SGI™ DataSync™ Programmer’s Guide

007-4463-001

CONTRIBUTORS
Written by Ken Jones

Illustrated by Chrystie Danzer and Dan Young

Edited by Susan Wilkening

Production by Bryan Perkins

Engineering contributions by Patrick Bouchaud, Stefan Eilemann, Phillipe Robert, and Yair Kurzion

Record of Revision

Version	Description
001	August 2001 Original publication.

Contents

Figures . . . vii
Audience . . . ix
Related Publications . . . ix
Conventions . . . x
Reader Comments . . . x

1. **SGI DataSync Overview** . . . 1
SGI Graphics Cluster System Architecture . . . 1
System Software . . . 3
What SGI DataSync Provides . . . 5

2. **Sharing Memory Across a Cluster** . . . 7
A Typical Cluster . . . 7
Basic Services . . . 8
dsDataId—A Sharing Primitive . . . 9
Session Initialization—An Overview . . . 10
Session Initialization—API Details . . . 12
Session—Run-Time Considerations . . . 13
Events . . . 14
Message Passing . . . 15
Sample Frame-Based Execution with Barriers . . . 15
Session State . . . 16
Exit Sequence . . . 17
Man Pages . . . 18

3. ClusterFly—A Sample Application 19
Basic Operations 19
Starting ClusterFly. 20
Running `cfly` Manually on Each Channel Node 20
Running `cfly` on the Cluster from a Single Node 20
ClusterFly Controls 21
Looking Around 22
Approaching the Building 22
More Controls 23
Other Motion Models 24
Flying 24
Trackball 25
Motion Using Paths 25
Quitting ClusterFly 25
An Overview of the Source Code 26
Changes to `perfly` 26
Initialization Sequence—Function **InitClusterViewState()** 27
Sending Data from the Master to the Slaves 30
Receiving Data from the Master 31
Synchronizing Master and Slaves 31
Design Decisions 32

Figures

Figure 1-1 Sample Configuration of an SGI Graphics Cluster Series 12 . . . 2
Figure 1-2 System Software Architecture. 4
Figure 2-1 Typical Cluster Block Diagram 8
Figure 2-2 Propagation of Allocated dsDataId Objects across a Cluster . . . 11
Figure 3-1 Section of the New Jerusalem City Hall 22

About This Guide

This guide describes the SGI DataSync component of the SGI Graphics Cluster, which provides high-end graphics for visual simulation and virtual reality applications. The SGI Graphics Cluster uses either the Linux or Windows NT operating system and incorporates proprietary hardware and software from SGI.

SGI DataSync is a software component of the SGI Graphics Cluster Series 12 and is available only for Linux. The SGI DataSync component provides an API that enables data sharing across a cluster.

Audience

This guide targets applications programmers interested in creating graphics applications that can run efficiently in a cluster environment.

Related Publications

The following SGI documents contain additional information that may be helpful:

- *SGI ImageSync User's Guide*
- *SGI SynaptIQ Administrator's Guide*
- *SGI Graphics Cluster Hardware User's Guide*
- *SGI Graphics Cluster Quick Start Guide*

To obtain SGI documentation, see the SGI Technical Publications Library at `http://techpubs.sgi.com.`

Conventions

The following conventions are used throughout this document:

Convention	Meaning
`command`	This fixed-space font denotes literal items such as commands, files, routines, path names, signals, messages, programming language structures, and URLs.
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.
interface	This font denotes the names of graphical user interface (GUI) elements such as windows, screens, dialog boxes, and menus. Functions are also 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 tell us. 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, the document number is located at the bottom of each page.)

You can contact us in any of the following ways:

- Send e-mail to the following address:

  `techpubs@sgi.com`

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

  `http://techpubs.sgi.com`

- Contact your customer service representative and ask that an incident be filed in the SGI incident tracking system.
- Send mail to the following address:

  Technical Publications
  SGI
  1600 Amphitheatre Pkwy., M/S 535
  Mountain View, California 94043-1351
- Send a fax to the attention of “Technical Publications” at +1 650 932 0801.

SGI values your comments and will respond to them promptly.

SGI DataSync Overview

This overview of SGI DataSync consists of the following sections:

- "SGI Graphics Cluster System Architecture"
- "System Software"
- "What SGI DataSync Provides"

SGI Graphics Cluster System Architecture

The SGI Graphics Cluster uses either the Linux or Windows NT operating system and incorporates proprietary hardware and software from SGI. The hardware consists of the following:

- A single master node
- Multiple visual channel nodes (each with a commercial graphics card)
- An Ethernet backbone, including a network switch
- An optional SGI ImageSync network

Figure 1-1 illustrates the hardware architecture.

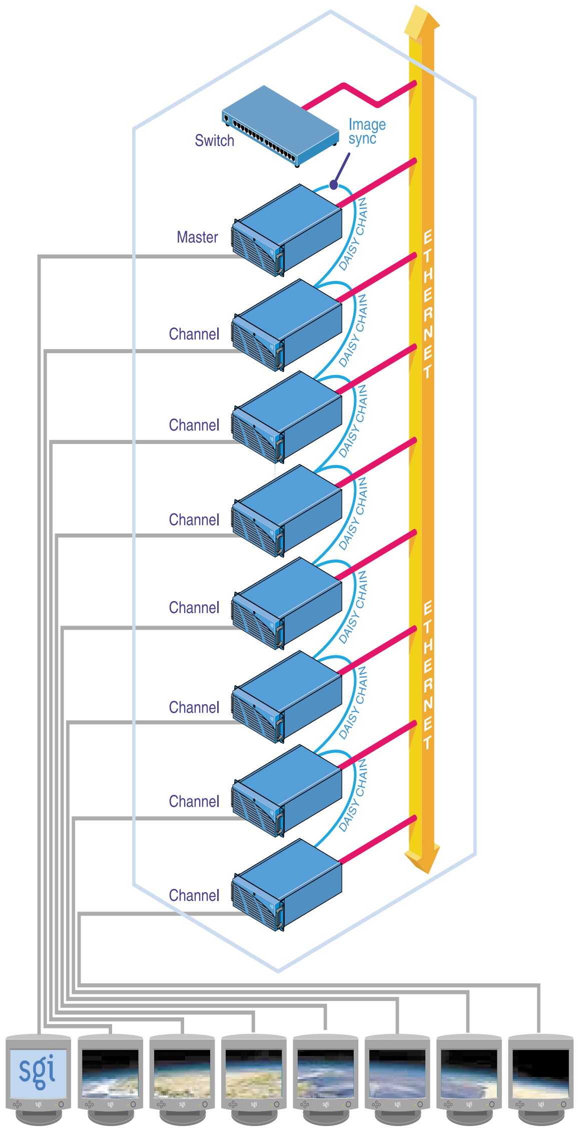

Figure 1-1 Sample Configuration of an SGI Graphics Cluster Series 12

System Software

The software for the SGI Graphics Cluster Series 12 consists of the following:

- Operating system (Windows NT 4.0 or SGI Linux kernel with XFS support (2.4.2-5SGI-137 or greater)
- SGI ImageSync device drivers
- SGI SynaptIQ (Linux systems only)
- SGI DataSync (Linux systems only)

Figure 1-2 illustrates the software architecture.

