# ELECTRONIC VISUALIZATION LABORATORY
# UNIVERSITY OF ILLINOIS AT CHICAGO

## Spatialized Sound Server Implementation


Edited by : Dipl. Medieninf. (FH) Severin S. Todt
VR4 Lab
University of Applied Sciences - Wedel
Feldstrasse 143
22880 Wedel
Germany

Supervised by : Daniel Sandin
Director
Electronic Visualization Laboratory
University of Illinois at Chicago
Engineering Research Facility (ERF)
842 W.Taylor Street
Chicago, IL 60607
USA

# Spatialized Sound Server Implementation

**Severin S. Todt**

Summer 2002

## Abstract

Within this project a sound server is developed capable of reproducing spatial sound impressions based on the position of sound sources and listener objects that can be positioned anywhere in a virtual world.
Based on the "Bergen Server" a former sound server implementation a spatialized sound server is developed using one to four loudspeakers to reproduce a spatial sound impression.
The user's position within his physical audio reproduction environment as well as the virtual sound sources' positions are taken into account to calculate a realistic sound characteristic.
With the listener changing his position or the sound sources being repositioned within the virtual world the generated sound characteristic is dynamically adjusted.
The client – server model implemented within this server allows a distributed application flow. A client module implementing a virtual reality application is controlling the audio processing tasks on the server module by sending messages to create a listener and sound sources as well as sending messages to update their positions. The final sound characteristic is calculated on the server side.

# Table of Contents

## Table of Figures

# 1. Introduction

Introducing the features and advantages of spatialized sound this part further discusses the efforts made so far to include sound in virtual environments at the Electronic Visualization Laboratory (EVL) at the University of Illinois at Chicago (UIC) where the spatialized sound server is being implemented within this project.

## 1.1. Spatialized Sound

Spatialized sound is understood as sound that enables the derivation of information about the sound's spatial characteristic, e.g. position, direction, movement, in a virtual environment, surrounding the listener.
A sound spatialization system is capable of positioning sounds at any position within the listener's surrounding space and to evaluate the appropriate sound issues in order to calculate and replay the adequate spatial sound impression.

## 1.2. The Contribution of Spatialized Sound to Virtual Environments

With the inclusion of spatialized sound in virtual reality environments the grade of the virtual environment's immersion is improving.
As a form of application feedback to the user spatialized sound can be helpful redundant to a visual cue or can provide feedback for actions and situations that are out of the listener's field of view.[1]
Invisible objects can easily be pursued or audibly observed through the use of spatialised sound. With sound integration the recognition of objects in a scene can be improved[2] and the time delay between object appearance and object perception can be shortened.[3]

---

[1] cp.: Begault, Durand R. : 3-D sound For Virtual Reality And Multimedia, Moffet Field, NASA Ames Research Center, Academics Press Professional, 1994, Reprint 2000, p.14

[2] cp.: Davis, Elisabeth T. a.o. : Can Audio Enhance Visual Perception and Performance in a Virtual Environment, Atlanta, Georgia Institute of Technology, 1999, p. 4

[3] cp.: Begault, Durand R. : Head-Up Auditory Displays for Traffic Collision Avoidance System Advisories, Moffet Field, NASA Ames Research Center, The Human Factors and Ergonomics Society Inc., 1993, p. 707

The benefits of spatialized sound integration contribute to the improvement of quality and ease of interaction.[4]

### 1.3. Project Focus

The focus of this project is set on implementing a spatialzed sound server. The implementation should utilise the already existing sound server, "Bergen Sound Server" used within the Virtual Reality Environment at EVL at UIC.
The "Bergen Sound Server" established in November 2000 by Dave Pape is used for reproduction of mono sound samples. The sound server is implemented to run either on Linux based systems or IRIX based SGI workstations.
For final sound replay the audio hardware installed at the workstation the audio server is installed and running on is used for reproduction. Currently standard audio hardware is used for replaying the mono sound samples.
Upgrading the server to a spatialized sound implementation has to be performed keeping a backward compatible version available for current applications.
For a successful sound server upgrade the server's data structure, layer model and communication model and the source code have to be analysed and documented.
Based on the achieved structures implementation strategies are to be evaluated and discussed to obtain an easy though performant approach for a spatialized sound server implementation.
In consequence of the project's time limitation a spatialization technique has to be chosen which is based on already available hardware configurations in a first step which influence the decisions made on the implementation strategy.

---

[4] cp.: Todt, Severin S., Integration of Multichannel Sound in Virtual Environments, University of Applied Sciences – Wedel, Wedel, Germany , February 2002, p. 2

## 2. "Bergen Sound Server" Analysis

The underlying data structures, layer models and communication models as well as an application's application flow using the "Bergen Sound server" have to be analysed and documented in order to obtain an overview and to identify the structures to be adjusted to obtain spatialized sound functionality.
Starting with a plain description of the sound server's current functionality an overview for the "Bergen Sound Server" is given, followed by an analyse of the layer model, the underlying data structures and the communication model.
The section closes describing a typical sound application session.

### 2.1. Functionality

Bergen is a very simple, freely redistributable audio server and client library. It was created for use in CAVE applications, to get around a few of the limitations in the VSS library; most of those limitations have been fixed, but as VSS is a fairly advanced tool, Bergen will stick around as a more basic alternative.[5]
The sound server's current version enables client applications to replay any sound sample (given in an "AIFF", "RAW" or "WAV" file format), describable as a tone with a certain frequency or known as "white noise" sound.
For best results only files given in a 16-bit, 2's-complement format should be used and the server should be run at the same frequency the used sample files are stored in to keep the influence of the server's available resample facility low.
For a distributed approach the "Bergen Sound Server" offers a server as well as a client module.
The client module is controlling the application flow and in contrast the server calculates the sound characteristics and replays it using the sound hardware available at the server side.

---

[5] cp.: Pape, Dave : Bergen Sound Server & Library, Version 0.4.1, Electronic Visualization Laboratory – Universit of Illinois at Chicago, http://www.evl.uic.edu/pape/sw/bergen/, 11/10/2002

Both, the client and the server module can run on the same machine, though in a distributed installation the two modules communicate based on a UDP network connection.
Presently most of the available applications using the "Bergen Sound Server" are based on the Ygdrasil framework, offering easy integration of sound in virtual environments.[6]
The Ygdrasil framework implements the sound server's client module and uses it for communication with the sound server.
Through sending messages to the server to reset a sound's amplitude the Ygdrasil framework thereby implements a distance model for the sound source being replayed, based on a linear fall off function including the definition of a near and far fall off range.

## 2.2. Layer Model

The sound server design is based on a four layer structure. The lowest level, the audio hardware forms the sound system's base. The base is used by a unique server module for audio reproduction. Besides the audio reproduction the server implements the memory management and the binary audio data file.
Several client modules are able to access the server module utilising a network connection. The client modules use the server to open, resample, adjust and replay any given audio sample file, a tone or even a white noise. The client module is free to be used within any application.

---

[6] cp.: Hill, Alex : Ygdrasil Documentation, Electronic Visualization Laboratory – Universit of Illinois at Chicago, http://www.evl.uic.edu/yg/about.html , Chicago, IL, USA, 2002

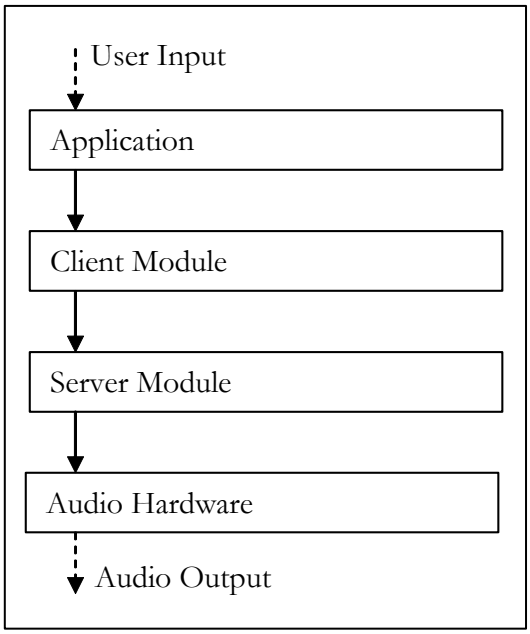

Figure 1 : "Bergen Server" Layer Model

### 2.3. Data Structures

The "Bergen Sound Server" architecture is constructed based on two separated data constructs.

Due to the client – server model there's a data structure available for the client side as well as for the server side.

The client side on the one hand simply spoken administrates identifiers to data structs managed at the server side. The client's efforts therefore are reduced to managing the own application's data identifiers and is freed from any data storage and memory management tasks.

On the other hand the server is dealing with the memory and binary data aspects for all the clients connected to the server. Therefore data structures are available for data storage and memory management.

Both sides have to have data structures necessary for network communication.

### 2.3.1. Client Data Structure

The client offers a "bergenServer" class which is the main structure enabling access to all of the belongings connected to sound reproduction.
It holds references to a unique "bergenUDPSocket" keeping track of the network connection data.
Beside the unique datastructure for network communication a list of "bergenSound" structures is administrated. The "bergenSound" structure can be implemented as a "bergenTone", "bergenSample" or "bergenWhiteNoise".
The "bergenTone" offers a constructor to create an tone with a certain frequency and holds attributes to control the tone.
With the "bergenSample" a structure is offered holding attributes describing the properties of a sound sample file as well as methods to influence the sound samples behaviour.
The "bergenWhiteNoise" as a third possibility offers a constructor to create a white noise sound and offers methods to influence it's behaviour. (see figures 2 – 7)

| class bergenServer | | | | |
|---|---|---|---|---|
| public | | bergenServer | (char *server=NULL,int port=0) | The constructor will make a UDP network connection to the snerd server program, sending it a "ping" message to verify the connection. The argument host, if given, is the name or IP address of the machine which snerd is running on; if it is not given, the value of the environment variable BERGEN_SERVER is used; if this variable is not set, the connection is made to the local machine. The argument port is the port number that the server is listening to; if it is not given (or is 0), the value of the environment variable BERGEN_PORT is used; if this variable is not set, the default port of 5900 is used. |
| public | | ~bergenServer | (void) | The destructor will automatically delete all sounds which are in its list of bergenSound objects (maintained by addSound() and removeSound). |
| public | void | reset | (void) | Re-initializes the connection to the server program, and re-creates all the sound objects (that it manages) on the server. This can be used to reset things if snerd crashes & is restarted, or if it is started after the bergenServer object is created |
| public | void | addSound | (class bergenSound *) | Adds a sound to the server object's list. This is used internally, by the bergenSound class, and should not be called by applications. The list consists of the references to the sound objects. The sound object data is managed by snerd |
| public | void | removeSound | (class bergenSound *) | Removes a sound from the server object's list. This is used internally, by the bergenSound class, and should not be called by applications |
| public | void | setDirectory | (char *) | Sets the default directory for sound objects. bergenSample objects will use this directory if their sample file name does not include a path. The directory is used by snerd to look up the sound files which are to be stored at sound server's location |
| public | char * | directory | (void) | Returns the default directory name set by setDirectory() |
| public | void | sendMessage | (char *msg) | Sends the given text string to the server program. This is mostly for internal use, by the bergenSound classes. The snerd analyses the message and reacts on it |
| public | int | receiveMessage | (char *msg,int size) | Gets the next message sent by the server program. This is mostly for internal use, by the bergenSound classes |
| private | void | connect | (void) | Tries to establish a connection to the snerd server. Error messages if no reply from the snerd server or no socket is given in the bergenServer |
| private | char * | serverName_ | | Name of the server |
| private | int | serverPort_ | | Port to be used for communication |
| private | struct _sound * | soundList_ | | List of Sounds (only the references) |
| private | char * | directory_ | | Default directory of the snerdServer |
| private | class bergenUDPSocket * | socket_ | | socket to be used for communication |

Figure 2 : class bergenServer overview

| class bergenUDPSocket | | | | |
|---|---|---|---|---|
| public | | bergenUDPSocket | (int port) | Creates socket for communication and binds to the given port |
| public | void | setDestination | (char *host,int port) | Tries to find host and if successfull sets address and port for future |
| public | void | setBlocking | (int block) | Sets or resets blocking size |
| public | void | send | (void *buffer,int size) | Is sending a buffer using opened socket |
| public | int | receive | (void *buffer,int size) | Receives an incomming message |
| public | void | reply | (void *buffer,int size) | sendsa received buffer back to sender |
| private | struct sockaddr_in | destAddr_ | | destination address for client server communication |
| private | struct sockaddr_in | lastFromAddr_ | | address the last message was received from |
| private | int | socket_ | | socket used for communication |

Figure 3 : class bergenUDPSocket overview

| class bergenSound | | | | |
|---|---|---|---|---|
| public | | bergenSound | (class bergenServer *server) | The constructor must be given a pointer to a server object; it will tell the server object to add this sound to its list of sounds |
| public | | ~bergenSound | (void) | The destructor does an automatic kill before the sound object is removed |
| public | int | handle | (void) | Returns the handle_ |
| public | class bergenServer * | server | (void) | Returns the server_ |
| public | virtual void | setAmplitude | (float amp) | Sets the sound's current amplitude; sends a message to the server program to accomplish this |
| public | virtual void | play | (void) | Sends a message to the server program to start playing the sound |
| public | virtual void | stop | (void) | Sends a message to the server program to stop playing the sound. The next time a play command is issued, the sound will start again from the beginning |
| public | virtual void | pause | (void) | Sends a message to the server program to pause the sound. The next time a play command is issued, the sound will resume from the point at which it was paused |
| public | virtual void | kill | (void) | Sends a message to the server program to remove the sound, and tells the server object to remove this sound from its list. The sound object should not be used after kill() is called; this is meant to be called from the destructor, and should not generally be used directly by applications |
| public | virtual void | killRemote | (void) | Sends a message to the server program to remove the sound, and tells the server object to remove this sound from its list |
| public | virtual void | createRemote | (void) | Creates Fullpath by calling fullpath and sends message to the associated server to create a sound sample and waits for the resulting handle to store |
| protected | class bergenServer * | server_ | | Reference to bergenServer |
| protected | int | handle_ | | Handle of the sound |
| protected | bool | isPlaying_ | | flag indexing if it is currently playing |

Figure 4: class bergenSound overview

| class bergenTone : public bergenSound | | | | |
|---|---|---|---|---|
| public | | bergenTone | (bergenServer *server) | Creates the tone sound object |
| public | virtual void | setFrequency | (float freq) | Sets the frequency of the sound to be freq cycles per second. The default frequency is 1000 Hz |
| public | virtual void | createRemote | (void) | Creates new Tone Objekt by sending a message to the server to create it |

Figure 5 : class bergenTone overview

| class bergenSample : public bergenSound | | | | |
|---|---|---|---|---|
| public | | bergenSample | (char *filename,bergenServer *server) | The constructor must be given the name of the sample file which the sound will play, in addition to the server object pointer. The Sound Object itself is created by snerd on snerd's side.Snerd is managing the data and return just the Handle to it |
| public | virtual void | setLoop | setLoop(int loop) | Sets the sound's looping flag -- if it is true (non-zero), the sound will\ loop continuously when it is played |
| public | virtual void | createRemote | createRemote(void) | Creates Fullpath bz calling fullpath and sends message to the associated server to create a sound sample and waits for the resulting handle to store |
| public | virtual float | duration | duration(void) | Returns the total playing time of the sample file, in seconds. If the server failed to find the sample file, this will return 0. Note that this function requires sending a message to the server and waiting for the response, so it is potentially slow |
| private | char | filename_ | | Filename if the auiofile |

