# OpenGL Multipipe™ SDK White Paper

## Introduction

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. As these types of evolving environments have numerous parameters, the applications must be sufficiently flexible and robust to accommodate their demands.

OpenGL Multipipe SDK 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. OpenGL Multipipe SDK 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

OpenGL Multipipe SDK 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 OpenGL Multipipe SDK 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. OpenGL Multipipe SDK 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 OpenGL Multipipe SDK now provides in software. OpenGL Multipipe SDK is structured to support such hardware as well as conventional graphics hardware.

## Integrated Support for Stereo and Immersive Environments

Along with its scalability features, OpenGL Multipipe SDK 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, OpenGL Multipipe SDK 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. OpenGL Multipipe SDK 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 OpenGL Multipipe SDK ideal for use in immersive environments.

## Application Structure

As an application will have to run in different configurations, OpenGL Multipipe SDK 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 illustrates the structure of an application based on OpenGL Multipipe SDK.

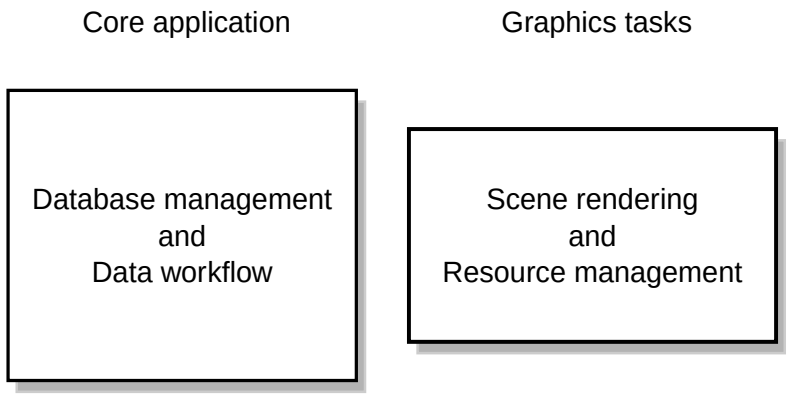


**Figure 1** OpenGL Multipipe SDK Application Structure

OpenGL Multipipe SDK 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™.

## Graphics Environment Description

The OpenGL Multipipe SDK configuration file uses a tree structure to describe the physical graphics resources. The root of the structure is the whole visualization facility and the leaves are the physical rendering layouts. Reading this configuration file, OpenGL Multipipe SDK determines the following:

- What physical pipes it must allocate
- What parallel tasks it must create
- How to synchronize the tasks until the per-view frustum is applied
- The final rendering framebuffer area

A typical structure, defined as a `config`, is based on following components:

| Component | Description |
|---|---|
| Config | The whole graphics environment where the application runs. One configuration can be described per file. |
| Pipe | The physical rendering engine. It is defined by the X™ display-screen mapping of the graphics hardware in the X11 abstraction scheme. A config can handle several pipes. |
| Window | A single GLX unit. It retains the X window attributes and GLX visuals and handles associated graphics context. Thus, it harnesses the base structure where a task (or thread) will be created for parallel rendering. |
| Channel | A view definition of the scene. It defines a part of the window where the scene will be drawn. It specifies how the scene will be projected and displayed. Each channel specification determines the projection parameters or the physical surface (wall) where the view will be displayed. Channels defined in the same window render in the same graphics context. |
| Projection | A physical projection definition. It defines the projection parameters (field of view, heading, and position) in the physical display environment. |
| (Ortho)Wall | A physical display surface definition. It defines the rectangular area issued from the projection. |
| Compound | A decomposition definition. It defines the type of decomposition, source channels, and destination channel. Compounds can be combined as a graph of channel parts. |

A `config` structure can be graphically represented as in Figure 2.

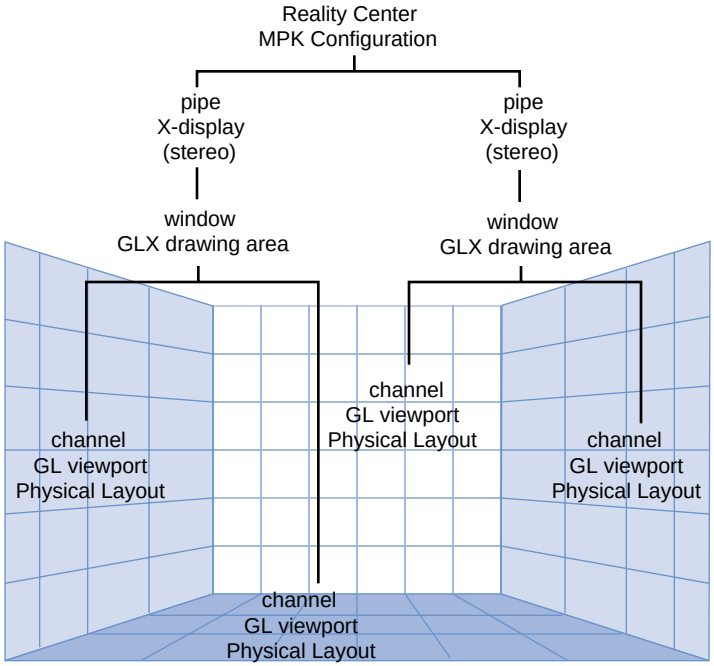


**Figure 2** Configuration File Mapping onto a Reality Center Environment

Figure 2 shows the configuration of an application running on a two-pipe platform, two windows handling the GLX context, and four channels. Example 1 shows a skeletal configuration file that describes the environment.

