Visualization and Virtual Reality Research Group
School of Computing

Heterogeneous Collaborative Visualization

Summary

This project has developed a library called Motlab for heterogeneous ("motley") collaborative visualization. We have implemented a proof-of-concept application from scratch, extended the existing University of Leeds pathology visualization tool, and have set up some collaboration techniques ready for research. Motlab has been used on the University of Leeds Powerwall (50 mega pixel display) to visualize map data (the whole of the UK using 1:50000 ordinance survey data) collaboratively from Powerwall-to-desktop. Motlab is cross-platform compatible, and so the desktop can be running either Windows or Linux platforms. Motlab has been released as open source software and is available on Sourceforge.net along with the new proof-of-concept application designed for collaboratively visualizing 2D map data, which includes the collaboration techniques that would be suitable for research into heterogeneous collaborative visualization. The library and software application are documented within.

Compiled by:
Trevor Dodds, Roy Ruddle and Ken Brodlie
University of Leeds

Contents

1 Heterogeneous Collaborative Visualization

Collaborative visualization takes place when multiple users visualize the same dataset, and technology can support synchronous collaboration by allowing users to see the same data at the same time. The diversity of available visualization hardware means that some collaborations will be heterogeneous, for example one user may have a Powerwall, the other may not.

This project has set up a library and proof-of-concept software application for heterogeneous collaborative visualization. The library, called Motlab, was intended to allow the easy implementation of techniques to aid collaborative visualization and was built with heterogeneous systems in mind (i.e. it expects each end user to have different resolutions/viewports, and different requirements: the user on the Powerwall may want to augment their main view of the dataset with a thumbnail view, the user on the desktop may not). The project sets the stage for the undertaking of research that develops techniques based on known principles of computer supported cooperative work and collaborative interaction in general and evaluates them in the context of Powerwall-desktop collaboration. The library supports the following functionality:

- Robust networking using TCP/IP, taking the low level operations out of the hands of the application developer.
- A simple system of 'data units' is used, specifically designed with heterogeneous collaboration in mind.
- Audio capture from the microphone.
- Audio streaming (e.g. from a microphone input or audio file).
- A simple system to transfer the audio across a network for real time audio communication.
- A simple system to transfer data files across a network.

There are different types of 'data units' available to the application developer. There are units to transfer input data, positional data (e.g. position of an object), information about a user's viewport, audio data, file transfer and more, but the number of units is kept small (11 in total) to increase simplicity in the design. A generic data unit is available for application developers to transfer their own types of data.

The following section describes how to download the library and proof-of-concept application for visualizing 2D map data. This application is called Map, and has simple functionality for visualizing

2D data along with techniques to do this collaboratively. The techniques supported are a thumbnail view, showing the whole dataset, and viewboxes representing the viewports of the other users. As other users pan and zoom around the data, the viewboxes are updated in real time, to provide awareness of what others are looking at. A developer could easily modify this program to include their own data set, which could potentially be vector data (e.g. GML, Geography Markup Language) or raster images.

Further, there is another application available that demonstrates the audio communication functionality provided by Motlab. This application is called Talker. Talker has no graphical interface, it simply runs in the background and transmits audio from the microphone to all other users connected (a maximum of ten users by default). It can therefore be ran in parallel with other applications, and it's lack of graphical interface means the source code is shorter and easier to understand.

Motlab, Map and Talker are all released as open source projects. This report contains a userguide, and a reference manual.

2 Userguide

2.1 Download Motlab

Motlab can be downloaded from `http://sourceforge.net/projects/motlab/`. Binary and source code packages are available.

2.2 Build Motlab

The library and example programs can be used by downloading the binary packages. There is an alternative option to build from source. This can be done using one of the source packages available, or by using the latest source revision. Both the source packages and the repository contain the Motlab library code and some example programs. To get the latest source revision, checkout the subversion repository using:

```
$ svn checkout https://motlab.svn.sourceforge.net/svnroot/motlab
```

The library can be built using the CMake cross-platform make utility as follows (see `http://www.cmake.org/`). First, navigate to the build directory (i.e. motlab/trunk/build if using a working copy of the subversion repository), then run:

```
$ cmake ../src

$ make
```

This will create a subdirectory called 'lib' which will contain the .so library file. This can be installed system wide (put in /usr/lib or /usr/local/lib) or used from it's current location. To use from it's current location, set LD_LIBRARY_PATH to point to the location of the library.

Windows users: The library has been tested and compiled using Microsoft Visual Studio 2008. The project files can be generated using CMake. You will need to define MOTLAB_EXPORTS when compiling the library (DLL), but not when building the applications that use this library. Install OpenAL before building Motlab.

2.3 Build example apps

The Motlab library comes with two example applications, 'Talker' and 'Map'. Talker provides an example of using Motlab for audio transmission across a network. The advantage of doing this through Motlab is the developer has complete control over the audio data, i.e. they can choose to store the data to disk and playback at a later time, for asynchronous collaboration. Note that Talker just uses the audio functionality, and has no graphical output. This means it could be easily run in parallel with another application. It does not require any push-to-talk operation, the audio is transmitted continuously.

Map provides a collaborative data visualization tool. By default it is set up to visualize 2D data (e.g. map data). The map datafile could be vector based (e.g. GML, Geography Markup Language) or raster. Due to licensing restrictions a datafile is not provided, but the example program draws some simple polygon data using OpenGL. The source code provides comments to indicate where the data should be loaded and rendered, so a developer could easily adapt this. In addition, since the rendering is performed using OpenGL, it would be easy to adapt this example to a 3D visualization.

Map provides some basic collaboration tools: a thumbnail view showing the whole dataset, and viewboxes rendered onto the thumbnail showing the viewports of each other. As users pan and zoom around the dataset, the viewboxes are updated in real time. Map also takes advantage of the audio functionality in Motlab, and provides a push-to-talk operation.

2.3.1 Talker

Build talker by navigating into the build directory (e.g. talker/trunk/build if using a working copy of the subversion repository) and typing:

```
$ cmake ../src
$ make
```

Run a talker server using:

```
$ ./talk -s
```

Run a client using:

```
$ ./talk [-i <ip_address>]
```

The optional argument `<ip_address>` is the IP address or hostname of the sever (default localhost). If you run two clients on the same machine, you will be able to hear yourself (the audio will feed back). If you cannot hear yourself, check your audio settings.

2.3.2 Map

Build map by navigating into the build directory (e.g. map/trunk/build if using a working copy of the subversion repository) and typing:

```
$ cmake ../src
$ make
```

Run a map server using:

```
$ ./map -s
```

Run a map client using:

```
$ ./map [-i <ip_address>]
```

The optional argument `<ip_address>` is the IP address or hostname of the sever (default localhost). The application can be tested on the same machine by running two clients. Two graphical windows will open, displaying some example polygon data. The viewbox of the other window will be visible in the thumbnail view. The controls for navigation are as follows:

- **Page up** — Zoom in
- **Page down** — Zoom out
- **Arrow keys** — Pan up/down/left/right
- **Space** — Transmit audio (push-to-talk)

2.4 Using Motlab

The Motlab library was designed to allow the easy implementation of techniques to aid collaborative visualization and was built with heterogeneous systems in mind. The library uses a simple system of data units to transmit different types of information. The library converts these units into a stream of bytes for transmission over TCP/IP, and deals with converting them back into units for processing by the application. The library is designed for use with a client-server architecture, so that when shared objects are being manipulated simultaneously by multiple clients, the server can maintain the shared state of the environment and resolve any conflicts.

2.4.1 Networking tutorial

Motlab has three main classes for dealing with networking: `Net`, `Client` and `Server`. `Client` and `Server` deal with the low level operations for sending and receiving data between the client and server processes. `Net` translates the data units into network safe bytes ready for transmission, and translates the bytes received back into data units. All the application developer needs to see are the high level data units containing information that is useful to them.

Motlab has a built-in queue for sending data units. A call to `client.sendData();` by the client process will attempt to send all the data in the queue, but if it is unsuccessful (e.g. the socket is blocked, or not all the data could be sent) then some data will remain on the queue ready for the next attempt. The application designer can decide when units should be queued and when Motlab should attempt to transmit the contents of the queue. This allows the designer to control the rate of transmission.

The server process receives data units from the network into a single queue for processing, using `server.recvAll();`. The server process has a send queue for each client so that data can optionally be sent to individual clients, and not transmitted to all users.

The typical usage of Motlab is as follows. NB: `net`, `client`, and `server` are instances of `Net`, `Client` and `Server` respectively, and `unit` is a variable of type `Unit`.

1. The client process has some information to send. The client populates a data unit (a variable named `unit`) with the relevant information.

2. The client process calls `net.addUnit(unit, client);` to translate the data unit into network safe bytes and add the resulting data to the send queue. When the client process is ready to transmit, it calls `client.sendData();`.

3. The server process gets data from the network using `server.recvAll();` which adds any available data to the receive queue. The server gets the units from the queue by using `unit = server.getDataUnit(net);`.

4. The server processes the unit as necessary, and sends back any relevant information (e.g. state update) by creating its own data units. The server sends these units to the clients using `net.addUnitAll(unit, server, -1);` to add the data to all client's queues, and then `server.sendAll();` to attempt to send all data from the queues.

5. The client process gets any data from the network using `unit = client.recvDataUnit(net);` and processes it accordingly.