Figure 6 : class bergenSample overview

| class bergenWhiteNoise : public bergenSound | | | | |
|---|---|---|---|---|
| public | | bergenWhiteNoise | (bergenServer *server) | Creates the white-noise sound object |
| public | virtual void | createRemote | (void) | Creates new White Noise Objekt by sending a message to the server to create it |

Figure 7 : class bergenWhiteNoise overview

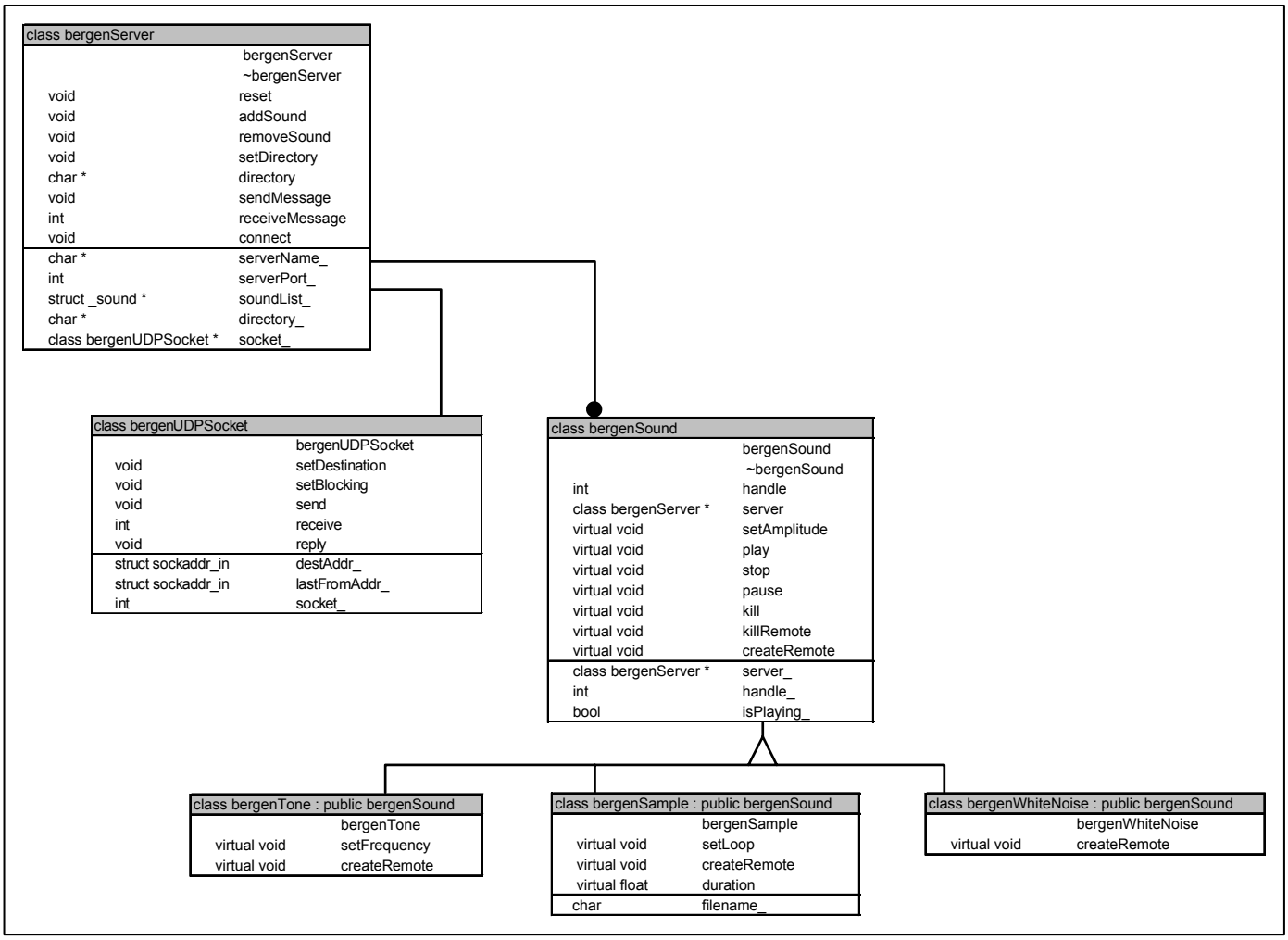


Figure 8 : OMT Diagram client components

### 2.3.2. Server Data Structure

On the server side the main data structure is given within the "snerdServer" class. The "snerdServer" is holding references to a "bergenUDPSocket" used for communication and a "snerDB" class, a database implementation managing binary audio data.
With these references the "snerdServer" data structure offers access to available sound sources and processes any network communication tasks.
The network communication is performed using the "bergenUDPSocket" using a similar data structure as used at the client side.
Beside managing a list of sound objects containing the appropriate digital audio data the "snerdDB" offers functions to operate on these sound data objects.

Offering functionality to update a sound objects buffer or setting an object's attributes controlling the behaviour of any sound object within the "snerdDB" sound list is possible. For this purpose any message received over the network interface or sent by the "snerdServer" connection addressed to any sound object is processed to the appropriate sound object by the "snerdDB".
A sound object can either be a of a "WhiteNoise", "Tone" or "SampleFile" type.
As described within the client data structure section[7] a "WhiteNoise" offers functionality and structures to reproduce a white noise. The "Tone" offers access for producing and replaying a tone of a certain given frequency. Based on a given binary audio file the "SampleFile" offers access to the file's binary data as well as functionality to read, update and replay the audio data. (see figures 9 – 14)

| class snerdServer | | | | |
|---|---|---|---|---|
| public | void | init | (int bufSize, int sampleRate, class bergenUDPSocket *sock) | snerdServer Constructor initialises snerdServerObject with given buffer size, sample rate and the network connection socket |
| public | void | mainLoop | (void) | once called the mainLoop functions only ends when the sner server attribute "looping_" is disabled. With evry loop new arriving messages are processed, the sound objects are updated and the updated and mixed audiobuffe ris written to the audiodevice |
| public | void | returnMessage | (void *msg,int len) | Sends back the message the client just sent to the server |
| public | void | returnInt | (int val) | Sends back theint the client just sent to the server |
| public | void | returnFloat | (float val) | Sends back the float the client just sent to the server |
| private | void | processMessages | (void) | If a message was received over the socket, the message is send to splitMessage for getting messages isolated. The isolated massages are parsed by sending them to parseMessage if they are no sound related ones. Sound messages are forwarded to the according sound object |
| private | int | splitMessage | (char *message,char **strings,int max) | The one message string is split into isolated messages. Ecah message is either seperated by \t \n or ' '. The amount of messages found is returned |
| private | void | parseMessage | (char **message,int numStrings) | Parses the message string and makes according function calls |
| private | void | parseNewSoundMessage | (char **message,int numStrings) | Parses the message for sound objects newly to generate |
| private | class snerdDB * | db_ | | Reference to the sound object "data base" |
| private | class bergenUDPSocket * | socket_ | | Reference to the socket used for network communication |
| private | bool | looping_ | | Flag setting snerd's looping state |

Figure 9 : class snerdServer overview

[7] cp.: section 2.3

| class snerdDB | | | | |
|---|---|---|---|---|
| public | | snerdDB | (int bufsize=1000) | Initialises the snerdDB object with initial values and allocates memory for 'replay buffer' where the samples of all audio objects are mixed into |
| public | | ~snerdDB | (void) | Frees memory allocated for audio object list and 'replay' buffer |
| public | void | addSound | (ASound *) | Adds a sound object to the list of sound objects |
| public | ASound * | findHandle | (int) | Searches the handle to a sound object by given id and returns the handle |
| public | void | update | (void) | Calls every sound objects update routine and mixes the updated buffers down to one the 'replay' buffer |
| public | short * | buffer | (void) | Returns the snerdDb's 'replay' buffer |
| public | int | bufferSize | (void) | Returns the buffersize of the snerDB object |
| public | void | reset | (void) | Deletes all Sounds in snerdDB |
| public | void | setSampleRate | (int rate) | sets the sample rate used for play back |
| public | void | setGain | (float) | Sets the gain for the snerdDB |
| private | short * | buffer_ | | 'Replay' buffer of the snerdDB |
| private | int | bufferSize_ | | size of the 'replay' buffer |
| private | ASound ** | sounds_ | | List of References to sound objects |
| private | int | maxSoundIndex_ | | Amount of sounds in th elist |
| private | int | sampleRate_ | | actual sample rate used fro replay |
| private | float | gain_ | | actual overall gain for replay |

Figure 10 : class snerdDB overview

| class ASound | | | | |
|---|---|---|---|---|
| public | | Asound | (int bufsize=1000) | Initialises ASound Object with default values and allocates memory for storing audio data |
| public | | ~Asound | (void) | The memory occupied by the buffer is freed |
| public | virtual short * | updateBuffer | (void) | Returns the actual buffer |
| public | virtual int | removable | (void) | returns flag to initialise that object kann be removed |
| public | virtual int | active | (void) | Returns 0 to determine that Object is not playing |
| public | int | handle | (void) | Returns the handle_ on the object |
| public | short * | buffer | (void) | Returns the reference to the object's audio buffer |
| public | int | bufferSize | (void) | Returns the size of the audio buffer |
| public | virtual void | setAmplitude | (float amp) | Sets the amplitude to the new given amplitude |
| public | virtual float | amplitude | (void) | Gets the object's current amplitude |
| public | virtual void | play | (void) | not implemented |
| public | virtual void | stop | (void) | not implemented |
| public | virtual void | pause | (void) | not implemented |
| public | virtual void | kill | (void) | not implemented |
| public | virtual void | message | (char **msg,int num) | Executes the right action refereing to the messages coming in |
| public | void | setSampleRate | (int rate) | Sets the object's sampling rate to the given sampling rate |
| protected | short * | buffer_ | | Buffer holding the Asamples samples |
| protected | int | bufferSize_ | | Size of the buffer_ |
| protected | int | handle_ | | Handle to the Asample |
| protected | float | amplitude_ | | Asamples amplitude |
| protected | int | sampleRate_ | | Asamples sample rate |
| private | static int | NextHandle | | Handle to the next Sample |

Figure 11 : class ASound overview

| class SampleFile : public ASound | | | | |
|---|---|---|---|---|
| public | | SampleFile | (char *filename,int bufsize=1000) | |
| public | | ~SampleFile | (void) | |
| public | virtual short * | updateBuffer | (void) | Returns the actual buffer |
| public | virtual int | removable | (void) | returns flag to initialise that object kann be removed |
| public | virtual int | active | (void) | Returns 0 to determine that Object is not playing |
| public | virtual void | play | (void) | Sets the playing_ flag for the sample file |
| public | virtual void | stop | (void) | Sets back the playing_ flag and sets the file reading position to the start of the audiofile |
| public | virtual void | pause | (void) | Unsets the playing_ flag |
| public | virtual void | kill | (void) | Sets the killed_ flag and closes the associated audio file |
| public | virtual void | message | (char **,int) | Executes the right action refereing to the messages coming in |
| public | virtual void | setLoop | (int loop) | Sets loop_ flag |
| public | virtual int | loop | (void) | Is returning the loop state of the SampleFile Object |
| public | virtual char * | filename | (void) | Is returning the filename of the audio file that is associated with the sample file object |
| private | int | readFrames | (void) | Decides which method to use for reading audiofile and deligates |
| private | void | convertFrames | (void) | Converts sample format to standard c datatypes |
| private | void | initializeRawFile | (void) | Initialises SampleFile Object with initial values for mono file on stereo replay hardware (frameSize_=2) |
| private | void | initializeAFFile | initializeAFFile(void) | Initialises SampleFile Object with values taken from the AIFF file header |
| private | int | readFrames | (void) | Decides which type of audio file and delegates to either readRawFrames or readAFFFrames |
| private | int | readRawFrames | readRawFrames(void) | Reads the samples out of a raw audio file |
| private | int | readAFFrames | readAFFrames(void) | Reads the samples out of a AIFF audio file into tempbuffer_ |
| private | char * | filename_ | | Filename of the audio file to read from |
| private | int | isRawFile_ | | Flag if audio file is a raw file |
| private | int | rawfd_ | | |
| private | AFfilehandle | afFile_ | | |
| private | int | loop_ | | flag if playing looped |
| private | int | playing_ | | flag indexing if source is plazing |
| private | int | channels_ | | amount of channels of the audio file (may be a stereo file) |
| private | int | sampleFormat_ | | sort of AIFF sampleformat if AIFF samplefile |
| private | int | sampleWidth_ | | 16 or 8 (16bit or 8bit format possible) |
| private | int | bytesPerSample_ | | 2 or 1 (16bit or 8bit) |
| private | int | fileRate_ | | SampleRate of the audio file |
| private | unsigned char * | tempBuf_ | | Buffer the audio files are read to before converting them |
| private | short * | inputBuf_ | | Buffer holding the finally converted audio samples |
| private | int | inputFrames_ | | Frames to be read from file ? |
| private | int | frameSize_ | | size of one frame |
| private | int | numFrames_ | | Total amount of frames in file ? |
| private | int | killed_ | | flag inexing if samplefile is marked to be killed |

Figure 12 : class SampleFile overview

| class Tone : public ASound | | | | |
|---|---|---|---|---|
| public | | Tone | Tone(int bufsize=1000); | Initializes ToneFile Object with initial values |
| public | | ~Tone | ~Tone(void); | Calls kill routine in order to set killed_ flag to be marked for removable |
| public | virtual short * | updateBuffer | (void) | Calculates samples to be played next |
| public | virtual int | removable | (void) | Calls kill routine in order to set killed_ flag to be marked for removable |
| public | virtual int | active | (void) | Returns the value of the playing flag to determine if WhiteNoise Objekt is playing |
| public | virtual void | play | (void) | Sets the playing_ flag for the sample file |
| public | virtual void | stop | (void) | Sets back the playing_ flag and sets the file reading position to the start of the audiofile |
| public | virtual void | pause | (void) | Unsets the playing_ flag |
| public | virtual void | kill | (void) | Sets the killed_ flag and closes the associated audio file |
| public | virtual void | message | (char **msg,int) | Executes the right action refereing to the messages coming in |
| public | virtual void | setFrequency | (int freq) | Sets the frequency to the given value |
| private | int | frequency_ | | Frequency of the Tone |
| private | int | playing_ | | flag indexing if source is plazing |
| private | int | killed_ | | flag inexing if samplefile is marked to be killed |
| private | unsigned int | count_ | | Position in Wave of the Tone to be produced as a reference for continous tone reproduction |

Figure 13 : class Tone overview

| class WhiteNoise : public ASound | | | | |
|---|---|---|---|---|
| public | | WhiteNoise | (int bufsize=1000) | Initialises White Noise Object with default values |
| public | | ~WhiteNoise | (void) | Stops audio play and set killed flag by calling kill |
| public | virtual short * | updateBuffer | updateBuffer(void) | Calculates samples to be played next |
| public | virtual int | removable | removable(void) | Calls kill routine in order to set killed_ flag to be marked for removable |
| public | virtual int | active | active(void) | Returns the value of the playing flag to determine if WhiteNoise Objekt is playing |
| public | virtual void | play | play(void) | Sets the playing_ flag for the sample file |
| public | virtual void | stop | stop(void) | Sets back the playing_ flag and sets the file reading position to the start of the audiofile |
| public | virtual void | pause | pause(void) | Unsets the playing_ flag |
| public | virtual void | kill | kill(void) | Sets the killed_ flag and closes the associated audio file |
| private | int | playing_ | | flag indexing if source is plazing |
| private | int | killed_ | | flag inexing if samplefile is marked to be killed |

