ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, IL 60439

ANL/MCS-TM-218

CAVEcomm Users Manual

by

Terrence L. Disz, Michael E. Papka,
*Michael Pellegrino, and Matthew Szymanski**

Mathematics and Computer Science Division

Technical Memorandum No. 218

December 1996

*Address: The Electronic Visualization Lab, The University of Chicago, 851 S. Morgan St. (M/C 154), Chicago, IL 60607

This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

Contents

CAVEcomm Users Manual

by

Terrence L. Disz, Michael E. Papka
Michael Pellegrino, and Matthew Szymanski

Abstract

The CAVEcomm library is a set of routines designed to generalize the communications between virtual environments and supercomputers.

1 Library Model

The model for the library is that of a client/server. Somewhere at a known URL is a broker whose purpose is to mediate the communications between various client applications (CAVE, simulations, etc.).

1.1 The Broker

The broker maintains tables of all clients and of all sessions (applications) that are registered with it.

The table of clients includes such elementary information as the client's capabilities (machine type, model, etc.) as well as basic application information (CAVE application, IBM simulation, etc.). Clients can query the broker for this information.

The table of sessions includes information on all sessions (applications) that any registered client can run. The table maintains such information as executable names, data files, command line arguments, etc. Of course, clients cannot attach to sessions outside of the realm of their physical machine. For example, clients cannot run a CAVE application unless CAVE functiexists onality on their machine. Clients can query the broker for this session information.

The broker executable is in the bin subdirectory within the CAVEcomm distribution.

To run the broker, type `./c2cbroker` - commandline arguments

Commandline arguments include the following:

- -c2cdbg { *100-140* }

  Specifies the level of debuging messages output by the broker, 100 being the lowest level of **c2c** internal messages and 140 being the greatest.

- -c2curl_filename { *my_url_file* }

  Contains the URL for the broker, the idea being that this file lives in some common shared file space that all users of a given instance can access.

- -c2c_port { *port number* }

  Specifies the particular port that the broker will be listening on, thereby allowing users to pick an arbitrary port number and use that port over and over, thus eliminating the need to change the config files.

Example: `./c2cbroker -c2c_port 10001 -c2cdbg 140`

This command will start a broker process on the machine from which it is run, listening on port 10001, and printing out all the internal c2c debug messages available.

1.2 The Client

Each CAVEcomm client is an application (CAVE, simulation, etc.) that has the power to attach to any other application on the broker. A simulation that is registered with the broker can be accessed by any other application on the broker such as visualization applications or virtual environments.

2 Communications Transport

The CAVEcomm library uses the Nexus communications library developed at the Mathematics and Computer Science Division at Argonne National Laboratory. (See http://www.mcs.anl.gov/nexus.)

While Nexus is the communications transport, the user does not need any prerequisite knowledge in using it. All Nexus communications calls are encapsulated in the library; thus, the user is unaware of the Nexus presence.

By using Nexus, the CAVEcomm library is able to support various communications protocols such as TCP/IP sockets and shared memory, as well as various message-passing libraries.

3 CAVEcomm Library

All functions in the CAVEComm library start with the prefix **c2c**. The CAVEcomm library contains functions that are broken down into various sections.

3.1 General Routines

int c2cGetDebugLevel(void); Get the current maximum debugging level in an application.

void c2cInit(int *argc, char *argvg[],CaveEnv *env); Start all communications functionality. Its arguments are the argument count, argument vector, and a structure describing the application's operating environment. It *must* be the first CAVEcomm library call made (except for c2cReadConfigFile or if the world routines are used). If the CAVEcomm library is used with the CAVE library, the call to c2cInit must occur *after* the call to *CAVEInit* is made.

void c2cPrintf(int level, char *format, ...); Print the text. The text consists of a chracter string describing how to format the text and any applicable variables contained in the format. This function is similar to the printf function found in the standard I/O C library. The *level* parameter tells the library at what level to print the text.

int c2cReadConfigFile(char *filename,c2cConfig *config); Read *filename* for application definitions. The definitions are placed into *config*, which contains all known application definitions. If *filename* is null, the default file .c2cConfig is attempted to be opened. If *filename* or .c2cConfig cannot be opened, E_OPEN_CONFIG_FILE is returned; otherwise, E_SUCCESS is returned.

void c2cSetDebugLevel(int level); Set the current maximum debugging level in an application.

void c2cTerminate(void); End all communications functionality. It is the last CAVEcomm library call made (unless the world routines are used).

3.2 Broker Routines

HostId c2cBrokerAttach(URL url); Attach to the broker listed by *url*. The URL must be of the format **c2cBroker://{ip hostname}:{ip port}/**. Supplying a URL of a different format will give unpredictable results. A HostId is returned uniquely describing that broker.

int c2cBrokerDetach(HostId host); Detach from the broker specified by *host*. No further references can be made to it host after it is detached. E_INVALID_HOST is returned if HostId does not match a host that was previously attached with c2cBrokerAttach; otherwise, E_SUCCESS is returned.

int c2cBrokerKill(HostId host); Terminate the broker specified by *ost*. All database information on the broker is disposed of, and all programs connected are terminated. Most programs will never issue this call. E_INVALID_HOST is returned if HostId does not match a host that was previously attached with c2cBrokerAttach; otherwise, E_SUCCESS is returned.

CAVEId c2cGetClientId(HostId host,char *name); Query the broker *host*, and return the assigned CAVEId of the client *name*. The CAVEId is returned if the client name is found; otherwise, E_INVALID_CLIENT is returned. If the host is invalid, E_INVALID_HOST is returned.

int c2cGetClients(HostId host,int *num_clients,CaveEnv **client_list); Retrieve a list of all clients currently registered on broker *host*. The number of clients registered is returned is in *num_clients* along with an array of all the registered CaveEnv structures. E_INVALID_HOST is returned if HostId does not match a host that was previously attached to with c2cBrokerAttach; otherwise, E_SUCCESS is returned.

SessionId c2cGetSessionId(HostId host,char *name); Query the broker *host*, and return the assigned SessionId of the session *name*. The SessionId is returned if the session name is found; otherwise, E_INVALID_SESSION is returned. If the host is invalid, E_INVALID_HOST is returned.

int c2cGetSessions(HostId host,int *num_sessions,CaveSession **session_list); Retrieve a list of all sessions currently registered on broker *host*. The number of sessions registered is

returned is in *num_sessions* along with an array of all the registered CaveSession structures. E_INVALID_HOST is returned if HostId does not match a host that was previously attached to with c2cBrokerAttach; otherwise, E_SUCCESS is returned.

int c2cKillSession(HostId host,SessionId session,CAVEId cave); Not yet implemented.

CAVEId c2cRegister(HostId host,CaveEnv *env); Register the application on broker *host* with the environment *env*. All subsequent queries to the broker with respect to clients will reflect that application's presence. E_INVALID_HOST is returned if HostId does not match a host that was previously attached with c2cBrokerAttach. E_CLIENT_EXISTS is returned if the client name is already used on the broker. If there are no errors, a CAVEId (0 or higher) is returned.

int c2cSessionAttach(HostId host,SessionId session, CAVEId cave); Not yet implemented.