The following code shows an example of the above process, from the Map application. The first extract is from the input loop in the `Clientcontrol` class (see clientcontrol.cpp). The client process transmits a user's keypresses to the server. This system was designed so that multiple users could control the same object, and the server would process the keypresses and maintain the shared state of the environment. It is used here for controlling the position of the user's map object (this information is used by other clients to draw a visual representation of each client's viewport over a thumbnail view of the map data, see 2.3). The server transmits updates about the map's position at a given frequency (20Hz), and only transmits when the state has changed since the last transmission. The clients can optionally maintain their own local version of the object if latency is an issue.

```
void Clientcontrol::inputloop()
{
  if (input.getGrabbed()) keys = input.check(keys);

  if (keys != keysOld) {
    // if keys have changed, send them to server

    // ... snip code that starts audio capture if using push-to-talk ...
```

```
    keysOld = keys;

    // make unit (part 1)
    unit.keys.flag = UNIT_KEYS; // specify the type of the data unit
    unit.keys.id = myId; // populate the data unit
    unit.keys.bits = keys;

    // add the data unit to the send queue (part 2)
    net.addUnit(unit, client);

  }
}
```

The data is transmitted by the client at the end of the client's network loop using `client.sendData();` (part 2). The server receives the data from the network in the class `Servercontrol` (see servercontrol.cpp). It does this during it's network loop, an extract of which is shown below. The server also checks for closed connections and new connections during the network loop, but the code for that has been omitted here for clarity.

```
void Servercontrol::networkloop()
{
  // ... snip code for dealing with new and closed connections ...
  server.doSelect(); // poll the socket

  // receive data from the network into receive queue (part 3)
  server.recvAll();

  unit = server.getDataUnit(net); // get the next data unit from the queue

  while (unit.flag > -1) { // if we have a data unit
    process(unit); // process it
    unit = server.getDataUnit(net); // get the next data unit
  }
```

The server can decide when it's time to send out the shared state (and only send updates for objects that have changed, to save network bandwidth). The full conditional code is not shown below to make it clearer — see networkloop() in servercontrol.cpp for the full code.

```
  // ... if it's time to send out positions ...
  // ... (e.g. a 20th of a second has passed) ...
  // ... and if the positions have changed ...
  unit.position.flag = UNIT_POSITION;
  unit.position.id = i;
  unit.position.x = map[i].getX(); // get the coordinates of the map object
  unit.position.y = map[i].getY();
  unit.position.z = map[i].getZoom();

  // part 4 -- add the data to all the queues
  net.addUnitAll(unit, server, -1);
  // ... end if ...

  server.sendAll(); // attempt to send all data units from the queues
} // end networkloop()
```

The client's network loop is shown below. This deals with receiving data from the server.

```
void Clientcontrol::networkloop()
{
  client.doSelect(); // poll the socket

  // receive a data unit from the network (part 5)
  unit = client.recvDataUnit(net);

  while (unit.flag > -1) { // if there is a data unit
    process(unit); // process it
    client.doSelect(); // poll the socket again (is it still readable?)
    unit = client.recvDataUnit(net); // continue receiving data units
  }

  client.sendData();
}
```

Both the client and server call a function called `process();` to process their data units. Those functions simply look at the data unit flag to determine what the unit is, and then interpret the data accordingly. For example, when the server detects a new client has connected, it assigns it an ID by transmitting the unit called `UNIT_ASSIGN`. The client receives the data unit and processes it as follows:

```
void Clientcontrol::process(Unit unit)
{
  switch (unit.flag) {
    case UNIT_ASSIGN:
      myId = unit.assign.id;
      cout << "Received my client ID: " << myId << endl;
      users++; // increment the number of users connected
      break;

    case ...
      // ... continue for other units ...

    default:
      cerr << "Error, flag not found: " << unit.flag << endl;
  }
}
```

This completes the tutorial for using Motlab to send and receive data. For full documentation on the `Net`, `Client` and `Server` classes, and a complete reference of all available data units, refer to the Doxygen documentation (enclosed).

3 Reference manual

The following pages contain a reference manual for the Motlab library. This completes the documentation of the library. For any queries about licensing issues (e.g. releasing the library under a different license) please contact us.

Motlab

1.0

Generated by Doxygen 1.7.3

Wed Apr 27 2011 14:23:26

Contents

Chapter 1

Class Index

1.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 2

Class Index

2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 3

Class Documentation

3.1 Client Class Reference

`#include <client.h>`

Public Member Functions

- **Client** ()
- void **init** (**Outverbose** &o, int p, int f, int u[ ])
- int **getBufferSpace** ()
- bool **openConnection** (const char ∗ip)
- int **closeConnection** ()
- bool **getConnected** () const
- int **addData** (uint16_t data)
- int **addData** (uint32_t data)
- int **addData** (const char ∗data, int amount)
- void **doSelect** ()
- void **sendData** ()
- **Unit recvDataUnit** (**Net** &net)
- int **getStatSent** ()
- int **getStatRecvd** ()

Protected Attributes

- **Outverbose** ∗ **out**

Private Member Functions

- int **findUnit** (char ∗, int)

Private Attributes

- char **ipAddress** [16]
- struct hostent * **host**
- struct sockaddr_in **serverAddress**
- int **sockfd**
- fd_set **readSocks**
- fd_set **writeSocks**
- fd_set **exceptionSocks**
- int **numSocksReadable**
- int **PORT**
- int **flagsize**
- int **unitsize** [UNITS]
- int **statSent**
- int **statRecvd**
- **Ringbuf sendBuf**
- bool **connected**
- int **recvSize**

3.1.1 Detailed Description

A client class.

Deals with sending and receiving data to and from the server. The data to be sent is buffered by **addData()** (p. 6) and transmitted upon a call to **sendData()** (p. 8).

See also

addData() (p. 6)
sendData() (p. 8)

3.1.2 Constructor & Destructor Documentation

3.1.2.1 Client::Client ( )

Constructor.

3.1.3 Member Function Documentation

3.1.3.1 int Client::addData (uint16_t *data*)

Add integer (2 bytes) to buffer for sending.

Parameters

data	the integer data (2 bytes)

3.1.3.2 int Client::addData (uint32_t *data*)

Add integer (4 bytes) to buffer for sending.

Parameters

data	the integer data (4 bytes)

3.1.3.3 int Client::addData (const char * *data*, int *amount*)

Add binary data to buffer for sending.

Parameters

data	the binary data.
amount	number of bytes in array.

3.1.3.4 int Client::closeConnection ( )

Close connection to server.

3.1.3.5 void Client::doSelect ( )

Check if the socket can be read from / written to, using select() system call. Call this once per network loop, at the start of the loop.

3.1.3.6 int Client::findUnit (char *, int) [private]

Find a data unit amongst the bytes.

3.1.3.7 int Client::getBufferSpace ( )

Find space remaining on buffer.

3.1.3.8 bool Client::getConnected ( ) const

Are we connected? True/false.

3.1.3.9 int Client::getStatRecvd ( )

Get stats on amount of data received.

Returns

bytes received.

3.1.3.10 int Client::getStatSent ( )

Get stats on amount of data sent.

Returns

bytes sent.

3.1.3.11 void Client::init (Outverbose & *o*, int *p*, int *f*, int *u[]*)

Initialisation function.

Parameters

o	the output object.
p	the port number.
f	the data flag size.
u	the data unit sizes.

3.1.3.12 bool Client::openConnection (const char * *ip*)

Set up socket and connect to server.

Parameters

ip	IP address.

3.1.3.13 Unit Client::recvDataUnit (Net & *net*)

Receive data from socket. Call **doSelect()** (p. 7) first.

See also

doSelect() (p. 7)

Parameters

net	shared instance of net.

3.1.3.14 void Client::sendData ( )

Send data from buffer. Call **doSelect()** (p. 7) first.

See also

doSelect() (p. 7)

3.1.4 Member Data Documentation

3.1.4.1 bool Client::connected [private]

Are we connected?

3.1.4.2 fd_set Client::exceptionSocks [private]

Sockets to watch for exceptions on.

3.1.4.3 Client::flagsize [private]

The flag size for the data units.

3.1.4.4 struct hostent* Client::host [private]

Host entity.

3.1.4.5 char Client::ipAddress[16] [private]

Store IP address of server in case need to reconnect.

3.1.4.6 int Client::numSocksReadable [private]

Number of sockets readable.

3.1.4.7 Outverbose* Client::out [protected]

The class used for sending output to the user.

3.1.4.8 Client::PORT [private]

The port number.

3.1.4.9 fd_set Client::readSocks [private]

Sockets to read from.

3.1.4.10 int Client::recvSize [private]

The amount of data to receive.

3.1.4.11 Ringbuf Client::sendBuf [private]

The send buffer.

3.1.4.12 struct sockaddr_in Client::serverAddress [private]

Server (p. 35) address information.

3.1.4.13 int Client::sockfd [private]

Socket file descriptor.

3.1.4.14 Client::statRecvd [private]

The total amount of bytes received, used for making statistics on the data transfer rate.

3.1.4.15 Client::statSent [private]

The total amount of bytes sent, useful for making statistics on the data transfer rate.

3.1.4.16 Client::unitsize[UNITS] [private]

The size of each data unit.

3.1.4.17 fd_set Client::writeSocks [private]

Sockets to write to.

