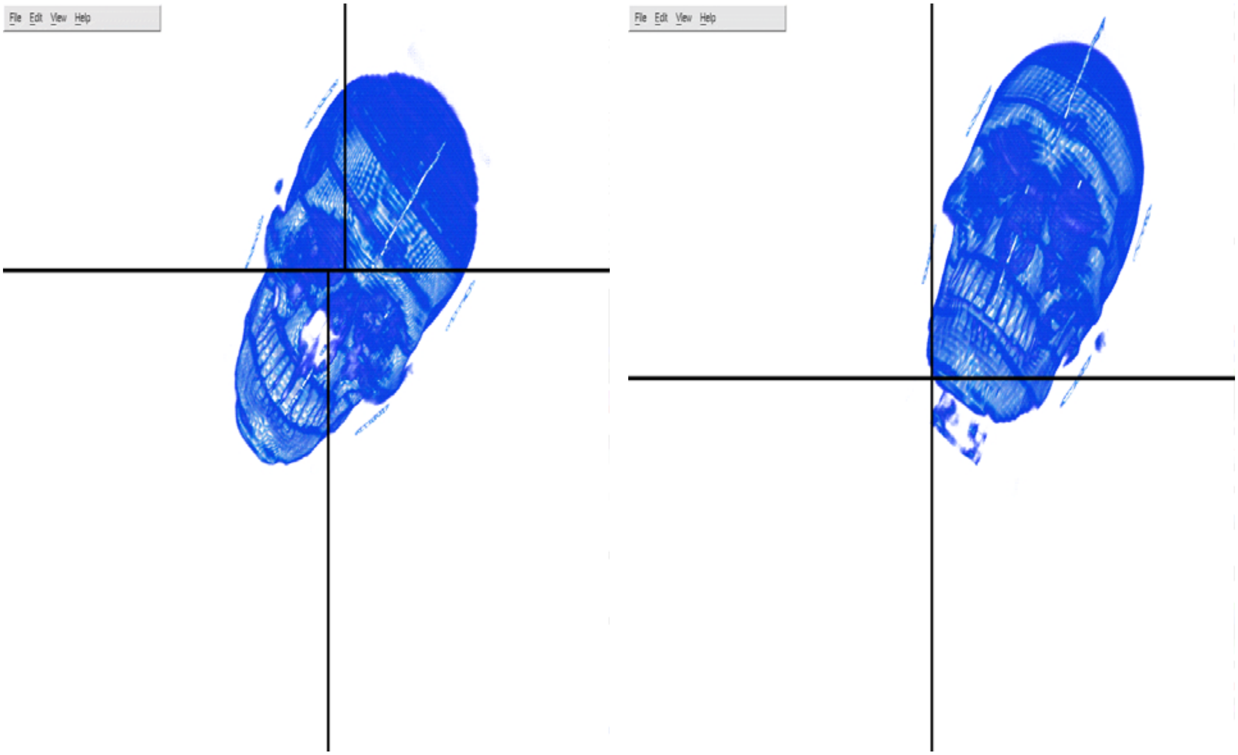


Figure 4-9 Dynamic Versus Static Load Balancing

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 47, you can combine the various decomposition modes to fix different performance bottlenecks.

Using Scalable Graphics Hardware

In contrast to most of the compounds described in Chapter 4, “Compounds”, scalable graphics hardware offers a hardware solution to joining or cascading the video output of two or more graphics pipes and outputting them in a single video output. Scalable graphics hardware provides nearly perfect scaling of both geometry rate and fill rate on some applications.

This chapter describes how you use MPK in conjunction with a SGI Video Digital Multiplexer (DPLEX) and an SGI Scalable Graphics Compositor. The following topics are described:

- “Using MPK with a DPLEX”
- “Using MPK with an SGI Scalable Graphics Compositor”

Using MPK with a DPLEX

A DPLEX is an optional daughtercard that permits multiple graphics hardware pipelines to work simultaneously on a single visual application. DPLEX hardware is available on Silicon Graphics Onyx2, SGI Onyx 3000, and SGI Onyx 300 systems. This section describes how you create the DPLEX compound in MPK and shows a configuration file example. For an overview of the DPLEX hardware, see the document *Onyx2 DPLEX Option Hardware User’s Guide*.

To enable DPLEX decomposition, you must specify the `DPLEX` mode along with the `HW` flag in the configuration file. The destination channel’s pipe is used to control the hyperpipe. Naturally, this is the display pipe of the DPLEX cascade. The use of the `NOCOPY` flag is mandatory to suppress pixel transfer.

Example 5-1 shows a three-pipe DPLEX cascade with the pipe associated to channel `channel::1` being the display pipe. The order of the channels reflects the order of the pipes in the DPLEX cascade.