**Example 1** Skeletal Configuration File

```
config {
    pipe {
        window {
            viewport [ parameters1 ]
            channel {
                viewport [ parameters2 ]
                  .
                  .
                  .
             }
        }
    }
    pipe {
        window {
            viewport [ parameters1 ]
            channel {
                viewport [ parameters2 ]
                  .
                  .
                  .
             }
        }
    }
}
```

## Displaying Multiple and Wider Fields of View

There are various schemes to address multiple views. Two such schemes are the following:

- Multiple views per pipe
- Single view per pipe

## Multiple Views Per Pipe

The multiple-views-per-pipe scheme dedicates one channel per view, each defined by its respective frustum, heading, position, and drawing area. Each channel shares the same graphics context in a window. Each view is drawn in a different area of the same framebuffer on the same pipe and owns the same GLX visual. The channels of the same window are drawn sequentially. Often, the framebuffer region corresponding to the view is output as a video signal and projected as well. Figure 3 illustrates this scheme.

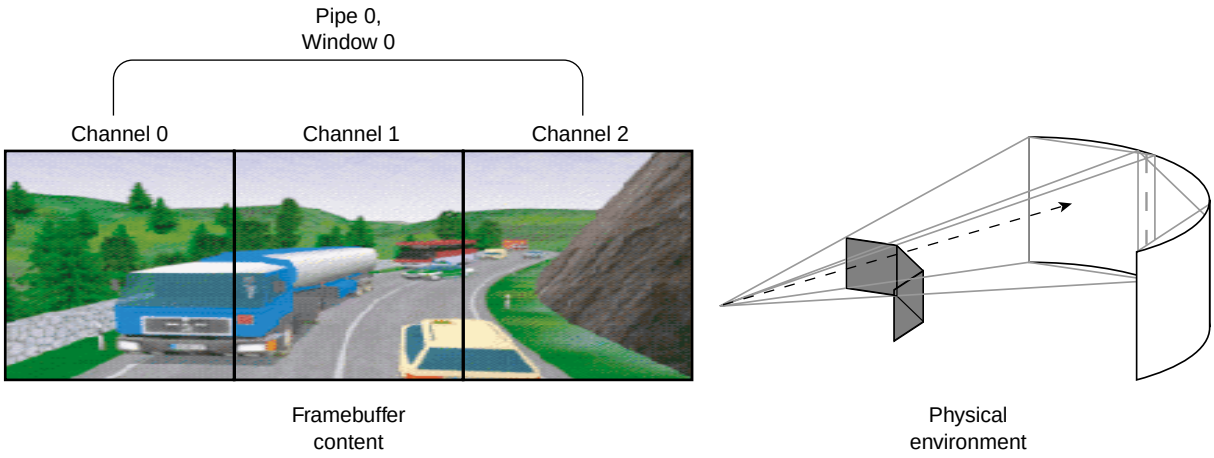


**Figure 3** Multiple Views Per Pipe

## Single View Per Pipe

The single-view-per-pipe scheme dedicates one pipe per view. In this case, a unique pipe-window-channel link is created for each view and the views are rendered in parallel on the pipes. This adds graphics power to the application as the graphics load on each pipe will be lower. At the same time, the rendering task needs to be distributed over the pipes accordingly and the resulting draw processes need to be synchronized. Figure 4 illustrates this scheme.