SessionId c2cSessionCreate(HostId host,CaveSession *ses); Create a session on broker *host* with session *ses*. All subsequent queries to the broker with respect to sessions will reflect that application session's presence. E_INVALID_HOST is returned if HostId does not match a host that was previously attached with c2cBrokerAttach. E_SESSION_EXISTS is returned if the session name is already used on the broker. If all goes well, E_SUCCESS is returned.

int c2cSessionDetach(HostId host,CAVEId cave); Not yet implemented.

int c2cSubscribe(HostId host,CAVEId datasource,StreamType stream,void *callback); Inform the broker *host* that your application would like receive stream data of type *stream* from *datasource*, and call *callback* when any incoming data is to be processed. The remote application will then start streaming the requested data to the application. E_INVALID_HOST is returned if HostId does not match a host that was previously attached with c2cBrokerAttach. E_INVALID_CAVE_ID is returned if *datasource* is invalid on *host*. E_INVALID_STREAM is returned if *datasource* does not have *stream* available. E_SUCCESS is returned upon successful subscription to *stream*.

int c2cUnregister(HostId host,CAVEId cave); Tell the broker *host* to remove an application from the client list. All subsequent queries to the broker with respect to clients will not reflect that application's presence. E_INVALID_HOST is returned if HostId does not match a host that was previously attatched with c2cBrokerAttach. E_INVALID_CAVE_ID is returned if *cave* is invalid on *host*. E_SUCCESS is returned upon successful subscription to *stream*.

int c2cUnsubscribe(HostId host,CAVEId datasource,StreamType stream,void *callback); Inform broker *host* that an application would like cancel the streaming of data of type *stream* from *datasource* with the matching callback *callback*. The remote application no longer streams data to that application. E_INVALID_HOST is returned if HostId does not match a host that was previously attatched with c2cBrokerAttach. E_INVALID_CAVE_ID is returned if *datasource* is invalid on *host*. E_INVALID_STREAM is returned if *datasource* does not have *stream* available. E_INVALID_CALLBACK is returned if a callback is given that was not used for a subscription to *stream*. E_SUCCESS is returned upon successful unsubscription to *stream*.

3.3 Data Routines

void c2cFreeDataBuffer(void *buffer); Free the data buffer *buffer* (accessed with c2cGet*n* calls) from memory.

void c2cFreePackBuffer(c2cBuffer **buffer); Free the data buffer *buffer* (accessed with c2cPack*n* calls) from memory.

void c2cGetChar(void *buffer,char *data,int size); Get *size* characters from *buffer*, and put them into *data*.

void c2cGetDouble(void *buffer,double *data,int size); Get *size* doubles from *buffer*, and put them into *data*.

void c2cGetFloat(void *buffer,float *data,int size); Get *size* floats from *buffer*, and put them into *data*.

void c2cGetInt(void *buffer,int *data,int size); Get *size* integers from *buffer*, and put them into *data*.

void c2cGetLong(void *buffer,long *data,int size); Get *size* long integers from *buffer*, and put them into *data*.

void c2cInitPackBuffer(c2cBuffer **buffer); Initialize data buffer *buffer* for future packing.

void c2cPackChar(c2cBuffer **buffer,char *data,int size); Add *size* blocks of character *data* to *buffer* for broadcast later.

void c2cPackDouble(c2cBuffer **buffer,double *data,int size); Add *size* blocks of double *data* to *buffer* for broadcast later.

void c2cPackFloat(c2cBuffer **buffer,float *data,int size); Add *size* blocks of float *data* to *buffer* for broadcast later.

void c2cPackInt(c2cBuffer **buffer,int *data,int size); Add *size* blocks of integer *data* to *buffer* for broadcast later.

void c2cPackLong(c2cBuffer **buffer,long *data,int size); Add *size* blocks of long *data* to *buffer* for broadcast later.

int c2cRegisterStream(StreamType stream,void *subscribecallback,void *unsubscribecallback); Register a stream in an application. No streams can be subscribed to until they are registered. If *subscribecallback* is supplied (it is not null), that callback will be executed every time the stream is subscribed to. If *unsubscribecallback* is supplied (it is not null), that callback will be executed every time the stream is unsubscribed to.
E_STREAM_REGISTERED is returned if the *stream* is already registered. E_SUCCESS is returned otherwise.

int c2cSendStream(CAVEId sourceid, c2cBuffer **buffer,StreamType stream); Cast a buffer of data *buffer* (previously packed with c2cPack*n* routines) of stream type *stream* to all applications subscribed to that stream with a source identifier of *sourceid*. The source identifier tells the remote processes who sent them the stream. E_INVALID_STREAM is returned if the stream was not registered; otherwise, E_SUCCESS is returned.

int c2cSendHeadTracker(float x,float y,float z,float a,float e,float r); Cast local CAVE head tracker information (passed via parameters) to all applications subscribed to the stream S_HEAD_TRACKER. E_INVALID_STREAM is returned if the stream was not registered; otherwise, E_SUCCESS is returned.

int c2cSendUser(CAVEId sourceId, CAVEUser *user); Cast local CAVE user (all trackers, buttons, joystick and world info) to all applications subscribed to the stream S_CAVE_USER. E_INVALID_STREAM is returned if the stream was not registered; otherwise, E_SUCCESS is returned.

int c2cSendWandButtons(CAVEId sourceId, int b1,int b2,int b3,int b4); Cast local CAVE wand button information (passed via parameters) to all applications subscribed to the stream S_WAND_BUTTON. E_INVALID_STREAM is returned if the stream was not registered; otherwise, E_SUCCESS is returned.

int c2cSendWandJoystick(CAVEId sourceId, float joyx,float joyy); Cast local CAVE wand joystick information (passed via parameters) to all applications subscribed to the stream S_WAND_JOYSTICK. E_INVALID_STREAM is returned if the stream was not registered; otherwise, E_SUCCESS is returned.

int c2cSendWandTracker(float x,float y,float z,float a,float e,float r); Cast local CAVE wand tracker information (passed via parameters) to all applications subscribed to the stream S_WAND_TRACKER. E_INVALID_STREAM is returned if the stream was not registered; otherwise, E_SUCCESS is returned.

int c2cSendWorldPosition(float x,float y,float z,float a,float e,float r); Cast local CAVE world position (passed via parameters) to all applications subscribed to the stream S_WORLD_POSITION. E_INVALID_STREAM is returned if the stream was not registered; otherwise, E_SUCCESS is returned.

int c2cUnregisterStream(StreamType stream); Unregister a stream with an application. No remote applications can subscribe to a stream once it is unregistered. E_INVALID_STREAM is returned if the stream was not registered; otherwise, E_SUCCESS is returned.

3.4 World Routines

void c2cDrawAllUsers(void); Draw all users that are tracked via c2cTrackUserInit.

void c2cDrawSomeUsers(int count,char *users[]); Draw a list of users that are tracked via c2cTrackUserInit. The number of users in the list as well as the application names of the users to draw are passed as parameters.

int c2cDrawUser(char *user); Draw a user that is tracked via c2cTrackUserInit. The user to draw is specified by giving his applicaion name for *user*. E_INVALID_CLIENT is returned if an invalid client name is given. E_SUCCESS is returned otherwise.

void c2cTrackUserInit(char *user,void *callback); Start tracking *user* (specified by application name). The rendering of the user can be user defined by supplying a callback. Whenever a draw command is issued, this callback will be executed. If null is given as the callback, a default stick-man representation of the user will be rendered. E_INVALID_CLIENT is returned if *user* is invalid. E_INVALID_STREAM is returned if *user* does not have tracking available. E_SUCCESS is returned upon successful tracking initializaion.

int c2cTrackUserExit(char *user,void *callback); Cancel the tracking of *user* (specified by application name). The callback of the tracked user *must* match that given when initiated. E_INVALID_CLIENT is returned if *user* is invalid. E_INVALID_STREAM is returned if *user* does not have tracking available. E_SUCCESS is returned upon successful tracking exiting.

void c2cWorldDataInit(void); Initialize necessary data structures needed for world communications. This *must* be the first CAVEcomm library call issued in an application (except for c2cReadConfigFile). If the application running is a CAVE application, it *must* occur before the call to *CAVEInit* is made. In addition, if any CAVEcomm application wishes to use the user-tracking facilities, it *must* make a call to c2cWorldDataInit (whether the application is CAVE based or not).

void c2cWorldDataSubscribe(char *user,StreamType stream,void *callback); Subscribe to *user* (specified by application name) for data stream *stream* and to call *callback* when the data is received. E_INVALID_CLIENT is returned if *user* is invalid. E_INVALID_STREAM is returned if *user* does not have *stream* available. E_SUCCESS is returned upon successful data subscription.

void c2cWorldDataUnsubscribe(char *user,StreamType stream,void *callback); Unsubscribe to the data stream *stream* that was previously subscribed to from *user* (specified by his application name) with *callback*. E_INVALID_CLIENT is returned if *user* is invalid. E_INVALID_STREAM is returned if *user* does not have *stream* available. E_INVALID_CALLBACK is returned if a callback is given that was not used for a subscription to *stream*. E_SUCCESS is returned upon successful data unsubscription.

void c2cWorldExit(void); Terminate all World functionality. This *must* be the last CAVEcomm library call issued in an application.