Example 5-1 A Typical DPLEX Compound

```
compound {
    mode [ DPLEX HW NOCOPY ]
    channel “channel::1”

    region {
        channel “channel::1”
    }

    region {
        channel “channel::2”
    }

    region {
        channel “channel::3”
    }
}
```

Using MPK with an SGI Scalable Graphics Compositor

This section gives a brief overview of the SGI Scalable Graphics Compositor and how to use it with MPK. For more information on the compositor, including the details of the hardware setup, refer to the document *SGI InfinitePerformance: Scalable Graphics Compositor User's Guide*.

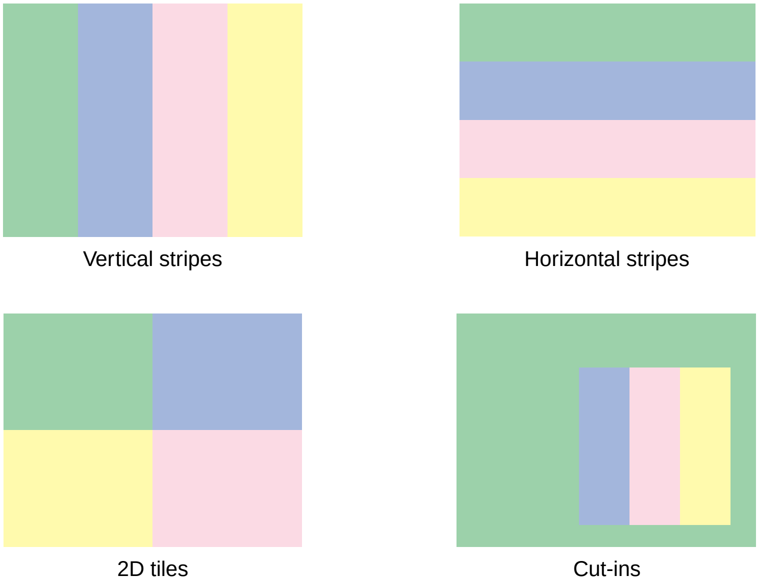
Note: The compositor is currently supported by InfinitePerformance graphics systems only.

How the Compositor Functions

The compositor receives two to four input signals and outputs a single signal either in analog or digital format. Hence, it can handle spatial composition of four inputs which enables multiple pipes to contribute to a single output. Four different composition schemes are available:

- Vertical stripes
- Horizontal stripes
- 2D tiles
- Cut-ins

The following figure illustrates the various hardware composition schemes.



Vertical stripes

Horizontal stripes

2D tiles

Cut-ins

Figure 5-1 Hardware Composition Schemes

The following items are noteworthy regarding the compositor's capabilities:

- For every output pixel, the compositor averages all values from all the pipes. Among other things, this provides applications with the means to do full-scene antialiasing (FSAA) in hardware.
- Stereo is supported only for analog output.
- Due to restrictions imposed by the SGI Graphics Compositor, MPK does not allow the mixing of the various hardware decomposition modes—for example, two vertical stripes with two horizontal stripes.

Note: For more information on the current limitations and anomalies associated with the use of the SGI Scalable Graphics Compositor, refer to the hardware documentation.

MPK Specifications

In order to use the compositor with MPK, you must specify the `2D`, `EYE`, or `FSAA` compound mode along with the `HW` flag. If you do not specify the `NOCOPY` flag, copying is performed even though the compositor is being used. Example 4-8 on page 45 shows a configuration file entry for an FSAA compound using the SGI Scalable Graphics Compositor.

Example 5-2 shows how a 2 x 2 tiling scheme might look in a configuration file.

Example 5-2 A 2 x 2 Tiling Scheme in a Configuration File

```
compound {
    mode    [ 2D HW NOCOPY ]
    channel “channel0”

    region {
        viewport    [ 0., 0.5, .5, .5 ]
        channel “channel0”
    }
    region {
        viewport    [ 0.5, 0.5, .5, .5 ]
        channel “channel1”
    }

    region {
        viewport    [ 0., 0., .5, .5 ]
        channel “channel2”
    }

    region {
        viewport    [ .5, 0., .5, .5 ]
        channel “channel3”
    }
}
```

Note the following:

- You must specify a destination channel if the compositor is to be used. Otherwise, MPK uses a software fallback solution.
- MPK does not require that the destination channel be used as a source channel—that is, it does not have to contribute to the rendering.