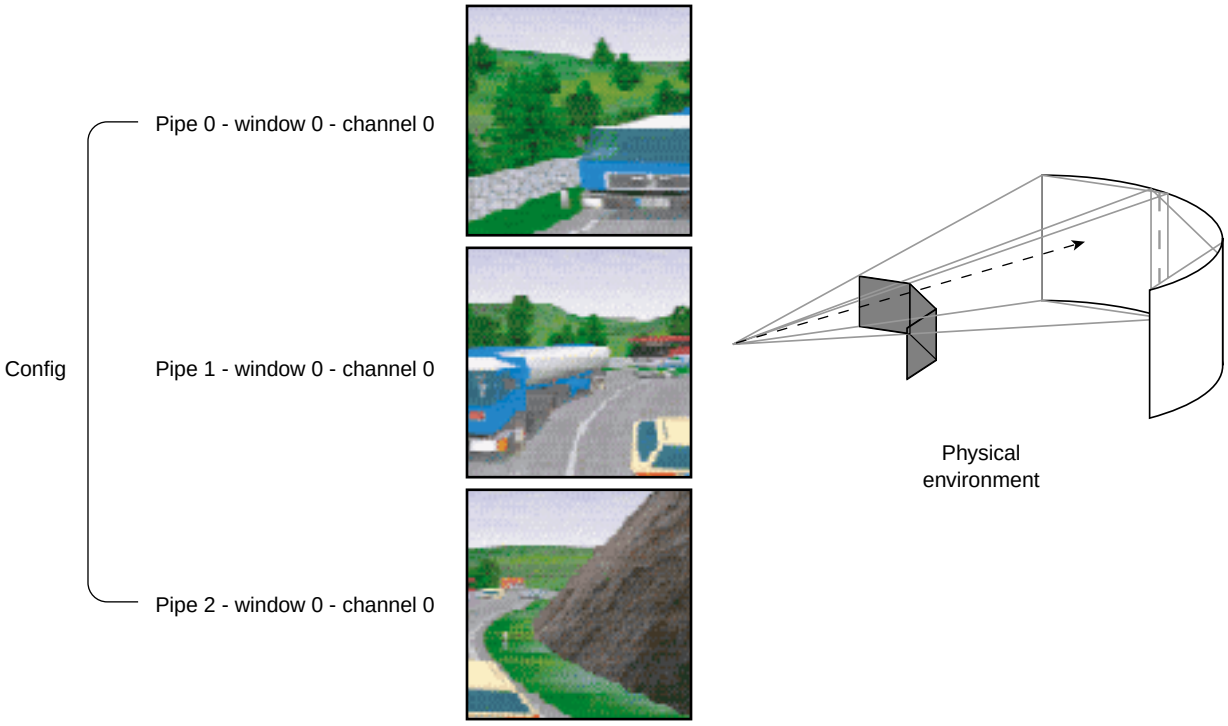


**Figure 4** Single View Per Pipe

In both cases, an application can be dynamically configured to run in mono or stereo through the configuration file. In the multiple-views-per-pipe scheme, the single pipe-window GLX visual is set in stereo mode. In the single-view-per-pipe scheme, each pipe-window GLX visual is set in stereo mode. Stereo mode can also be configured through scalable rendering, as explained in following sections.

## Scalable Rendering

To achieve greater application performance, OpenGL Multipipe SDK 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 5 illustrates the role of source and destination channels in scalable rendering.

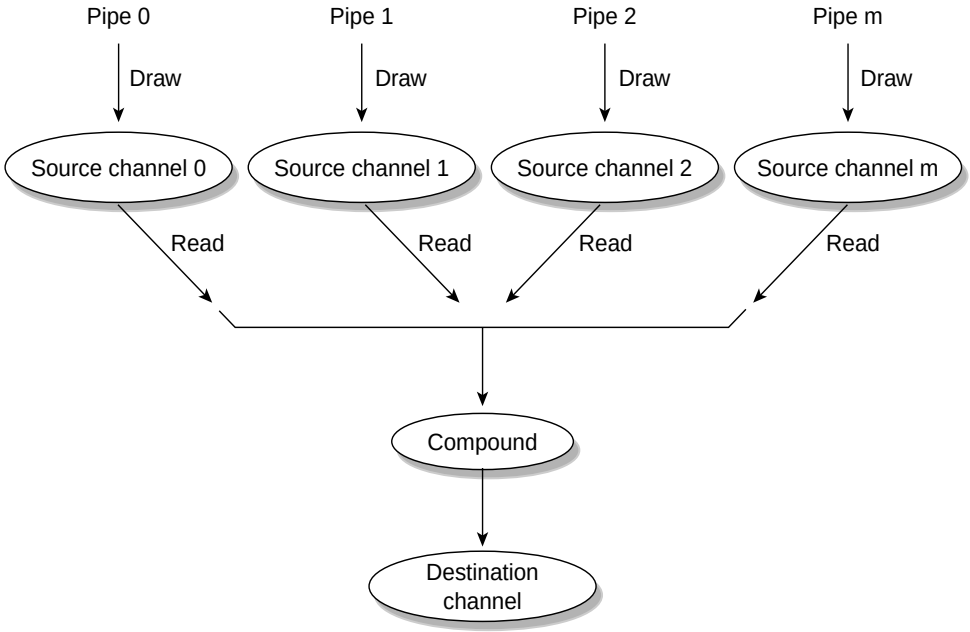


**Figure 5** Source and Destination Channels

The following decomposition modes are available and are described in the subsequent subsections:

- “Frame Decomposition” on page 10
- “Temporal Decomposition” on page 16
- “Multilevel Decomposition” on page 18

Each decomposition mode improves performance 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 | As 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 6 illustrates screen decomposition.