CAVEId c2cWorldGetClientId(char *name); Query the broker, and return the assigned CAVEId of the client *name*. The CAVEId is returned if the client name is found; otherwise, E_INVALID_CLIENT is returned.

SessionId c2cWorldGetSessionId(char *name); Query the broker, and return the assigned SessionId of the session *name*. The SessionId is returned if the session name is found; otherwise, E_INVALID_SESSION is returned.

void c2cWorldInit(int *argc,char *argv[]); Initiate all World functionality. If the CAVEcomm library is used with the CAVE library, it must occur *after* the call to *CAVEInit* is made.

int c2cWorldSendStream(c2cBuffer **buffer,StreamType stream); Cast a buffer of data *buffer* (previously packed with c2cPack*n* routines) of stream type *stream* to all applications subscribed to that stream. E_INVALID_STREAM is returned if the stream was not registered; otherwise, E_SUCCESS is returned.

3.5 Global Variables

The CAVEcomm library maintains the following global variable that the user can access:

CaveEnv c2cEnv This variable maintains all local environment information that was registered with c2cInit.

3.6 Data Structures

The CAVEcomm library has data structures that are accessible by the user.

3.6.1 CaveEnv

```
/* Structure defining CAVE environment */
typedef struct caveenv
{
  char name[C2C_NAME_SIZE]; /* Unique name for this CAVE */
  int type; /* Type of session this is (CAVE, I-Desk, etc...) */
  int id; /* Id of the client */
}CaveEnv;
```

3.6.2 CaveSession

```
/* Structure describing a CAVE session */
typedef struct cavesession
{
  CAVEId owner;         /* Who owns the session on the broker */
  char name[C2C_NAME_SIZE]; /* Unique name describing the session */
  char pathname[C2C_PATH_SIZE]; /* Where to find the application */
  char execname[C2C_EXE_SIZE]; /* Executable name */
  char args[C2C_ARGS_SIZE]; /* Command line arguments */
  int id;                       /* Id of the session */
}CaveSession;
```

3.6.3 CAVEUser

```
/* Structure defining CAVE user */
```

```
typedef struct caveuser
{
  float headx,heady,headz, /* Head tracker location */
        heada,heade,headr, /* Head tracker orientation */
wandx,wandy,wandz, /* Wand tracker location */
wanda,wande,wandr, /* Wand tracker orientation */
worldx,worldy,worldz, /* World location */
worlda,worlde,worldr; /* World orientation */
  int   but1,but2, /* Button information */
        but3,but4;
  float joyx,joyy; /* joystick information */
}CAVEUser;
```

3.6.4 c2cConfig

```
/* Structure defining a CAVEcomm session */
typedef struct c2cconfig
{
  URL broker; /* URL to connect to broker with */
  CaveEnv env; /* Local environment parameters */
  CaveSession session; /* Session info for this CAVE app */
}c2cConfig;
```

4 CAVEcomm Configuration File

The CAVEcomm library looks for a configuration file (.c2cConfig by default) that defines certain aspects of the application. The format for the configuration file is *keyword [option]*. Keywords are not case sensitive. Application definitions are as follows:

Broker *"Broker URL"* :
A known URL of the broker that one is to register and attach to. The broker URL *must* be of the format **c2cBroker:**//{**ip hostname**}:{**ip port**}/.

DebugLevel *level* :
The level for debugging the application. Debugging will be set at *level* as soon as the parser finishes processing the DebugLevel keyword.

ClientName *"Client application name"* :
The name of the application that is used when registering on the broker.

ClientAppType *type* :
The application type describing what type of application it is (CAVE, ImmersaDesk, simulation).

SessionName *"Session name"* :
Name of the session given to the broker upon session creation.

SessionPath *"Pathname to application"* :
Pathname of the executable for the session. It is given to the broker upon session creation.

SessionExe *"Session executable name"* :
Executable name of the application for the session. It is given to the broker upon session creation.

SessionArgs *"Command line arguments"* :
Command line arguments needed to run the application. They are given to the broker upon session creation.

5 CAVEcomm Constants

5.1 Stream Types

S_HEAD_TRACKER Head tracker information

S_WAND_TRACKER Wand tracker information

S_WAND_BUTTON Button information

S_WAND_JOYSTICK Joystick information

S_WORLD_POSITION CAVE location in the world

S_CAVE_USER User information (trackers/buttons etc... all in one)

5.2 Session Types

ST_CAVE CAVE session

ST_CAVE_SIMULATOR CAVE simulator session

ST_IMMERSADESK ImmersaDesk session

ST_SIMULATION Data simulation of some sort

5.3 Data Types

DT_CHAR Character data

DT_INT Integer data

DT_LONG Long integer data

DT_FLOAT Float data

DT_DOUBLE Double data

5.4 Error Messages As Seen by User

E_SUCCESS No error at all

E_BROKER_ATTACH Attach to the broker failed

E_SUBSCRIBE_NO_STREAM Subscribe to non-existant stream

E_INVALID_HOSTID Bad host id given

E_INVALID_CALLBACK Bad callback given

E_INVALID_STREAM Stream never registeged

E_STREAM_REGISTERED Stream already registered

E_INVALID_HOST Invalid host id given

E_INVALID_CAVEID Invalid CAVE id given

E_INVALID_BROKER_URL Invalid broker URL given

E_INVALID_CLIENT Invalid client name given

E_INVALID_SESSION Invalid session name given

E_CLIENT_EXISTS Client already registered on broker

E_SESSION_EXISTS Session already created on broker

E_OPEN_CONFIG_FILE Error opening configuration file

5.5 Various Array Sizes

C2C_NAME_SIZE Size of session string

C2C_URL_SIZE Size of a URL string

C2C_PATH_SIZE Size of path string

C2C_EXE_SIZE Size of exe string

C2C_ARGS_SIZE Size of argument string

5.6 Miscellaneous Constant Definintions

C2C_CONFIG_FILE Default config file to open if none is given

C2C_BROKER_URL_FILE Default filename to house broker URL

CONFIG_FILE_LINE_SIZE Max size of a file line in configuration file

CONFIG_OPTION_SIZE Max size of an option in configuration file

C2C_SUBSCRIBE Flag for World subscribe operation

C2C_UNSUBSCRIBE Flag for World unsubscribe operation

6 Sample Applications

6.1 New Samples

Multiple CAVE OpenGL Application

Makefile (CAVE to CAVE)