Configuration File Format

This chapter contains the following topics:

- "File Format"
- "Defining MPK Data Structures"
- "Specifying Global Attributes"

File Format

This section describes the format you must use to create an MPK configuration file. The format of the configuration file closely follows the conventions for the Open Inventor file format. The following items are described:

- "Specifying Comments (#)"
- "Specifying Delimiters"
- "Specifying MPK Data Structures"
- "Specifying Values within a Field"

Specifying Comments (#)

MPK considers any items between a number sign (#) and the end of the line to be a comment. The number sign can be anywhere on the line.

Specifying Delimiters

White space delimits most elements in the configuration file—for example, a field name and its value. Exceptions are noted where they apply. Extra white space created by spaces, tabs, and new lines is ignored.

Specifying MPK Data Structures

An MPK data structure consists of the following sequence of elements:

1. Data structure type (`config`, `pipe`, `window`, `channel`, or `compound`)
2. Open brace ({)
3. Field specifications (if any), followed by child structures (if any)
4. Close brace (})

The following lines show the syntax symbolically:

```
data-structure-type {
    field-specs
    child-specs
}
```

The following is an example of a `channel` data structure:

```
channel {
    viewport        [ 0., 0., 1., 1. ]
    projection {
        origin      [ 0., 0., 0. ]
        distance    3.
        fov         [ 54., 47. ]
        hpr         [ 0., 0., 0. ]
    }
}
```

The later section "Defining MPK Data Structures" on page 64 describes the special requirements for defining each of the data structures.

Specifying Values within a Field

There are three types of MPK fields:

- Single-value fields

  These fields have the following syntax:

  name *value*

- Multiple-value fields

  These fields have the following syntax:

  name [*value1*, *value2*, . . . *valuen*]

- Composite fields

  These fields have the following syntax:

  ```
  name {
          subfields-specs
  }
  ```

The fields can appear in any order.

The values you specify in a field are determined by the field type. The field types and accepted formats are described in Table 6-1.

Table 6-1 Field Types and Formats

Field Type	Accepted Format
`int`	Use an integer in decimal, hexadecimal, or octal format. Examples: 55 0xff 0177
`float`	Use an integer or floating point number. Examples: 10 10. 10.3 1.3e–2

Table 6-1 Field Types and Formats **(continued)**

Field Type	Accepted Format
`string`	Use double quotation marks (“ “) around the value. Example: `name` “3-pipes”

`enum`	Use a mnemonic. Examples:
	`true-color`
	`direct-color`

`boolean`	Use a mnenomic (`y` or `n`) or an integer (`0` or `1`).

Defining MPK Data Structures

The earlier section “Specifying MPK Data Structures” on page 62 describes the general format of an MPK data structure as follows:

```
data-structure-type {
      field-specs
      child-specs
}
```

This section describes the field specifications and child specifications required to define the following MPK data structures:

- `config`
- `pipe`
- `window`
- `channel`
- `compound`

Additionally, you can include a `global` data structure to define defaults for global attributes. The later section “Specifying Global Attributes” on page 80 describes how you do so.

The `config` Data Structure

The `config` data structure encapsulates the other data structures and as such defines the overall configuration. It has the following form:

```
config {
    name      "config-name"
    mode      stereo-mono
    mono      "shell command1"
    stereo    "shell command2"
    runon      n
    pipe-1-specs
    pipe-2-specs
         .
         .
         .
    pipe-n-specs
    compound-specs
}
```

Every `config` data structure requires a `pipe` data entry for each pipe you want to use in your system. Section "The pipe Data Structure" on page 66 describes the *pipe-i-specs* fields, section"The compound Data Structure" on page 77 describes the *compound-specs* field, and Table 6-2 describes the other `config` fields.

Table 6-2 The `config` Fields

Field	Description
`name`	The `name` field is a string identifier for the current configuration.
`mode`	The `mode` field characterizes the initial configuration state: either `mono` or `stereo`.
`mono` `stereo`	These fields describe the shell command to execute when changing to mono or stereo mode. By default, no command is executed.
`runon`	The `runon` field contains the default processor ID for all configuration threads—that is, the processor to which every window thread will be assigned, unless specified otherwise by the window `runon` field. A negative `runon` value frees the thread to run on whatever processor the system deems suitable. The default is –1.

The pipe Data Structure

A pipe data structure has the following form:

```
pipe {
    name        “pipe-name”
    display      ”display-name”
    attributes {
        mono {
            width     w
            height    h
        }
        stereo {
            type      stereo-type
            width     w
            height    h
            offset    o
        }
    }
    window-specs
}
```

Every pipe data structure must contain a window entry. The section “The window Data Structure” on page 68 describes the *window-specs* field. Table 6-3 describes the other fields of an pipe data structure.