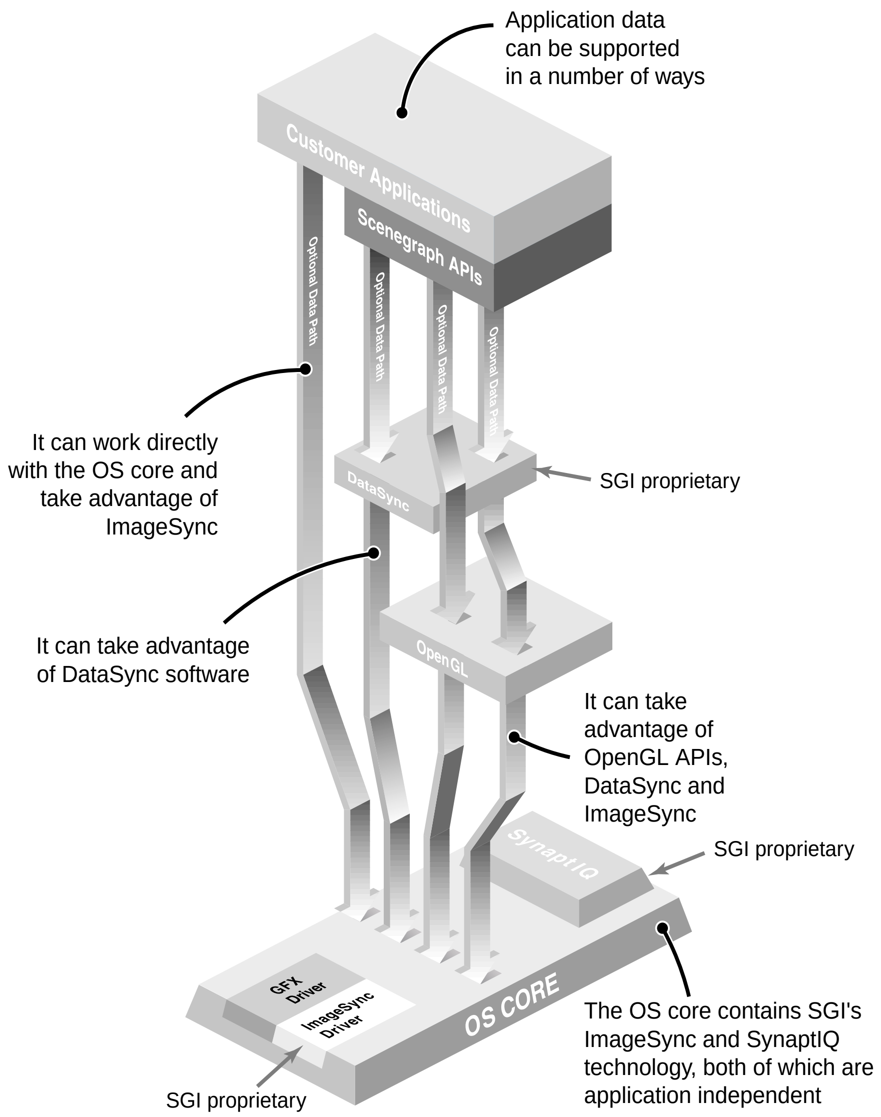
Application data can be supported in a number of ways
Customer Applications
Scenegraph APIs
Optional Data Path
Optional Data Path
Optional Data Path
Optional Data Path
It can work directly with the OS core and take advantage of ImageSync
DataSync
SGI proprietary
It can take advantage of DataSync software
OpenGL
It can take advantage of OpenGL APIs, DataSync and ImageSync
SynaptIQ
SGI proprietary
GFX Driver
ImageSync Driver
OS CORE
The OS core contains SGI's ImageSync and SynaptIQ technology, both of which are application independent
SGI proprietary

Figure 1-2 System Software Architecture

What SGI DataSync Provides

SGI DataSync attempts to provide a simulated single-system image (SSI) interface to distributed applications running on a cluster. Although compute applications have long used packages like Message Passing Interface (MPI) to parallelize codes across a cluster, a similar fabric has been missing in the rendering arena. SGI DataSync handles the complexity of breaking up and reassembling graphics data in a cluster environment and of rendering an application across multiple pipes.

The software core creates a *shared arena* style module within each node. A library call then creates named blocks within this shared arena. These blocks of memory can then be selectively shared across the nodes to provide a transparent mechanism for sharing data.

Sharing Memory Across a Cluster

SGI DataSync is an API enables data sharing across a cluster. It aims at imitating a shared memory system such as SGI Onyx on a cluster of computers with no physical shared memory.

SGI DataSync implements a client-server model for control operations and a symmetric, shared-memory model for data exchanges. This document describes the SGI DataSync API in the following sections:

- "A Typical Cluster"
- "Basic Services"
- "dsDataId—A Sharing Primitive"
- "Session Initialization—An Overview"
- "Session Initialization—API Details"
- "Session—Run-Time Considerations"
- "Events"
- "Message Passing"
- "Sample Frame-Based Execution with Barriers"
- "Session State"
- "Exit Sequence"
- "Man Pages"

A Typical Cluster

This section describes a typical cluster and names its components. A cluster consists of multiple PCs. Each one of these PCs is a cluster node. Figure 2-1 shows a cluster consisting of four nodes. Commonly, one of these nodes has the host name `master-channel`, and the others have the names `channel0`, `channel1`, and so on.