The documentation for this class was generated from the following file:

- include/motlab/client.h

3.2 Net Class Reference

`#include <net.h>`

Public Member Functions

- **Net** ()
- ~**Net** ()
- void **init** (**Outverbose** &o)
- void **setAudioDataSize** (int s)

- int * **getUnitsize** ()
- int **getFlagsize** ()
- **Unit bytesToUnit** (int unitId, char *data)
- int **unitToBytes** (**Unit** unit, char *data)
- void **addUnit** (**Unit** unit, **Client** &client)
- void **addUnit** (int cid, **Unit** unit, **Server** &server)
- void **addUnitAll** (**Unit** unit, **Server** &server, int source)

Private Member Functions

- int **readInt** (char *&data)
- float **readFloat1** (char *&data)
- float **readFloat2** (char *&data)
- int **writeShortInt** (**Ringbuf** &buf, uint16_t data)
- int **writeInt** (**Ringbuf** &buf, uint32_t data)
- int **writeFloat1** (**Ringbuf** &buf, float data)
- int **writeFloat2** (**Ringbuf** &buf, float data)

Private Attributes

- **Outverbose** * **out**
- int **flagsize**
- int **unitsize** [UNITS]
- int **maxSize**
- int **maxClients**
- int **audioDataSize**
- char * **audioData**

3.2.1 Detailed Description

This class takes a data unit and converts it to bytes to be sent using the client/server classes. It also takes bytes received from the client/server classes and converts them back into data units.

See also

Client (p. 5)
Server (p. 35)

3.2.2 Constructor & Destructor Documentation

3.2.2.1 Net::Net ( )

Constructor.

3.2.2.2 Net::~Net ( )

Destructor.

3.2.3 Member Function Documentation

3.2.3.1 void Net::addUnit (Unit *unit,* Client & *client*)

Add a data unit ready to be sent.

This is used by the client.

Parameters

unit	the data unit.
client	the client instance.

3.2.3.2 void Net::addUnit (int *cid,* Unit *unit,* Server & *server*)

Add a data unit ready to be sent to a particular client.

This is used by the server.

Parameters

cid	the client ID.
unit	the data unit.
server	the server instance.

3.2.3.3 void Net::addUnitAll (Unit *unit,* Server & *server,* int *source*)

Add unit to all client buffers, optionally excluding the source client (where data originates from).

Source can be set to -1 to send out to them all. This is used by the server.

Parameters

unit	the data unit
server	the server instance.
source	the ID of the client that sent this data to the server, or -1 to sent out to all.

3.2.3.4 Unit Net::bytesToUnit (int *unitId,* char * *data*)

Convert bytes into unit.

Parameters

unitId	the flag.
data	the bytes of data.

Returns

the data unit.

3.2.3.5 int Net::getFlagsize ( )

Get the size of the data unit flag.

Returns

the flag size.

3.2.3.6 int* Net::getUnitsize ( )

Get the size of the data units.

Returns

a pointer to an array of data unit sizes, where the arrays length is UNITS.

3.2.3.7 void Net::init (Outverbose & *o*)

Initialisation function.

Parameters

o	the output object.

3.2.3.8 float Net::readFloat1 (char *& *data*) [private]

Read a 4 byte float from the byte stream. This is not network safe, since some architectures may represent floats differently. However it uses less bytes than the network safe counterpart.

Parameters

data	the data to read.

See also

readFloat2() (p. 14).

3.2.3.9 float Net::readFloat2 (char *& *data*) [private]

Read a 4 byte integral part followed by 4 byte fraction (* 10000). This is network safe, but uses more bytes than the alternative.

Parameters

data	the data to read.

See also

readFloat1() (p. 13).

3.2.3.10 int Net::readInt (char *& *data*) [private]

Read an integer from the 4 bytes of data.

Parameters

data	the data to read.

3.2.3.11 void Net::setAudioDataSize (int *s*)

Set the size of a block of audio data. This should be set to talk.getChunkBytes().

Parameters

s	the size of the audio data chunk.

See also

Talk (p. 54)

3.2.3.12 int Net::unitToBytes (Unit *unit,* char * *data*)

Convert unit into network safe bytes.

Parameters

unit	the unit.
data	the bytes of data.

Returns

size of data in bytes.

3.2.3.13 int Net::writeFloat1 (Ringbuf & *buf,* float *data*) [private]

Write a 4 byte float to the buffer. This is not network safe, since some architectures may represent floats differently. However, it uses less bytes than the network safe counterpart.

Parameters

buf	the buffer to write to.
data	the data to write.

See also

writeFloat2() (p. 15).

3.2.3.14 int Net::writeFloat2 (Ringbuf & *buf,* float *data*) [private]

Write a 4 byte integral part followed by 4 byte fraction ($*$ 10000) to the buffer. This is network safe, but uses more bytes than the alternative.

Parameters

buf	the buffer to write to.
data	the data to write.

See also

writeFloat1() (p. 15).

3.2.3.15 int Net::writeInt (Ringbuf & *buf,* uint32_t *data*) [private]

Write a 4 byte integer to the buffer.

Parameters

buf	the buffer to write to.
data	the data to write.

3.2.3.16 int Net::writeShortInt (Ringbuf & *buf,* uint16_t *data*) [private]

Write a 2 byte integer to the buffer.

Parameters

buf	the buffer to write to.
data	the data to write.

3.2.4 Member Data Documentation

3.2.4.1 char* Net::audioData [private]

A pointer to a chunk of audio data.

3.2.4.2 int Net::audioDataSize [private]

The size of the audio data chunks (bytes).

3.2.4.3 Net::flagsize [private]

The size of the flag used in the data units (bytes).

3.2.4.4 Net::maxClients [private]

The maximum number of clients.

3.2.4.5 Net::maxSize [private]

The maximum data unit size (bytes).

3.2.4.6 Outverbose* Net::out [private]

A pointer to the shared output object.

3.2.4.7 Net::unitsize[UNITS] [private]

The various sizes of the data units (bytes).

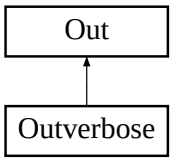
The documentation for this class was generated from the following file:

- include/motlab/net.h

3.3 Out Class Reference

`#include <outstd.h>`

Inheritance diagram for Out:



Public Member Functions

- **Out** ()
- ∼**Out** ()
- void **init** ()
- void **endWin** ()
- int **getWidth** ()
- int **getHeight** ()
- void **addCh** (int y, int x, char c)
- void **add** (const char ∗msg)
- void **add** (std::string msg)
- void **add** (char c)
- void **add** (int n)
- void **addln** (int n)
- void **addln** (const char ∗msg)
- void **addln** (std::string msg)
- void **refreshScreen** ()
- int **getCh** ()
- char **get** ()
- char **getIn** (int c)
- void **putIn** (const char ∗str)
- void **scrollInput** ()
- void **scrollOutput** ()
- void **setMenu** (const char ∗str)
- void **setCursor** (bool c)
- void **openLog** (const char ∗log)
- void **closeLog** ()
- bool **getOpenedLog** ()

Private Member Functions

- **Out** (const **Out** &o)
- **Out** & **operator=** (const **Out** &o)

Private Attributes

- bool **initialised**
- int **outputX**
- int **outputY**
- int **inputX**
- int **inputY**
- int **menuPos**
- int **cursor**
- char **outputLines** [MAX_LINES][MAX_COLS]
- char **inputLines** [MAX_LINES][MAX_COLS]
- int **outputLine**

- int **outputChar**
- int **inputLine**
- int **inputChar**
- int **cols**
- int **lines**
- std::ofstream **logfile**
- bool **openedLog**

3.3.1 Detailed Description

The output class.

This can be implemented using Curses, OpenGL, or simply using the standard output stream.

3.3.2 Constructor & Destructor Documentation

3.3.2.1 Out::Out (const Out & *o*) [private]

The constructor called with an instance of **Out** (p. 16).

Declared private, but not defined, to prevent copying.

Parameters

o	the output instance to be copied (not used).

3.3.2.2 Out::Out ( )

Constructor.

3.3.2.3 Out::~Out ( )

Destructor.

3.3.3 Member Function Documentation

3.3.3.1 void Out::add (const char * *msg*)

Add character string to output.

Parameters

msg	the output message.

3.3.3.2 void Out::add (std::string *msg*)

Add string to output.

Parameters

msg	the output message.

3.3.3.3 void Out::add (char *c*)

Add character to output.

Parameters

c	the character.

3.3.3.4 void Out::add (int *n*)

Add integer to output.

Parameters

n	the integer.

3.3.3.5 void Out::addCh (int *y*, int *x*, char *c*)

Add character to ouput.

Parameters

y	y coordinate.
x	x coordinate.
c	the character.

3.3.3.6 void Out::addln (int *n*)

Add integer to output, followed by newline.

Parameters

n	the integer.

3.3.3.7 void Out::addln (const char * *msg*)

Add character string to output, followed by newline.

Parameters

msg	the output message.

3.3.3.8 void Out::addln (std::string *msg*)

Add string to output, followed by newline.

Parameters

msg	the output message.

3.3.3.9 void Out::closeLog ( )

Close log file.

3.3.3.10 void Out::endWin ( )

End output.

3.3.3.11 char Out::get ( )

Get input into the inputLines array.

Designed to call **getIn()** (p. 21) with **getCh()** (p. 20).

Returns

the input character, which may have been modified by **getIn()** (p. 21).