Table 6-3 The `pipe` Fields

Field	Description
`name`	The `name` field is a string identifier for the current pipe.
`display`	The `display` field specifies the name of the X display for the current pipe.

`attributes`	The `attributes` field is a composite field with the following subfields:
	`mono`
	`stereo`
	These subfields are in turn composite fields. The subfield `mono` has the following fields:
	`width` *w*
	`height` *h*
	The values for *w* and *h* must be integers. The subfield `stereo` has the following fields:
	`typ`e *stereo-type*
	`width` *w*
	`height` *h*
	`offset` *o*
	The values for *w*, *o*, and *h* must be integers. The value for *stereo-type* can be one of the following:
	`quad`
	`rect`
	`top`
	`bottom`
	`user`
	If no stereo type is specified, `quad` is used.

Example 6-1 is an example of a `pipe` definition:

Example 6-1 A Sample `pipe` Definition

```
pipe {
    display     “:0.0”
    window {
        runon       2
        viewport    [ 0., 0., 1., 1. ]
        channel {
            viewport        [ 0., 0., 1., 1. ]
            projection {
                origin      [ 0., 0., 0. ]
                distance    3.
                fov         [ 54., 47. ]
                hpr         [ 0., 0., 0. ]
            }
        }
    }
}
```

The `window` Data Structure

A `window` data structure has the following form:

```
window {
    name        “win-name”
    viewport    [x, y, width, height]
    runon        n
    attributes   attribute-specs
    channel-specs
}
```

Every window data structure requires a channel entry. The section "The channel Data Structure" on page 74 describes the *channel-specs* field. Table 6-4 describes the other fields of a window data structure.

Table 6-4 The window Fields

Field	Description
name	The name field is a string identifier for the current window.
viewport	The viewport field specifies the fractional viewport (position and size) of the current window relative to the display dimensions. The fractional viewport format is [*x, y, width, height*] with all parameters in the range 0.0 to 1.0. See the section "Specifying Global Attributes" on page 80 for more information on the following related global variables: MPK_PATTR_MONO_HEIGHT MPK_PATTR_MONO_WIDTH MPK_PATTR_STEREO_HEIGHT MPK_PATTR_STEREO_OFFSET MPK_PATTR_STEREO_TYPE MPK_PATTR_STEREO_WIDTH
runon	The runon field contains the default processor ID for the current window thread—that is, the processor to which the window thread will be assigned. A negative runon value frees the thread to run on whatever processor the system deems suitable. If this field is not specified, it is inherited from the runon field of the config data structure. The default is –1.
attributes	The attributes field specifies the X Window System default visual attributes and other related information, such as whether window managers decorations should be present or not. The attributes field is a composite field with the following subfields: hints planes transparent These subfields are in turn composite fields. which are described in the tables that follow.

Table 6-5 describes the structure and values of the hints subfields.

Table 6-5 Window Attributes— `hints` Subfields

Subfield	Valid Values	Description

`visual`	`true-color`
	`pseudo-color`
	`direct-color`
	`static-color`
	`grayscale`
	`static-gray`
		Specifies the type of GLX visual to be used.
		Related global variable:
		`MPK_WATTR_HINTS_VISUAL`

`caveat`	`none`
	`slow`
	`non-conformant`
		Specifies a caveat for selecting the framebuffer, such as one of the following:
		`MPK_GLX_SLOW`
		`MPK_GLX_NON_CONFORMANT`
		`MPK_GLX_NOCAVEAT`
		Related global variable:
		`MPK_WATTR_HINTS_CAVEAT`

`transparent`	`y or n`	Determines if a visual should be opaque or transparent. Related global variable: `MPK_WATTR_HINTS_TRANSPARENT`
`X-renderable`	`y or n`	Determines which visuals are selected. If true, only visuals which have an associated X visual are selected. Related global variable: `MPK_WATTR_HINTS_X_RENDERABLE`
`rgba`	`y or n`	Specifies if an RGBA or color-index visual is selected`.` Related global variable: `MPK_WATTR_HINTS_RGBA`
`doublebuffer`	`y or n`	Specifies if a double- or single-buffer visual is selected. Related global variable: `MPK_WATTR_HINTS_DOUBLEBUFFER`

Table 6-5 Window Attributes— `hints` Subfields **(continued)**

Subfield	Valid Values	Description
`stereo`	`y or n`	Specifies if a stereo-capable visual is selected Related global variable: `MPK_WATTR_HINTS_STEREO`

`drawable`	`window`
	`pbuffer`
	`pixmap`
		Specifies the type of drawable to be used for rendering.
		Related global variable:
		`MPK_WATTR_HINTS_DRAWABLE`