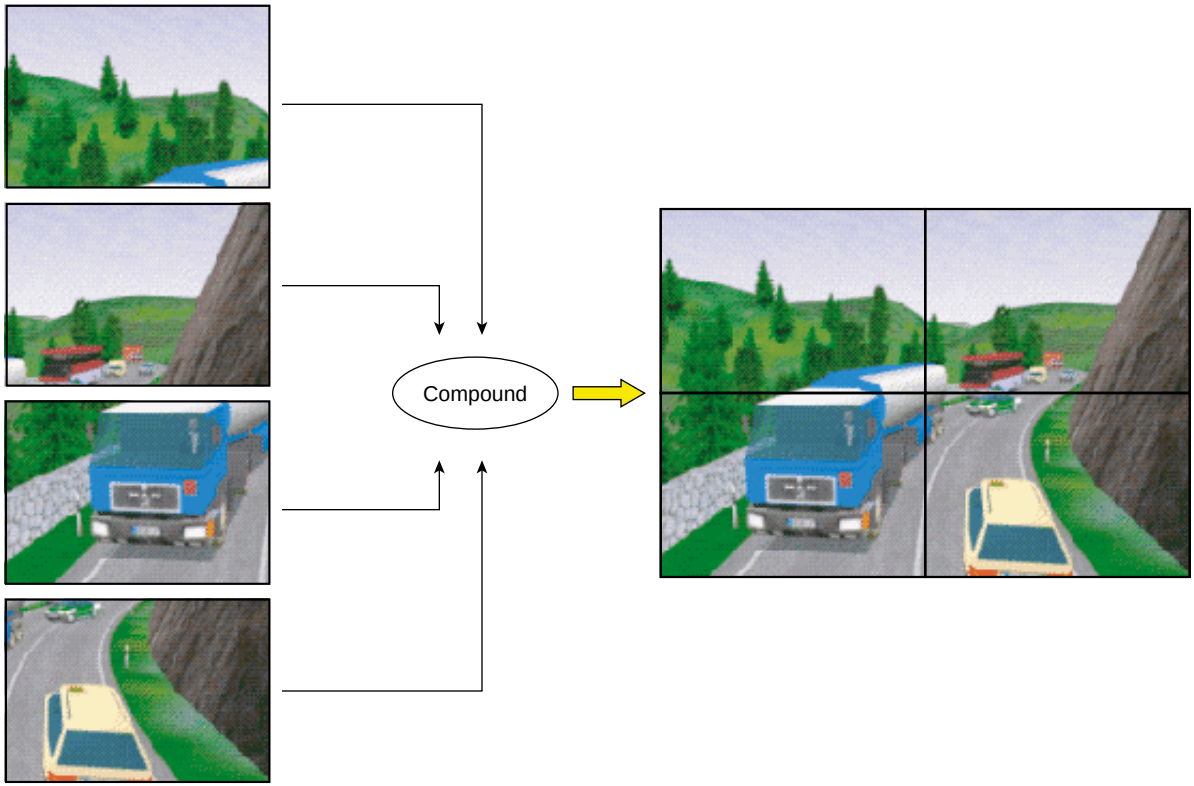


**Figure 6** Screen Decomposition

Example 2 shows the configuration file specifications for the screen decomposition illustrated in Figure 6.

**Example 2** 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, 1. ]
                    channel "source0"
                   }
# The top right of "destination" image will be
# rendered on "source1"...
          region {
                   viewport [ .5, .5, 1., 1. ]
                    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., 1., .5 ]
                  channel "destination"
                 }
        }
```

### 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 OpenGL Multipipe SDK 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. In addition, it also scales the texture memory capacity of the system by the number of pipes.

Figure 7 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.

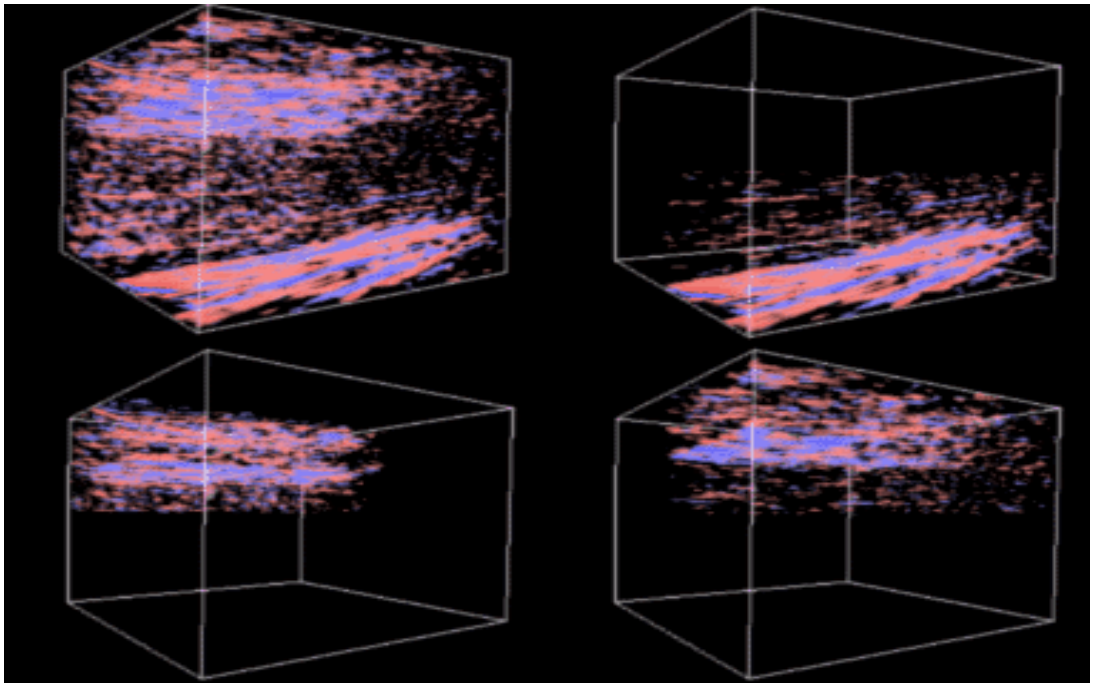

**Figure 7** Database Decomposition

Example 3 shows the configuration file specifications for the database decomposition illustrated in Figure 7.

**Example 3** 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”
                   }
             }
```