See also

getCh() (p. 20).
getIn() (p. 21).

3.3.3.12 int Out::getCh ( )

Get character from keyboard.

Note

If implemented using curses getch() then this also performs a refresh.

Returns

the input character.

3.3.3.13 int Out::getHeight ( )

Get height of output.

Returns

height of output (number of rows).

3.3.3.14 char Out::getIn (int *c*)

Get input into the inputLines array.

Parameters

c	the character to put into the inputLines array.

Returns

the input character (may have been modified, e.g. changing carriage return and newline character to '
').

3.3.3.15 bool Out::getOpenedLog ( )

Is the log file open?

Returns

true if open, false otherwise.

3.3.3.16 int Out::getWidth ( )

Get width of output.

Returns

width of output (number of columns).

3.3.3.17 void Out::init ( )

Initialisation.

3.3.3.18 void Out::openLog (const char * *log*)

Open log file.

Parameters

log	the name of the log file.

3.3.3.19 Out& Out::operator= (const Out & *o*) [private]

The overload of the assignment operator.

Declared private, but not defined, to prevent assignment.

Parameters

o	the output instance (not used).

3.3.3.20 void Out::putIn (const char * *str*)

Put some stuff into the inputLines array, as if the user typed it themselves.

Parameters

str	the string to add to the inputLines array.

3.3.3.21 void Out::refreshScreen ( )

Refresh the output screen.

Note

This is needed if using curses. However, it is not needed if using curses getch() which also performs a refresh.

See also

getCh() (p. 20).

3.3.3.22 void Out::scrollInput ( )

Shift input lines up one.

3.3.3.23 void Out::scrollOutput ( )

Shift output lines up one.

3.3.3.24 void Out::setCursor (bool *c*)

Turn cursor on/off.

Parameters

c	boolean representing on or off.

3.3.3.25 void Out::setMenu (const char * *str*)

Set the menu string.

Parameters

str	the menu string.

3.3.4 Member Data Documentation

3.3.4.1 Out::cols [private]

The width of the output in columns.

3.3.4.2 int Out::cursor [private]

The cursor type.

3.3.4.3 bool Out::initialised [private]

Are we initialised?

3.3.4.4 Out::inputChar [private]

The column number (number of characters so far) we are using for the inputLines array.

See also

inputLines (p. 23).

3.3.4.5 Out::inputLine [private]

The line number we are using for the inputLines array.

See also

inputLines (p. 23).

3.3.4.6 char Out::inputLines[MAX_LINES][MAX_COLS] [private]

Array storing the input lines.

3.3.4.7 Out::inputX [private]

The x coordinate of the input cursor.

3.3.4.8 Out::inputY [private]

The y coordinate of the input cursor.

3.3.4.9 Out::lines [private]

The width of the output in lines.

3.3.4.10 std::ofstream Out::logfile [private]

The filename of a logfile.

3.3.4.11 int Out::menuPos [private]

The menu position.

3.3.4.12 bool Out::openedLog [private]

Store if log file is opened.

3.3.4.13 Out::outputChar [private]

The column number (number of characters so far) we are using for the outputLines array.

See also

outputLines (p. 24).

3.3.4.14 Out::outputLine [private]

The line number we are using for the outputLines array.

See also

outputLines (p. 24).

3.3.4.15 char Out::outputLines[MAX_LINES][MAX_COLS] [private]

Array storing the output lines.

3.3.4.16 Out::outputX [private]

The x coordinate used for output.

3.3.4.17 Out::outputY [private]

The y coordinate used for output.

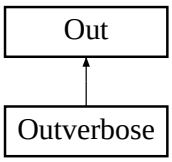
The documentation for this class was generated from the following file:

- include/outstd.h

3.4 Outverbose Class Reference

`#include <outverbose.h>`

Inheritance diagram for Outverbose:



Public Member Functions

- **Outverbose** ()
- ∼**Outverbose** ()
- void **setVerbosity** (verboseEnum v)
- void **outputVerbosity** (verboseEnum v)
- void **add** (const char ∗msg, verboseEnum v)
- void **add** (std::string msg, verboseEnum v)
- void **add** (char c, verboseEnum v)
- void **add** (int i, verboseEnum v)
- void **addln** (int i, verboseEnum v)
- void **addln** (const char ∗msg, verboseEnum v)
- void **addln** (std::string msg, verboseEnum v)
- void **adderr** (const char ∗msg, verboseEnum v)
- void **adderr** (std::string msg, verboseEnum v)
- void **adderr** (char c, verboseEnum v)
- void **adderr** (int i, verboseEnum v)
- void **adderrln** (int i, verboseEnum v)
- void **adderrln** (const char ∗msg, verboseEnum v)
- void **adderrln** (std::string msg, verboseEnum v)
- **Outverbose** & **operator**<< (const verboseEnum v)

- **Outverbose** & **operator**<< (const char *str)
- **Outverbose** & **operator**<< (const std::string &str)
- **Outverbose** & **operator**<< (const char c)
- **Outverbose** & **operator**<< (const int i)

Private Member Functions

- bool **checkVerbosity** (verboseEnum v)

Private Attributes

- verboseEnum **verbose**
- verboseEnum **outVerbosity**

3.4.1 Detailed Description

The **Outverbose** (p. 25) class.

This extends the output class to deal with stream operators, error streams, and verbosity.

3.4.2 Constructor & Destructor Documentation

3.4.2.1 Outverbose::Outverbose ( )

Constructor.

3.4.2.2 Outverbose::~Outverbose ( )

Destructor.

3.4.3 Member Function Documentation

3.4.3.1 void Outverbose::add (const char * *msg*, verboseEnum *v*)

Output a message.

Parameters

msg	the character string to output.
v	the programmer's verbosity level.

3.4.3.2 void Outverbose::add (std::string *msg*, verboseEnum *v*)

Output a message.

Parameters

msg	the string to output.
v	the programmer's verbosity level.

3.4.3.3 void Outverbose::add (char *c*, verboseEnum *v*)

Output a character.

Parameters

c	the character to output.
v	the programmer's verbosity level.

3.4.3.4 void Outverbose::add (int *i*, verboseEnum *v*)

Output an integer.

Parameters

i	the integer to output.
v	the programmer's verbosity level.

3.4.3.5 void Outverbose::adderr (const char * *msg*, verboseEnum *v*)

Output a message, and copy to stderr.

Parameters

msg	the character string to output.
v	the programmer's verbosity level.

3.4.3.6 void Outverbose::adderr (std::string *msg*, verboseEnum *v*)

Output a message, and copy to stderr.

Parameters

msg	the string to output.
v	the programmer's verbosity level.

3.4.3.7 void Outverbose::adderr (char *c*, verboseEnum *v*)

Output a character, and copy to stderr.

Parameters

c	the character to output.
v	the programmer's verbosity level.

3.4.3.8 void Outverbose::adderr (int *i,* verboseEnum *v*)

Output an integer, and copy to stderr.

Parameters

i	the integer to output.
v	the programmer's verbosity level.

3.4.3.9 void Outverbose::adderrln (const char * *msg,* verboseEnum *v*)

Output a message and newline, and copy to stderr.

Parameters

msg	the character string to output.
v	the programmer's verbosity level.

3.4.3.10 void Outverbose::adderrln (std::string *msg,* verboseEnum *v*)

Output a message and newline, and copy to stderr.

Parameters

msg	the character string to output.
v	the programmer's verbosity level.

3.4.3.11 void Outverbose::adderrln (int *i,* verboseEnum *v*)

Output an integer and newline, and copy to stderr.

Parameters

i	the integer to output.
v	the programmer's verbosity level.

3.4.3.12 void Outverbose::addln (int *i,* verboseEnum *v*)

Output an integer and newline.

Parameters

i	the integer to output.
v	the programmer's verbosity level.

3.4.3.13 void Outverbose::addln (const char * *msg*, verboseEnum *v*)

Output a message and newline.

Parameters

msg	the character string to output.
v	the programmer's verbosity level.

3.4.3.14 void Outverbose::addln (std::string *msg*, verboseEnum *v*)

Output a message and newline.

Parameters

msg	the string to output.
v	the programmer's verbosity level.

3.4.3.15 bool Outverbose::checkVerbosity (verboseEnum *v*) [private]

Check if the output should be displayed, based on the user's setting (verbose) and the programmer's setting v, which would usually be set to outVerbosity. If the programmer's setting is louder or equal to the user's setting, then the output should be displayed, (i.e. if (v $>$ 1 - verbose) return true; else return false;).

Parameters

v	the programmer's verbosity setting, which should be set to outVerbosity under normal circumstances.

Returns

true if output should be displayed, or false if it should be suppressed.

3.4.3.16 Outverbose& Outverbose::operator<< (const char * *str*)

Output a string.

Parameters

str	the string to output.

3.4.3.17 Outverbose& Outverbose::operator<< (const char *c*)

Output a character.

Parameters

c	the character to output.

3.4.3.18 Outverbose& Outverbose::operator<< (const verboseEnum *v*)

Change the programmer's verbosity setting.

Parameters

v	the verbosity setting.

3.4.3.19 Outverbose& Outverbose::operator<< (const int *i*)

Output an integer.

Parameters

i	the integer to output.

3.4.3.20 Outverbose& Outverbose::operator<< (const std::string & *str*)

Output a string.

Parameters

str	the string to output.