`direct`	`y or n`	Specifies if a direct or indirect context is created. Related global variable: `MPK_WATTR_HINTS_DIRECT`
`largest`	`y or n`	Determines whether the largest available pbuffer is allocated. It will be if a pbuffer drawable is used and this flag is set. Related global variable: `MPK_WATTR_HINTS_LARGEST`
`preserved`	`y or n`	Determines whether the content of the framebuffer is preserved. It will be if a pbuffer drawable is used and this flag is set. Related global variable: `MPK_WATTR_HINTS_PRESERVED`
`decoration`	`y or n`	Determines if the window should have window manager decorations. Related global variable: `MPK_WATTR_HINTS_DECORATION`

`xinerama`	`y or n`	Indicates that this window are created using Xinerama if `xinerama` is `y`. If it is set to `n`, the window is created Xinerama-aware.
		Related global variables:
		`MPK_WATTR_HINTS_XINERAMA`
		`MPK_XINERAMA`

Table 6-6 describes the structure and values of the `planes` subfields. In all instances in this table, the variables denote integers.

Table 6-6 Window Attributes—`planes` Subfields

Subfield	Valid Values	Description
`level`	*x*	Specifies the buffer level. Positive values correspond to overlay buffers and negative values correspond to underlay buffers. Related global variable: `MPK_WATTR_PLANES_LEVEL`
`depth`	*x*	Specifies the minimum depth buffer size. Related global variable: `MPK_WATTR_PLANES_DEPTH`
`stencil`	*x*	Specifies the minimum stencil buffer size`.` Related global variable: MPK_WATTR_PLANES_STENCIL
`samples`	*x*	Specifies the minimum number of multi-sample buffers. Related global variable: `MPK_WATTR_PLANES_SAMPLES`
`auxiliary`	*x*	Specifies the minimum number of auxiliary buffers. Related global variable: `MPK_WATTR_PLANES_AUX`
`color`	*x*	Specifies the minimum color-index buffer size. Related global variable: `MPK_WATTR_PLANES_COLOR`

Table 6-6 Window Attributes—`planes` Subfields **(continued)**

Subfield	Valid Values	Description

`rgba`	*[r, g, b, a]*	Specifies the minimum RGBA buffer size.
		Related global variables:
		`MPK_WATTR_PLANES_RED`
		`MPK_WATTR_PLANES_GREEN`
		`MPK_WATTR_PLANES_BLUE`
		`MPK_WATTR_PLANES_ALPHA`

`accum`	*[r, g, b, a]*	Specifies the minimum RGBA accumulation buffer size.
		Related global variables:
		`MPK_WATTR_PLANES_ACCUM_RED`
		`MPK_WATTR_PLANES_ACCUM_GREEN`
		`MPK_WATTR_PLANES_ACCUM_BLUE`
		`MPK_WATTR_PLANES_ACCUM_ALPHA`

Table 6-7 describes the structure and values of the `transparent` subfields. In all instances in this table, the variables denote integers.

Table 6-7 Window Attributes—`transparent` Subfields

Subfield	Valid Values	Description
`index`	*x*	Specifies the index value for the transparent *color*. Related global variables: `MPK_WATTR_TRANSPARENT_INDEX`

`rgba`	*[r, g, b, a]*	Specifies the RGBA value for the transparent *color*.
		Related global variables:
		`MPK_WATTR_TRANSPARENT_RED`
		`MPK_WATTR_TRANSPARENT_GREEN`
		`MPK_WATTR_TRANSPARENT_BLUE`
		`MPK_WATTR_TRANSPARENT_ALPHA`

The `channel` Data Structure

A `channel` data structure has the following form:

```
channel {
    name         "channel-name"
    viewport    [x,y, width, height]
    ortho-wall   ortho-wall-specs
    wall         wall-specs
    projection   projection-specs

}
```

The `channel` data structure is the lowest-level data structure—that is, it has no child data structures. Table 6-8 describes the fields of a `channel` data structure.

Table 6-8 The `channel` Fields

Field	Description
`name`	The `name` field is a string identifier for the current channel. You must specify the `name` field if your configuration file contains a compound that references this channel.
`viewport`	The `viewport` field specifies the fractional viewport (position and size) of the channel relative to the parent window dimensions. The fractional viewport format is [*x*, *y*, *width*, *height*] with all parameters in the range `0.0` to `1.0`.

Table 6-8 The `channel` Fields **(continued)**

Field	Description
`wall`	The `wall` field contains the modeling coordinates of the bottom-left, bottom-right, and top-left corners of the channel's projection rectangle in the real world. This field is a composite field with the following subfields: `bottom_left` [*x*, *y*, *z*] `bottom_right` [*x*, *y*, *z*] `top_left` [*x*, *y*, *z*] See Example 3-1 on page 23 for an example of specifying the `wall` field. You must specify one of the modeling coordinates fields: `wall`, `projection`, or `ortho-wall`. MPK uses the last specified modeling transformation—that is, either `wall` or `projection`—unless you set the channel to orthographic projection by specifying the `ortho-wall` field.