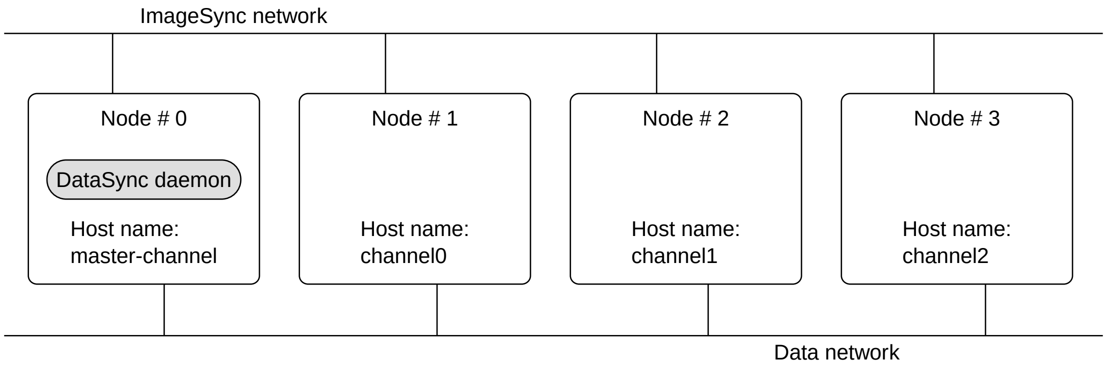

Figure 2-1 Typical Cluster Block Diagram

The node with the host name `master-channel` has a special designation: it is the master of the SGI ImageSync network. The SGI ImageSync network enables frame synchronization of all video outputs of cluster nodes.

SGI DataSync provides the API for sharing data and synchronizing processes on each one of the cluster nodes. Typically, SGI DataSync uses the node with the host name `master-channel` for running its daemon. All cluster nodes may run processes (multiple processes per node) sharing data using SGI DataSync.

Basic Services

When a collection of processes on cluster nodes has to share some data, it has to connect to an SGI DataSync session. SGI DataSync provides a server process that manages sessions. Processes can create or join sessions on the server. Once connected to a session, processes can share data with other processes in the same session.

When connecting to a server session, a process receives access to the following synchronization primitives:

- Shared memory blocks
- Messages

- Cluster events (for example, nodes added or dropped from a session)
- Barriers
- Semaphores
- Locks

dsDataId—A Sharing Primitive

In order to share data, cluster nodes have to create a correspondence between memory buffers on each node. In order to create this correspondence, SGI DataSync lets an application attach each memory buffer to a unique identifier. Two nodes attaching memory blocks to the same unique identifier are sharing the contents of their memory blocks.

SGI DataSync identifies blocks of data and other synchronization primitives among cluster nodes using unique identifiers called dsDataId objects. Each node on the cluster can allocate multiple dsDataId objects and these are unique across the entire cluster. When two nodes on a cluster operate on the same dsDataId identifier, they access the same object.

A dsDataId object can have one of the following types:

`DS_MEMORY`	A memory block identifier
`DS_BARRIER`	A barrier identifier
`DS_SEMA`	A semaphore identifier
`DS_LOCK`	A lock identifier

A dsDataId object of type `DS_MEMORY` does not allocate any memory. Attaching a block of previously allocated memory to a dsDataId object of type `DS_MEMORY` makes this memory block shared.

dsDataId objects of types `DS_BARRIER`, `DS_SEMA`, and `DS_LOCK` do not need additional attachment to local constructs. For example, an application can join a barrier by naming a dsDataId object of type `DS_BARRIER`.

For each type of dsDataId object, SGI DataSync provides a different API set. For example, on dsDataId objects of type `DS_MEMORY`, SGI DataSync provides interfaces for the following actions:

- Allocate/free memory for a dsDataId object.
- Map/unmap (attach/detach) a block of physical memory to a dsDataId object.
- Mark a dsDataId memory block as dirty.
- Synchronize contents of all dirty memory blocks.

Session Initialization—An Overview

This section describes the order of events in a basic SGI DataSync session (dsSession). This high-level description includes the required operations without exposing API details, which are described in the following subsection.

The following steps describe the initialization of a session.

1. Start the server process on one cluster node.
2. Repeat the following steps for each process needing to share data:
    a. Initialize SGI DataSync.
    b. Acquire a handle to the SGI DataSync server.
    c. Try to create a dsSession on the server. If you succeed, this process is the first process in the dsSession. If you fail, a dsSession already exists and the process has to log in to the existing dsSession.
    d. Add a new user (dsUser) to the dsSession. A dsUser is a connection to a dsSession. All dsSession-related operations use the dsUser pointer.
3. If this process is the first process in a dsSession, do the following:
    a. Allocate unique identifiers for shared data blocks.
    b. Allocate a special identifier for a dsSession information block.
    c. Allocate local memory for each identifier.
    d. Attach each identifier to its newly allocated memory.
    e. Store the unique identifiers in the dsSession information block.
    f. Realize the dsSession.
4. For each process other than the first one in a dsSession, do the following:
    a. Query the dsSession for its information identifier (a dsDataId indentifier).

b. Allocate memory for the dsSession information.

c. Attach newly allocated memory to the dsSession information identifier.

d. Retrieve the unique identifiers that the first process stored in the dsSession information block.

e. Allocate memory for each unique identifier.

f. Attach the newly allocated memory to the new identifier.

Figure 2-2 demonstrates the process of allocating new dsDataId objects and distributing them to all cluster nodes. Node A is the first node to join a session. It allocates three dsDataId objects (D1, D2, and D3) and three memory blocks (A1, A2 and A3). Node A then maps A1 to D1, A2 to D2, and A3 to D3. In order to inform node B of the newly allocated dsDataId objects, Node A stores D2 and D3 in memory block A1 and designates dsDataId D1 as the session information block.