3.4.3.21 void Outverbose::outputVerbosity (verboseEnum *v*)

Ouput the verbosity level of the parameter in a human readable form (e.g. Quiet, Normal, Loud).

Parameters

v	the verbosity level to output.

See also

verboseEnum.

3.4.3.22 void Outverbose::setVerbosity (verboseEnum *v*)

Set verbosity.

Parameters

v	the user's verbosity level.

See also

verbose (p. 31).

3.4.4 Member Data Documentation

3.4.4.1 Outverbose::outVerbosity [private]

The programmer's verbosity setting. For example, error messages can be output using VERBOSE_LOUD, and minor system information with VERBOSE_QUIET. Everything else can be output with VERBOSE_NORMAL. The value can be set using the stream operator, e.g. out << VERBOSE_LOUD << "Fatal error.\n";

See also

Outverbose (p. 25)& **operator<<(const verboseEnum)** (p. 30);
verboseEnum.
checkVerbosity() (p. 29).

3.4.4.2 Outverbose::verbose [private]

The user's setting for verbosity. For example, if they specify -v on the command line, this can be set to VERBOSE_LOUD, and a -q can be used to set this to VERBOSE_QUIET. If they don't specify anything, then this can default to VERBOSE_NORMAL.

See also

verboseEnum.
checkVerbosity() (p. 29).

The documentation for this class was generated from the following file:

- include/outverbose.h

3.5 Ringbuf Class Reference

`#include <ringbuf.h>`

Public Member Functions

- **Ringbuf** ()
- **Ringbuf** (const int size)
- ∼**Ringbuf** ()
- void **allocate** (const int size)
- int **peek** (char ∗data, int amount)
- int **read** (char ∗data, int amount)
- int **write** (const char ∗data, int amount)
- int **getLength** () const
- int **grabline** (char ∗data, int max)
- void **output** ()

Private Member Functions

- int **update** ()
- int **read** (char ∗data, int amount, bool peek)

Private Attributes

- int **BUFFER_SIZE**
- int **length**
- char ∗ **buffer**
- char ∗ **writePtr**
- char ∗ **readPtr**
- char ∗ **endPtr**
- bool **allocated**

3.5.1 Detailed Description

A ringbuffer class.

3.5.2 Constructor & Destructor Documentation

3.5.2.1 Ringbuf::Ringbuf ( )

Constructor.

3.5.2.2 Ringbuf::Ringbuf (const int *size*)

Constructor with size allocation.

Parameters

size	the size of the buffer.

3.5.2.3 Ringbuf::∼Ringbuf ( )

Destructor.

3.5.3 Member Function Documentation

3.5.3.1 void Ringbuf::allocate (const int *size*)

Allocate buffer memory.

Parameters

size	the size of the buffer.

3.5.3.2 int Ringbuf::getLength ( ) const

Get the length of data on buffer.

Returns

the length of the buffer.

3.5.3.3 int Ringbuf::grabline (char ∗ *data*, int *max*)

Get an ASCII line of data.

Parameters

data	a pointer to where the data should be copied to.
max	the maximum amount of data to read.

Note

If max is not big enough, then you may get no data or only part of a line.

Returns

the number of bytes read, or -1 on error.

3.5.3.4 void Ringbuf::output ( )

Output buffer to STDOUT.

3.5.3.5 int Ringbuf::peek (char ∗ *data*, int *amount*)

Read but don't move read cursor or change length.

Parameters

data	a pointer to where the data should be copied to.
amount	the amount of data to read from the buffer.

Returns

the number of bytes read.

3.5.3.6 int Ringbuf::read (char * *data*, int *amount*)

Read data from buffer, advance read cursor.

Parameters

data	a pointer to where the data should be copied to.
amount	the amount of data to read from the buffer.

Returns

the number of bytes read.

3.5.3.7 int Ringbuf::read (char * *data*, int *amount*, bool *peek*) [private]

Read characters from buffer into data. If peeking, don't change readPtr or length.

Parameters

data	the pointer to a memory location for writing.
amount	the amount of bytes to copy to data.
peek	if true then don't change readPtr or length.

Returns

the number of bytes read (may be less than amount if amount is more than buffer length).

3.5.3.8 int Ringbuf::update ( ) [private]

Wrap around, and error check.

3.5.3.9 int Ringbuf::write (const char * *data*, int *amount*)

Write data to buffer, advance write cursor.

Parameters

data	a pointer to the data.
amount	the amount of data to write.

Returns

the number of bytes successfully written.

3.5.4 Member Data Documentation

3.5.4.1 bool Ringbuf::allocated [private]

Have we allocated memory?

3.5.4.2 char* Ringbuf::buffer [private]

A pointer to the buffer data.

3.5.4.3 int Ringbuf::BUFFER_SIZE [private]

The buffer size.

3.5.4.4 Ringbuf::endPtr [private]

The end pointer.

3.5.4.5 int Ringbuf::length [private]

Amount of data on buffer.

3.5.4.6 Ringbuf::readPtr [private]

The read pointer.

3.5.4.7 Ringbuf::writePtr [private]

The write pointer.

The documentation for this class was generated from the following file:

- include/motlab/ringbuf.h

3.6 Server Class Reference

`#include <server.h>`

Public Member Functions

- **Server** ()
- void **init** (**Outverbose** &o, int p, int b, int f, int u[ ])
- bool **startListening** ()
- int **getClients** ()
- int **addData** (int cid, uint16_t data)
- int **addData** (int cid, uint32_t data)
- int **addData** (int cid, float data)
- int **addData** (int cid, const char ∗data, int amount)
- void **addDataAll** (const char ∗data, int size, int source)
- void **doSelect** ()
- void **recvAll** ()
- void **sendAll** ()
- bool **checkNewConnections** (**Net** &net)
- int **checkClosedConnections** ()
- void **closeAll** ()
- **Unit getDataUnit** (**Net** &net)

Protected Attributes

- **Outverbose** ∗ **out**

Private Member Functions

- void **sendData** (int client)
- int **recvData** (int fd, char ∗data)
- int **findUnit** (char ∗data, int datasize)
- bool **acceptConnection** (**Net** &net)
- void **closeConnection** (int fd)

Private Attributes

- struct hostent ∗ **host**
- struct sockaddr_in **myaddr**
- int **listenfd**
- int **clientfd** [MAX_CLIENTS]
- bool **clientActive** [MAX_CLIENTS]
- int **clients**
- fd_set **readSocks**
- fd_set **writeSocks**
- fd_set **exceptionSocks**
- int **numSocksReadable**
- int **PORT**
- int **BACKLOG**

- int **flagsize**
- int **unitsize** [UNITS]
- **Ringbuf sendBuf** [MAX_CLIENTS]
- **Ringbuf recvBuf**
- int **recvSize**

3.6.1 Detailed Description

A server class.

Deals with receiving data from clients (e.g. keypresses/releases), and transmitting data back (e.g. the shared state).

3.6.2 Constructor & Destructor Documentation

3.6.2.1 Server::Server ( )

Constructor

3.6.3 Member Function Documentation

3.6.3.1 bool Server::acceptConnection (Net & *net*) [private]

Accept a new connection.

Parameters

net	an instance of net.

Returns

true if connection successfully accepted, false otherwise.

3.6.3.2 int Server::addData (int *cid*, uint16_t *data*)

Add integer (2 bytes) to buffer for sending.

Parameters

cid	client ID of recipient.
data	the data to send.

Returns

the number of bytes written, or -1 on error.

3.6.3.3 int Server::addData (int *cid,* uint32_t *data*)

Add integer (4 bytes) to buffer for sending.

Parameters

cid	client ID of recipient.
data	the data to send.

Returns

the number of bytes written, or -1 on error.

3.6.3.4 int Server::addData (int *cid,* float *data*)

Add float (4 bytes) to buffer for sending.

Parameters

cid	client ID of recipient.
data	the data to send.

Returns

the number of bytes written, or -1 on error.

3.6.3.5 int Server::addData (int *cid,* const char * *data,* int *amount*)

Add binary data to buffer for sending.

Checks it fits on the buffer first - simpler than ending up with half a data unit being written, with application having to store the rest (i.e. app can just drop the whole data unit instead).

Parameters

cid	client ID of recipient.
data	data to send.
amount	number of bytes in array.

Returns

the number of bytes written, or -1 on error. Returns 0 if it doesn't fit (nothing written).

3.6.3.6 void Server::addDataAll (const char * *data,* int *size,* int *source*)

Add data to all clients, optionally excluding source (originating client).

Parameters

data	data to send.
size	size of data (bytes).
source	ID of source client to be excluded, or -1 to transmit to all clients.

3.6.3.7 int Server::checkClosedConnections ( )

Check for closed connections and remove them.

Returns

ID of next inactive client.

3.6.3.8 bool Server::checkNewConnections (Net & *net*)

Check for new connections to the server. Call **doSelect()** (p. 39) first.

See also

doSelect() (p. 39)

Parameters

net	shared instance of net.

Returns

true if a new connection has been accepted.

3.6.3.9 void Server::closeAll ( )

Close all active connections.

3.6.3.10 void Server::closeConnection (int *fd*) [private]

Close connection with client.

Parameters

fd	the file descriptor to close.

3.6.3.11 void Server::doSelect ( )

Check if the socket can be read from / written to, using select() system call. Call this once per network loop, at the start of the loop, and after any read operation.