Make sure to replace text in double brackets (`[[ ]]`) with the appropiate information for your site.

```
LIBS =     -L. -L [[ Path to CAVE lib ]] \
    -L[[ Path to CAVEcomm lib ]] \
    -L[[ Path to nexus lib ]]
INCL = -I. -I[[ Path to CAVE includes ]] \
    -I[[ Path to CAVEcomm includes ]]
TARGET = cave1

CFLAG = -c
LFLAG = -o $(TARGET)


OBJS = main.o
CFILES = main.C

ALL_LIB = $(LIBS) -lcave_ogl -lGL -lGLU -lc2c_ogl -lX11 -lXi -lnexuslite_opt -lm
ALL_INCL = $(INCL)

$(TARGET): $(OBJS)
    CC $(LFLAG) $(OBJS) $(ALL_LIB)
main.o: main.C
    CC -c $(ALL_INCL) main.C
```

```
clean:
    rm -f *.o $(TARGET)
```

CAVE One Code

The following piece of code implements the code needed for a simple CAVE-to-CAVE application. The user in CAVE One will see the virtual representation of a user from CAVE Two. In the code included below, code specific to the CAVEcomm library is enclosed in double brackets ([[ ]]).

```
// CAVE1

//  Simple CAVE-to-CAVE application with CAVE functionality
//
//  This program contacts the CAVE application called "CAVE2" and
//  tracks its user.  You will see the remote user navigating through
//  the coordinate space with the default user representation


#include <cave_ogl.h>
#include <c2c.h>

void display(void);
void frameFunction(void);
void navigate(void);
void navTranslate(float);


// Display function, responsible for clearing the buffer,
// setting up the navigation matrix, and rendering
void display(void) {

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // Navigate the CAVE's coordinates
    CAVENavTransform();

        // Draw remote user
    c2cDrawAllUsers();
}

// Frame function, gets called once per frame.  Is responsible
// for any calculation, and/or interactions that must be performed
void frameFunction() {

    // Do navigation calculations
```

```
    navigate();
}



//////////////////////////  NAVIGATION ROUTINES ////////////////////////


// Translate the CAVE by some amount 'dz' along the direction
// the wand is being pointed
void navTranslate(float dz) {

    float tx, tz;
    float vec[3];

    CAVEGetVector(CAVE_WAND_FRONT, vec);

    tx = dz * vec[0];
    tz = dz * vec[2];

    CAVENavTranslate(tx, 0, tz);
}

// Navigate function, gets called by frame function,
// measures Joystick, and detemines if and how much
// CAVE should be navigated.
void navigate() {

    float dz;
    float azi;

    if(CAVE_JOYSTICK_Y > 0.2) {
            dz = .2*(CAVE_JOYSTICK_Y);
        navTranslate(dz);
    }
    if(CAVE_JOYSTICK_Y < -0.2) {
            dz = .2*(CAVE_JOYSTICK_Y);
        navTranslate(dz);
    }

    if(CAVE_JOYSTICK_X > 0.2) {
        azi = -CAVE_JOYSTICK_X;
        CAVENavRot(azi, 'y');
    }
    if(CAVE_JOYSTICK_X < -0.2) {
```

```
        azi = -CAVE_JOYSTICK_X;
        CAVENavRot(azi, 'y');
    }
}



//////////////////////////////////// MAIN ////////////////////////////////////


// Main function
void main(int argc,char *argv[])
{ /* main */
    int rc;                             /* Return Code */


 [[ c2cWorldDataInit();            /* Perform PreInitialization */ ]]

    CAVEConfigure(&argc,argv,NULL);        /* Get CAVE configuration info */
    CAVEInit();                         /* Start CAVE application */
    CAVEDisplay(display,0);        /* Assign drawing function */
    CAVEFrameFunction(frameFunction,0);
 [[ c2cWorldInit(&argc,argv);          /* Start CAVE-to-CAVE functionality */

    rc = -1;
    while (rc != E_SUCCESS) {
    printf("No One to ATTACH to yet\n");
        /* Track Argonne user with default user rendering */
    rc =  c2cTrackUserInit("CAVE2",NULL);
    } ]]

    while(!CAVEgetbutton(CAVE_ESCKEY)) {
    /* Do computation here or spin endlessly */

 [[ c2cUpdate();             /* This is needed every loop to
                       poll for a new event */ ]]

    } /* While */

 [[ c2cTrackUserExit("CAVE2",NULL);    /* Stop tracking Argonne user */
    c2cWorldExit();            /* End CAVE-to-CAVE functionality */ ]]

    CAVEExit();                /* End the CAVE application */
}
```

CAVE Two Code

The following piece of code implements the code needed for a simple CAVE to CAVE application. The user in CAVE Two will see the virtual representation of a user from CAVE One. In the code included below code specific to the CAVEcomm library is enclosed in double brackets (`[[ ]]`).

```
// CAVE2

//  Simple CAVE-to-CAVE application with CAVE functionality
//
//  This program contacts the CAVE application called "CAVE1" and
//  tracks its user.  You will see the remote user navigating through the
// coordinate space with the default (stick-man) user representation


#include <cave_ogl.h>
[[ #include <c2c.h> ]]

void display(void);
void frameFunction(void);
void navigate(void);
void navTranslate(float);


// Display function, responsible for clearing the buffer,
// setting up the navigation matrix, and rendering
void display(void) {

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    CAVENavTransform();

 [[ c2cDrawAllUsers();          /* Draw remote user */ ]]

}

// Frame function, gets called once per frame.  Is responsible
// for any calculation, and/or interactions that must be performed
void frameFunction() {

    navigate();
}
```

```
//////////////////////////  NAVIGATION ROUTINES ////////////////////////


// Translate the CAVE by some amount 'dz' along the direction
// the wand is being pointed
void navTranslate(float dz) {

    float tx, tz;
    float vec[3];

    CAVEGetVector(CAVE_WAND_FRONT, vec);
    tx = dz * vec[0];
    tz = dz * vec[2];
    CAVENavTranslate(tx, 0, tz);
}


// Navigate function, gets called by frame function,
// measures Joystick, and detemines if and how much
// CAVE should be navigated.
void navigate() {

    float dz;
    float azi;

    if(CAVE_JOYSTICK_Y > 0.2) {
            dz = .2*(CAVE_JOYSTICK_Y);
        navTranslate(dz);
    }
    if(CAVE_JOYSTICK_Y < -0.2) {
            dz = .2*(CAVE_JOYSTICK_Y);
        navTranslate(dz);
    }


    if(CAVE_JOYSTICK_X > 0.2) {
        azi = -CAVE_JOYSTICK_X;
        CAVENavRot(azi, 'y');
    }
    if(CAVE_JOYSTICK_X < -0.2) {
        azi = -CAVE_JOYSTICK_X;
        CAVENavRot(azi, 'y');
    }
}
```

```
//////////////////////////////// MAIN ////////////////////////////////


// Main function
void main(int argc,char *argv[])
{ /* main */
    int rc;                             /* Return Code */


 [[ c2cWorldDataInit();            /* Perform PreInitialization */ ]]

    CAVEConfigure(&argc,argv,NULL);        /* Get CAVE configuration info */
    CAVEInit();                /* Start CAVE application */
    CAVEDisplay(display,0);            /* Assign drawing function */
    CAVEFrameFunction(frameFunction,0);


 [[ c2cWorldInit(&argc,argv);          /* Start CAVE-to-CAVE functionality */

    rc = -1;
    while (rc != E_SUCCESS) {
        printf("No One to ATTACH to yet\n");
            /* Track Argonne user with default user rendering */
        rc =  c2cTrackUserInit("CAVE1",NULL);
    } ]]


    while(!CAVEgetbutton(CAVE_ESCKEY)) {
         /* Do computation here or spin endlessly */

      [[ c2cUpdate();             /* This is needed every loop to
                          poll for a new event */ ]]

    } /* While */


 [[ c2cTrackUserExit("CAVE1",NULL);        /* Stop tracking Argonne user */
    c2cWorldExit();                /* End CAVE-to-CAVE functionality */ ]]

    CAVEExit();                /* End the CAVE application */
} /* main */
```