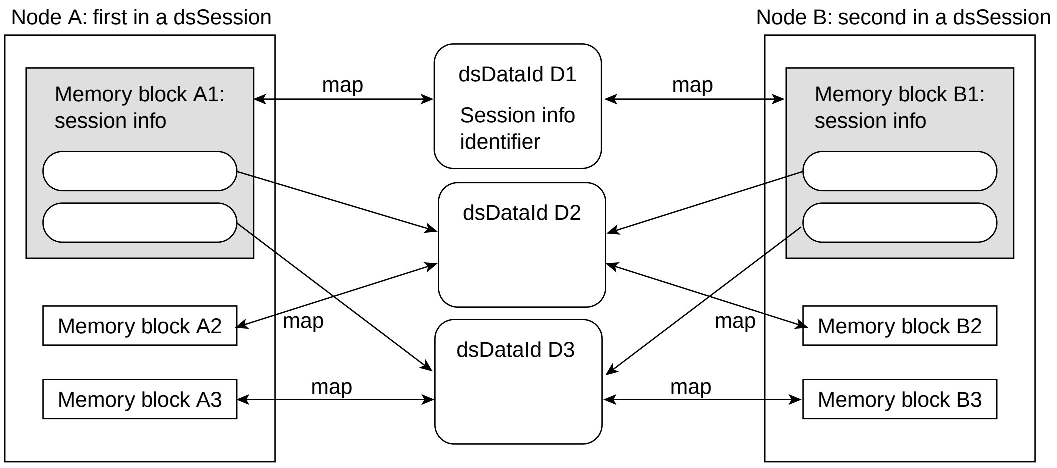

Figure 2-2 Propagation of Allocated dsDataId Objects across a Cluster

Node B would like to share blocks A2 and A3 with node A. Node B starts by allocating three memory blocks B1, B2, and B3. In order to make B2 and A2 shared, node B has to get dsDataId D2. It queries the dsSession for the dsDataId of its information block (D1), maps memory block B1 to it, and reads D2 and D3 out of its local B1.

In effect, the dsSession info block serves as a globally known name that all dsSession users can access.

Session Initialization—API Details

This section shows pseudo-code for implementing the example in Figure 2-2 using the SGI DataSync API.

```
dsInit(initParams);
server = dsOpen( openParams );
buffer1 = malloc ( ... );
buffer2 = malloc ( ... );
buffer2 = malloc ( ... );
session = server->addSession( “SessionName” );
if (session) // Success - I’m the first user of this session.
{
     user = session->addUser(“First”);
     // Create three dsDataId’s
     D1 = user -> alloc( DS_MEMORY, DS_PRESERVED );
     D2 = user -> alloc( DS_MEMORY, DS_UNPRESERVED );
     D3 = user -> alloc( DS_MEMORY, DS_UNPRESERVED );

     // Store dsDataId’s in dsSession information block
     buffer1 -> d2 = D2;
     buffer1 -> d3 = D3;

     // Attach dsDataId’s to physical memory.
     user -> mapMemory(D1, 0, buffer1, ... );
     user -> mapMemory(D2, 0, buffer2, ... );
     user -> mapMemory(D3, 0, buffer3, ... );

     // Notify SGI DataSync that the contents of buffer1 should be sent
to
     // all other cluster nodes that mapped memory to D1.
     user -> markMemory(D1, ...);
     user -> syncMemory();

     // Designate the dsDataId D1 as the session information
     // identifier.
     session -> putInfo(D1);

     // Finish session startup.
     session -> realize();
{
else
{
     session = server -> login(“SessionName”);
     user = addUser(“Second”);

     // Obtain dsDataId of session information block.
     D1 = session -> getInfo();

     // Attach memory to the session information dsDataId. Request an
     // immediate copy of the most up-to-date data in that buffer.
     user -> mapMemory(D1, buffer1, ..., DS_RDONLY | DS_COPY);

     // Get the dsDataId’s for D2 and D3 from the session information
     // block.
     D2 = buffer1 -> d2;
     D3 = buffer1 -> d3;

     user -> mapMemory(D2, buffer2, ...);
     user -> mapMemory(D3, buffer3, ...);


}
```

At the end of the above code sequence, each one of the processes on the dsSession has three local memory buffers and SGI DataSync is responsible for keeping their contents identical upon user request.

Session—Run-Time Considerations

SGI DataSync optimizes its processing based on the amount of network traffic that it produces. It does not propagate memory block changes unless specifically requested to do so by the application. On the other hand, an application does not have to request SGI DataSync to update its local memory blocks with information from other nodes. These updates happen transparently using a separate updating thread.

Assume `Buffer` is a local memory block mapped to dsDataId D. A frame of a producer process would therefore contain the following stages:

- Compute new data in `Buffer`.
- Mark D dirty.
- Call the **syncMemory()** function to propagate the contents of `Buffer` across the cluster.

A consumer of the shared data in `Buffer` would simply use its contents—that is, never calling any API to update it. Updates happen transparently without application intervention.

Note: Transparent updates can cause program errors and race conditions. The application should always assume that any shared buffer may be changing while being read. Shared buffers are just as sensitive to race conditions as true shared memory buffers in a multiprocessing environment.

The simple session in the preceding section does not synchronize the writers and readers of a shared memory block. Most applications require some level of synchronization among cluster processes. The following sections demonstrate how to use the SGI DataSync synchronization primitives.

Events

Many SGI DataSync API calls generate a notification event either for the dsUser calling the API or for all other connected dsUsers. Each dsUser can read a queue of incoming events in order to find out about these API calls. Event types break into the following groups:

- dsSession/dsUser activity
  - A session was added or removed on the SGI DataSync server.
  - A dsUser joined or left a dsSession.

  These events are important when a process needs to change its behavior based on the size or existence of dsSessions.
- dsDataId activity

  A memory block mapped to some dsDataId has been updated with new contents, or a dsDataId has been freed.
- Messages

  A message from another user has arrived.

An application may declare what type of events it wants to receive by calling **dsUser::handleEvent()**. All other events will be rejected upon arrival and will not fill up the event queue.

An application can poll for events using **dsUser::checkEvent()** and read the next event by calling **dsUser::nextEvent()**.

Message Passing

SGI DataSync provides a message passing mechanism among nodes connected to a dsSession. A dsUser can send a memory block as a message to either a single other dsUser or to all dsUsers on a dsSession.

Upon arrival of a message, the receiving dsUser receives a dsEvent of type `dsMessageEvent`.

Sample Frame-Based Execution with Barriers

This section demonstrates the use of barriers in the following setup: multiple cluster nodes read a shared memory block that one cluster node writes. The writer changes the contents of the buffer at regular frame intervals. The readers are to process new contents as it arrives.

In the following event-sequence example, process A is the writer and process B is one of the readers of a shared memory block D.