Figure 14 : class WhiteNoise overview

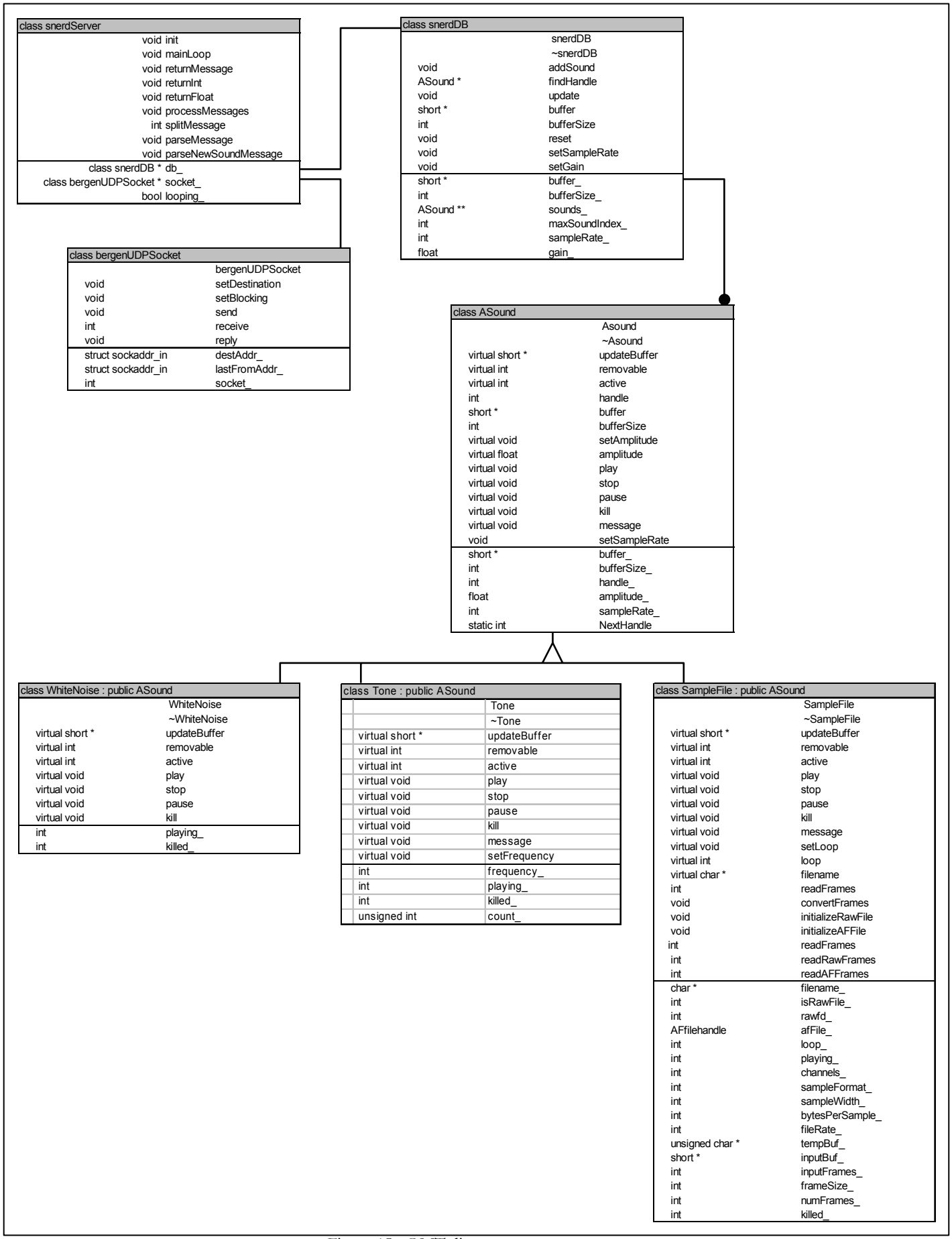


Figure 15 : OMT diagram server components

## 2.4. Communication and Data Flow Model

The components application, client module, server module and audio hardware, forming the layer model are communicating in between but are further communicating with objects they hold references to.
The communication is mainly a one way communication following the way from the top to the bottom, the hardware layer. The layers are offering services for the layer above them offering service and data.
The data flow therefore is being transferred in two directions. The superior layers offer information the underlying layer may need for fulfilling the provided services. The service's result is then either send back to the upper layer or further transmitted as base for following services to the lower layer.
The services can be categorised by the layer they are offered by.

### 2.4.1. Client Services and Dataflow

Any application using the sound server holds a reference to a "bergenServer" object. The "bergenServer" object holds a reference to a list of "bergenSound" objects which containing handles to binary data objects at the client side. It also holds a reference to a "bergenUDPSocket" object containing connection information used for client server communication.
The application uses the "bergenServer" object to build up a list of sounds. To build up the list of sounds the client is communicating with the "snerdServer" using the "bergenUDPSocket" object.
The client requests a new sound from the server which then constructs a "ASound" object and reaches back the reference to that object, so the client can identify it in the future.
To control the state of any sound, the application is sending request to change a sound's or the server's state to the "bergenServer" which then transfers it to the "snerdServer". Success or failure of any requests as well as the information about an object ,if requested, is reported to the application. (See figure 16)

### 2.4.2. Server Services and Dataflow

The server's main functionality is implemented in a "snerdServer" object. The "snerdServer" object is communicating with the client using a "bergenUDPSocket" object. The "snerdServer" object is coordinating the activities on the server side.
It holds a handle to the audio device used for replaying audio data and a reference to a "snerdDB" object.
The "snerdDB" object is treated as a source of digital audio on the one hand and as an access point to the list of server "ASound" objects on the other hand.
Viewed from the idea of digital audio data source, the "snerdDB" offers the functionality of keeping ready an updated audio buffer ready for replay.
To offer the audio buffer the "snerdDB" successfully calls every "ASound" in the list of sounds to return their updated buffers which are then mixed down to one buffer by the "snerdDB" object. The updated audio buffer is frequently requested by the "snerdServer" for continuous replay.
From the sound list access point of view, the "snerdDB" offers access to each sound source in the sound list which is used by the "snerdServer" object in order to set or query the state of any of the "ASound" objects within this list. (See figure 16)

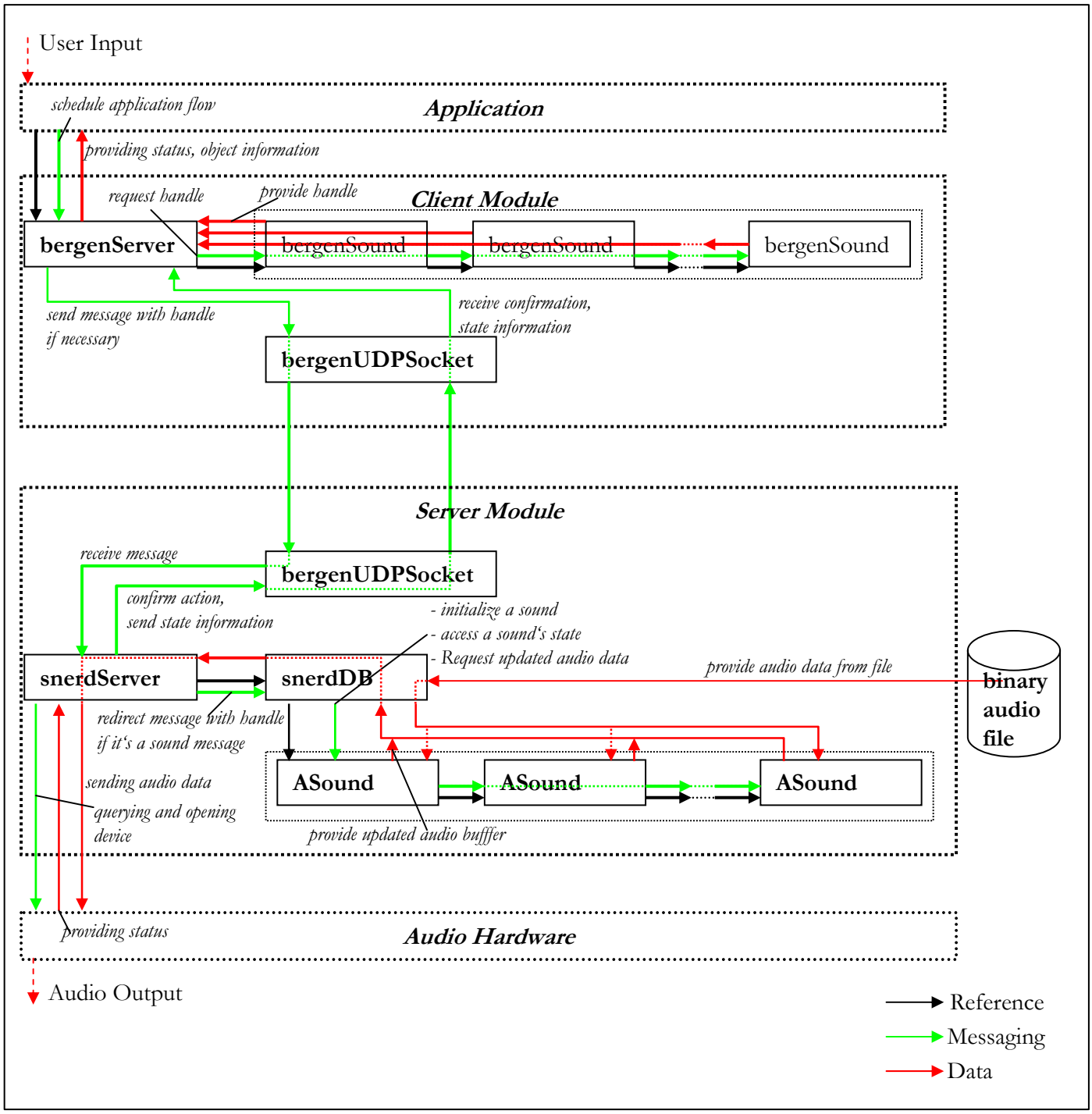


Figure 16 : Communication and Data Flow Model

### 2.4.3. Client – Server Communication

The client and server modules communicate using a UDP socket. Messages are send from the client side to the server side. The server is responding to certain messages sending a notification or a the performed action's return value back to the client.
The client is capable of controlling the "snerdDB" sound database, the "snerdServer" as well as individual sound objects.
Messages are assembled using a unique identifier, a command and optional attributes in the format :

<message> ::= <identifier> "," <command> "," <attribute>
<identifier> ::= {-1 ... MAX_INTEGER}
<command> ::= {"new sample" "new tone" "new whitenoise" "newsample"
"newtone" "newwhitenoise" "gain" "cd" "kill" "reset" "ping" }
<attribute> ::= { <floatvalue> <stringvalue> }
<floatvalue> ::= {0.0 ... MAX_FLOAT }
<charvalue> ::= { a ... z A ... Z}*

Using "-1" as identifier is marking the message to be assembled either for the "snerdDB", the "snerdServer" or sound objects to be built within the action triggered by the message itself.
Possible combinations can be retrieved by the table of messages. (See figure 17)

| Identifier | Command | Attribute | | Snerd Action |
|---|---|---|---|---|
| | | Type | Description | |
| -1 | new sample | char* | filename | Create a new sample object from the audio file with the given filename |
| -1 | newsample | char* | filename | Create a new sample object from the audio file with the given filename |
| -1 | new tone | int | frequency | Create a new tone object with the given frequency |
| -1 | newtone | int | frequency | Create a new tone object with the given frequency |
| -1 | new whitenoise | --- | | Create a new whitenoise object |
| -1 | newwhitenoise | --- | | Create a new whitenoise object |
| 0 ... MAX_INTEGER | gain | float | new gain | Adjust the gain of the sound object with the given identifier |
| -1 | cd | char* | new directory | Change the server's reference directory to the given one |
| 0 ... MAX_INTEGER | kill | --- | | Kill the sound object with the given identifier |
| -1 | reset | --- | | Reset the snerdDB |
| -1 | ping | --- | | Send back a "pong" message to the client as respond |
| 0 ... MAX_INTEGER | play | --- | | Set the 'play_' attribute of the sound object with the given identifier |
| 0 ... MAX_INTEGER | stop | --- | | Set the 'stop_' attribute of the sound object with the given identifier |
| 0 ... MAX_INTEGER | pause | --- | | Set the 'pause_' attribute of the sound object with the given identifier |
| 0 ... MAX_INTEGER | loop | {1,0} | new loop value | Set the 'loop_' attribute of the sound object with the given identifier to the given value |
| 0 ... MAX_INTEGER | setfrequency | int | new frequency | Set the 'frequency_' attribute of the sound object with the given identifier to the given value |

Figure 17 : Client – Server Communication Messages

### 2.4.4. Audio Processing Control

While the main necessary data is stored and managed at the server side, the client is controlling the audio processing. With the appropriate messages the client is able to control each of the involved sound's attributes as well as the "snerdServer" and the "snerdDB" itself. Any sound within the list managed by the "snerdDB" can be replayed by the "snerdServer".

The client module can control the state of a sound source through the handle that the client is managing within his list of sound handles. Through setting the state of "play_", "stop_" and "pause_" the client is able to determine if a sound is being played back. Setting a sound source's "amplitude_" attribute enables control of the intensity the sound is replayed with.

Within the current Ygdrasil framework currently used as client module at EVL an amplitude for each sound is calculated depending on it's distance from the listener in the virtual world.

The calculations performed on the client side are used to set each sound source's "amplitude_" attribute depending on a linear distance model using the "gain" message. During replay the "snerdServer" continuously calls the buffer update function of the "snerdDB".
The "snerdDB" object then is calling the buffer update function of each of the sounds within the sound list for which the "active_" and "play_" attribute is set.. The individual sound sources return a buffer of samples to be replayed next depending on their implementation. The buffers returned by the individual functions are downmixed to one singular buffer at the "snerdDB" object.
The singular buffer of the "snerdDB" object is used by the "snerdServer" for replaying it depending on the audio device being opened and the amount of channels it is opened with. (Currently every device is opened in one channel mode)

### 2.5. Sound Application Application Flow

For a executable sound application that uses the "Bergen Sound Server" the "snerdServer" must be running on either the local machine where the application itself is running on or on a remote workstation that is reachable over the network using TCP/IP. To start the "snerdServer" the sources have to be compiled and the "snerd" program must be started.[8]
Each application's life cycle passes through three stages.
Starting with the initialization phase, a "bergenServer" is initialised and connected to a running "snerdServer".
After having initialised a "bergenServer" the sounds being used within the application are initialised and registered to the "bergenServer" (e.g. by setting the amplitude).
In a second phase, the interaction phase, the behaviour of each sound object can be adjusted to control the generated sound characteristics which is calculated by the "snerdDB" and replayed by the "snerdServer".
Each application has to end with a termination phase in which all of the sounds and the "bergenServer" are to be killed.