CAVE Three Code

The following piece of code implements the code needed for a simple CAVE to CAVE application. The user in CAVE Three will see the virtual representation of a user from CAVE One and Two. In the code included below, code specific to the CAVEcomm library is enclosed in double brackets ([[ ]]).

```
// CAVE3

//  Simple CAVE-to-CAVE application with CAVE functionality
//
//  This program contacts the CAVE applications called "CAVE1" and
//  "CAVE2" and tracks its users, This application does not have
//  navigation.  You will see the remote users navigating
//  through the coordinate space with the default user representations


#include <cave_ogl.h>
[[ #include <c2c.h> ]]

void display(void);


// Display function, responsible for clearing the buffer,
// and rendering
void display(void) {

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

 [[ c2cDrawAllUsers();          /* Draw remote user */ ]]

}


void main(int argc,char *argv[])
{ /* main */
    int rc;                    /* Return Code */

 [[ c2cWorldDataInit();            /* Perform PreInitialization */ ]]

    CAVEConfigure(&argc,argv,NULL);        /* Get CAVE configuration info */
    CAVEInit();                /* Start CAVE application */
    CAVEDisplay(display,0);            /* Assign drawing function */

 [[ c2cWorldInit(&argc,argv);        /* Start CAVE-to-CAVE functionality */
```

```
    rc = -1;
    while (rc != E_SUCCESS) {
        printf("No One to ATTACH to yet\n");
        rc =  c2cTrackUserInit("CAVE1",NULL);/* Track Argonne user with default
              user rendering */
    }
      rc = -1;
    while (rc != E_SUCCESS) {
        printf("No One to ATTACH to yet\n");
        rc =  c2cTrackUserInit("CAVE2",NULL);/* Track Argonne user with default
              user rendering */
    } ]]
    while(!CAVEgetbutton(CAVE_ESCKEY)) {
        /* Do computation here or spin endlessly */

 [[ c2cUpdate();             /* This is needed every loop to
                       poll for a new event */ ]]

    } /* While */


 [[ c2cTrackUserExit("CAVE1",NULL);        /* Stop tracking Argonne user */
    c2cTrackUserExit("CAVE2",NULL);        /* Stop tracking Argonne user */
    c2cWorldExit();                /* End CAVE-to-CAVE functionality */ ]]

    CAVEExit();                /* End the CAVE application */
} /* main */
```

Inventor CAVE Application

Client C++ Code

The following piece of code implements the code needed for a simple Inventor CAVE-to-CAVE application. In the code included below, code specific to the CAVEcomm library is enclosed in double brackets ([[ ]]).

```
#include <<A HREF="clientH.html">client.h</A>>

#define WINWIDTH    512
#define WINHEIGHT    512

#define STREAM10    10
#define STREAM11    11
```

```
#define    NAVSPEED    1.0


// smObjectPosition is where the position data from the server is
// stored. This must be put into shared memory since, each of the
// graphics pipes has its own copy of the inventor objects, and
// each will need to access smObjectPosition to update its (pipe)
// cones position.
//
void *smArena;
float *smObjectPosition;


// root and the cone transform node are declared globally so that they
// can be accessed easily within the various functions
//
SoSeparator        *root;
SoTransform        *coneXForm;

c2cBuffer            *databuffer;


void INVInit(int, char**);
void initSharedMem();
void updateScene();
void initLighting();
void sendToServer();
void initCAVEcomm(int , char*[]);
void oglDraw();

void navigate();
void navTranslate(float);
void CAVENavRotateY();

void callbackFunction(void*, CAVEId);



void initSharedMem() {

    smArena = CAVEUserSharedMemory(1024);
    smObjectPosition = (float*)amalloc(3*sizeof(float), smArena);

}
```

```
void initLighting() {

    // Create some directional lights to be shared amongst
    // the scene hierarchies
    //
    SoDirectionalLight* directionalLight1 = new SoDirectionalLight;
    directionalLight1->direction.setValue(-1, -1, -1);
    SoDirectionalLight* directionalLight2 = new SoDirectionalLight;
    directionalLight2->direction.setValue(1, 1, 1);

    root->addChild(directionalLight1);
    root->addChild(directionalLight2);
}




void INVInit(int argc, char* argv[]) {

    // Initialize Inventor
    SoDB::init();

    // create the scene's root node
    //
    root = new SoSeparator;
    root->ref();


    myViewPort = new SbViewportRegion;


    // add some lighting
    initLighting();


    // Create a cone for the scene
    //
    SoSeparator *coneSeparator = new SoSeparator;
    coneXForm = new SoTransform;

    SoMaterial *coneMat = new SoMaterial;
    SoCone *myCone    = new SoCone;

    coneSeparator->addChild(coneXForm);
```

```
    coneSeparator->addChild(coneMat);
    coneSeparator->addChild(myCone);

    root->addChild(coneSeparator);


    // Create a node for the ogl rendering of the other
    // users
    SoCallback *oglCallback = new SoCallback;
    oglCallback->setCallback(oglDraw);
    root->addChild(oglCallback);


    // Initialize the CAVE-to-CAVE communications
    //

 [[ initCAVEcomm(argc, argv); ]]


}



[[
// This function is setting up two way communication between the server and
// client, although, in this application we are only doing one way communication.
// This app only receives position information of the cone from the server, but
// the code that follows will show you how to set up two way communication
// in case it is needed.
void initCAVEcomm(int argc, char* argv[]) {
    int subscribed = -1;
    int rc = -1;

    // Only set up the communication links once,
    //
    if(CAVEMasterDisplay()) {
        c2cWorldInit(&argc, argv);

        // Register a stream that sends to the server
        //
        c2cRegisterStream(STREAM11, NULL, NULL);    /* Register stream to cast */

        // Subscribe to the server, and pass a pointer to the  callback
        // function that will get executed when data is received
        // from the server.
```

```
        while(subscribed != E_SUCCESS) {
            subscribed = c2cWorldDataSubscribe("ANL_SERVER", STREAM10, callbackFunction);
            sleep(1);
            cout << "subscribed " << subscribed << endl;
        }

        // Initialize the user tracking, this way an avatar
        // of the server will be rendered when c2cDrawAllUsers()
        // gets exectued.  This function takes care of tracking
        // users in remote CAVEs for you.
        while(rc != E_SUCCESS) {
            rc = c2cTrackUserInit("ANL_SERVER", NULL);
            sleep(1);
        }
    }
    else {
        cout << "not master display" << endl;
        cout << "SUBSCRIBED SUCCESSFUL" << endl;
    }

}
]]


// A function that calls the c2c function to render
// users in remote CAVEs
//
void oglDraw() {
 [[ c2cDrawAllUsers(); ]]

}


// Gets called from c2cupdate, when data from the server
// has been received.  We store the data into a shared memory
// variable, since it will be used by each of the pipes
// to update its own scene graph
//
void callbackFunction(void* databuffer, CAVEId) {

 [[ c2cGetFloat(databuffer, smObjectPosition, 3);
    c2cFreeGetBuffer(databuffer); ]]

}
```

```
// Called from the frame function, it is called once per
// frame per pipe.  Each pipe's cone transform node gets
// updated with the data stored in shared memory.
//
void updateScene() {

    coneXForm->translation.setValue(smObjectPosition);

}


// the send to server routine that would be called
// via the frame function, if this app were sending
// data to the server.
//
void sendToServer() {

}



// A typical display function for any application that uses Inventor
// in the CAVE
//
void display() {

    XGetWindowAttributes(CAVEXdisplay, CAVEglxWindow, &winAtts);

    glPushAttrib(GL_ALL_ATTRIB_BITS);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glPushMatrix();

        myViewPort->setWindowSize(winAtts.width, winAtts.height);
        SoGLRenderAction myRenderAction(*myViewPort);

        CAVENavTransform();
        myRenderAction.apply(root);
    glPopMatrix();
    glPopAttrib();
}



// The frame function gets called once per frame per pipe.
```

```
// It's been added to the CAVEFrameFunction()
//
void frameFunction() {

    // You only want to send/receive data in the master proc
    // since that is where the c2c routines were setup
    //
    if(CAVEMasterDisplay()){

 [[ c2cUpdate();        /* This is needed every loop to
                       poll for a new event */ ]]

    }

    // THIS IS IMPORTANT
    // You only need this in a server/client that receives data.
    // Wait here for all procs to complete the above routines,
    // since c2cUpdate calls the callback function that receives
    // data from the other CAVEs.
    //
    // You do not want one of the non-master procs to continue
    // and try to update the scene using the shared mem data
    // if it hasn't been updated by the master proc yet.
    //
    CAVEDisplayBarrier();
    updateScene();

    navigate();

}




////////////////////   NAVIGATION ROUTINES  ////////////////////

void navigate() {

    float dz;

    if(CAVE_JOYSTICK_Y > 0.3) {
            dz = NAVSPEED;
        navTranslate(dz);
    }
    if(CAVE_JOYSTICK_Y < -0.3) {
```

```
            dz = -NAVSPEED;
        navTranslate(dz);
    }
    CAVENavRotateY();
}

void navTranslate(float dz) {

    float wandVector[3];

    CAVEGetVector(CAVE_WAND_FRONT, wandVector);
    wandVector[0] *= dz;
    wandVector[2] *= dz;

    CAVENavTranslate(wandVector[0], 0, wandVector[2]);

}


void CAVENavRotateY() {

    float azi;

    azi = NAVSPEED*2;

    if(CAVE_JOYSTICK_X > 0.3) {
        azi = CAVE_JOYSTICK_X * azi;
        CAVENavRot(-azi, 'y');
    }
    if(CAVE_JOYSTICK_X < -0.3) {
        azi = CAVE_JOYSTICK_X * azi;
        CAVENavRot(-azi, 'y');
    }

}



//////////////////////////////   MAIN   ////////////////////////////////

void main(int argc, char **argv) {

 [[ c2cWorldDataInit(); ]]
```

```
    CAVEConfigure(&argc,argv,NULL);
    initSharedMem();

    CAVEInit();
    CAVEInitApplication((CAVECALLBACK)INVInit, 2, argc, argv);
    CAVEDisplay(display, 0);
    CAVEFrameFunction(frameFunction, 0);

    while(!CAVEgetbutton(CAVE_ESCKEY)) {
    }


 [[ c2cTrackUserExit("ANL_SERVER",NULL);
        c2cWorldExit(); ]]

    CAVEExit();

}
```