1. At initialization time, the first process in the dsSession allocates a dsBarrier and store its dsDataId in the session information block.

   All other processes can retrieve the barrier dsDataId from the session information block.

2. For each frame, writer process A does the following:

   a. Changes the contents of the shared memory block D.

   b. Calls **dsUser::markMemory()** on D.

   c. Calls **dsUser::syncMemory()**.

   d. Enters the barrier.

3. For each frame, reader process B does the following:
   a. Enters the barrier.
   b. Reads and uses the contents of data block D.

The preceding sequence assumes that B's usage of the contents of D is always shorter than A's frame duration. In other words, at the time B finishes reading the block D, A has not yet started modifying it again. If this assumption is too strong, the application could use another barrier in order to avoid writing on D before B finishes its reading.

Note: Using a dsBarrier requires specifying the number of nodes expected to join the barrier. Since nodes can join or leave a dsSession at any time, the application may run into race conditions and hang on a barrier expecting too many participants.

Session State

Applications often have to know about all dsUsers connected to their session.They often wish to acquire a list of all other dsUsers on a cluster and compute their relative position on that list. The following code sample demonstrates how to compute this information:

```
// get the directory of all dsUsers in session
dsDirectory *dir = (dsDirectory *) session -> directory();

// Lock directory while we look through it.
dir -> lock();

// Loop through dsUsers until we find our user name.
for (i = 0 ; i < dir -> size() ; i ++)
     if (strcmp (myUserName, dir -> entry(i)) == 0)
     {
          myUserID = i;
          break;
     }

// Unlock directory so we can receive session updates from daemon.
dir -> unlock();
```

Exit Sequence

When exiting a session, all processes other than the first process in a session should log out of the session. The first process in the session must remove the session after all other processes have logged out.

Note: Removing a dsSession (by calling the **dsServer::removeSession()** function) before all dsUsers have logged out of it may cause the application to crash. When removing a dsSession, all dsSession and dsUser interfaces are disabled on the removed dsSession. This means that any call other than **dsServer::logout()** and **dsSession::removeUser()** will produce an error.

To expedite the exit sequence of your program, you can replace the **sleep(1)** call with **user->nextEvent(dsRemoveUserEvent).**

The following code sample demonstrates how the first process in a dsSession waits for all dsUsers to log out before it removes the dsSession:

```
// Figure out initial number of users in session.
dsDirectory *dir = (dsDirectory *)(session -> directory());
numUsers = dir -> size();

// Wait for all users to logout from the session.
while (numUsers > 1)
{
     dsDirectory *dir = (dsDirectory *)(session -> directory());
     numUsers = dir -> size();
     sleep (1);
}
// Remove dsSession from the master.
server -> removeSession (session_name);
// Exit SGI DataSync.
dsExit();
```

Note: Exiting an application without removing the dsSession leaves this session open on the SGI DataSync daemon. If you run the application again without restarting the server, there is no way to tell which process is the first in the session.

Man Pages

For further descriptions of the SGI DataSync API, see the following online man pages:

- `dsInit`
- `dsServer`
- `dsSession`
- `dsUser`
- `dsArena`
- `dsDirectory`
- `dsError`
- `dsEvent`
- `dsInitParams`
- `dsMessage`
- `dsOpen`
- `dsOpenParams`

ClusterFly—A Sample Application

This chapter demonstrates the basic DataSync functionality through the ClusterFly demo. The application ClusterFly is a basic visual simulation application that can load, store, and display scene databases in many common formats. This chapter describes how to use ClusterFly to look at several sample databases provided with the SGI Graphics Cluster.

The following sections describe ClusterFly:

- "Basic Operations"
- "An Overview of the Source Code"

Basic Operations

The ClusterFly demo provides a graphical user interface (GUI) with which you can control many visual simulation features such as time-of-day selection, fog density, and so on. These options all default to reasonable values; so, you do not need to learn about them before using ClusterFly.

Note: To faciliate the description of ClusterFly operation, this section assumes your cluster consists of the following four nodes: `master-channel`, `channel0`, `channel1`, and `channel2`.

The following sections describe ClusterFly operation:

- "Starting ClusterFly"
- "ClusterFly Controls"
- "Looking Around"
- "Approaching the Building"

- "More Controls"
- "Other Motion Models"
- "Motion Using Paths"
- "Quitting ClusterFly"

Starting ClusterFly

In order to run ClusterFly, start an identical copy of the program `cfly` on each of the channel nodes. You can either run `cfly` manually on each node, or you can use a script that runs `cfly` on them from a single node. The following subsections show each one of these operation modes using a data file called `town_ogl.pfb`.

Running `cfly` Manually on Each Channel Node

If you wish to run each node separately, on each node enter the following:

```
% setenv PFSERVER master-channel
% cfly town_ogl.pfb
```

Start at the leftmost node and advance to the right. As new nodes start, `cfly` breaks the viewer field of view among all the available hosts from left to right in the order that they were activated.

Running `cfly` on the Cluster from a Single Node

If you wish to start `cfly` from a single node, do the following:

1. Ensure all nodes have logged in and have executed the following command:

   ```
   % xhost +
   ```

2. Start `cfly` using the provided script on the node `master-channel`:

   ```
   % run_all_cfly town_ogl.pfb
   ```

   The following is the contents of the script `run_all_cfly`:

   ```
   #!/bin/csh
   # start ImageSync.
   /usr/share/ImageSync/bin/imagesync -v
   # Start cfly application on all channels listed in /etc/channel_list
   # Channels should be listed from left to right to ensure correct
   ```

```
# breakup of the cfly field of view.
set list=‘cat /etc/channel_list‘
# assume DataSync server runs on master host (this host)
set server=‘hostname‘
set c=0
set i=2
while ( $c <  $list[1] )
   rsh $list[$i] /usr/bin/X11/run_cfly $server $* &
   @ c = ($c + 1)
   @ i = ($i + 1)
   sleep 3
end
```

Each channel node contains the script `run_cfly`, which consists of the following:

```
#!/bin/csh
setenv PFSERVER $1
setenv DISPLAY :0
setenv __GL_SYNC_TO_VBLANK 1
shift
/usr/bin/X11/cfly $*
```

Note that the script `run_all_cfly` uses the channel information file `/etc/channel_list`. The script assumes that this file lists channel nodes in the order determined by the left-to-right position of their display monitors. If your channels come up with incorrect view offsets, check the file `/etc/channel_list`.