Table 6-8 The `channel` Fields **(continued)**

Field	Description
`ortho-wall`	The `ortho-wall` field contains an alternate wall description that, if specified, will be used when the channel orthographic frustum is applied. The format for the field values is the same as that of the `wall` field. You must specify one of the modeling coordinates fields: `wall`, `projection`, or `ortho-wall`. MPK uses the last specified modeling transformation—that is, either `wall` or `projection`—unless you set the channel to orthographic projection by specifying the `ortho-wall` field.
`projection`	The `projection` field contains the modeling coordinates and characteristics of an imaginary projection system that would produce the channel's projection rectangle. This is a composite field with the following subfields: `origin` [*x*, *y*, *z*] `distance` *d* `fov` [*a*, *b*] `hpr` [*h*, *p*, *r*] The `hpr` field represents the head, pitch, and roll and describes Euler angles with respect to the OpenGL convention—that is, the counter-clockwise rotation around the Y axis (head), X axis (pitch), and Z axis (roll) viewed from the positive side of the axis. See Example 3-2 on page 25 for an example of specifying the `projection` field. You must specify one of the modeling coordinates fields: `wall`, `projection`, or `ortho-wall`. MPK uses the last specified modeling transformation—that is, either `wall` or `projection`—unless you set the channel to orthographic projection by specifying the `ortho-wall` field.

The compound Data Structure

A compound data structure is not a part of the pipe-window-channel hierarchy. The compound data structure is subordinate only to the config or another compound data structure.

A compound data structure has the following form:

```
compound {
    name        "compound-name"
    channel     "channel-name"
    mode        [ mode flags ]
    format      [ format1 format2 ... formatn ]
    region      region-specs
}
```

Table 6-9 describes the fields of a compound data structure.

Table 6-9 The compound Fields

Field	Description
name	The name field is a string identifier for the current compound.
channel	The channel field identifies the destination channel for the compound. You must specify the name as defined in the name field of the associated channel data structure. If you do not specify this field and the compound has a parent, then its value is inherited from the parent. If the resulting channel is still unspecified, then the compound will simply maintain time consistency of the views across all of its regions with respect to their respective frame latency. Once a channel is involved in a compound, you must explicitly specify any other use of that channel with another compound. This is true also for mode-selective compounds.

Table 6-9 The compound Fields **(continued)**

Field	Description
mode	The mode field specifies the decomposition mode (2D, 3D, DB, DPLEX, EYE, FSAA, or HMD) and optionally mode flags (ADAPTIVE, ADAPTIVE_H, ADAPTIVE_V, ASYNC, HW, MONO, NOCOPY, or STEREO). The following are examples: mode [3D] mode [DB MONO] mode [EYE STEREO ASYNC] mode [2D ASYNC] mode [DPLEX NOCOPY] For descriptions of the decomposition modes, see section “Building Compounds” in Chapter 4. For the use of 2D, FSAA, HW , DPLEX, and NOCOPY in scalable hardware solutions, see Chapter 5, “Using Scalable Graphics Hardware”. For the use of ADAPTIVE, ADAPTIVE_H, and ADAPTIVE_V, see section “Automatic Load Balancing for Compounds” in Chapter 4. ASYNC indicates that the pixel transfer from the regions to the destination channel should be delayed to the next frame. Despite an additional one-frame latency, this setting may have a noticeable influence on the compound performance—especially for 2D and DB decompositions. MONO or STEREO indicates that the decomposition should only be activated when the configuration is in the corresponding stereo mode. Note that a window containing only STEREO-active channels will simply not be launched when the configuration is in MONO mode. NOCOPY indicates that no pixel transfer should occur between the compound and its regions; this is typically useful when you use hardware video compositing equipment—for example, SGI Video Digital Multiplexer (DPLEX).