### Eye Decomposition

Eye decomposition is well-suited for stereo or multiple-view rendering. Each pipe renders a particular view (left, right, others). The final rendering depends on the type of display. As illustrated in Figure 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.

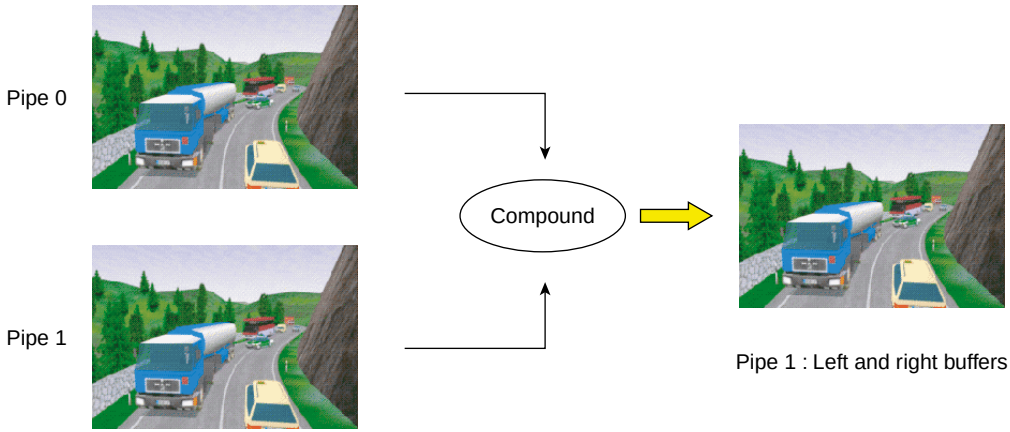


**Figure 8** Eye Decomposition

Example 4 shows the configuration file specifications for the eye decomposition illustrated in Figure 8.

**Example 4** 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.

## 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 software 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 9 illustrates frame multiplexing on a four-pipe system.

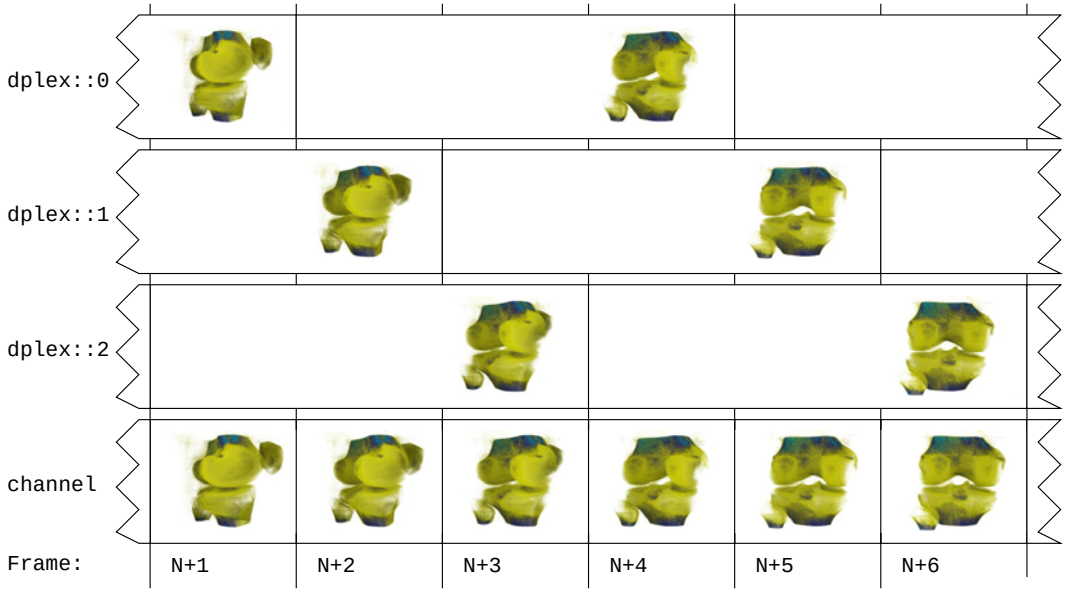


**Figure 9** Frame Multiplexing

Frame multiplexing scales globally 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.