ClusterFly Controls

The ClusterFly GUI provides a control panel with an assortment of controls. You can operate the control panel using the mouse buttons and the keyboard attached to node `channel0` (or the node that started running `cfly` first).

Many other keys on the keyboard are active and can be used to control ClusterFly even when the control panel is not displayed. For full details, the `perfly` man page contains a list of these key sequences and their effects, as well as details on motion models.

Entering `cfly -H` at the command line displays a list of the command-line options. For instance, the ClusterFly program allows several motion models; the `–d` on the command line tells the program to start in the Drive model, which provides an easy way to drive or walk through a scene while maintaining a fixed height above the ground.

Looking Around

Look around the scene using the mouse. First, place the cursor in the center of the simulation window. Now depress the middle mouse button and move the mouse to the left or right to turn in place; you will continue to pivot until you place the cursor back in the center of the screen.

Do not worry if you inadvertently start moving around, lose sight of the building, or otherwise lose position or control. Just move the cursor into the control panel area and click the **Reset All** button on the control panel to get back to the original setup.

Approaching the Building

To approach the City Hall model, turn until you are facing it (if you are not already facing it) and then center the mouse in the screen. Depress the left mouse button briefly to start accelerating forward. When you release the button, you will continue gliding forward at constant speed and can hold down the middle mouse button to steer. The ClusterFly application shows you how the basic visual simulation tools work. This example uses a section of the New Jerusalem City Hall (see Figure 3-1).

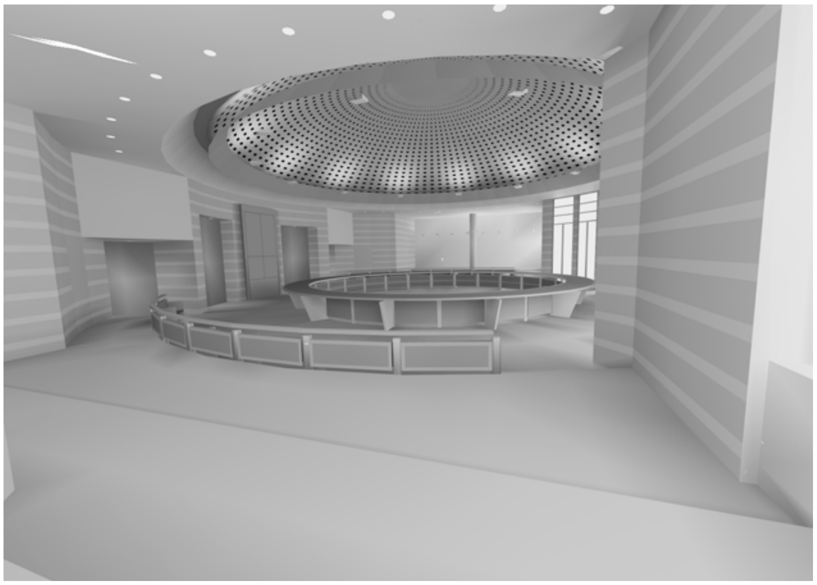

Figure 3-1 Section of the New Jerusalem City Hall

Tap the middle mouse button to stop in front of the building (if you actually entered the building, remember the **Reset All** button). Now accelerate backward by pressing the right button. When you are as far back as you want to go, hold down the left mouse button to gradually slow down, or tap the middle mouse button to stop immediately.

Now use the left mouse button again to start moving forward and drive slowly into the model. Notice that the walls closest to you are cut away at first so you can see inside; when you are completely inside the building, those walls reappear. Drive around and explore the building. Tap the middle mouse button to stop before you run into anything (but do not worry—at this point you will bounce off any walls you hit). If the walls get in your way, you can turn off collision detection with the button labeled **Collide** on the control panel, or press the `c` key on the keyboard.

More Controls

To see the underlying geometry used to create the model, click the **Style** button in the control panel, or press the `w` key on the keyboard. This changes the display to wireframe mode. In this mode you can more easily see how many polygons are used to represent an object. This information can be helpful when you are tuning a database, because it is important to know when the number of polygons becomes a limiting factor. To turn wireframe mode off, just click the **Style** button (or press `w`) again. The `W` key can be used to cycle through several different drawing styles.

To close the entire control panel (and devote the entire screen to the model), click the GUI **`Off`** button at the upper right of the control panel, or just press the `F1` key. Press the `F1` key again to restore the control panel. The GUI is part of `libpfutil`. See the sample program, `/usr/share/Performer/src/pguide/libpfutil/utilui.c`.

If you click the **Stats** button in the control panel, a transparent panel showing scene statistics appears overlaid on the screen. The buttons next to the **Stats** button allow you to choose one of the available statistical displays. Try moving around in the scene while watching how the statistics change. Note in particular that the number of triangles being considered for rendering changes drastically depending on where you look; this demonstrates OpenGL Performer’s use of culling to ignore objects that are completely outside the field of vision. For more information about culling and the statistics panels, see the *OpenGL Performer Programmer’s Guide*.

The control panel’s field-of-view slider can be used to select a wide angle view, up to 100 degrees.

As you travel through the building, try turning on the fog effect by clicking the **Fog** button. Experiment with the fog density and other controls. (Remember: If you have closed the control panel, the `F1` key restores it.)

Other Motion Models

So far you have been driving. There are other default motion models provided through the `libpfui` library. These motion models can be subclassed to create your own models. You can find the source code for these motion models in the directory `/usr/share/performer/src/lib/libpfui/`.

Flying

The Fly motion model provides an alternative to the Drive model. This model allows full motion in three dimensions (unlike the Drive model, which does not allow vertical motion). The mouse in the Fly model is used in much the same way to control motion, but when steering, the vertical position of the mouse in the window controls your vertical tilt. You can select this mode by pressing the right mouse button on the button marked **Drive** and select **Fly** from the menu.

As when driving, the left mouse button makes you go forward and the right mouse button makes you go backward. As long as either button is pressed you will continue to accelerate.

You turn by holding down the middle mouse button and moving the cursor away from the center of the simulation window. Moving the cursor left or right causes left or right turns, respectively. Moving the cursor up or down causes the view direction to tilt up or down, respectively. The rate of turning and tilting is scaled by the distance of the cursor from the center of the simulation window; that is, no change of direction occurs when the cursor is at the center and full-speed rotation occurs at the edges of the window.