Client Header Code

The following piece of code implements the code needed for a simple Inventor CAVE-to-CAVE application. In the code included below, code specific to the CAVEcomm library is enclosed in double brackets ([[ ]]).

```
#ifndef CLIENT_H_
#define CLIENT_H_

#include <cave_ogl.h>
#include <GL/glx.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include <stdio.h>
#include <unistd.h>
#include <stream.h>
[[ #include <c2c.h> ]]
#include <malloc.h>
#include <sys/types.h>

#include <Inventor/SoDB.h>
#include <Inventor/actions/SoGLRenderAction.h>
#include <Inventor/nodes/SoCube.h>
#include <Inventor/nodes/SoDirectionalLight.h>
#include <Inventor/nodes/SoPointLight.h>
#include <Inventor/nodes/SoLightModel.h>
#include <Inventor/nodes/SoMaterial.h>
```

```
#include <Inventor/nodes/SoTransform.h>
#include <Inventor/nodes/SoSeparator.h>
#include <Inventor/nodes/SoGroup.h>
#include <Inventor/nodes/SoSphere.h>
#include <Inventor/nodes/SoCallback.h>
#include <Inventor/nodes/SoPerspectiveCamera.h>
#include <Inventor/nodes/SoRotor.h>
#include <Inventor/nodes/SoCone.h>
#include <Inventor/nodes/SoDrawStyle.h>
#include <Inventor/nodes/SoComplexity.h>
#include <Inventor/nodes/SoScale.h>
#include <Inventor/nodes/SoSwitch.h>
#include <Inventor/nodes/SoLineSet.h>
#include <Inventor/actions/SoWriteAction.h>
#include <Inventor/SoOffscreenRenderer.h>

XWindowAttributes    winAtts;
extern Display        *CAVEXdisplay;
extern Window        CAVEglxWindow;
SbViewportRegion    *myViewPort;

extern SbViewportRegion*    myViewPort;
#endif
```