Example 5 shows the configuration file specifications for the screen decomposition illustrated in Figure 9.

**Example 5** Frame Multiplexing Compound in a Configuration File

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

        region { channel “dplex::0” }
        region { channel “dplex::1” }
        region { channel “dplex::2” }
}
```

### Data Streaming

Data streaming (also referred to as 3D decomposition) is similar to database decomposition in the sense 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 10. 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 10, 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.

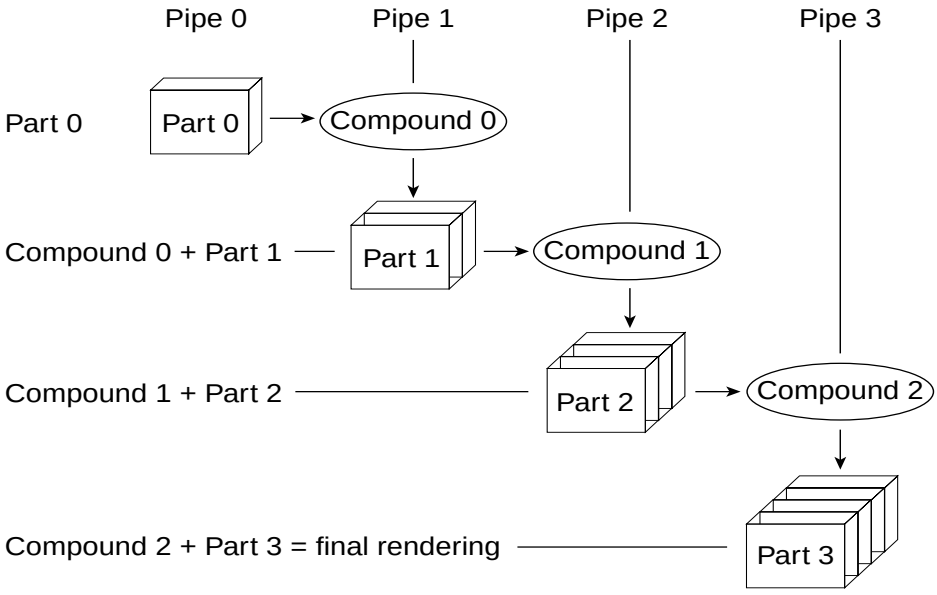


**Figure 10** Data Streaming Decomposition

## Multilevel Decomposition

OpenGL Multipipe SDK 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 with a limiting transport delay or can use an eye and database decomposition scheme for stereo volume rendering.

Figure 11 shows a four-pipe solution using an eye and database decomposition scheme.

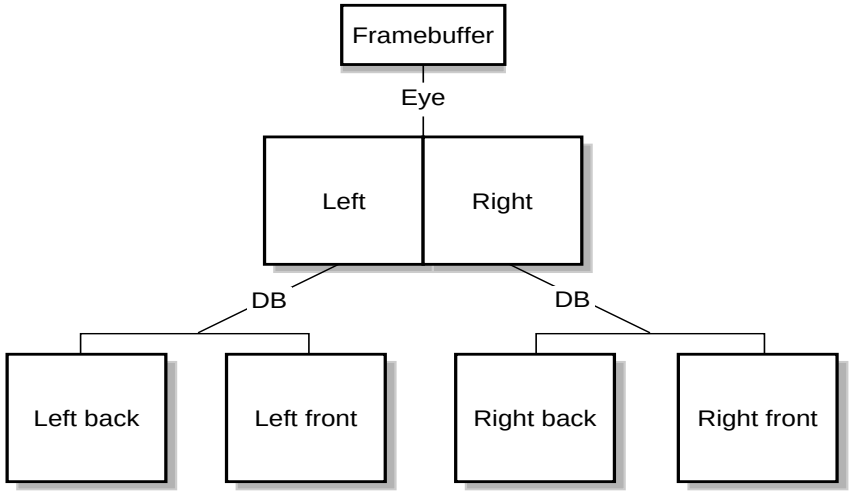


**Figure 11** Eye-DB Multilevel Decomposition

Example 6 shows the configuration file specifications for the multilevel decomposition illustrated in Figure 11.

**Example 6** 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”
            }
        }
    }
}
```

## 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:

| | |
|---|---|
| 2D | Use this scheme if your application is fill-limited. You can also scale geometry performance and texture memory size by using view-frustum culling techniques. |
| DB | Use this scheme when your application’s frame rendering can be sequenced into equally consuming phases. This requires being able 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. Note that this scheme introduces added latency. |
| Eye | Use this scheme for stereo viewing. |
| DPLEX | Use this scheme for general load balancing where the application maintains a reasonably steady frame rate. |

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

### Customized Assembly

OpenGL Multipipe SDK allows you to customize the assembly of the source and destination channels by providing callbacks that are invoked in the compound traversal. The application can provide two callbacks that are invoked before (pre-assemble callback) and after (post-assemble callback) the channel is traversed.