If you want to maintain a steady velocity rather than accelerating, hold down the middle mouse button to steer while using the left and right buttons to control the speed. To stop, tap the middle mouse button.

Trackball

The Trackball motion model provides a third option for controlling motion. You can select this mode by pressing the right mouse button on the button marked **Fly** and selecting **Trackball** from the menu.

In trackball mode, when you drag with the middle button the object rotates about its center as if it were attached to a large trackball that fills the screen. Dragging up and down causes rotation about the horizontal axis parallel to the screen; dragging left and right causes rotation about the vertical axis parallel to the screen.

By dragging with the left mouse button, you can translate the object in the direction you drag: left, right, up or down. By dragging with the right mouse button, you can translate the object in and out of the screen. In all cases, if you release the mouse button while dragging, the motion continues on its own.

Motion Using Paths

There are other approaches to traveling through a scene than the models described here. For instance, you can build a specific path into the viewer, to prevent the user from straying outside your model. The Path model is supported by a general path-following system in the `libpfutil` library. Many simulation applications require path support for such objects as cars, trucks, and people (in driver-training software); waiting aircraft both on the ground and in the air (in flight simulation); and opposing forces in military trainers. Path support in `libpfutil` allows paths of varying speeds to be built from line segments and arcs with automatic fillet construction between segments for smooth transitions.

Quitting ClusterFly

When you want to quit `cfly`, either press the `Esc` key on the keyboard of `channel0` (or the node that started running `cfly` first) or click the **Quit** button on the ClusterFly GUI of `channel0` (or the node that started running `cfly` first).

An Overview of the Source Code

The ClusterFly demo is a good demonstration of DataSync and OpenGL Performer in action because it is a complete application. It is, however, a large and complex piece of code. The program `cfly` is based on the `perfly` sample code. This section assumes that you have a basic knowledge of the `perfly` sample code. An overview of `perfly` is available in the OpenGL Performer documentation.

This overview of the `cfly` source code consists of the following subsections:

- "Changes to perfly"
- "Initialization Sequence—Function InitClusterViewState()"
- "Sending Data from the Master to the Slaves"
- "Receiving Data from the Master"
- "Synchronizing Master and Slaves"
- "Design Decisions"

Changes to `perfly`

This subsection describes the changes made to `perfly` in order to create `cfly`.

In general, all the global information in perfly is shared among the cluster nodes. This information is stored in a global variable named ViewState. The naive approach would assign ViewState for sharing via the DataSync API. Unfortunately, ViewState contains pointers to node-local objects such as pfDCS nodes. Sharing the exact contents of ViewState would therefore share the pointers to the pfDCS nodes (incorrect). Moreover, the intent is to share information in two phase-shifted stages:

- Right before starting a new frame (before calling **pfFrame()**), share latency critical information like the viewer camera position.
- Right after a frame and before suspending in wait for a new frame (between **pfFrame()** and **pfSync()**), share the bulk of the required information—for example, draw modes, positions of vehicles, time of day, etc.

There are two new memory structures containing data for the two phase shifted stages:

- ClusterViewState_pre for sharing data before calling **pfFrame()**

- ClusterViewState_post for sharing data between **pfFrame()** and **pfSync()**

The intent was to keep the changes to the `perfly` source code to the necessary minimum. The new file `cluster.C` contains all the code accessing the DataSync cluster API. The original code contains new function calls in three places:

Initialization	In function **initViewState()**, call **InitClusterViewState()** for initializing DataSync.
Pre-Frame	In function **localPreFrame()**, call functions **sendClusterViewState_pre()** or **recvClusterViewState_pre()**, depending on the variable `ViewState -> clusterRole`. The first node to start up `cfly` has clusterRole `ROLE_MASTER`. All other nodes have clusterRole `ROLE_SLAVE`. These functions propagate the latency-critical portion of ViewState.
Post-Frame	In function **updateSim()**, call functions **sendClusterViewState_post()** or **recvClusterViewState_post()**, depending on clusterRole. These functions propagate the non-latency-critical portion of ViewState.

The `perfly` code contains a few other changes. In multiple places a node with clusterRole `ROLE_SLAVE` should skip some standard operations—for example, setting camera position.

Initialization Sequence—Function InitClusterViewState()

The code starts by initializing the DataSync library. Initializing requires initialization parameters as in the following code:

```
dsInitParams *initParams = new dsInitParams();
dsInit(initParams);
```

The program must now get a handle to the DataSync server. We use the environment variables `PFSERVER` and `PFPORT` to select the server host and port. Once selected, the program calls **dsOpen()** and receives a handle to the server as shown in the following:

```
dsOpenParams *openParams = new dsOpenParams();
if ((server = getenv ("PFSERVER")) != NULL)
     openParams -> setAddress(server);
else
     openParams -> setAddress(server_name);

if ((port = getenv ("PFPORT")) != NULL)
```

```
     openParams -> setPort(atoi(port));
dsServer *svr = dsOpen( openParams );
```

The program now allocates memory in the Performer shared arena for all the data structures that it wants to share via DataSync. These include the pre-**pfFrame()** and post-**pfFrame()** structures as well as a DataSync session information block. The following code shows the memory allocation:

```
ViewState -> SessionData = (ClusterSessionData *)
                     pfCalloc (sizeof (ClusterSessionData), 1,
                     pfGetSharedArena());
ClusterViewState_pre = (ClusterSharedViewState_pre *)
                     pfCalloc (sizeof (ClusterSharedViewState_pre), 1,
                     pfGetSharedArena());
ClusterViewState_post = (ClusterSharedViewState_post *)
                     pfCalloc (sizeof (ClusterSharedViewState_post), 1,
                     pfGetSharedArena());
```

The program tries to add a session to the DataSync server. Only one cluster node can add a session of a given name. The DataSync server returns a non-`NULL` result to that node. The program designates this node to be the master of the cluster. All other nodes are slaves of that master. The following code shows adding a session and finding the master:

```
dsSession *session = svr->addSession( session_name );
if (session) // Success - I'm the first use of this session.
{
    ViewState -> clusterRole = ROLE_MASTER;
    sprintf (ViewState -> UserName, "master");
    ViewState -> SessionUser = session->addUser(ViewState -> UserName);
    ....
}
else
{
    ViewState -> clusterRole = ROLE_SLAVE;
    sprintf (ViewState -> UserName, "slave[%d]", getpid());
    dsSession *session = svr->login( session_name );
    ViewState -> SessionUser = session->addUser(ViewState -> UserName);
    .....
}
```

A master receives a pointer to the newly created session and is considered logged in to the session. A slave has to log in to a session in order to receive a pointer. Once a session pointer is obtained, the master and the slave have to add themselves as users to a session. A dsUser is the only way a process can interact with a session. One process may have multiple dsUsers to a single session. In the `cfly` implementation, each process has only one dsUser.