Server C++ Code

The following piece of code implements the code needed for a simple Inventor CAVE-to-CAVE application. In the code included below, code specific to the CAVEcomm library is enclosed in double brackets ([[ ]]).

```
#include "server.h"

#define WINWIDTH    512
#define WINHEIGHT    512

#define STREAM10    10
#define STREAM11    11

#define    NAVSPEED    1.0

// root and the cone transform node are declared globally so that they
// can be accessed easily within the various functions
//
SoSeparator        *root;
SoTransform        *coneXForm;
```

```
[[ c2cBuffer            *databuffer ]];

void INVInit(int, char**);
void initSharedMem();
void updateScene();
void initLighting();
void sendToClient();
void initCAVEcomm(int , char*[]);
void oglDraw();

void grabMove(SoTransform*, int);
void navigate();
void navTranslate(float);
void CAVENavRotateY();

void callbackFunction(void*, CAVEId);


void initSharedMem() {
}


// A typical initialization function that initializes Inventor
// and creates some scene objects.
//
void INVInit(int argc, char* argv[]) {

    // Initialize Inventor
    SoDB::init();

    // create the scene's root node
    //
    root = new SoSeparator;
    root->ref();


    myViewPort = new SbViewportRegion;


    // add some lighting
    initLighting();


    // create a cone for the scene
    //
```

```
    SoSeparator *coneSeparator = new SoSeparator;
    coneXForm = new SoTransform;

    SoMaterial *coneMat = new SoMaterial;
    SoCone *myCone    = new SoCone;

    coneSeparator->addChild(coneXForm);
    coneSeparator->addChild(coneMat);
    coneSeparator->addChild(myCone);

    root->addChild(coneSeparator);


    // Create a node for the ogl rendering of the other
    // users
    SoCallback *oglCallback = new SoCallback;
    oglCallback->setCallback(oglDraw);
    root->addChild(oglCallback);


    // Initialize the CAVE to CAVE communications
    //
    initCAVEcomm(argc, argv);

}


// Create some lighting nodes and add them to the scene
//
void initLighting() {

    // Create some directional lights to be shared amongst
    // the scene hierarchies
    //
    SoDirectionalLight* directionalLight1 = new SoDirectionalLight;
    directionalLight1->direction.setValue(-1, -1, -1);
    SoDirectionalLight* directionalLight2 = new SoDirectionalLight;
    directionalLight2->direction.setValue(1, 1, 1);

    root->addChild(directionalLight1);
    root->addChild(directionalLight2);
}


// This function is setting up two way communication between the server and
```

```
// client, although, in this application we are only doing one way communication.
// This app only sends position information of the cone to the client, but
// the code that follows will show you how to set up two way communication
// in case it is needed.
//
void initCAVEcomm(int argc, char* argv[]) {

    int subscribed = -1;
    int rc = -1;

    // Only set up the communication links once,
    //
    if(CAVEMasterDisplay()) {
        [[ c2cWorldInit(&argc, argv); ]]

        // Register a stream that sends to the client
        //
        [[ c2cRegisterStream(STREAM10, NULL, NULL); ]]

        // Subscribe to the client, and pass a pointer to the  callback
        // function that will get executed when data is received
        // from the client.
        [[ while(subscribed != E_SUCCESS) {
            subscribed = c2cWorldDataSubscribe("ANL_CLIENT", STREAM11, callbackFunction);
            sleep(1);
            cout << "subscribed " << subscribed << endl;
        } ]]
        // Initialize the user tracking, this way an avatar
        // of the client will be rendered when c2cDrawAllUsers()
        // gets exectued.  This function takes care of tracking
        // users in remote CAVEs for you.
        [[ while(rc != E_SUCCESS) {
            rc = c2cTrackUserInit("ANL_SERVER", NULL);
            sleep(1);
        } ]]
    }
    else  {
        cout << "not master display" << endl;
        cout << "SUBSCRIBED SUCCESSFUL" << endl;
    }
}


// A function that calls the c2c function to render
// users in remote CAVEs
```

```
//
void oglDraw() {
    [[ c2cDrawAllUsers(); ]]
}



// This callback function would get executed if the client
// were to send information to the server
//
void callbackFunction(void* databuffer, CAVEId) {

    [[ c2cFreeGetBuffer(databuffer); ]]

}



// Update the scene, pretty self explanatory.
// If button 1 is pressed and the wand is moved
// the cone moves relative to the wand's motion
//
void updateScene() {

    //move cone
    //
    if(CAVEButtonChange(1) == 1)
        grabMove(coneXForm, 1);
    if(CAVEBUTTON1)
        grabMove(coneXForm, 0);

}



// Code for grabbing and moving the cone
//
void grabMove(SoTransform* trans, int initialPress) {

    static float oldPos[3], newPos[3], diffPos[3];
    SbVec3f oldVec, newVec;
    int i;

    if(initialPress == 1) {
        CAVEGetPosition(CAVE_WAND, oldPos);
```

```
    }

    else {
        CAVEGetPosition(CAVE_WAND, newPos);
        oldVec = trans->translation.getValue().getValue();
        for(i=0; i<3; i++)
            diffPos[i] = newPos[i] - oldPos[i];
        newVec.setValue(diffPos);
        newVec += oldVec;
        trans->translation.setValue(newVec);
        for(i=0; i<3; i++)
            oldPos[i] = newPos[i];

    }
}


// This function gets called once per frame via the master display proc,
// it is responsible for getting the cone's current position and storing
// the values in a static array, and sends the values to the client
//
void sendToClient() {

    static float position[3];

    coneXForm->translation.getValue().getValue(position[0], position[1],
                          position[2]);
 [[ c2cInitPackBuffer(&databuffer); ]]        /* Initialize pack buffer */
 [[ c2cPackFloat(&databuffer, position, 3); ]]
 [[ c2cWorldSendStream(&databuffer,STREAM10); ]] /* Broadcast your stream */
 [[ c2cFreePackBuffer(&databuffer); ]]        /* Free sending data buffer */
}



// A typical display function for any application that uses Inventor
// in the CAVE
//
void display() {

    XGetWindowAttributes(CAVEXdisplay, CAVEglxWindow, &winAtts);

    glPushAttrib(GL_ALL_ATTRIB_BITS);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

```
    glPushMatrix();

        myViewPort->setWindowSize(winAtts.width, winAtts.height);
        SoGLRenderAction myRenderAction(*myViewPort);

        CAVENavTransform();
        myRenderAction.apply(root);
    glPopMatrix();
    glPopAttrib();
}


// The frame function gets called once per frame per pipe.
// It's been added to the CAVEFrameFunction()
//
void frameFunction() {

    // You only want to send/receive data in the master proc
    // since that is where the c2c routines were setup
    //
    if(CAVEMasterDisplay()){

 [[ c2cUpdate(); ]]
        sendToClient();
    }

    // THIS IS IMPORTANT
    // You only need this in a server/client that receives data.
    // Wait here for all procs to complete the above routines,
    // since c2cUpdate calls the callback function that receives
    // data from the other CAVEs.
    //
    // You do not want one of the non-master procs to continue
    // and try to update the scene using the shared mem data
    // if it hasn't been updated by the master proc yet.
    //
    CAVEDisplayBarrier();
    updateScene();

    navigate();

}
```

```
////////////////////   NAVIGATION ROUTINES  ////////////////////

void navigate() {

    float dz;

    if(CAVE_JOYSTICK_Y > 0.3) {
            dz = NAVSPEED;
        navTranslate(dz);
    }
    if(CAVE_JOYSTICK_Y < -0.3) {
            dz = -NAVSPEED;
        navTranslate(dz);
    }
    CAVENavRotateY();
}

void navTranslate(float dz) {

    float wandVector[3];

    CAVEGetVector(CAVE_WAND_FRONT, wandVector);
    wandVector[0] *= dz;
    wandVector[2] *= dz;

    CAVENavTranslate(wandVector[0], 0, wandVector[2]);

}

void CAVENavRotateY() {

    float azi;

    azi = NAVSPEED*2;

    if(CAVE_JOYSTICK_X > 0.3) {
        azi = CAVE_JOYSTICK_X * azi;
        CAVENavRot(-azi, 'y');
    }
    if(CAVE_JOYSTICK_X < -0.3) {
        azi = CAVE_JOYSTICK_X * azi;
        CAVENavRot(-azi, 'y');
    }

}
```

```
//////////////////////////////   MAIN   ////////////////////////////////

void main(int argc, char **argv) {
 [[ c2cWorldDataInit(); ]]

    CAVEConfigure(&argc,argv,NULL);
    initSharedMem();

    CAVEInit();
    CAVEInitApplication((CAVECALLBACK)INVInit, 2, argc, argv);
    CAVEDisplay(display, 0);
    CAVEFrameFunction(frameFunction, 0);

    while(!CAVEgetbutton(CAVE_ESCKEY)) {
    }

 [[ c2cTrackUserExit("ANL_SERVER",NULL);
        c2cWorldExit(); ]]
    CAVEExit();

}
```

Server Header Code

The following piece of code implements the code needed for a simple Inventor CAVE-to-CAVE application. In the code included below, code specific to the CAVEcomm library is enclosed in double brackets ([[ ]]).