Table 6-9 The `compound` Fields **(continued)**

Field	**Description**

`format`	The `format` field specifies the format of the pixel data that has to be transferred between the compound and its regions as a combination of `RGBA`, `DEPTH`, and `STENCIL`. It will be inherited by the compound regions. The following are the possible combinations:
	`format [ RGBA ]` # default
	`format [ RGBA DEPTH ]`
	`format [ RGBA DEPTH STENCIL ]`

`region`	The `region` field specifies a portion of the compound destination channel and the channel where this portion should be rendered. Depending on the compound `mode` field, the portion described can be either a sub-viewport of the destination channel [`2D`], a portion of the application database [`DB` or `3D`], a specific eye view [`EYE` or `HMD`], or a pipelined, de-multiplexed rendering cycle [`DPLEX`]. The format for each follows:
	`2D`:
	viewport [*x*, *y* ,*width*, *height*]
	channel "*channel-name*"
	`DB` or `3D`:
	range [*a* ,*b*]
	channel "*channel-name*"
	`EYE`
	`eye` *left-right*
	`channel` "*channel-name*"
	`HMD`
	`eye` *left-right*
	`channel` "*channel-name*"
	`DPLEX`
	`channel` "*channel-name*"

See Chapter 4, "Compounds" for examples of `compound` definitions.

Specifying Global Attributes

A `global` data structure allows you to specify default values for MPK attributes:

- Stereo and pipe display attributes
- Window attributes
- Channel attributes

To specify a default value for an attribute in the configuration file, use the following construct:

```
global {
    attribute1   value
    attribute2   value
        .
        .
        .
    attributen   value
}
```

Your default declarations should precede the definition of the `config` data structure in the configuration file. The following is an example of default declarations:

```
global {
    MPK_DEFAULT_EYE_OFFSET  .035
    MPK_WATTR_PLANES_ALPHA  1
}
```

Table 6-10 provides the data type, default value, and description for the MPK global attributes.

Table 6-10 MPK Global Attributes

Variable	Data Type	Default Value	Description
`MPK_CATTR_NEAR`	`float`	0.01	Specifies the default near distance of the channel. This value is preempted by the function **mpkChannelSetNearFar()**.
`MPK_CATTR_FAR`	`float`	100.	Specifies the default far distance of the channel. This value is preempted by the function **mpkChannelSetNearFar()**.

Table 6-10 MPK Global Attributes **(continued)**

Variable	Data Type	Default Value	Description
MPK_DATTR_FULLSTEREO_HEIGHT	int		Deprecated. Use MPK_PATTR_STEREO_HEIGHT and MPK_PATTR_STEREO_TYPE. Specifies the height of the display to be used by the function **mpkWindowUpdatePixelViewport()** for full-stereo mode instead of that returned by the X11 **DisplayHeight()** function.
MPK_DATTR_FULLSTEREO_WIDTH	int		Deprecated. Use MPK_PATTR_STEREO_WIDTH and MPK_PATTR_STEREO_TYPE. Specifies the width of the display to be used by the function **mpkWindowUpdatePixelViewport()** for full-stereo mode instead of that returned by the X11 **DisplayWidth()** function.
MPK_DATTR_FULLSTEREO_OFFSET	int	532	Deprecated. Use MPK_PATTR_STEREO_OFFSET and MPK_PATTR_STEREO_TYPE. Specifies the offset of the display to be used by the function **mpkWindowUpdatePixelViewport()** for full-stereo mode instead of that returned by the X11 **DisplayHeight()** and **DisplayWidth()** functions.
MPK_DATTR_MONO_HEIGHT	int		Deprecated. Use MPK_PATTR_MONO_HEIGHT . 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.

Table 6-10 MPK Global Attributes **(continued)**

Variable	Data Type	Default Value	Description
MPK_DATTR_MONO_WIDTH	int		Deprecated. Use MPK_PATTR_MONO_WIDTH . 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_DATTR_QUADSTEREO_HEIGHT	int		Deprecated. Use MPK_PATTR_STEREO_HEIGHT and MPK_PATTR_STEREO_TYPE. Specifies the height of the display to be used by **mpkWindowUpdatePixelViewport()** for quad-stereo mode instead of that returned by the X11 **DisplayHeight()** function.
MPK_DATTR_QUADSTEREO_WIDTH	int		Deprecated. Use MPK_PATTR_STEREO_WIDTH and MPK_PATTR_STEREO_TYPE. Specifies the width of the display to be used by the function **mpkWindowUpdatePixelViewport()** for quad-stereo mode instead of that returned by the X11 **DisplayWidth()** function.
MPK_DEFAULT_EYE_OFFSET	float	0.035	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_PATTR_MONO_HEIGHT	int	492 is used for MPK_STEREO_REC, MPK_STEREO_BOT, and MPK_STEREO_TOP.	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.