During the execution of `cfly`, the program has to receive DataSync events when other cluster nodes enter or leave the session. The program informs the DataSync server what events we wish to handle by calling the following function:

```
ViewState -> SessionUser -> handleEvent (session_events);
```

On the master side, the program has to create all the DataSync shared components: a session-info block, shared-memory buffers, and barriers. The program allocates shared data IDs for these elements in the following code:

```
dsDataId session_data_id = ViewState -> SessionUser
                   ->alloc( DS_MEMORY, DS_PRESERVED );
ViewState -> SessionData -> data_pre = ViewState -> SessionUser
                   ->alloc( DS_MEMORY, DS_UNPRESERVED );
ViewState -> SessionData -> data_post = ViewState -> SessionUser
                   ->alloc( DS_MEMORY, DS_UNPRESERVED );
ViewState -> SessionData -> barrier_pre = ViewState -> SessionUser
                   ->alloc(DS_BARRIER, DS_UNPRESERVED);
ViewState -> SessionData -> barrier_post = ViewState -> SessionUser
                   ->alloc(DS_BARRIER, DS_UNPRESERVED);
```

The program only allocates these shared Data IDs on the master. The slaves receive these the IDs using a session-info block as described in the following paragraphs.

For barrier objects, the previous allocations are all that are needed. However, DataSync does not allocate any memory for memory objects. We have to attach a local memory buffer (allocated earlier by the application) to each shared data ID in order to enable sharing. The program does this in the following calls (on the master):

```
        ViewState -> SessionUser -> mapMemory(
                            ViewState -> SessionData -> data_pre,
                            0,
                            ClusterViewState_pre,
                            sizeof(ClusterSharedViewState_pre),
                            DS_WRONLY);
         ViewState -> SessionUser -> mapMemory(
                            ViewState -> SessionData -> data_post,
                            0,
                            ClusterViewState_post,
                            sizeof(ClusterSharedViewState_post),
                            DS_WRONLY);
        ClusterViewState_post -> exitFlag = 0;
        ViewState -> SessionUser -> mapMemory(
                            session_data_id,
                            0,
                            ViewState -> SessionData,
                            sizeof(ClusterSessionData),
                            DS_WRONLY );
```

On the slaves, the program maps the above shared memory blocks as read-only.

The only pre-defined shared memory block on the DataSync server is the session info. The master has to notify the slaves of its newly allocated shared data IDs by storing their IDs in the session info as follows:

```
session->putInfo( session_data_id );
```

Slaves can call **dsSession::getInfo()** to retrieve the shared data ID used for the session info and map a local memory block to it:

```
dsDataId cluster_data_id = session->getInfo();

ViewState -> SessionUser -> mapMemory(
          cluster_data_id,
          0,
          ViewState -> SessionData,
          sizeof(ClusterSessionData),
          DS_RDONLY|DS_COPY);
```

When done with its initialization, the master calls the following function:

```
session->realize();
```

All pending slaves continue running their initialization code.

Sending Data from the Master to the Slaves

The functions **sendClusterViewState_pre()** and **sendClusterViewState_post()** package portions of the ViewState structure and inform DataSync to propagate them to the slaves. This section will focus on **sendClusterViewState_pre()** because it is shorter.

As explained earlier, some members of ViewState are pointers to local objects; so, the program has to package them before sharing across a cluster. One such example is `ViewState -> sceneDCS`. The program cannot share its pointer over the cluster. Instead, the program calls the following function:

```
    ViewState -> sceneDCS -> getMat(ClusterViewState_pre -> sceneDCS);
```

Once done packing all latency-critical members in ClusterViewState_pre, the program informs DataSync that the buffer is not clear:

```
ViewState -> SessionUser -> markMemory (
          ViewState -> SessionData ->data_pre,
          0,
          sizeof (ClusterSharedViewState_pre));
```

The program then makes DataSync propagate all marked buffers across the cluster as shown in the following:

```
ViewState -> SessionUser -> syncMemory();
```

Receiving Data from the Master

DataSync does not have any API for receiving data. The local memory blocks that each slave creates are updated transparently. Slaves can assume that their input buffers are up-to-date and process them. The `cfly` slaves unpack input buffers into ViewState and make the necessary changes in the displayed GUI.

Synchronizing Master and Slaves

In order to force synchronization between the master and the slaves, each joins two barriers each frame. The master joins barriers right after calling **dsUser::syncMemory()**. The slaves join the same barriers before reading their shared memory copies. This ensures that slaves do not run too fast and process the same shared memory before the master finished updating it. It also ensures that slaves do not run too slow and miss complete updates of their shared memory.

Joining a barrier requires knowing its shared data ID and the number of users that should also join this barrier. The following call joins a barrier:

```
ViewState -> SessionUser -> barrier (
          ViewState -> SessionData -> barrier_pre,
          ViewState -> numUsers);
```

In order to maintain an up-to-date count on the number of users in a session, the program makes the following calls:

```
dsDirectory *dir = (dsDirectory *)
                   (ViewState -> ClusterSession -> directory());
ViewState -> numUsers = dir -> size();
```

The function **queryClusterRank()** computes the size of a cluster and the relative position of the current node in the cluster.

Design Decisions

The program `cfly` uses shared memory to share the contents of ViewState. Another possible approach would send messages every time some value changes in the ViewState structure.

- Advantages for shared memory approach
  - Much simpler. The application looks more like a multipipe Onyx application.
  - Any cluster node that joins the session late sees the full contents of ViewState. There is no need to request updates from one of the nodes.
- Advantage for message passing

  When users do not make many GUI selections, there is less traffic on the network. The application only has to send camera position and the position of moving models every frame.