These callbacks can be used, for example, to composite the source channels in a particular sorted order using a certain blending mode. This is essential for applications like volume rendering, where the composition must be done in a front-to-back or back-to-front order. The example shown in Figure 7 uses the preceding technique to ensure correct alpha blending of the source channels.

## Programming Model

The OpenGL Multipipe SDK programming model reflects the “natural” application framework of OpenGL and adds a thread-safe implementation. In this model, the application has to provide only function callbacks while the core of OpenGL Multipipe SDK handles the synchronization and multithreading of rendering processes. Among other things, it functions as a multithreaded OpenGL Utility Toolkit (GLUT). The application developer defines a set of callback functions for each of the configuration file components (pipes, windows, channels, and compound). These functions are executed appropriately by the tasks (created by the core) during the frame rendering. The core then is able to manage the scheduling and the synchronization of the tasks. These callback functions are of several types:

| Callback Type | Description |
|---|---|
| Initialization | Contributes to component creation (pipes, windows, and channels) and sets initial parameters values. These callbacks are invoked when the application calls **mpkConfigInit()**. |

| | |
|---|---|
| Update | Defines actions to execute during each frame refresh. As they are executed for each new frame, the code defines how to render the channels but also the general updates done on the global context handled by each window. These callbacks are invoked when the application calls **mpkConfigFrame()**. |
| Event | Specifically for windows, defines an action to execute for a given input event (mouse, keyboard buttons, and the like). |

Along with providing an update-channel callback, OpenGL Multipipe SDK allows you to provide a clear-channel callback. This callback is invoked before the channel pre-assemble and update callbacks are invoked.

OpenGL Multipipe SDK takes care that the callbacks receive the correct frame data pointer depending on their latency. You pass a frame data pointer to each call of **mpkConfigFrame()**. The callback set using **mpkConfigSetDataFreeCB()** is invoked every frame to free frame data no longer needed for rendering.

Processes are created implicitly for each window and the synchronization occurs inside a frame generation. The developer can choose the multiprocess mechanism—`fork`, `sproc`, or `pthread`.

## Typical Code Example

Example 7 illustrates the programming model used in OpenGL Multipipe SDK. The program shows the following critical steps:

1. An initialization phase configuring data and environment ended by **mpkConfigInit()**

   At this step, all the callback pointers are given to OpenGL Multipipe SDK core rendering.

2. A loop where **mpkConfigFrame()** is the core rendering function that calls the right callback functions
3. A cleanup phase before exit

Figure 12 shows these critical steps in the overall program structure.