---

[8] cp.: Pape, Dave : Bergen Sound Server & Library, Version 0.4.1, Electronic Visualization Laboratory – Universit of Illinois at Chicago, http://www.evl.uic.edu/pape/sw/bergen/, 11/10/2002

Sample Code Application Flow :

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include "bergenServer.h"
#include "bergenTone.h"
#include "bergenWhiteNoise.h"
int main(int argc,char **argv)
{
/************************
 * Initialization Phase *
 ************************/
bergenServer * server = new bergenServer;
bergenTone * tone = new bergenTone(server);
/************************
 * Interaction Phase    *
 ************************/
tone->setAmplitude(0.25);
tone->play();
sleep(3);
tone->setFrequency(2000.0);
sleep(1);
tone->setFrequency(500.0);
sleep(1);
/************************
 * Termination Phase    *
 ************************/
delete tone;
delete server;
return 0;
}
```

# 3. Sound Spatialization

Various techniques are applicable to obtain the functionality of spatialized sound. The techniques are differentiated by three criteria.
Most obvious the techniques use different audio hardware set-ups for audio reproduction. There are setups available using either loudspeakers or headphones while the technique using loudspeakers are further divided through the amount of channels used for reproduction.
Some techniques applicable to either headphone or loudspeaker are less obviously distinguishable by the software rendering technique to calculate the final sound characteristics.
The main quality aspect that the available techniques are differentiated by is the ability to mediate the intended spatial impression with the sound characteristics being replayed. Within this section varying techniques are introduced and discussed.

## 3.1. Loudspeaker Array based Audio Reproduction

Using loudspeakers to reproduce spatial sound characteristics a techniques are used based on stereophonic techniques. Using stereophonic techniques, each loudspeaker is connected to a different separate channel for audio reproduction.
With stereophonic techniques sound appearing to originate from a position anywhere on a direct path between the used loudspeakers is reproduced. This positional effect is realized using the available loudspeakers and adjusting the gain of each channel in order to determine the position of the virtual sound source. (See figure 18)

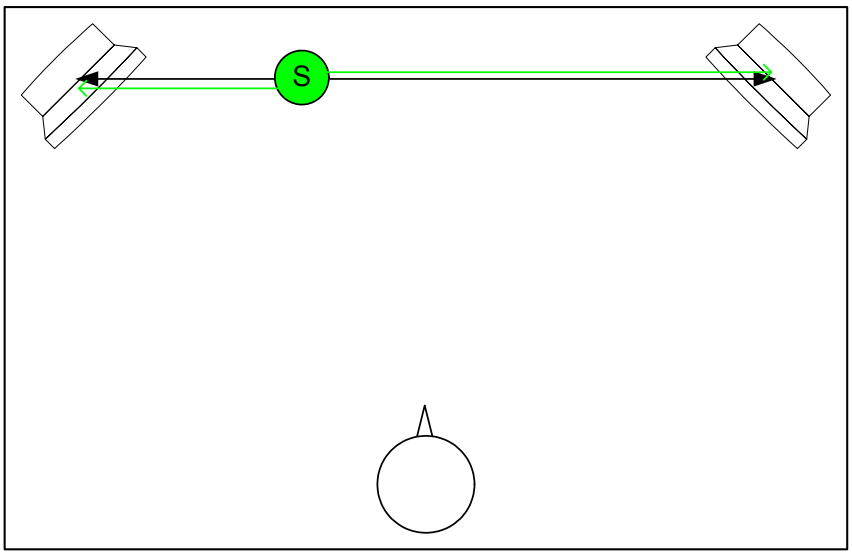

Figure 18 : Stereophonic sound positioning using two loudspeakers

The position of the sound being reproduced is determined by the position of the loudspeakers within the physical reproduction environment. The room spanned by vectors between the loudspeakers is limiting the positions from which the sounds can appear to originate from. For greater freedom of positioning the amount of channels is increased to widen the room.

For a listener situated in a position within the room spanned by the vectors between the loudspeaker it is then possible to derive information about a sound's origin's direction relative to his position. (See figure 19)

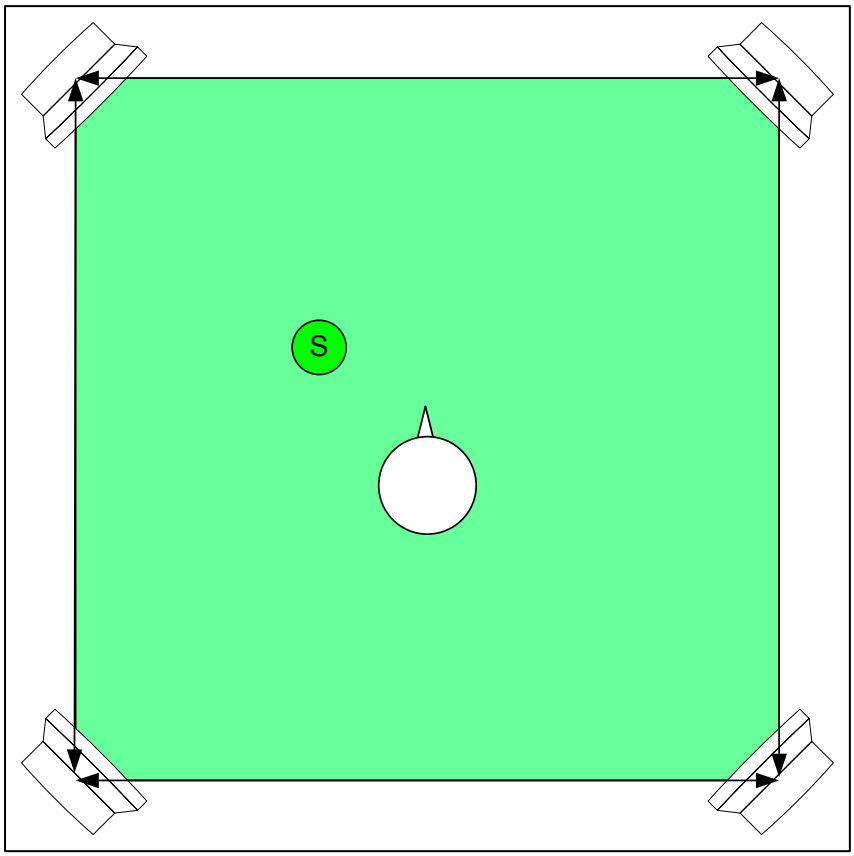


Figure 19 : Sound positioning using four loudspeakers

Adding an additional adjustment of a sound's overall amplitude it is possible to mediate the impression of distance.

With directional and distance information the listener is able to determine a sound's position which then can appear to be outside the room limited by the vectors between the loudspeakers. (See figure 20)

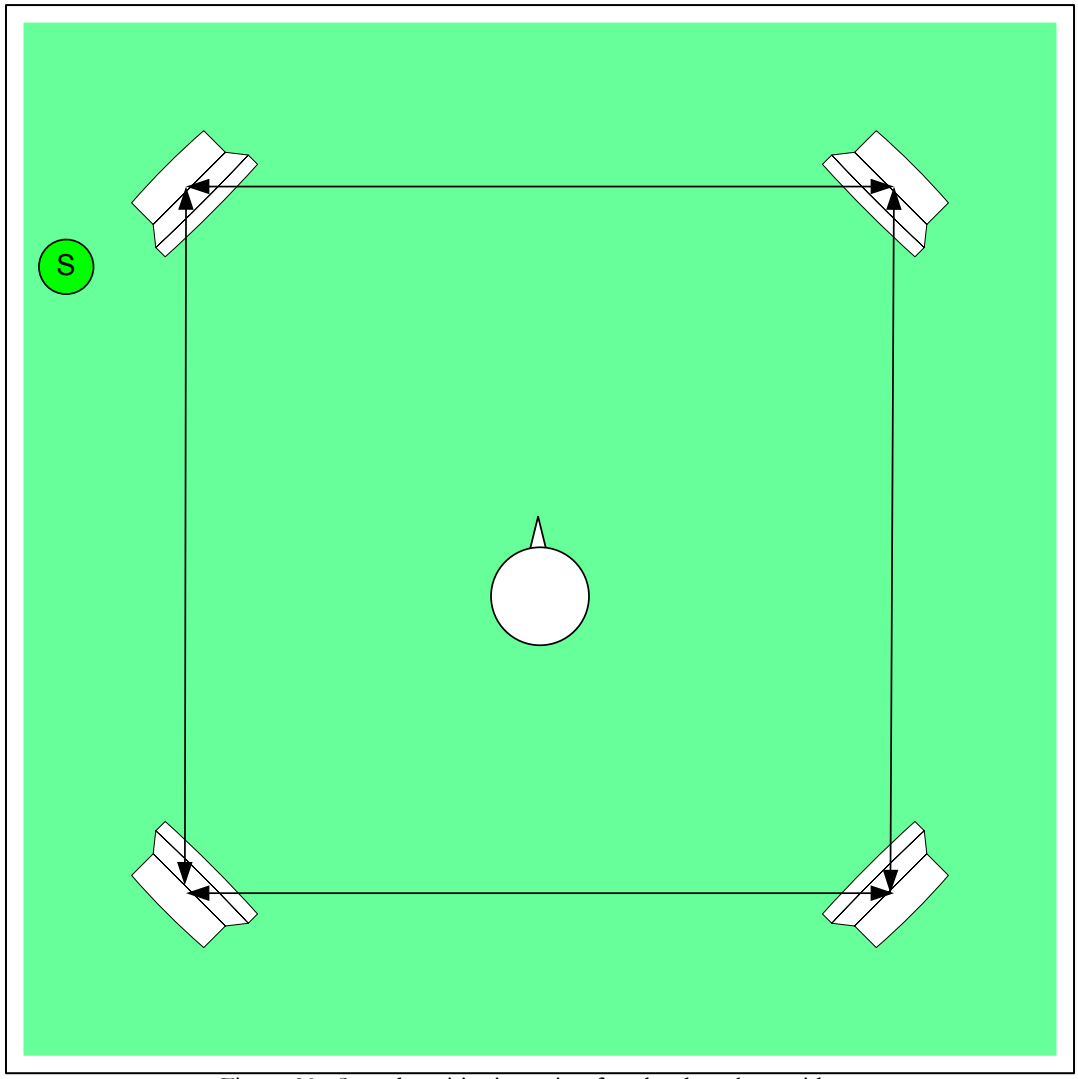


Figure 20 : Sound positioning using four loudspeakers with additional overall amplitude adjustment

Since stereophonic techniques are reproducing the position of a sound source the coordinates of the sound source's representation within the application relative to the listener as well as the position of the real listener within the physical reproduction facility have to determinable and stable.

### 3.1.1. 1 Channel Loudspeaker Reproduction

If only one channel is available for audio reproduction the room spanned by the loudspeaker vectors is limited to a point. Sounds being replayed always appear to originate from the direction of the loudspeaker's position.
Adjusting the overall amplitude of the sound makes distance information available for the user.
This technique can be used with several speakers connected to one audio channel. Then the direction of sound is set to the loudspeaker set-up's hotspot. The hotspot is determined as the centre of the room which is spanned by the loudspeaker's vectors.
A headphone attached to one audio channel can also be used. Using headphones the sound's direction seems to be inside the head.
Since sounds being replayed always appear to originate from one point, either the loudspeaker's position or the loudspeaker's hotspot, no directional information can be mediated with one channel loudspeaker reproduction techniques.
Beside the directional information a sound source's distance from the listener can be encoded using one channel only. Adjusting the sound source's amplitude following a chosen distance model it is possible to simulate the sound source to originate from a position further away or closer to the listener. Several distance models are applicable.

#### 3.1.1.1. Inverse Square Law Distance Attenuation

Several studies have proven that a decrease of 6dB in a sound source's intensity with every doubling of distance results in the most natural impression when applied as a distance model for distance attenuation.[9]
With the Inverse Square Law an initial reference distance is given describing a sphere around the sound source's centre at which's surface the sound is heard unattenuated. Moving further away from the sound source's the sound source's initial amplitude is attenuated by 6dB with every doubling of the reference distance. Moving closer to the sound source than the reference distance, the sound source's amplitude is increased by 6dB with every bisecting of distance.

---

[9] cp.: Begault, Durand R. : Preferred Sound Intensity Increase For Sensation Of Half Distance, Moffet Field, AES 93rd Convention San Francisco, 1992

Following the Inverse Square Law a sound source's amplitude is determined by a logarithmic formula (See figure 21)

$$\text{Amplitude} = \text{Initial Amplitude} - 20 * \log10 * \left(1 + \frac{\text{Actual Distance} - \text{Reference Distance}}{\text{Reference Distance}}\right)$$

Figure 21 : Inverse Square Law

To offer anybody working with the Inverse Square Law more influence on the attenuation performed with increase of distance a roll off factor is involved in a modified version of the Inverse Square Law algorithm. Using a roll off factor it is possible to increase or decrease the attenuation performed with a doubling of distance.(See figure 22)

$$\text{Amplitude} = \text{Initial Amplitude} - 20 * \log10 * \left(1 + \text{Roll of Factor} * \frac{\text{Actual Distance} - \text{Reference Distance}}{\text{Reference Distance}}\right)$$

Figure 22 : Inverse Square Law with Roll off Factor

A roll off factor of 1 does not influence the effect of the Inverse Square Law. A factor $> 1$ does strengthen the effect while a factor $< 1$ weakens the effect.

#### 3.1.1.2. Clamped Inverse Square Law Distance Attenuation

The Clamped Inverse Square Law is a further modification of the Inverse Square Law. Introducing a maximum and minimum amplitude the amplitude calculated by the Inverse Square Law is clamped down to the maximum and minimum amplitude values. (See figure 23)

$$\text{Amplitude} = \text{Initial Amplitude} - 20 * \log10 * \left(1 + \text{Roll of Factor} * \frac{\text{Actual Distance} - \text{Reference Distance}}{\text{Reference Distance}}\right)$$

$$\text{Amplitude} = \text{Max}(\text{Amplitude}, \text{Minimum Amplitude})$$

$$\text{Amplitude} = \text{Min}(\text{Amplitude}, \text{Maximum Amplitude})$$

Figure 23 : Clamped Inverse Square Law

### 3.1.1.3. Linear Distance Attenuation

With the linear distance attenuation the sound source's amplitude is attenuated between a given reference distance and a fall off distance. The initial amplitude is given for the sound source being closer or at the same distance given by the reference distance. For sound sources being as far away as given by the fall off distance or even further away, the amplitude is zero.

To attenuate the amplitude between the two distances a linear approach is used. (See figure 24)

$$\text{Amplitude} = 1 - \frac{\text{Actual Distance} - \text{Reference Distance}}{\text{Fall off Distance}}$$

Figure 24 : Linear Distance Attenuation

### 3.1.1.4. Distance Attenuation by Factor

Given a reference distance and an attenuation factor the sound source is attenuated for distances greater than reference distance. For sound sources closer to or as far away as reference distance the initial amplitude is not attenuated. For greater distances the sound source is attenuated by distance times attenuation factor. (See figure 25)

$$\text{Amplitude} = 1 - (\text{Actual Distance} - \text{Reference Distance}) * \text{Attenuation Factor}$$

Figure 25 : Distance Attenuation by Factor

### 3.1.2. 2 Channel Audio Reproduction

Using two channels for audio reproduction a sound source can appear to originate from a direction placed on a direct path between the loudspeakers attached to the two channels. (See figure 18)

Adding overall amplitude adjustment then enables the sound source to be located at a position on a path from the listener in the estimated direction. (See figure 26)

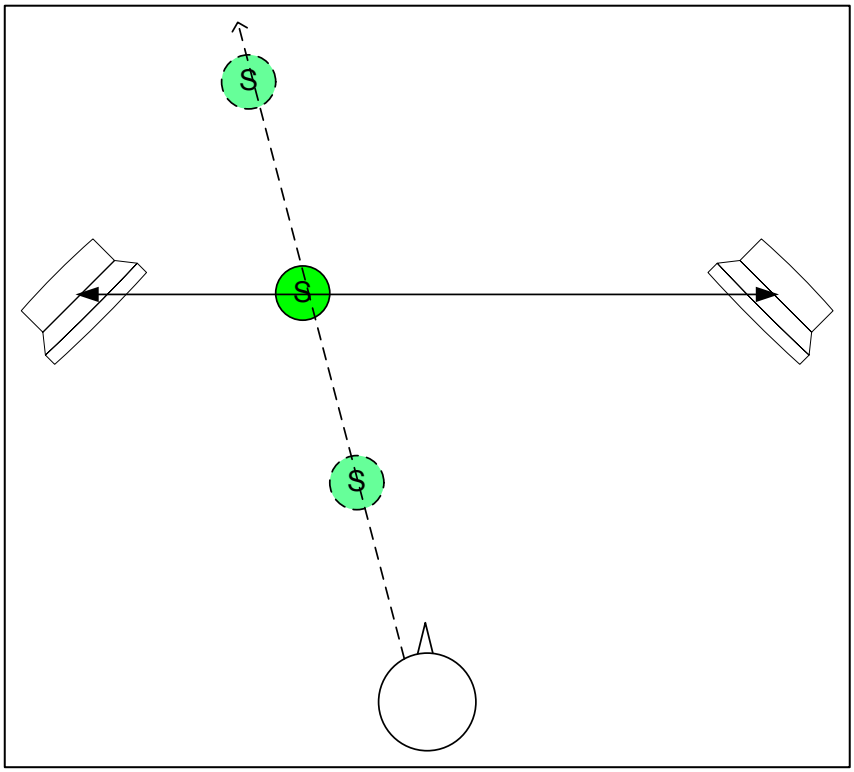

Figure 26 : 2 Channel Audio Reproduction

To calculate each individual loudspeaker's amplitude according to the sound source's position and the loudspeaker set-up the "Chowning Panning Law" is used. With the "Chowning Panning Law" each loudspeaker's gain is adjusted according to the speaker angle ( $\alpha$ ) and the sound source's angle ( $\beta$ ). (See figure 27 and figure 28)

$$\text{Amplitude}_{\text{Speaker 1}} = \sqrt{\frac{\alpha_{\text{Speaker 2}} - \beta}{\alpha_{\text{Speaker 1}} - \alpha_{\text{Speaker 2}}}}$$

$$\text{Amplitude}_{\text{Speaker 2}} = \sqrt{\frac{\alpha_{\text{Speaker1}} - \beta}{\alpha_{\text{Speaker 2}} - \alpha_{\text{Speaker 1}}}}$$

Figure 27 : Chowning Panning Law

To calculate correct amplitude adjustment for sound sources being positioned behind the listener, the sound sources are temporary mirrored at the listener coordinate system's x-axis for calculation.

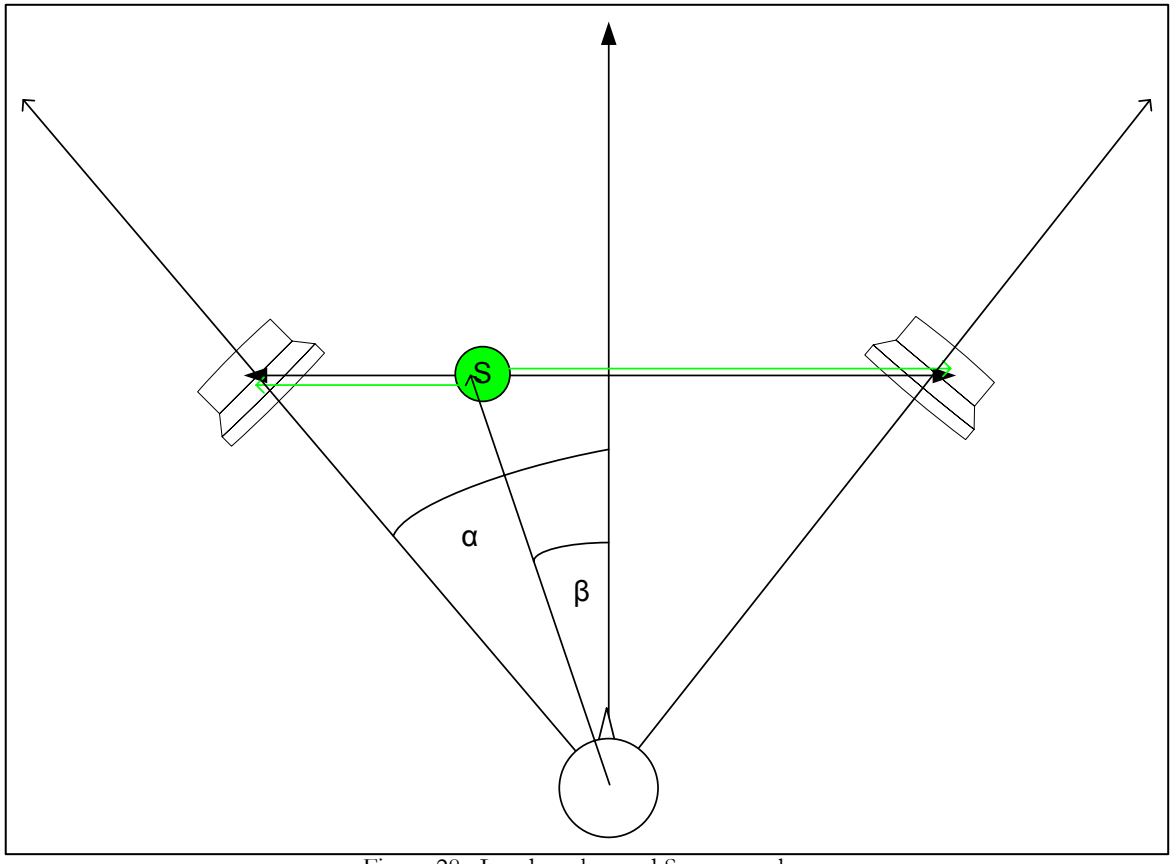

Figure 28 : Loudspeaker and Source angle

Using two channels for audio reproduction is applicable to headphone reproduction but is connected to further challenges in calculating the appropriate sound characteristics. Since the headphones are mounted at the listeners head, the listener's orientation has to be taken account. With turning the head, the position of the audio set-up is changed leading to the need to adjust the channel's amplitude regarding the new headphone orientation. For a successful implementation an accurate and low latency listener tracking is needed, hence a two channel implementation using headphones is not recommendable.

### 3.1.3. Multichannel Audio Reproduction

Techniques exceeding the amount of two channels used for spatialization are termed multichannel audio systems.
Using a set-up of three loudspeakers is doable but mostly not implemented, since the major audio replay techniques offer pairs of channels for audio reproduction.

#### 3.1.3.1. 4 Channel Audio Reproduction

Choosing a four channel loudspeaker set-up a plane becomes available on which the sound source can be positioned on. The plane is spanned by the vectors between the loudspeakers. (See figure 19)