3.6.3.12 int Server::findUnit (char * *data*, int *datasize*) [private]

Find the data unit from bytes of data.

Parameters

data	a pointer to some data.
datasize	the size of the data.

Returns

the data unit found, or -1 if not found.

3.6.3.13 int Server::getClients ( )

Get the number of clients.

Returns

the number of clients.

3.6.3.14 Unit Server::getDataUnit (Net & *net*)

Read a data unit from the receive buffer.

Parameters

net	shared instance of net.

Returns

next data unit from the receive buffer.

3.6.3.15 void Server::init (Outverbose & *o*, int *p*, int *b*, int *f*, int *u[]*)

Initialise server.

Parameters

o	output instance.
p	port number.
b	backlog (number of pending connection requests allowed).
f	data unit flagsize.
u	data unit sizes.

3.6.3.16 void Server::recvAll ( )

Receive from all clients into receive buffer. Call **doSelect()** (p. 39) first. Use **getDataUnit()** (p. 40) to get a unit from the receive buffer for processing.

See also

doSelect() (p. 39)
getDataUnit() (p. 40)

3.6.3.17 int Server::recvData (int *fd,* char * *data*) [private]

Receive data from socket.

Parameters

fd	the file descriptor for the socket.
data	a pointer to where the data should be written, of size recvSize.

See also

recvSize (p. 43).

Returns

flag of unit found or -1 if not found.

3.6.3.18 void Server::sendAll ( )

Send out the buffered data to all clients. Call **doSelect()** (p. 39) first.

See also

doSelect() (p. 39)

3.6.3.19 void Server::sendData (int *client*) [private]

Send data from buffer.

Parameters

client	the client ID of the buffer/client to use.

3.6.3.20 bool Server::startListening ( )

Start listening.

Returns

true on success, false otherwise.

3.6.4 Member Data Documentation

3.6.4.1 Server::BACKLOG [private]

The number of pending connections allowed.

3.6.4.2 bool Server::clientActive[MAX_CLIENTS] [private]

Store which fd's are active.

3.6.4.3 int Server::clientfd[MAX_CLIENTS] [private]

File descriptors of clients.

3.6.4.4 int Server::clients [private]

Number of clients connected.

3.6.4.5 fd_set Server::exceptionSocks [private]

Sockets to watch for exceptions on.

3.6.4.6 Server::flagsize [private]

The flag size for the data units.

3.6.4.7 struct hostent* Server::host [private]

Host entity.

3.6.4.8 int Server::listenfd [private]

Socket file descriptor to listen on.

3.6.4.9 struct sockaddr_in Server::myaddr [private]

My address information.

3.6.4.10 int Server::numSocksReadable [private]

Technically number of bits set in read/write/exceptions by select.

3.6.4.11 Outverbose* Server::out [protected]

The class used for sending output to the user.

3.6.4.12 Server::PORT [private]

The port number.

3.6.4.13 fd_set Server::readSocks [private]

Sockets to read from.

3.6.4.14 Server::recvBuf [private]

The receive buffer. All data units received from clients are put into one buffer.

See also

recvAll (p. 41).

3.6.4.15 int Server::recvSize [private]

The maximum amount of data to receive in one go.

This should be set to the size of the largest data unit + flagsize.

See also

flagsize (p. 42).

3.6.4.16 Server::sendBuf[MAX_CLIENTS] [private]

The send buffers, one for each client.

3.6.4.17 Server::unitsize[UNITS] [private]

The sizes of the data units.

3.6.4.18 fd_set Server::writeSocks [private]

Sockets to write to.

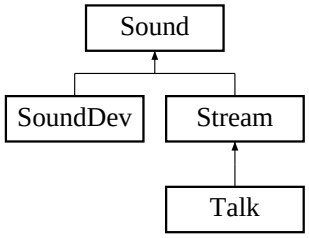
The documentation for this class was generated from the following file:

- include/motlab/server.h

3.7 Sound Class Reference

`#include <sound.h>`

Inheritance diagram for Sound:



Public Member Functions

- **Sound** ()
- ~**Sound** ()
- void **initOutput** (**Outverbose** &o)
- void **setPlayDevice** (ALCdevice *dev)

Protected Member Functions

- ALenum **check** (const char *errorMaker)
- ALCenum **checkAlc** (const char *errorMaker)

Protected Attributes

- **Outverbose** * **out**
- ALCdevice * **playDevice**

3.7.1 Detailed Description

A basic **Sound** (p. 44) class to hold a playback device and perform error checking.

This is extended by **Stream** (p. 49) (for streaming playback), and Dev (to hold capture devices and deal with opening playback device and contexts).

3.7.2 Constructor & Destructor Documentation

3.7.2.1 Sound::Sound ( )

Constructor.

3.7.2.2 Sound::~Sound ( )

Destructor.

3.7.3 Member Function Documentation

3.7.3.1 ALenum Sound::check (const char * *errorMaker*) [protected]

Performs alGetError().

Call after an OpenAL function. If there is an error then it is printed to the output object, along with information on the place where the error occured (errorMaker).

Parameters

errorMaker	a character string containing information on what made the error.

3.7.3.2 ALCenum Sound::checkAlc (const char * *errorMaker*) [protected]

Context error check. Performs alcGetError().

Call after an ALC function. Similar to **check()** (p. 45).

See also

check() (p. 45)

3.7.3.3 void Sound::initOutput (Outverbose & *o*)

Initialise output instance.

Parameters

o	output instance.

3.7.3.4 void Sound::setPlayDevice (ALCdevice * *dev*)

Set the play device.

Parameters

dev	the play device.

3.7.4 Member Data Documentation

3.7.4.1 Outverbose* Sound::out [protected]

The instance of the output object.

3.7.4.2 ALCdevice* Sound::playDevice [protected]

Playback device is stored in the parent class because it's needed to **checkAlc()** (p. 45).

See also

checkAlc() (p. 45)

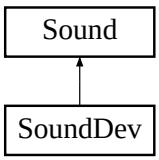
The documentation for this class was generated from the following file:

- include/motlab/sound.h

3.8 SoundDev Class Reference

`#include <dev.h>`

Inheritance diagram for SoundDev:



Public Member Functions

- **SoundDev** ()
- ~**SoundDev** ()
- bool **grab** ()
- bool **release** ()
- bool **checkPlayContext** ()
- ALCdevice * **getPlayDevice** ()
- bool **openCaptureDevice** (ALCuint freq, ALCenum format, ALCsizei internalSamples)
- bool **closeCaptureDevice** ()
- bool **checkCaptureDevice** ()
- ALCdevice * **getCaptureDevice** ()

Private Member Functions

- bool **openPlayDevice** ()
- bool **createContext** ()
- bool **destroyContext** ()
- bool **closePlayDevice** ()

Private Attributes

- U **__pad0__**:/utils/motlabsvn/motlab/trunk/include/motlab/dev.h" 2 private: ALCdevice ∗captureDevice
- ALCcontext ∗ **context**

3.8.1 Detailed Description

Class to deal with the sound capture device and playback contexts.

Play device is dealt with in **Sound** (p. 44) class, because it's needed for the checkAlc OpenAL call.

3.8.2 Constructor & Destructor Documentation

3.8.2.1 SoundDev::SoundDev ( )

Constructor.

3.8.2.2 SoundDev::∼SoundDev ( )

Destructor.

3.8.3 Member Function Documentation

3.8.3.1 bool SoundDev::checkCaptureDevice ( )

Check capture device is not null (here to be consistent with play device).

Returns

true if it's not null, false if it is.

3.8.3.2 bool SoundDev::checkPlayContext ( )

Check we've got play device and context ready to go.

3.8.3.3 bool SoundDev::closeCaptureDevice ( )

Close capture device.

Returns

true on success, false on failure.

3.8.3.4 bool SoundDev::closePlayDevice ( ) [private]

Close play device.

Returns

true on success, false on failure.

3.8.3.5 bool SoundDev::createContext ( ) [private]

Create context.

Returns

true on success, false on failure.

3.8.3.6 bool SoundDev::destroyContext ( ) [private]

Destroy contect.

Returns

true on success, false on failure.

3.8.3.7 ALCdevice* SoundDev::getCaptureDevice ( )

Get the capture device.

Returns

the capture device.

3.8.3.8 ALCdevice* SoundDev::getPlayDevice ( )

For passing to other instances.

3.8.3.9 bool SoundDev::grab ( )

Open play device and create context.

3.8.3.10 bool SoundDev::openCaptureDevice (ALCuint *freq*, ALCenum *format*, ALCsizei *internalSamples*)

Open the capture device.

Parameters

freq	the capture frequency.
format	the sound format (e.g. AL_FORMAT_MONO16 - see the OpenAL reference manual).
internalSamples	size of internal OpenAL capture buffer, measured in samples.

Returns

true on success, false on failure.

3.8.3.11 bool SoundDev::openPlayDevice ( ) [private]

Open the play device.

Returns

true on success, false on failure.

3.8.3.12 bool SoundDev::release ( )

Destroy context and close play device.

3.8.4 Member Data Documentation

3.8.4.1 U SoundDev::__pad0__ [private]

The capture device.

3.8.4.2 ALCcontext* SoundDev::context [private]

The OpenAL context.