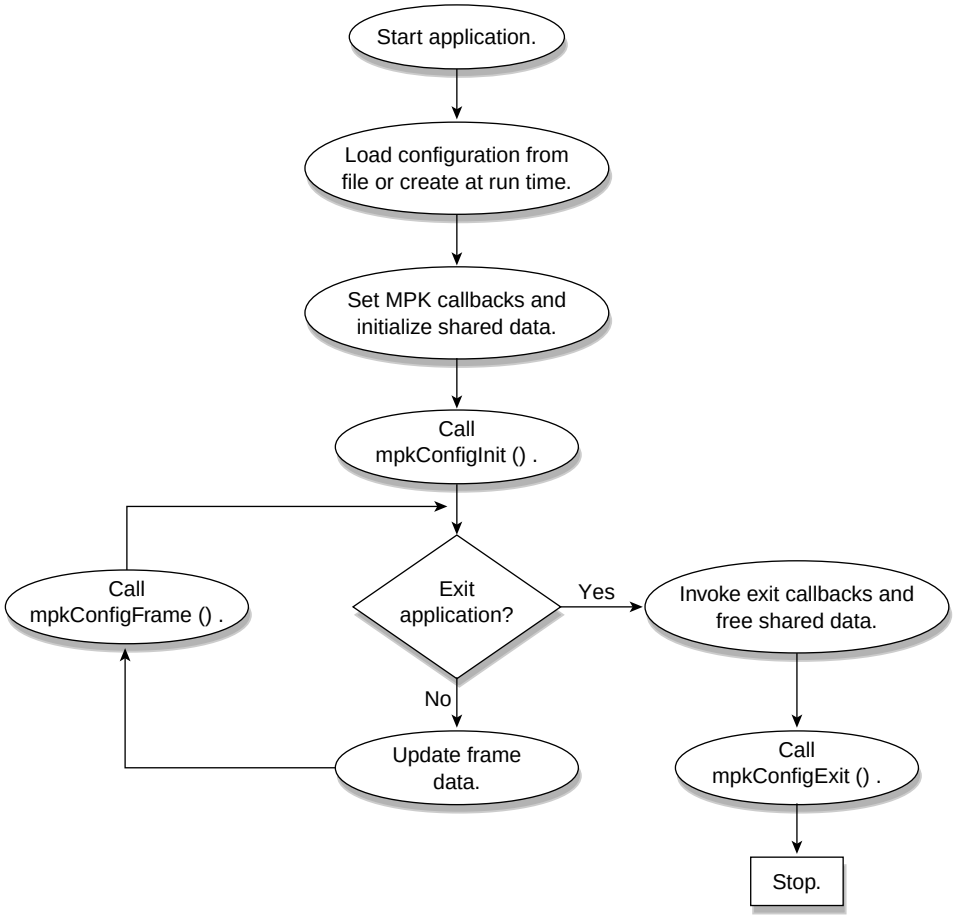


**Figure 12** Program Flowchart

**Example 7** Sample Program

```
////////////////////////////////////////////////
// application structure...
////////////////////////////////////////////////

main( int argc, char *argv[] )
{
    mpkInit();
    MPKConfig *config = mpkConfigLoad( ".mpconfig" );

    mpkConfigSetPipeInitCB( config, ... );
    mpkConfigSetWindowInitCB( config, ... );
    mpkConfigSetChannelInitCB( config, ... );
    mpkConfigSetDataFreeCB( config, ... );

    mpkConfigInit( config );

    while ( !exit ) {
        ...
        // update database
        ...
        framedata = newFrameData( db );
        mpkConfigFrame( config, framedata );
    }

    mpkConfigSetPipeExitCB( config, ... );
    mpkConfigSetWindowExitCB( config, ... );
    mpkConfigSetChanneExitCB( config, ... );

    mpkConfigExit( config );
}

////////////////////////////////////////////////////////////
// per-frame data
////////////////////////////////////////////////////////////

FrameData *newFrameData( Database *db )
{
    // Allocate memory for the frame data
    FrameData *frameData = (FrameData *) mpkMalloc (sizeof(FrameData)
);

    // copy relevant information from database into frame data
    ...
```

```
    return frameData;
}

void freeFrameData( MPKConfig *config, void *data )
{
    FrameData *frameData = (FrameData *) data;

    // Free the allocated memory
    mpkFree(frameData);
}
```

Figure 13 illustrates the task scheduling and callback invocations.

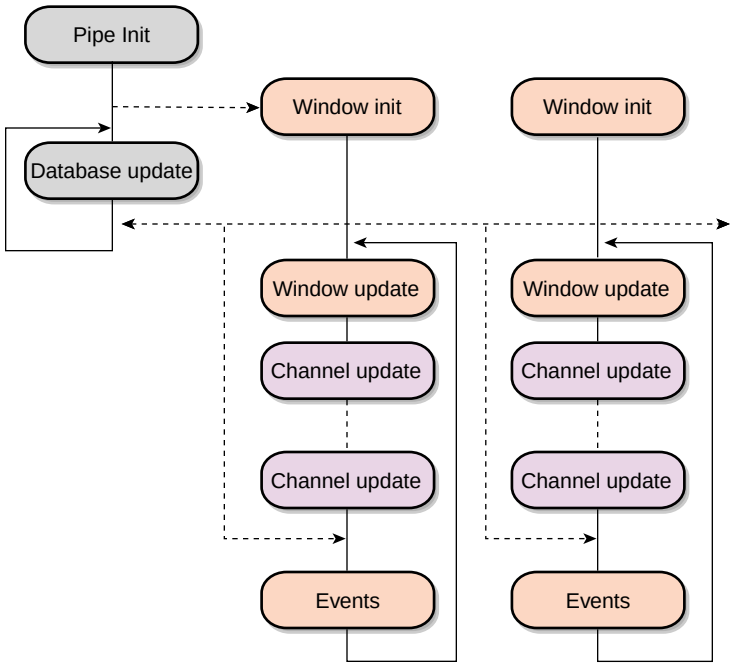


**Figure 13** Program Execution Model

Example 8 shows a one-pipe, one-window configuration file that can be used with the sample program in Example 7.

**Example 8** 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”, none ]
        stereo  [ “/usr/gfx/setmon -n str_top”, 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 ]
                        }
            }
        }
    }
}
```

## Download and Use It!

OpenGL Multipipe SDK, which gives application developers the leverage to extend their products to immersive environments, is free for the downloading. It can be accessed at the following URL:

`http://www.sgi.com/software/multipipe/sdk`

You can also find more documentation and resources from this webpage.