Table 6-10 MPK Global Attributes **(continued)**

Variable	Data Type	Default Value	Description
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	492 for full	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	532	Specifies the offset of the display to be used by the function **mpkWindowUpdatePixelViewport()** for rect and bottom stereo modes.
MPK_PATTR_STEREO_TYPE	enum	none	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.
MPK_WATTR_HINTS_CAVEAT	enum	MPK_UNDEFINED	Specifies the caveats associated with the window framebuffer configuration. Accepted values are MPK_GLX_SLOW, MPK_GLX_NOCAVEAT, and MPK_GLX_NON_CONFORMANT.
MPK_WATTR_HINTS_DECORATION	boolean	MPK_UNDEFINED	Specifies whether the window should have window manager decorations.
MPK_WATTR_HINTS_DIRECT	boolean	MPK_UNDEFINED	Specifies whether the window GLX context should be direct.

Table 6-10 MPK Global Attributes **(continued)**

Variable	Data Type	Default Value	Description
`MPK_WATTR_HINTS_DOUBLEBUFFER`	`boolean`	`1`	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()**.
`MPK_WATTR_HINTS_DRAWABLE`	`enum`	`MPK_UNDEFINED`	Specifies the window drawable type. Accepted values are `MPK_GLX_WINDOW`, `MPK_GLX_PBUFFER`, and `MPK_GLX_PIXMAP`.
`MPK_WATTR_HINTS_LARGEST`	`boolean`	`MPK_UNDEFINED`	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`	`MPK_UNDEFINED`	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`	`1`	Specifies whether RGBA visuals are used. If the hint is not set, a color-index visual is used.
`MPK_WATTR_HINTS_STEREO`	`boolean`	`MPK_UNDEFINED`	Specifies whether the window framebuffer configuration should support quad-buffer stereo.
`MPK_WATTR_HINTS_THREAD`	`boolean`	`MPK_UNDEFINED`	Specifies whether the window should be made a separate thread from the application.
`MPK_WATTR_HINTS_TRANSPARENT`	`boolean`	`MPK_UNDEFINED`	Specifies whether the window framebuffer configuration should be transparent.

Table 6-10 MPK Global Attributes **(continued)**

Variable	Data Type	Default Value	Description
MPK_WATTR_HINTS_VISUAL	enum	MPK_UNDEFINED	Specifies the window visual type. Accepted values are MPK_GLX_TRUE_COLOR, MPK_GLX_PSEUDO_COLOR, MPK_GLX_DIRECT_COLOR, MPK_GLX_STATIC_COLOR, MPK_GLX_GRAYSCALE, and MPK_GLX_STATIC_GRAY.
MPK_WATTR_HINTS_X_RENDERABLE	boolean	MPK_UNDEFINED	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	Conditional. See the description.	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	MPK_UNDEFINED	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	MPK_UNDEFINED	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	MPK_UNDEFINED	Specifies the minimum number of accumulation green bitplanes. This attribute is ignored if the RGBA hint of the window is not set.

Table 6-10 MPK Global Attributes **(continued)**

Variable	Data Type	Default Value	Description
MPK_WATTR_PLANES_ACCUM_RED	int	MPK_UNDEFINED	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	0	Specifies the minimum number of alpha bitplanes. This attribute is ignored if the RGBA hint of the window is not set.
MPK_WATTR_PLANES_AUX	int	MPK_UNDEFINED	Specifies the number of auxiliary buffers.
MPK_WATTR_PLANES_BLUE	int	1	Specifies the minimum number of blue bitplanes. This attribute is ignored if the RGBA hint of the window is not set.
MPK_WATTR_PLANES_COLOR	int	MPK_UNDEFINED	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	1	Specifies the minimum size of the depth buffer.
MPK_WATTR_PLANES_GREEN	int	1	Specifies the minimum number of green bitplanes. This attribute is ignored if the RGBA hint of the window is not set.
MPK_WATTR_PLANES_LEVEL	int	0	Specifies the window buffer level.
MPK_WATTR_PLANES_RED	int	1	Specifies the minimum number of red bitplanes. This attribute is ignored if the RGBA hint of the window is not set.
MPK_WATTR_PLANES_SAMPLES	int	MPK_UNDEFINED	Specifies the minimum number of samples required in the multi-sample buffer.
MPK_WATTR_PLANES_STENCIL	int	MPK_UNDEFINED	Specifies the minimum size of the stencil buffer.

Table 6-10 MPK Global Attributes **(continued)**

Variable	Data Type	Default Value	Description
MPK_WATTR_TRANSPARENT_ALPHA	int	MPK_UNDEFINED	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_BLUE	int	MPK_UNDEFINED	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_GREEN	int	MPK_UNDEFINED	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_INDEX	int	MPK_UNDEFINED	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	MPK_UNDEFINED	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.
MPK_XINERAMA	boolean	1	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.

You can find more information about the window attributes specifications in the `glXChooseFBConfigSGIX`(3G) and `glXChooseVisual`(3G) man pages.

Index