```
#ifndef SERVER_H_
#define SERVER_H_

#include <cave_ogl.h>
#include <GL/glx.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include <stdio.h>
#include <unistd.h>
#include <stream.h>
[[ #include <c2c.h> ]]
#include <malloc.h>
#include <sys/types.h>

#include <Inventor/SoDB.h>
#include <Inventor/actions/SoGLRenderAction.h>
```

```
#include <Inventor/nodes/SoCube.h>
#include <Inventor/nodes/SoDirectionalLight.h>
#include <Inventor/nodes/SoPointLight.h>
#include <Inventor/nodes/SoLightModel.h>
#include <Inventor/nodes/SoMaterial.h>
#include <Inventor/nodes/SoTransform.h>
#include <Inventor/nodes/SoSeparator.h>
#include <Inventor/nodes/SoGroup.h>
#include <Inventor/nodes/SoSphere.h>
#include <Inventor/nodes/SoCallback.h>
#include <Inventor/nodes/SoPerspectiveCamera.h>
#include <Inventor/nodes/SoRotor.h>
#include <Inventor/nodes/SoCone.h>
#include <Inventor/nodes/SoDrawStyle.h>
#include <Inventor/nodes/SoComplexity.h>
#include <Inventor/nodes/SoScale.h>
#include <Inventor/nodes/SoSwitch.h>
#include <Inventor/nodes/SoLineSet.h>
#include <Inventor/actions/SoWriteAction.h>
#include <Inventor/SoOffscreenRenderer.h>

XWindowAttributes    winAtts;
extern Display        *CAVEXdisplay;
extern Window        CAVEglxWindow;
SbViewportRegion    *myViewPort;

extern SbViewportRegion*    myViewPort;
#endif
```

Supercomputer to CAVE Application (Coming Soon)

CAVE Side C Code
CAVE Side Header C Code
Supercomputer Side C Code
Supercomputer Side Header Code

6.2 Old Samples

Below is a sample application using the CAVEComm library in a CAVE application.

```
/*
  Simple CAVE-to-CAVE application with CAVE functionality

  This program contacts the CAVE application called "Argonne CAVE" and
```

```
  tracks its user.  You will see the remote user navigating through the
  coordinate space with the default (stick-man) user representation
*/
#include <cave.h>                       /* CAVE library */
#include <c2c.h>                        /* CAVE-to-CAVE library */

void UserDraw(void);                    /* Prototype for drawing function */
int connection;

void main(int argc,char *argv[])
{ /* main */
    int rc;                /* Return Code */

  c2cWorldDataInit();            /* Perform PreInitialization */
  CAVEConfigure(&argc,argv,NULL);       /* Get CAVE configuration info */
  CAVEInit();                           /* Start CAVE application */
  CAVEDisplay(UserDraw,0);              /* Assign drawing function */
  c2cWorldInit(&argc,argv);        /* Start CAVE-to-CAVE functionality */

    rc = -1;
    connection = 0;
    while (rc != E_SUCCESS)
    {
    printf("No One to ATTACH to yet\n");
    rc =  c2cTrackUserInit("Argonne CAVE2",NULL);/* Track Argonne user with default
          user rendering */
        sleep(1);
    }/* End of WHILE Loop */
    connection = 1;

  while(!getbutton(ESCKEY))             /* Wait for escape key */
  { /* While */
    /* Do computation here or spin endlessly */
    c2cUpdate(); /* <---------------------------------- NEW FUNCTION  */
  } /* While */
  c2cTrackUserExit("Argonne CAVE2",NULL);/* Stop tracking Argonne user */
  c2cWorldExit();                       /* End CAVE-to-CAVE functionality */
  CAVEExit();                           /* End the CAVE application */
} /* main */

/*
  User drawing function with CAVE-to-CAVE functionality
*/
void UserDraw(void)
{ /* UserDraw */
```

```
  float bg[3] = {0,0,0};

  c3f(bg);
  clear();                              /* Clear frame buffers */
  zclear();
    c2cDrawAllUsers();          /* Draw remote user */
} /* UserDraw */
```

Here is its corresponding .c2cConfig file:

```
c2cBroker://cavesound.mcs.anl.gov:1743/
ClientApptype CAVE
Clientname "CAVEComm Test"
```

Simulation Sample Application

Below is an example of a client/server CAVEComm application. The server casts its application name to everyone subscribed to it (in this case, everyone subscribed to stream APP_NAME_STREAM).

Server program:

```
/*
  Simple CAVE-to-CAVE application without CAVE functionality

  This server sends a stream of data (its app name) to everyone
  subscribed to it
*/
#include <c2c.h>                        /* CAVE-to-CAVE library */
#include <stdio.h>;                     /* Standard I/O library */

#define APP_NAME_STREAM 10              /* Requested stream id */

void main(int argc,char *argv[])
{ /* main */
  c2cBuffer *databuffer;                /* Data buffer to pack */
  long longdata = 12345678;
  float floatdata = 9876.543;
  double doubledata = 5738.5674;

  c2cWorldInit(&argc,argv);             /* Start CAVE-to-CAVE functionality */
  c2cRegisterStream(APP_NAME_STREAM,NULL,NULL); /* Register stream to cast */
  while(1)                              /* Wait for escape key */
  { /* While */
    c2cInitPackBuffer(&databuffer);     /* Initialize pack buffer */
```

```
    c2cPackChar(&databuffer,&c2cEnv.name,/* Pack the sending buffer */
                C2C_NAME_SIZE);
    c2cPackInt(&databuffer,&c2cEnv.type,1);
    c2cPackLong(&databuffer,&longdata,1);
    c2cPackFloat(&databuffer,&floatdata,1);
    c2cPackDouble(&databuffer,&doubledata,1);
    c2cWorldSendStream(&databuffer,APP_NAME_STREAM);/* Broadcast your stream */
    c2cFreePackBuffer(&databuffer);     /* Free sending data buffer */
    c2cUpdate();
  } /* While */
  c2cWorldExit();                       /* End CAVE-to-CAVE functionality */
} /* main */
```

This is the server's corresponding .c2cConfig file:

```
Broker c2cBroker://cavesound.mcs.anl.gov:1743/
ClientApptype SIMULATION
Clientname "Argonne Simulation Server"
```

Client program:

```
/*
  Simple CAVE-to-CAVE application without CAVE functionality

  This client receives a stream and processes it in UserData callback
*/
#include <c2c.h>                        /* CAVE-to-CAVE library */
#include <stdio.h>                      /* Standard I/I */

#define APP_NAME_STREAM 10              /* Requested stream id */

void UserData(void *data, CAVEId id); /* Callback prototype */

void main(int argc,char *argv[])
{ /* main */
  c2cWorldInit(&argc,argv);              /* Start CAVE-to-CAVE functionality */
  c2cWorldDataSubscribe("Argonne Simulation Server",  /* Subscribe to stream */
                        APP_NAME_STREAM,UserData);
  while(1)                               /* Spin endlesslyy */
  { /* While */
    c2cUpdate();        /* Do some computation here or spin endlessly */
  } /* While */
  c2cWorldExit();                       /* End CAVE-to-CAVE functionality */
} /* main */
```

```
/*
  This is the callback called when the stream data is received
*/
void UserData(void *databuffer, CAVEId id)
{ /* UserData */
  char remotedata[C2C_NAME_SIZE];       /* Data buffers */
  int type;
  long longdata;
  float floatdata;
  double doubledata;

  c2cGetChar(databuffer,remotedata,C2C_NAME_SIZE); /* Read the data */
  c2cGetInt(databuffer,&type,1);
  c2cGetLong(databuffer,&longdata,1);
  c2cGetFloat(databuffer,&floatdata,1);
  c2cGetDouble(databuffer,&doubledata,1);
  c2cFreeGetBuffer(databuffer);         /* Get rid of data buffer */
  printf("Received data character %s\n",remotedata);
  printf("                integer %d\n",type);
  printf("                   long %d\n",longdata);
  printf("                  float %f\n",floatdata);
  printf("                 double %f\n",doubledata);
  printf("from source %d\n",id);
} /* UserData */
```

This is the client's corresponding .c2cConfig file:

```
Broker c2cBroker://cavesound.mcs.anl.gov:1743/
ClientApptype SIMULATION
Clientname "Argonne Simulation Client"
```

Acknowledgments

The following people have contributed to the development of the CAVEcomm library: Terry Disz, Ian Foster, Terry Franguiadakis, Jonathan Geisler, Dan Heath, Ivan Judson, Bob Olson, Mike Papka, Mike Pellegrino, Rick Stevens, Matthew Szymanski, and Steve Tuecke.