Until an all over amplitude adjustment is added the plane is restricted by the loudspeaker positions. Including the amplitude adjustment disposes this restrictions and opens up the plane. (See figure 20)
For four channels the Ambisonic technique is used. Ambisonic is an amplitude panning method in which a sound signal is applied to all loudspeakers placed evenly around the listener with adjusted amplitudes for each individual loudspeaker. (See figure 29)

$$\text{Amplitude} = \frac{1}{N} * (1+2\cos(\alpha))$$

Figure 29 : Ambisonic Spatialization

Each loudspeaker's amplitude is calculated depending on the total amount of loudspeakers, the angle to the loudspeaker the amplitude is calculated for and the angle to the sound source that is to spatialized.
Using the Ambisonic technique for spatialization the sound signal emanates from all loudspeakers which causes spatial artefacts.
Second order Ambisonic applies prominently lower absolute values to loudspeakers on the opposite side of the current panning direction resulting in lower spatial artefacts. (See figure 30)
For best results second order Ambisonic is used for spatialization with the spatialized sound server implementation.

$$\text{Amplitude} = \frac{1}{N} * (1+2\cos(\alpha) + 2\cos(2\alpha))$$

Figure 30 : Second Order Ambisonic Spatialization

### 3.1.3.2. 5.1/6.1/7.1 Channel Audio Reproduction

Adding more channels in the same plane can improve the precision with which the direction can be determined.

Relative to a fixed listener position, assumed to be in the hotspot of the loudspeaker set-up, the amount of channels in the front is increased to obtain an improved resolution in mediating the directional component of a sound. (See figure 27)

In addition to the main speakers an additional speaker is implemented within these techniques which is not providing any directional information. The additional low frequency speaker is omitting low frequency omni directional sounds which do not have to be spatialized, since the human hearing is not capable of locating low frequency sound sources.

Due to the set-up's asymmetric set-up it is barely used within Virtual Reality environments with flexible listener movement.

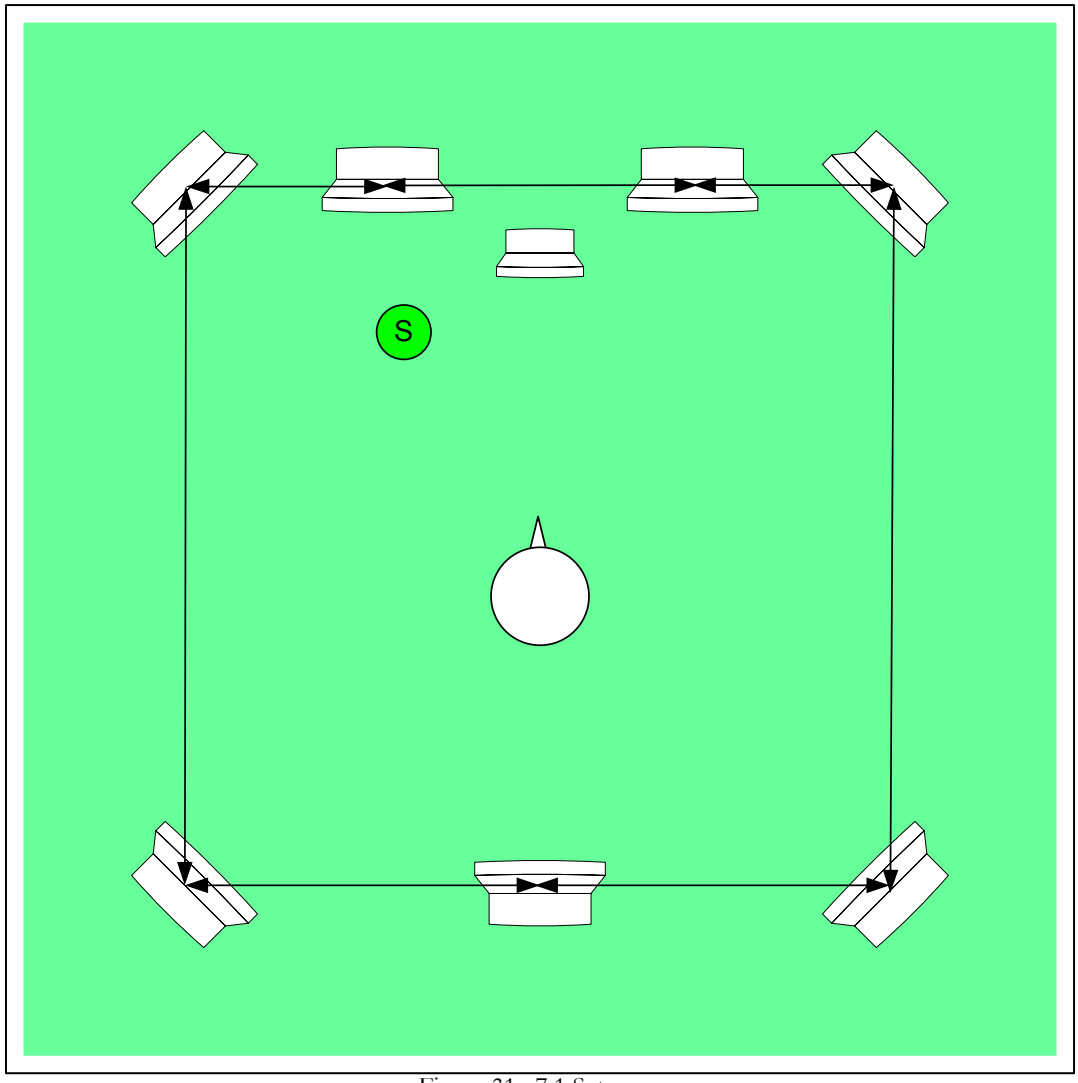


Figure 31 : 7.1 Setup

### 3.1.3.3. 8 Channel Audio Reproduction

In order to obtain spatialized sound exceeding the dimensionality of a plane it is necessary to chose a loudspeaker set-up that is spread in three dimensions.
For a regular geometrical loudspeaker set-up at least eight channels must be chosen. Using eight channels the vectors between the loudspeakers span a room in shape of cube with a listener hotspot in the centre of the cube. (See figure 32)
Using a regular geometric set-up guaranties the same directional resolution for all directions independent of the position within the loudspeaker set-up.

Until the amplitude adjustment is integrated a sound can be replayed appearing to originate from a position within the cube.
The additional amplitude adjustment opens up the boundaries of the cube. The loudspeakers' amplitudes may be calculated analogue to the four channel set-up using the Ambisonic technique

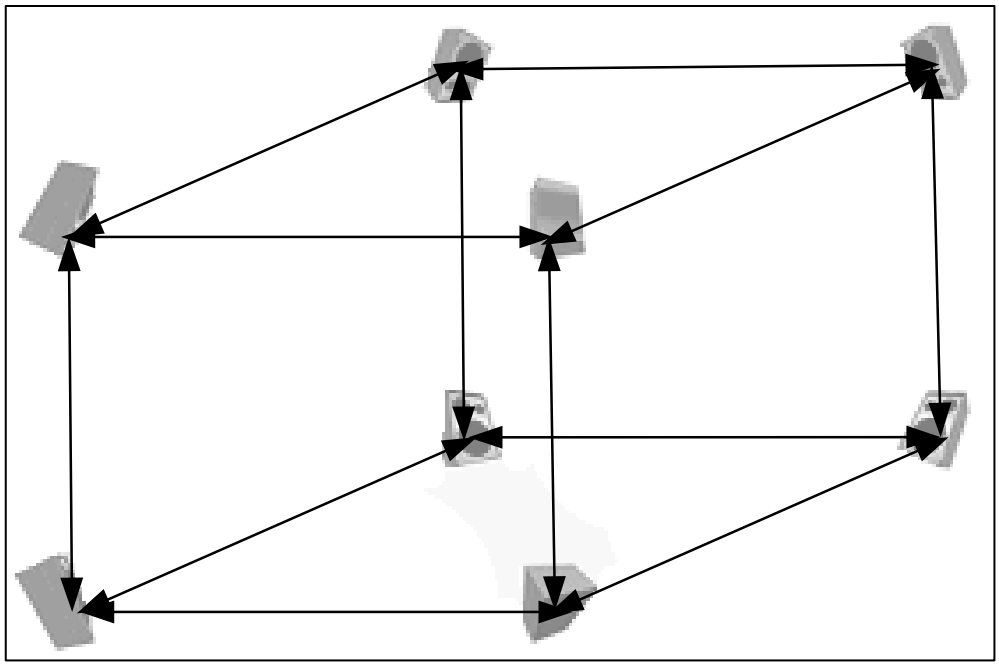

Figure 32 : Eight channel cube set-up

### 3.1.3.4. Sound Sphere Audio Reproduction

With further increment of the amount of channel used for spatialization the directional precision can be improved.Adding additional speakers an regular geometric set-up is should be chosen to keep a constant resolution for all directions.
Adding four additional speakers could lead to two hexagonal set-ups on top of each other. With the increasing amount of speakers the set-up is coming closer to form a sphere surrounding the listener.
Due to human physical acoustic limitations in resolving a sound's directional limitation exeeding a certain amount of speakers is not improving directional resolution.[10]

---

[10] cp.: Kendall, Gary : A 3-D sound Primer, http://www.northwestern.edu/musicschool/classes/3D/pages/3DsoundPrimer.html, 08/02/2002

### 3.2. Headphone based Audio Reproduction

Beside the mentioned implementation of mainly loudspeaker concerned one and two channel reproduction techniques[11] binaural spatialization techniques are used for headphone based spatialization.

Binaural techniques take advantage of the fact, that human hearing is based on the signals received at the two ear drums. Any spatial information is derived from difference between the two audio waves received at the ear drums.

Depending on the position of a sound source the audio wave emitted by that sound source is arriving at the ear drums at a slightly different time (interaural time difference) and with a slightly different intensity (interaural intensity difference).

In addition to the differences in time and intensity the audio wave is influenced by the listener's body. The shoulders, the head as well as the outer ears and other factors partly reflect, diffract or attenuate the audio wave arriving from the sound source.[12]

The individual influences on an arriving audio wave a describable within a functional coherence. The head related transfer function (HRTF) is describing an individual's influence on the audio wave perception at the ear drums. The HRTF can be analysed using special microphones placed close to the individual's ear drums.[13]

Knowing a person's HRTF makes it possible to simulate the conditions at the person's ear drums for any given audio wave.

This approach is used for headphone based spatialized sound. For any given sound source the HRTF is calculated with regard to the desired sound source position.

The resulting sound characteristics appears to the listener as under natural conditions.

Since the estimation of individual HRTF is complicated, time consuming and expensive, non individual HRTF have been derived from several individual HRTF which take the major aspects of human spatial audio perception into account while neglecting other more specific aspects.

---

[11] cp.: sections 3.1 and 3.2

[12] cp.: Tonnesen, Cindy; Steinmetz, Joe : 3-D sound Synthesis, Washington, The Encyclopedia of Virtual Environments, http://www.hitl.washington.edu/scivw/EVE/I.B.1.3DSoundSynthesis.html, 1993

[13] cp.: Gardner, William G.: 3-D Audio Using Loudspeakers, Boston, Massachusetts Institute of Technology, Kluwer Academic Publishers, Norwell, Massachusetts, 1998

The more specific aspects are not relevant for the bigger part of listening situations but lead to increased directional failure in critical listening situations, e.g. when the listener tries to determine the position of a sound source on the median plane which devides the listener head between the left and the right ear.[14]

The binaural technique is also applicable using two loudspeakers. Using loudspeakers, part of the HRTF is estimated twice since the sound cannot be provided directly at the listener's ear drums resulting in a adulterated sound characteristics. To decrease the influence of sound waves being perceived at the wrong ear drum crosstalk cancellation techniques have to be implemented.[15]

For fixed listener positions and orientation using headphone techniques is reliable and produces good results.

Since the conditions at the eardrums are simulated with the binaural audio techniques they have to be adjusted, whenever the user is changing his position or orientation, since the conditions at the ear drums do change according to change in position and orientation.

For a successful implementation that allows listener movement an accurate and low latency listener tracking system has to be established to react to the changing listening conditions.[16]

---

[14] cp.: Begault, Durand R. : 3-D sound For Virtual Reality And Multimedia, Moffet Field, NASA Ames Research Center, Academics Press Professional, 1994, Reprint 2000, p.41

[15] cp.: Gardner, William G.: 3-D Audio Using Loudspeakers, Boston, Massachusetts Institute of Technology, Kluwer Academic Publishers, Norwell, Massachusetts, 1998

[16] cp.: sections 3.1 and 3.2

### 3.3. Applicable Techniques at EVL

With the installed audio hardware at EVL a two channel loudspeaker set-up is applicable on all SGI IRIX based as well as PC Linux based systems.
The SGI Indigo2 workstations available at EVL do support four channel output, which can be used for a four channel loudspeaker implementation.
Since there is no user tracking available at the workstations offering two channel output, due to the time horizon of this project and the demand to minimize the need for additional interaction devices to be worn by the application user headphone based techniques are not to be supported within this project.

# 4. Spatialized Sound Server Implementation

To implement a spatialized sound server that is useable not only with the current EVL laboratory environment but open to be used with different hardware and software configurations the implementation must support several different techniques for sound spatialization as well as it's design must be open for extension in the future and configurable for individual needs.
The layer model as well as the communication model of the non spatialized "snerd" implementation can be used for the spatialized approach. However the data structures and the messages used for communication have to be adjusted in order to fulfil the needs of a spatialized approach. Besides the minimum data structures needed further data structures must be accessible to individually configure the sound server's behaviour and there must be messages available to set and query these newly introduced attributes.
There must be a data structure holding information about the audio set-up used for audio reproduction as well as the attributes necessary for the currently active spatialization technique.
As the Virtual Reality user navigates through the virtual world, his position and orientation within the virtual world must be tracked. The user's navigated coordinates must be stored in a separate data structure.
To finally calculate a sound characteristic that matches the current allocation of the sound source within the virtual world with respect to the navigated user's position, the sound source's positional information must be managed in a data structure.
The newly introduced data structures can be used to calculate the spatialized sound characteristic. Nevertheless the sound characteristic is calculated for the listener being positioned in the physical audio set-up's hotspot. For situations in which the listener is physically moving to a position different than the hotspot, the spatialization technique must take the new physical listener position into account. For this purpose the physical listener's position must be captured in an other data structure.

### 4.1. Spatialized Sound Server Data Structures

The data structures forming the data model for sound spatialization can be subdivided into the user data structure capturing data concerning the user, the sound source data structure holding information about the sound sources and the physical listener data structure keeping track of the physical aspects of the audio reproduction environment. An overview of the current class hierarchy within the non spatialized sound server implementation is given in the "Non Spatialized Class Hierarchy Overview". (See figure 33) It is used as a base for further development.

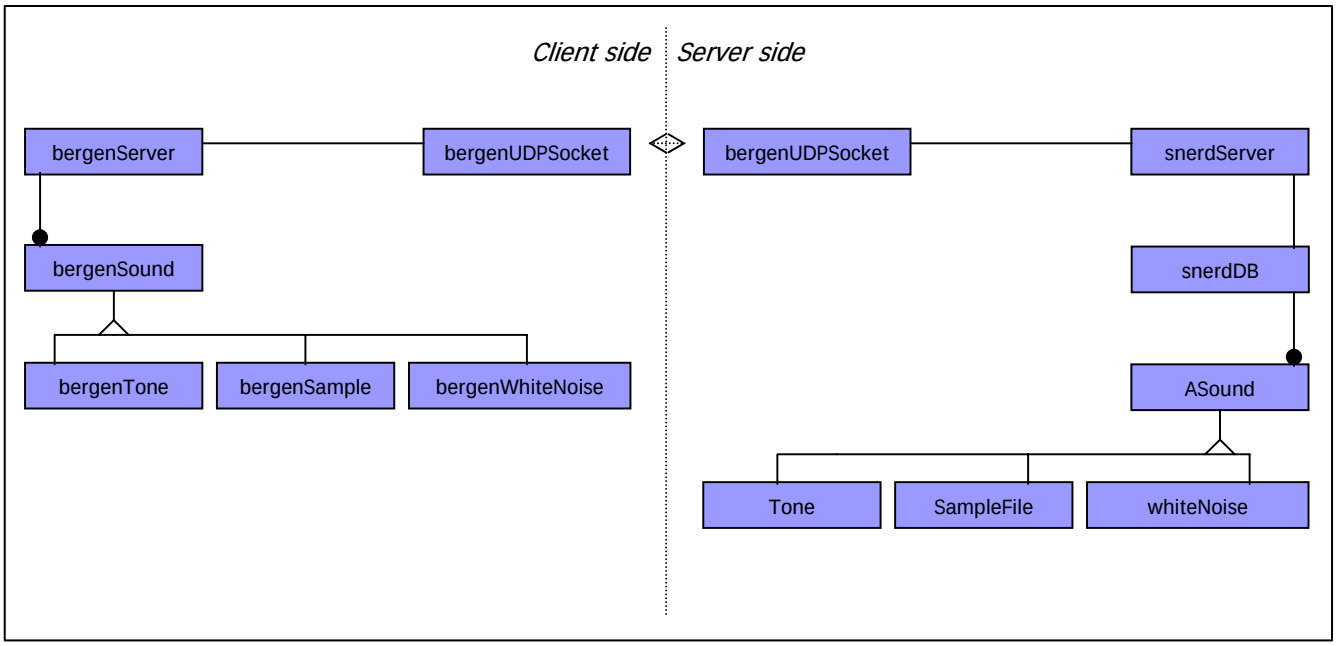


Figure 33 : Non Spatialized Class Hierarchy Overview

#### 4.1.1. User Data Structure

The Virtual Reality application's world is perceived from a determined user position within the virtual world. As the user navigates through the virtual world his position and orientation is changing. The virtual world is perceived under changed conditions according to the new user's attributes.