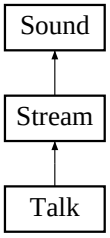
The documentation for this class was generated from the following file:

- include/motlab/dev.h

3.9 Stream Class Reference

`#include <stream.h>`

Inheritance diagram for Stream:



Public Member Functions

- **Stream** ()
- ∼**Stream** ()
- void **initStream** ()
- int **getChunkBytes** ()
- int **getChunksRecvd** ()
- void **receive** (const char ∗chunk)
- bool **update** ()
- bool **playing** ()

Protected Attributes

- ALCenum **format**
- ALCuint **freq**
- float **chunkSeconds**
- ALCint **chunkSamples**
- ALCint **chunkBytes**
- ALCsizei **internalSamples**

Private Member Functions

- bool **queue** ()
- int **unqueuePlayedBuffers** ()
- bool **stream** (ALuint buffer)

Private Attributes

- **Ringbuf streamBuf**
- ALuint **streamSource**
- ALuint **streamSwapBuf** [STREAM_SWAP_MAX]
- int **swap**
- int **swaps**
- int **minQueueSize**
- bool **initialisedStream**

3.9.1 Detailed Description

Class to deal with streaming sound.

For example, this could be audio read from a file, or from a microphone input. This class is extended by **Talk** (p. 54) to stream from microphone input.

This class extends **Sound** (p. 44) to inherit the playback device and error checking.

3.9.2 Constructor & Destructor Documentation

3.9.2.1 Stream::Stream ( )

Constructor.

3.9.2.2 Stream::~Stream ( )

Destructor.

3.9.3 Member Function Documentation

3.9.3.1 int Stream::getChunkBytes ( )

Get number of bytes in chunk.

Returns

number of bytes in chunk.

3.9.3.2 int Stream::getChunksRecvd ( )

Get number of chunks on buffer (i.e. streamBuf.getLength() / chunkBytes).

Returns

number of chunks on buffer.

3.9.3.3 void Stream::initStream ( )

Initialise stream.

3.9.3.4 bool Stream::playing ( )

Are we playing sound?

Returns

true if playing, false otherwise.

3.9.3.5 bool Stream::queue ( ) [private]

Attempt to queue next part of stream.

Returns

true on success, or false if there is no more stream to queue (i.e. if **stream()** (p. 52) returns false.

See also

stream() (p. 52).

3.9.3.6 void Stream::receive (const char * *chunk*)

Receive bytes into stream buffer.

Parameters

chunk	a chunk of audio data.

3.9.3.7 bool Stream::stream (ALuint *buffer*) [private]

Attempts to add the next bit of the stream, from streamBuf, to the OpenAL buffer.

Parameters

buffer	the OpenAL buffer.

Returns

true if the next bit of stream has been added to buffer, or false if there's not enough on streamBuf for next chunk.

3.9.3.8 int Stream::unqueuePlayedBuffers ( ) [private]

Unqueue the played buffers.

Returns

the number of buffers unqueued.

3.9.3.9 bool Stream::update ( )

Update stream buffer. Call once per sound loop. This function unqueues what's played, and tries to keep queue topped up (if there's anything left on the stream buffer). It deals with starting playback, which is done only when the queue is above minQueueSize.

Returns

true if source is still active, false otherwise.

3.9.4 Member Data Documentation

3.9.4.1 ALCint Stream::chunkBytes [protected]

number of bytes in chunk.

3.9.4.2 ALCint Stream::chunkSamples [protected]

number of samples per chunk.

3.9.4.3 float Stream::chunkSeconds [protected]

length of capture chunks in seconds.

3.9.4.4 ALCenum Stream::format [protected]

format of captured samples.

3.9.4.5 ALCuint Stream::freq [protected]

total number of samples per second.

3.9.4.6 bool Stream::initialisedStream [private]

Have we initialised the stream?

3.9.4.7 ALCsizei Stream::internalSamples [protected]

size of internal OpenAL capture buffer.

3.9.4.8 Stream::minQueueSize [private]

The minimum queue size before playback starts/resumes. This removes the popping effect but adds latency.

3.9.4.9 Ringbuf Stream::streamBuf [private]

The stream buffer.

3.9.4.10 Stream::streamSource [private]

The source for the stream.

3.9.4.11 Stream::streamSwapBuf[STREAM_SWAP_MAX] [private]

The swap buffer for the stream. Play one, queue the other, then swap.

3.9.4.12 Stream::swap [private]

The next swap buffer to be filled and queued.

3.9.4.13 Stream::swaps [private]

The number of swap buffers. Must be less than STREAM_SWAP_MAX.

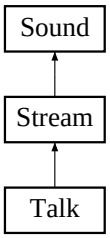
The documentation for this class was generated from the following file:

- include/motlab/stream.h

3.10 Talk Class Reference

`#include <talk.h>`

Inheritance diagram for Talk:



Public Member Functions

- **Talk** ()
- ∼**Talk** ()
- void **setSoundDev** (**SoundDev** &sd)
- void **setCaptureDevice** (ALCdevice ∗dev)
- bool **openCaptureDevice** ()
- bool **closeCaptureDevice** ()
- void **captureStart** ()
- void **captureStop** ()

- bool **getCapturing** ()
- void **capture** (int id, **Net** &net, **Client** &client)
- bool **captureRaw** (char ∗buf)

Private Attributes

- ALCdevice ∗ **captureDevice**
- **SoundDev** ∗ **soundDev**
- bool **capturing**

3.10.1 Detailed Description

A class for capturing microphone input, and transmitting across a network. Extends **Stream** (p. 49) to support the streaming of playback. Uses **Net** (p. 10) and **Client** (p. 5) for transmition.

3.10.2 Constructor & Destructor Documentation

3.10.2.1 Talk::Talk ( )

Constructor.

3.10.2.2 Talk::∼Talk ( )

Destructor.

3.10.3 Member Function Documentation

3.10.3.1 void Talk::capture (int *id,* Net & *net,* Client & *client*)

Capture sound and transmit. If there are chunkSamples or more on the internal OpenAL ring buffer, then grab the samples into a data unit and queue for transmition to server using **Net** (p. 10).

Parameters

id	our client ID.
net	instance of **Net** (p. 10) object.
client	instance of **Client** (p. 5) object.

3.10.3.2 bool Talk::captureRaw (char ∗ *buf*)

Capture sound and store raw audio data in buffer. Do not transmit. Application can use **getChunkBytes()** (p. 51) to determine the minimum buffer size.

Parameters

buf	pointer to a buffer of size chunkBytes.

Returns

true if there were enough samples (i.e. they were copied to the buffer), false otherwise.

3.10.3.3 void Talk::captureStart ( )

Start capturing.

3.10.3.4 void Talk::captureStop ( )

Stop capturing.

3.10.3.5 bool Talk::closeCaptureDevice ( )

Close capture device.

Returns

true on success, false on failure.

3.10.3.6 bool Talk::getCapturing ( )

Are we capturing?

Returns

true if capturing, false otherwise.

3.10.3.7 bool Talk::openCaptureDevice ( )

Open capture device.

Returns

true on success, false on failure.

3.10.3.8 void Talk::setCaptureDevice (ALCdevice * *dev*)

Set from **SoundDev** (p. 46).

Parameters

dev	capture device

3.10.3.9 void Talk::setSoundDev (SoundDev & *sd*)

Set **SoundDev** (p. 46) instance for opening capture device.

Parameters

sd	**SoundDev** (p. 46) instance

3.10.4 Member Data Documentation

3.10.4.1 ALCdevice* Talk::captureDevice [private]

The capture device.

3.10.4.2 bool Talk::capturing [private]

Are we capturing sound?

3.10.4.3 SoundDev* Talk::soundDev [private]

The sound device.

The documentation for this class was generated from the following file:

- include/motlab/talk.h

3.11 Transfercontrol Class Reference

`#include <transfercontrol.h>`

Public Member Functions

- void **init** (**Outverbose** &o)
- void **setReadPath** (std::string s)
- void **setWritePath** (std::string s)
- bool **checkFilename** (std::string s)
- bool **start** (int source, int dest, int id, char *filename, **Net** &net, **Client** &client)
- void **send** (int source, int dest, int id, char *filename, **Net** &net, **Client** &client)
- bool **receive** (**Unit** unit)
- void **go** (**Net** &net, **Client** &client)

Private Attributes

- **Outverbose** ∗ **out**
- std::string **readPath**
- std::string **writePath**
- std::deque< **Transfersend** > **transfersend**
- std::deque< **Transferrecv** > **transferrecv**

3.11.1 Detailed Description

Control file transfer across the network.

3.11.2 Member Function Documentation

3.11.2.1 bool Transfercontrol::checkFilename (std::string *s*)

Check filename contains no directories (i.e. can only read/write to base path).

Returns

true if ok, false otherwise.

3.11.2.2 void Transfercontrol::go (Net & *net,* Client & *client*)

Request next pieces of data, remove old transactions.

Parameters

net	instance of **Net** (p. 10).
client	instance of **Client** (p. 5).

3.11.2.3 void Transfercontrol::init (Outverbose & *o*)

Initialisation of output object.

Parameters

o	instance of output object.

3.11.2.4 bool Transfercontrol::receive (Unit *unit*)

Receive a data unit from the network, and allocate to the relevant transferrecv.

Parameters

unit	the data unit received (unit.transfer).

3.11.2.5 void Transfercontrol::send (int *source*, int *dest*, int *id*, char * *filename*, Net & *net*, Client & *client*)

Set up a file transfer to be sent.