Assuming a user right-handed Cartesian coordinate system with the user being positioned in the origin at all times and navigating the coordinate system within the world coordinate system it is then possible to describe all objects in the virtual world in navigated user co-ordinates. Using navigated user coordinates to describe all objects' position and orientation introducing a separate data structure to keep track of the user attributes is redundant and therefore not to be implemented.

### 4.1.2. Sound Source Data Structure

Sound sources are the only objects to be described within the virtual world that are relevant for spatialized audio processing at the moment. At least a sound source's positional information as well as the attributes needed for distance attenuation must be storable within the data structure.

In order to widen the current sound server implementation the client module as well as the server module must be adjusted.

The final class hierarchy is summarized within the "Spatialized Class Hierarchy Overview". (See figure 34)

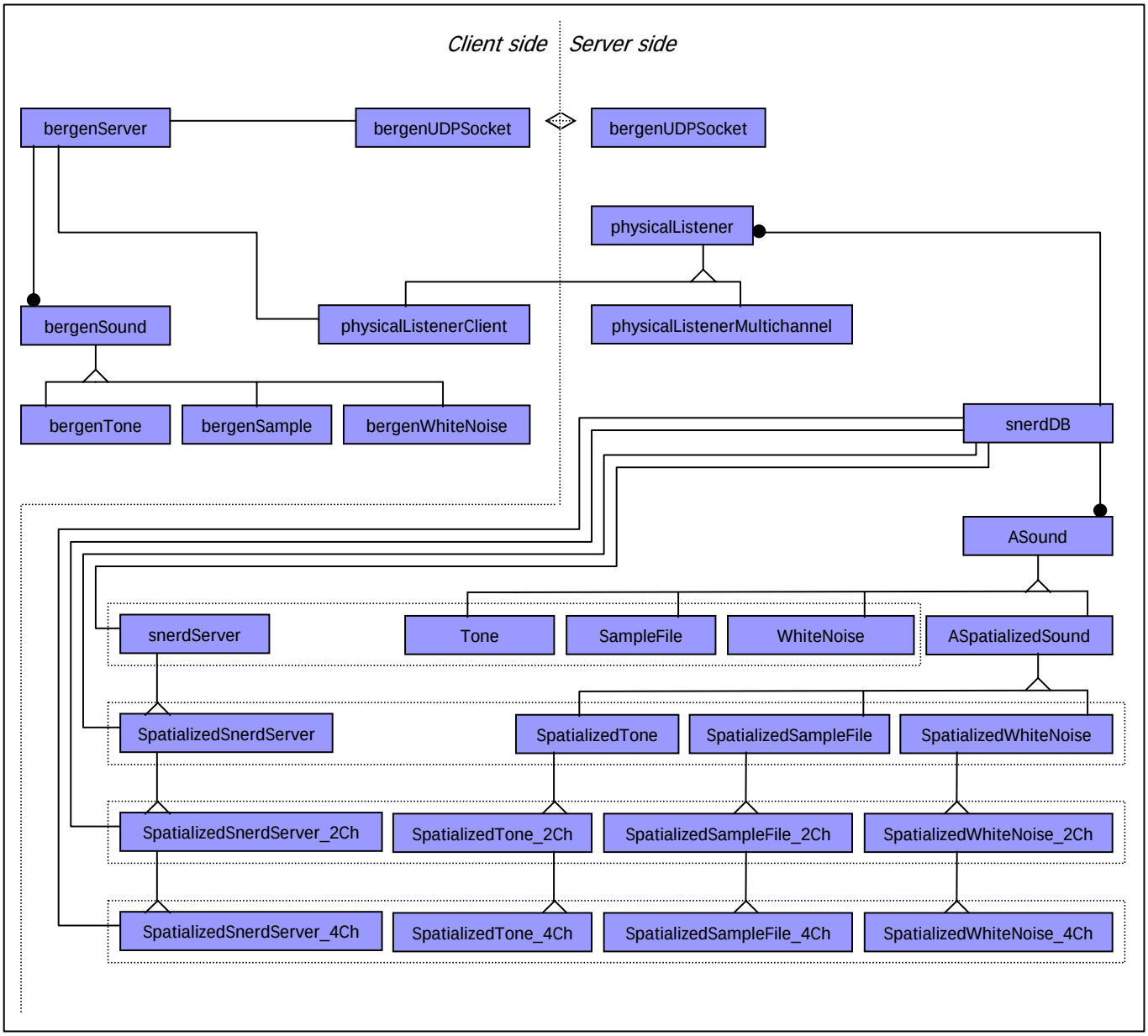


Figure 34 : Spatialized Class Hierarchy Overview

#### 4.1.2.1. Sound Source Data Structure – Server Component

The sound source's position is describable in terms of x, y, z coordinates.

To support different distance attenuation models several attributes related to sound propagation have to be manageable.

With the current implementation only a simple sound source model is assumed neglecting a more specific sound propagation model and sound source shape.

The current sound model assumes every sound source to be an omnidirectional sound source omitting sound waves equally in every direction. Neither the sound source's shape nor it's size is taken into account for audio processing.
Sound sources can appear to be ambient sound sources which are heard all over the virtual world without spatialization and some sound sources may show different behaviour with distance thus need different distance attenuation models. Attributes to chose between the different models for distance attenuation as well as for sound spatialization have to be offered.
Since all sound objects are managed by a unique "snerdDB" sound database which might be used by different clients, but each sound source's sound characteristics has to be calculated with respect to the connected client's limitations, a reference to the client's physical attributes also stored within the "snerDB" has to be stored with every sound source as well as a reference to the "snerDB" itself.
Beside the upgraded sound source attributes functions have to be implemented to set and get any desired attribute. Further functions have to be implemented to calculate each individual sound source's spatial sound characteristic including distance attenuation. Each supported spatialization technique requires a separate implementation for the spatialization calculation for each sound type. Hence there are nine different sound implementations (three different spatialization techniques (one channel, two channel, four channel) for three different sound types (tone, whitenoise, sample)).
For this purpose a new class is derived from "Snerd's" original "Asound" class introducing the newly needed attributes and function prototypes as well as the commonly used implementations. (See figure 34 for class hierarchy and figure 35 for detailed data structure). For specialized spatialization techniques the attributes and the already implemented functionality are derived from the "ASpatializedSound" class, while the individual algorithm for sound spatialization is implemented separately with each subclass.
With the introduction of new attributes and new functionality messages to access them from the sound server's client module are implemented within the message routine.

| class ASound | | | | |
|---|---|---|---|---|
| public | | ASpatializedSound | (int bufsize=1000) | Initialises ASound Object with default values and allocates memory for storing audio data |
| public | | ~ASpatializedSound | (void) | The memory occupied by the buffer is freed |
| public | virtual short * | updateBuffer | (void) | Returns the actual buffer |
| public | virtual int | removable | (void) | returns flag to initialise that object kann be removed |
| public | virtual int | active | (void) | Returns 0 to determine that Object is not playing |
| public | int | handle | (void) | Returns the handle_ on the object |
| public | short * | buffer | (void) | Returns the reference to the object's audio buffer |
| public | int | bufferSize | (void) | Returns the size of the audio buffer |
| public | virtual void | setAmplitude | (float amp) | Sets the amplitude to the new given amplitude |
| public | virtual float | amplitude | (void) | Gets the object's current amplitude |
| public | virtual void | play | (void) | not implemented |
| public | virtual void | stop | (void) | not implemented |
| public | virtual void | pause | (void) | not implemented |
| public | virtual void | kill | (void) | not implemented |
| public | virtual void | message | (char **msg,int num) | Executes the right action refereing to the messages coming in |
| public | void | setSampleRate | (int rate) | Sets the object's sampling rate to the given sampling rate |
| public | void | setPosition | (float x, float y, float z) | Sets the position to the given one |
| public | void | setAttenuationType | (AttenuationType type) | Sets the attenuation type to the given one |
| public | float | attenuatedGain | (void) | Calculates the attenuation for the sound source depending on it's position |
| public | float | spatializedGain | (int speakerIndex) | Calculates the amplitude for a given speaker depending on the spatialization technique |
| public | void | setMinGain | (float gain) | Sets the min gain to the given one |
| public | void | setMaxGain | (float gain) | Sets the max gain to the given one |
| public | void | setReferenceDistance | (float distance) | Sets the reference distance to the given one |
| public | void | setFalloffDistance | (float distance) | Sets the falloff distance to the given one |
| public | void | setFalloffFactor | (float factor) | Sets the falloff factor to the given one |
| protected | short * | buffer_ | | Buffer holding the Asamples samples |
| protected | int | bufferSize_ | | Size of the buffer_ |
| protected | int | handle_ | | Handle to the Asample |
| protected | float | amplitude_ | | Asamples amplitude |
| protected | int | sampleRate_ | | Asamples sample rate |
| protected | SpatializationTechnique | spatializationTechnique_ | | Technique used for sound spatialization |
| private | static int | NextHandle | | Handle to the next Sample |
| private | AttenuationType | attenuationType_ | | Attenuation Type used for distance attenuation |
| private | int | physicalListenerHandle_ | | Handle to the physical listener object experiencing this sound |
| private | snerdDB * | sDB_ | | Reference to the snerdDB managing this sound |
| private | float | positionX_ | | X coordinate of this sound |
| private | float | positionY_ | | Y coordinate of this sound |
| private | float | positionZ_ | | Z coordinate of this sound |
| private | float | minGain_ | | Minimum gain of the sound source |
| private | float | maxGain_ | | Maximum gain of the sound source |
| private | float | referenceDistance_ | | Reference distance used for distance attenuation |
| private | float | falloffDistance_ | | Falloff Distance used for distance attenuation |
| private | float | falloffFactor_ | | Falloff Factor used for distance attenuation |

Figure 35 : class ASpatializedSound overview

#### 4.1.2.2. Sound Source Data Structure – Client Component

On the client side the class hierarchy is not to be widened for a spatialized sound implementation in order to obtain backwards compatibility for older client applications. Thus the current client module implementation is to be adjusted to be ready for the spatialized sound approach.

For the client sided sound representation data structure new attributes to store the positional information as well as the attributes needed for sound spatialization and sound attenuation have to be implemented as well as the functionality to remotely set and get the according values.

Since the different sound types are all derived from the "BergenSound" class the additional attributes and functionality have to be implemented at this top class only. Therefore the client module's data structures are widened according to the client module data structure table for the "BergenSound" class. (See figure 36)

| class bergenSound | | | | |
|---|---|---|---|---|
| public | | bergenSound | (class bergenServer *server) | The constructor must be given a pointer to a server object; it will tell the server object to add this sound to its list of sounds |
| public | | ~bergenSound | (void) | The destructor does an automatic kill before the sound object is removed |
| public | int | handle | (void) | Returns the handle_ |
| public | class bergenServer * | server | (void) | Returns the server_ |
| public | virtual void | setAmplitude | (float amp) | Sets the sound's current amplitude; sends a message to the server program to accomplish this |
| public | virtual void | play | (void) | Sends a message to the server program to start playing the sound |
| public | virtual void | stop | (void) | Sends a message to the server program to stop playing the sound. The next time a play command is issued, the sound will start again from the beginning |
| public | virtual void | pause | (void) | Sends a message to the server program to pause the sound. The next time a play command is issued, the sound will resume from the point at which it was paused |
| public | virtual void | kill | (void) | Sends a message to the server program to remove the sound, and tells the server object to remove this sound from its list. The sound object should not be used after kill() is called; this is meant to be called from the destructor, and should not generally be used directly by applications |
| public | virtual void | killRemote | (void) | Sends a message to the server program to remove the sound, and tells the server object to remove this sound from its list |
| public | virtual void | createRemote | (void) | Creates Fullpath by calling fullpath and sends message to the associated server to create a sound sample and waits for the resulting handle to store |
| public | virtual void | setPosition | (float x, float y, float z) | Sets the sound source's position |
| public | virtual float * | position | (void) | Gets the sound source's position |
| public | virtual void | setAttenuationType | (AttenuationTape type) | Sets the souind source's attenuation type |
| public | AttenuastionType | attenuationType | (void) | Gets the sound source's attenuation type |
| public | virtual void | setMinGain | (float gain) | Sets the sound source's max gain |
| public | virtual float | minGain | (void) | Gets the sound source's max gain |
| public | virtual void | setMaxGain | (float gain) | Sets the sound source's min gain |
| public | virtual float | maxGain | (void) | Gets the sound source's min gain |
| public | virtual void | setReferenceDistance | (float distance) | Sets the sound source's reference distance |
| public | virtual float | referenceDistance | (void) | Gets the sound source's reference distance |
| public | virtual void | setFalloffDistance | (float distance) | Sets the sound source's falloff distance |
| public | virtual float | falloffDisance | (void) | Gets the sound source's falloff distance |
| public | virtual void | setFalloffFactor | (float factor) | Sets the sound source's falloff factor |
| public | virtual float | falloffFactor | (void) | Gets the sound source's falloff factor |
| protected | class bergenServer * | server_ | | Reference to bergenServer |
| protected | int | handle_ | | Handle of the sound |
| protected | bool | isPlaying_ | | flag indexing if it is currently playing |
| protected | AttenuastionType | attenuationType_ | | Sound source's attenuation type |
| protected | float | positionX_ | | Sound source's position's x co-ordinate |
| protected | float | positionY_ | | Sound source's position's y co-ordinate |
| protected | float | positionZ_ | | Sound source's position's z co-ordinate |
| protected | float | minGain_ | | Sound source's max gain |
| protected | float | maxGain_ | | Sound source's min gain |
| protected | float | referenceDistance_ | | Sound source's reference distance |
| protected | float | falloffDistance_ | | Sound source's falloff distance |
| protected | float | falloffFactor_ | | Sound source's falloff factor |

Figure 36 : class bergenSound overview

### 4.1.3. Physical Listener Data Structure

Beside the navigated user coordinates and the sound sources which are describable relative to the navigated coordinates within the virtual world, the physical reproduction conditions must be taken into account for sound characteristic caqlculations.
The aspects describing the reproduction conditions are physical attributes that can be measured and which are connected to the physical environment used for audio reproduction as well as the listener's physical appearance. Since all relevant attributes are describing the aspects of the listener's environment as well as himself the data structure can be best identified as physical listener data structure.
Introducing a physical listener two types of physical set-ups can be distinguished. There are audio set-ups available using only one channel for audio reproduction and these using more than one. For one channel systems the position of the speaker relative to the listener is not used for audio processing, hence it's storage within the physical listener data structure is not necessary. For multichannel systems indeed storing the position of each used speaker is essential since the spatialization effect is generated using the speakers' position relative to the listener.[17]
For physical environment attribute management tasks a class "PhysicalListener" has been implemented. The "PhysicalListener" class covers all aspects needed to represent a one channel audio reproduction environment (See figure 37). Because the "PhysicalListener" class does not implement any communication functionality nor communication data structures for client – server communication a "PhysicalListenerClient" class is derived from it. For multichannel implementations an additional "PhysicalListenerMultichannel" class is also derived from the "PhysicalListener" class which then offers functionality and data aspects for a number of loudspeakers. (See figure 34 for class hierarchy)

[17] cp.: section 3.1.3

| class PhysicalListener | | | | |
|---|---|---|---|---|
| public | | PhysicalListener | (void) | Physical Listener Constructor |
| public | | ~PhysicalListener | (void) | Physical Listener Destructor |
| public | virtual void | setPosition | (float x, float y, float z) | Set the listener position to the given co-ordinates |
| public | virtual float * | Position | (void) | Returns the current listener position |
| public | virtual void | updateSpeakers | (void) | not implemented |
| public | virtual int | handle | (void) | Return the handle to this listener |
| public | virtual void | message | (char ** msg, int num) | not implemented |
| public | virtual void | kill | (void) | Set the kill_ flag |
| public | virtual int | removable | (void) | Check if kill_ flag is set |
| public | float | positionX_ | | Listener's x co-ordinate |
| public | float | positionY_ | | Listener's y co-ordinate |
| public | float | positionZ_ | | Listener's z co-ordinate |
| protected | int | handle_ | | Handle for this listener |
| protected | static int | NextHandle_ | | Handle to next listener |
| protected | int | killed_ | | Flag if this listener is marked to be destroyed |

Figure 37 : class PhysicalListener overview

#### 4.1.3.1. Physical Listener Data Structure – Server Component

The audio set-up used by the server for audio reproduction is known on the server side only. Hence the representation of the environment's set-up can be limited to the data structure on the server side. The client module has to operate independent of the audio environment used by the server module and the client has no influence on the used audio set-up. Therefore representing the audio environment on the client side is not needed. Since no special attributes are needed for a single channel audio reproduction there is a derived "PhysicalListenerMultichannel" class for multichannel reproduction only (See figure 37).

| class PhysicalListenerMultichannel | | | | |
|---|---|---|---|---|
| public | | PhysicalListenerMultichannel | (void) | Physical Listener Multichannel Constructor |
| public | | ~PhysicalListenerMultichannel | (void) | Physical Listener Multichannel Destructor |
| public | virtual void | setPosition | (float x, float y, float z) | Set the listener position to the given co-ordinates |
| public | virtual float * | Position | (void) | Returns the current listener position |
| public | virtual void | setSpeakerAmount | (int amount) | Set the amount of speakers used within the audio environment |
| public | virtual void | setSpeakerPosition | (int speakerIndex, float x, float y, float z) | Set the position of the speakerIndex's position |
| public | virtual float * | initialSpeakerPosition | (int speakerIndex) | Return the initial speaker position |
| public | virtual float * | actualSpeakerPosition | (int speakerIndex) | Return the actual speaker position |
| public | virtual int | speakerAmount | (void) | Return the amount of speakers used for this environment |
| public | virtual void | updateSpeakers | (void) | Calculate the actual speaker position depending on the listener position |
| public | virtual int | handle | (void) | Return the handle to this listener |
| public | virtual void | message | (char ** msg, int num) | Parse message and apply appropiate action |
| public | virtual void | kill | (void) | Set the kill_ flag |
| public | virtual int | removable | (void) | Check if kill_ flag is set |
| public | virtual float | positionX_ | | Listener's x co-ordinate |
| public | virtual float | positionY_ | | Listener's y co-ordinate |
| public | virtual float | positionZ_ | | Listener's z co-ordinate |
| protected | virtual int | handle_ | | Handle for this listener |
| protected | virtual static int | NextHandle_ | | Handle to next listener |
| protected | virtual int | killed_ | | Flag if this listener is marked to be destroyed |
| protected | int | speakerAmount_ | | Amount of speakers used within the audio environment |
| protected | SpeakerInformation * | speakers | | Struct holding the actual and initial speaker position |

Figure 37 : class PhysicalListenerMultichannel overview

The "PhysicalListenerMultichannel" class offers attributes to store the initial position of the speakers, measured with the physical listener positioned in the audio set-up's hotspot as well as functionality to calculate their "actualPostion" attributes in case the listener is moving within the audio environment.
Currently only loudspeaker based spatialization techniques are used so the physical listener's orientation is neglectable. The position indeed has to be tracked at all times. A listener's experience to hear a simulated sound source to originate from a position to his right for example is obtained by replaying the sound with equal amplitude on each of the speakers on the physical listener's right side. If the sound source is positioned within the area spanned by the loudspeakers (see figure 39) and the listener is moving within it, the loudspeakers' position relative to the listener has to be updated. If not unexpected spatialization failures will appear. In the example, a listener moving to a position in front of the sound source's position will then still experience the sound source to originate from his right and not from behind as expected. Additional the intensity of the speaker the listener is coming closer to increases leading to a further failure in spatialization. The increase in intensity lets the listener experience the sound source's origin closer to the speaker he just walked closer to (see figure 40).
Calculating the actual loudspeaker position depending on the physical listener's position leads to a correct sound characteristic calculation of the current situation. In the example, knowing one loudspeaker's position behind will pan the intensity away from the loudspeaker in front to the ones in the back to simulate the sound source to originate form the desired position.
Extracting the listener's distance to the loudspeakers from his position can help to eliminate spatialization failure through a loudspeaker's increase in intensity but is not implemented with the current spatialized sound server since the effect is barely noticeable in current CAVE application areas. Due to the small dimensions of existing CAVE environments and the expected low increase with the listener moving closer to the speaker the resulting spatialization failure is minimal. Situations with the listener being positioned directly at a loudspeaker's position will of course destroy the perceived sound impression but are rarely to be expected.

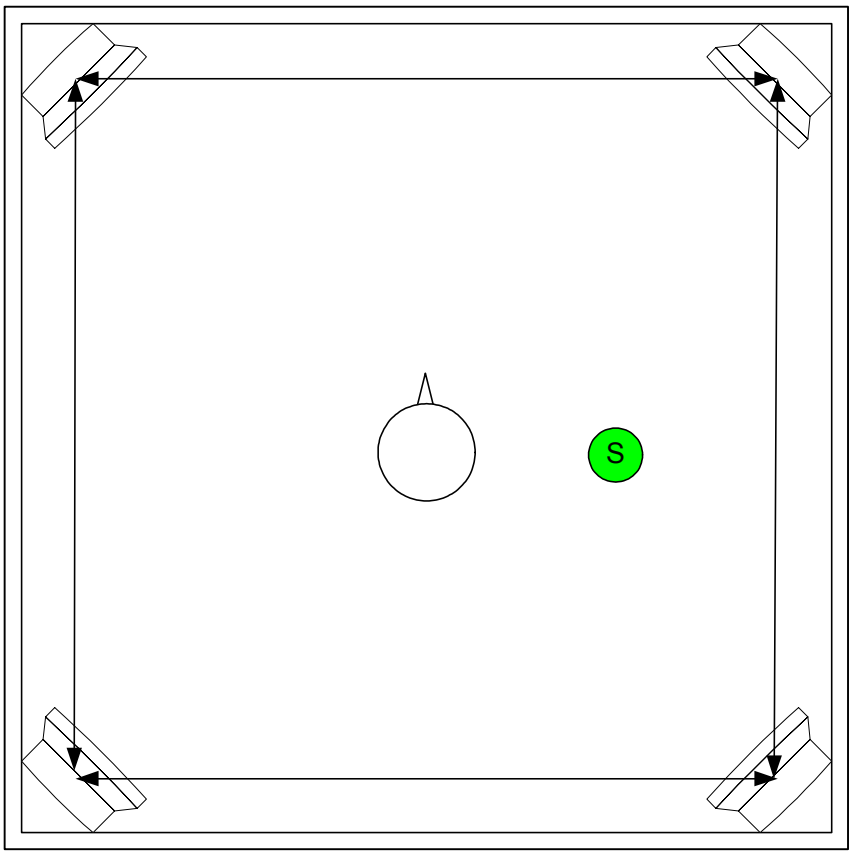


Figure 39 : Sound reproduction with listener in hotspot using 4 channels with 50% amplitude both of the right speakers

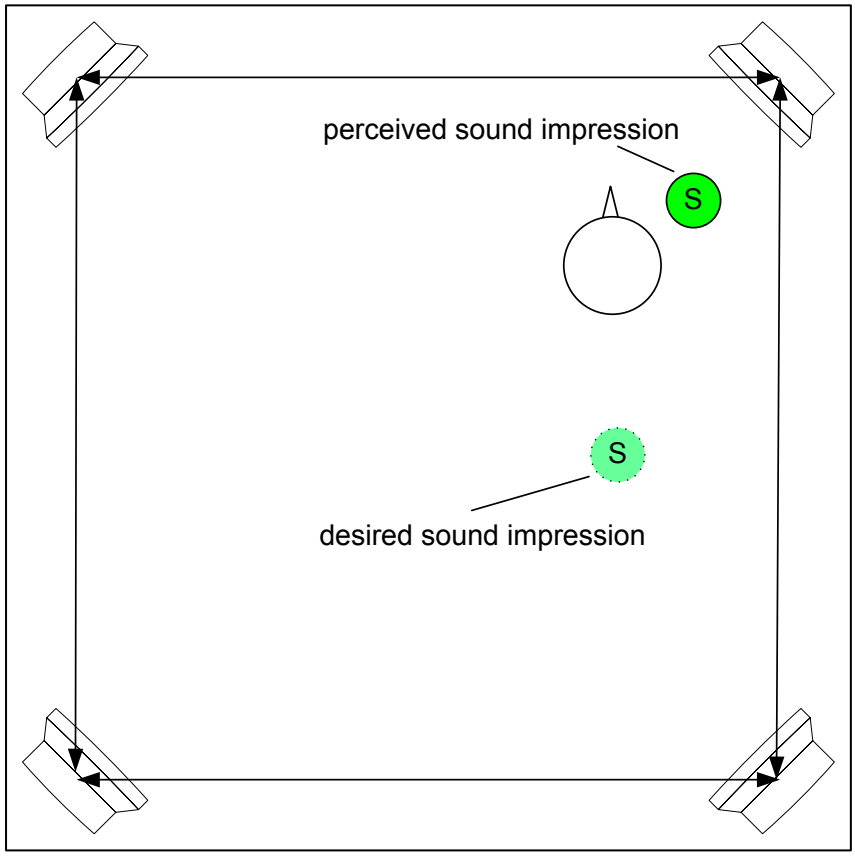


Figure 40 : Failure in sound reproduction through physical listener movement

#### 4.1.3.2. Physical Listener Data Structure – Client Component

The physical listener's position is available at the client side only. Each client connected to the sound server may have different physical listener's that are tracked for the application flow. The listener's positional information therefore must be kept individually within each client component.

Functionality to initially create a physical listener representation on the server side, managed within the "snerdDB" data structures, and to submit updated physical listener positions as well as to destroy the remote listener is implemented within the client component. To address individual listener representations on the server side each physical listener client data structure is holding an unique handle which is returned by the server while creating the remote listener. (See figure 41)

| class PhysicalListenerClient | | | | |
|---|---|---|---|---|
| public | | PhysicalListenerClient | (void) | Physical Listener Constructor |
| public | | ~PhysicalListenerClient | (void) | Physical Listener Destructor |
| public | virtual void | setPosition | (float x, float y, float z) | Set the listener position to the given co-ordinates |
| public | virtual float * | Position | (void) | Returns the current listener position |
| public | virtual void | updateSpeakers | (void) | not implemented |
| public | int | handle | (void) | Return the handle to this listener |
| public | bergenServer * | bergenServer | (void) | Return reference to bergenServer used for communication |
| public | virtual void | message | (char ** msg, int num) | not implemented |
| public | virtual void | kill | (void) | Set the kill_ flag |
| public | virtual void | killRemote | (void) | Kills listener representation on server side |
| public | virtual void | createRemote | (void) | Create listener representation on server side |
| public | virtual int | removable | (void) | Check if kill_ flag is set |
| public | float | positionX_ | | Listener's x co-ordinate |
| public | float | positionY_ | | Listener's y co-ordinate |