Parameters

source	source client ID.
dest	destination client ID.
id	transaction ID.
filename	name of file (should be in read path).
net	instance of **Net** (p. 10).
client	instance of **Client** (p. 5).

3.11.2.6 void Transfercontrol::setReadPath (std::string *s*)

Base path to read from.

Parameters

s	base path

3.11.2.7 void Transfercontrol::setWritePath (std::string *s*)

Base path to write to.

Parameters

s	base path

3.11.2.8 bool Transfercontrol::start (int *source*, int *dest*, int *id*, char * *filename*, Net & *net*, Client & *client*)

Initiate file transfer from source to destination. The caller should give this transaction an ID that is unique to this source.

This function queues the transfer request to transferrecv.

Parameters

source	source client ID.
dest	destination client ID.
id	ID of this transaction.

filename	name of file (to be saved in write path).
net	instance of **Net** (p. 10).
client	instance of **Client** (p. 5).

Returns

true if started transaction, false if already got file. Note that 'false' does not imply an error, and is a signal to the application that it already has the file it requested (or a file with that name).

3.11.3 Member Data Documentation

3.11.3.1 Outverbose* Transfercontrol::out [private]

The output object.

3.11.3.2 std::string Transfercontrol::readPath [private]

The path to read files from.

3.11.3.3 std::deque<Transferrecv> Transfercontrol::transferrecv [private]

The transfer receive queue.

3.11.3.4 std::deque<Transfersend> Transfercontrol::transfersend [private]

The transfer send queue.

3.11.3.5 std::string Transfercontrol::writePath [private]

The path to write files to.

The documentation for this class was generated from the following file:

- include/motlab/transfercontrol.h

3.12 Transferrecv Class Reference

`#include <transferrecv.h>`

Public Member Functions

- **Transferrecv** (int **source**, int **dest**, int **id**, const char *__path__, const char *__filename__)

- bool **checkFileExists** (char ∗name, bool isdir)
- bool **getStarted** ()
- void **setStarted** (bool b)
- bool **getReady** ()
- void **setReady** (bool b)
- bool **getActive** ()
- int **getId** ()
- int **getSource** ()
- void **request** (**Net** &net, **Client** &client)
- void **receive** (const char ∗data, int amount)

Private Attributes

- int **id**
- char **filename** [MAX_FILENAME_SIZE]
- char **path** [MAX_FILENAME_SIZE]
- int **source**
- int **dest**
- bool **started**
- bool **active**
- bool **opened**
- bool **ready**
- std::FILE ∗ **file**

3.12.1 Detailed Description

Deal with receiving files. Each instance deals with a single transaction.

3.12.2 Constructor & Destructor Documentation

3.12.2.1 Transferrecv::Transferrecv (int *source*, int *dest*, int *id*, const char ∗ *path*, const char ∗ *filename*)

Constructor.

Parameters

source	the source client ID.
dest	the destination client ID.
id	the transaction ID.
path	the write path.
filename	the name of the file.

3.12.3 Member Function Documentation

3.12.3.1 bool Transferrecv::checkFileExists (char * *name,* bool *isdir*)

Check if file exists.

Parameters

name	name of file/directory.
isdir	is a directory.

Returns

true if exists, false otherwise.

3.12.3.2 bool Transferrecv::getActive ( )

Are we active?

Returns

true yes, false otherwise.

3.12.3.3 int Transferrecv::getId ( )

What's this transaction ID?

Returns

transaction ID.

3.12.3.4 bool Transferrecv::getReady ( )

Are we ready to receive?

Returns

true yes, false otherwise.

3.12.3.5 int Transferrecv::getSource ( )

Who's the source client?

Returns

source client ID.

3.12.3.6 bool Transferrecv::getStarted ( )

Have we started receiving?

Returns

true yes, false otherwise.

3.12.3.7 void Transferrecv::receive (const char * *data*, int *amount*)

Receive binary data.

Parameters

data	binary data.
amount	size of data (bytes).

3.12.3.8 void Transferrecv::request (Net & *net*, Client & *client*)

Request next piece of data.

Parameters

net	instance of **Net** (p. 10).
client	instance of **Client** (p. 5).

3.12.3.9 void Transferrecv::setReady (bool *b*)

Set ready to receive.

Parameters

b	true yes, false no.

3.12.3.10 void Transferrecv::setStarted (bool *b*)

Set started receiving.

Parameters

b	started receiving (true/false).

3.12.4 Member Data Documentation

3.12.4.1 bool Transferrecv::active [private]

Are we active?

3.12.4.2 int Transferrecv::dest [private]

The destination client.

3.12.4.3 std::FILE* Transferrecv::file [private]

The file.

3.12.4.4 char Transferrecv::filename[MAX_FILENAME_SIZE] [private]

The filename.

3.12.4.5 int Transferrecv::id [private]

The ID of this transaction.

3.12.4.6 bool Transferrecv::opened [private]

Have we opened the file?

3.12.4.7 char Transferrecv::path[MAX_FILENAME_SIZE] [private]

The file path.

3.12.4.8 bool Transferrecv::ready [private]

Are we ready?

3.12.4.9 int Transferrecv::source [private]

The source client.

3.12.4.10 bool Transferrecv::started [private]

Have we started transmission?

The documentation for this class was generated from the following file:

- include/motlab/transferrecv.h

3.13 Transfersend Class Reference

`#include <transfersend.h>`

Public Member Functions

- **Transfersend** (int **source**, int **dest**, int **id**, const char ∗**path**, const char ∗**filename**)
- int **getSource** ()
- int **getDest** ()
- int **getId** ()
- char ∗ **getFilename** ()
- bool **getActive** ()
- void **send** (**Net** &net, **Client** &client)

Private Attributes

- int **id**
- char **filename** [MAX_FILENAME_SIZE]
- char **path** [MAX_FILENAME_SIZE]
- int **source**
- int **dest**
- int **offset**
- bool **started**
- bool **active**
- std::FILE ∗ **file**

3.13.1 Detailed Description

Deal with sending files across network. Each instance deals with one transaction.

3.13.2 Constructor & Destructor Documentation

3.13.2.1 Transfersend::Transfersend (int *source,* int *dest,* int *id,* const char ∗ *path,* const char ∗ *filename*)

Constructor.

Parameters

source	the source client ID.
dest	the destination client ID.
id	the transaction ID.
path	the read path.
filename	the name of the file.

3.13.3 Member Function Documentation

3.13.3.1 bool Transfersend::getActive ( )

Are we active?

Returns

true yes, false no.

3.13.3.2 int Transfersend::getDest ( )

Who's the destination client?

Returns

destination client ID.

3.13.3.3 char* Transfersend::getFilename ( )

Get the filename.

Returns

filename.

3.13.3.4 int Transfersend::getId ( )

What's this transaction ID?

Returns

transaction ID.

3.13.3.5 int Transfersend::getSource ( )

Who's the source client?

Returns

source client ID.

3.13.3.6 void Transfersend::send (Net & *net,* Client & *client*)

Send next chunk of file.

Parameters

net	instance of **Net** (p. 10).
client	instance of **Client** (p. 5).

3.13.4 Member Data Documentation

3.13.4.1 bool Transfersend::active [private]

Are we active?

3.13.4.2 int Transfersend::dest [private]

The destination client.

3.13.4.3 std::FILE* Transfersend::file [private]

The file.

3.13.4.4 char Transfersend::filename[MAX_FILENAME_SIZE] [private]

The filename.

3.13.4.5 int Transfersend::id [private]

The ID of this transaction.

3.13.4.6 int Transfersend::offset [private]

The file offset.

3.13.4.7 char Transfersend::path[MAX_FILENAME_SIZE] [private]

The file path.

3.13.4.8 int Transfersend::source [private]

The source client.

3.13.4.9 bool Transfersend::started [private]

Have we started?

The documentation for this class was generated from the following file:

- include/motlab/transfersend.h

3.14 Unit Union Reference

`#include <network.h>`

Public Attributes

- int **flag**
- struct {
  int **flag**
  int **id**
  } **assign**

- struct {
  int **flag**
  int **id**
  char **data** [4000]
  } **audio**

- struct {
  int **flag**
  int **from**
  int **to**
  int **objectid**
  int **info_i**
  float **info_f**
  char **info_c** [TRANSFER_DATA_SIZE]
  } **generic**

- struct {
  int **flag**
  int **id**
  int **bits**
  } **keys**

- struct {
  int **flag**
  int **id**
  } **logoff**

- struct {
  int **flag**
  int **id**
  } **newclient**

- struct {
  int **flag**
  int **id**
  float **x**
  float **y**
  float **z**
  } **position**

- struct {
  int **flag**
  int **to**
  int **from**
  int **id**
  char **filename** [MAX_FILENAME_SIZE]
  char **data** [TRANSFER_DATA_SIZE]
  int **amount**
  } **transfer**

- struct {
  int **flag**
  int **to**
  int **from**
  int **id**
  } **transferfin**

- struct {
  int **flag**
  int **to**
  int **from**
  int **id**
  char **filename** [MAX_FILENAME_SIZE]
  } **transferreq**

- struct {
  int **flag**
  int **id**
  int **width**
  int **height**
  } **viewport**

3.14.1 Detailed Description

The data unit union.

3.14.2 Member Data Documentation

3.14.2.1 int Unit::flag

always have a flag, -1 is no unit received.

The documentation for this union was generated from the following file:

- include/motlab/network.h

Index