| public | float | positionZ_ | | Listener's z co-ordinate |
| protected | int | handle_ | | Handle for this listener |
| protected | static int | NextHandle_ | | Handle to next listener |
| protected | int | killed_ | | Flag if this listener is marked to be destroyed |
| protected | class bergenServer * | server_ | | bergenServer used for communication |

Figure 41 : class PhysicalListenerClient

### 4.2. Spatialized Sound Server Communication Model

With the introduction of new data structures and new functionality new messages have to be implemented within the client – server communication to access these.
Using the already existing UDP socket for network communication, the implemented client and server services as well as the dataflow model new messages are introduced keeping the concerted format to guarantee compatibility. (See figure 42)

| Identifier | Command | Attribute | | Snerd Action |
|---|---|---|---|---|
| | | Type | Description | |
| -1 | new sample | char* | filename | Create a new sample object from the audio file with the given filename |
| -1 | newsample | char* | filename | Create a new sample object from the audio file with the given filename |
| -1 | new tone | int | frequency | Create a new tone object with the given frequency |
| -1 | newtone | int | frequency | Create a new tone object with the given frequency |
| -1 | new whitenoise | --- | | Create a new whitenoise object |
| -1 | newwhitenoise | --- | | Create a new whitenoise object |
| 0 ... MAX_INTEGER | gain | float | new gain | Adjust the gain of the sound object with the given identifier |
| -1 | cd | char* | new directory | Change the server's reference directory to the given one |
| 0 ... MAX_INTEGER | kill | --- | | Kill the sound object with the given identifier |
| -1 | reset | --- | | Reset the snerdDB |
| -1 | ping | --- | | Send back a "pong" message to the client as respond |
| 0 ... MAX_INTEGER | play | --- | | Set the 'play_' attribute of the sound object with the given identifier |
| 0 ... MAX_INTEGER | stop | --- | | Set the 'stop_' attribute of the sound object with the given identifier |
| 0 ... MAX_INTEGER | pause | --- | | Set the 'pause_' attribute of the sound object with the given identifier |
| 0 ... MAX_INTEGER | loop | {1,0} | new loop value | Set the 'loop_' attribute of the sound object with the given identifier to the given value |
| 0 ... MAX_INTEGER | setfrequency | int | new frequency | Set the 'frequency_' attribute of the sound object with the given identifier to the given value |
| -1 | newListener | --- | | Create a new listener object |
| 0 ... MAX_INTEGER | setListenerPosition | float,float,float | x,y,z co-ordinates | Change a listener's position to given co-ordinates |
| 1 ... MAX_INTEGER | killListener | --- | | Destroy a listener |
| 1 ... MAX_INTEGER | setSoundPosition | float,float,float | x,y,z co-ordinates | Change a sound's position to given co-ordinates |
| 1 ... MAX_INTEGER | setSoundAttenuationType | AttenuationType * | | Set a sound's attenuation type to the given one |
| 1 ... MAX_INTEGER | setSoundMinGain | float | | Set a sound's minGain to the given one |
| 1 ... MAX_INTEGER | setSoundMaxGain | float | | Set a sound's maxGain to the given one |
| 1 ... MAX_INTEGER | setSoundReferenceDistance | float | | Set a sound's referenceDistance to the given one |
| 1 ... MAX_INTEGER | setSoundFalloffDistance | float | | Set a sound's falloffDistance to the given one |
| 1 ... MAX_INTEGER | setSoundFalloffFactor | float | | Set a sound's falloffFactor to the given one |

* AttenuationType : {NONE, LINEAR_FALLOFF_BY_FACTOR, LINEAR_FALLOFF_BY_DISTANCE,INVERSE_SQUARE_LAW, INVERSE_SQUARE_LAW_CLAMPED}

Figure 42 : Client – Spatialized Sound Server Communication Messages

# 5. Operating the Spatialized Sound Server

The final spatialized sound server implementation is executable on SGI Irix based platforms as well as on PC Linux based systems.
To operate the spatialized sound server the hardware used for audio reproduction has to be installed, the sources have to be compiled and the server itself has to be executed on the machine the audio hardware is connected to.

## 5.1. Installting the Audio Hardware

The audio hardware has to be installed depending on the spatialization technique that is to be used by the server and the system's hardware limits.
To enable a fully software controlled audio reproduction the loudspeakers used for audio reproduction are to be connected to the line-out output of the sound device for each of the spatialization set-ups.

### 5.1.1. One channel audio reproduction

Installing audio equipment for one channel audio reproduction presume an audio output facility at the server's workstation that offers at least one channel.

#### 5.1.1.1. One Channel PC Installation

Using a PC Linux based platform a standard stereo sound card is most commonly used for audio reproduction. The loudspeaker system is to be attached to one of the sound card's stereo outputs or to both of them. The same signals are generated for both of the stereo outputs with one channel audio reproduction.

#### 5.1.1.2. One Channel SGI Installation

Each SGI Irix based system offering an audio output device is capable of reproducing one channel spatialized sound. The same spatialized sound signal is generated for each of the available outputs. In order to connect the loudspeaker system to be used it may be attached to either one of the audio channels or to a selection of the available outputs.

### 5.1.2. Two Channel Audio Reproduction

Two separate output channels are necessary for spatialized two channel audio reproduction. Both PC Linux based platforms as well as SGI Irix based platforms are capable of reproducing two channel output with the current spatialized sound server implementation.
For a most convincing spatialization effect the loudspeakers are to be positioned in a wide stereophonic setup with the listener being positioned in the audio set-up's hotspot.[18] Using headphones instead is also possible.

#### 5.1.2.1. Two Channel PC Installation

A stereo sound card device is obligatory for a successful two channel spatialized sound reproduction. The loudspeaker system offering two separate channels is to be connected to the stereo sound card with each of the loudspeakers channel's to a separate output channel.

#### 5.1.2.2. Two Channel SGI Installation

Using the stereo output device of any SGI Irix based platform enables the connection of a stereo loudspeaker set-up. The separate sound device's outputs are to be attached to two separate channels at the loudspeaker set-up.

### 5.1.3. Four Channel Audio Reproduction

With the current spatialized sound server implementation four channel audio reproduction is available on SGI Irix based platforms only. The audio library currently used within the sound server environment for PC Linux based platforms is not supporting any direct way to open four separate channels for audio reproductions.

---

[18] cp.: Dickreiter, Michael: Handbuch der Tonstudiotechnik, München, Saur, K.G, 1997, p.126

For best results a symmetric stereophonic loudspeaker set-up is to chosen for audio reproduction.[19]

#### 5.1.3.1. Four Channel SGI Installation

With the "Iris Digital Media" library used by the actual sound server implementation four channel output is accessible using the default stereo sound device with an Indigo, Indigo$^2$ or Indy workstation.
Connecting two separate loudspeaker channels to the line output as performed with the two channel set-up, additional two separate channels can be connected to the "Headphone" output of the sound device. Using the "Iris Digital Media" library's four channel audio reproduction option the headphone output is internally used as a line-out output offering an additional separate third and fourth channel.[20]

## 5.2. Sound Server Configuration

Before using the spatialized sound server or even compiling the sound server's sources, the server has to be configured.
The chosen loudspeaker set-up has to be adjusted within the sound server application sources for two and four channel sound servers.
For configuration purposes the "main_spatializedSnerd_2Ch.cxx" and "main_spatializedSnerd_4Ch.cxx" files within the "server" directory have to be adjusted according to the loudspeakers' positions. Within the source files the "speakers" array has to be modified. For each speaker the x, y, z coordinates have to be declared relative to the listener position at the audio reproduction set-up's hotspot.
Having adjusted the loudspeakers' positions the sources have to be compiled.

---

[19] cp.: Pulkki, Ville, Spatial Sound Generation and Perception by Amplitude Panning Techniques, Helsinki, Laboratory of Acoustics an Audio Signal Processing - Helsinki University of Technology, Espoo 2001, Report 62, 2001, pp.17-18

[20] cp.: Creek Patricia a.o. : IRIS Digital Media Programming Guide, Silicon Graphics, Inc., 1994, pp. 43 - 113

### 5.3. Sound Server Compilation

To compile the spatialized sound server implementation the "Makefile" within the "server" directory has to be adjusted according to the operating system the server sources are compiled for. Within the "Makefile" the "include" entry has to be changed. For a compilation on a PC Linux based system the "Makedefs.linux" file is to be included the "Makedefs.sgi" for a SGI Irix based platform.
A final "make" will compile 3 versions of the spatialized sound server one version for each possible spatialization technique.

### 5.4. Sound Server Execution

Having compiled the server's sources, the sound server itself can be executed. Depending on the audio hardware configuration the according version of the server has to be executed. The "spatializedSnerdServer" is to be executed for one channel audio reproduction, "spatializedSnerdServer_2Ch" for two channel audio reproduction and "spatializedSnerdServer_4Ch" for four channel audio reproduction.

# 6. Demo Application

With the implementation of the spatialized sound server sound it is possible to integrate sound objects within virtual reality applications as additional feedback channel.
To prove the applicability of the spatialized sound server within virtual reality applications and to show the workability of the techniques used for sound spatialization a demo application was developed.

## 6.1. Demo Application Description

The demo application is a simple OpenGL application, which's graphical components are connected to the essential components of the spatialized sound server.
The application's scene consists of a graphical physical listener representation (represented as a large green sphere) and a sound source representation (represented as a small red sphere). An outlined square is showing the boundaries of the physical environment the physical listener is positioned in. (See figure 43)
Beside the application's graphical components a sound source is positioned at the same position as it's graphical representation and is following it's movements. An additional sound source playing at low amplitude is connected to the listener, following his movement to simulate an ambient sound source which is not attenuated. The ambient sound source has no graphical counterpart.
Both the listener and the sound source can be moved within the scene to simulate a moving sound source on the one hand and a physical listener moving within the audio set-up on the other hand.
The physical environment's boundaries might be crossed by the physical listener representation but may lead to unexpected spatialization effects, since the techniques used for sound spatialization are only defined for the listener being positioned within the area spanned by the vectors between the loudspeakers (the outlined square in the application). [21]

[21] cp.: Section 3.1

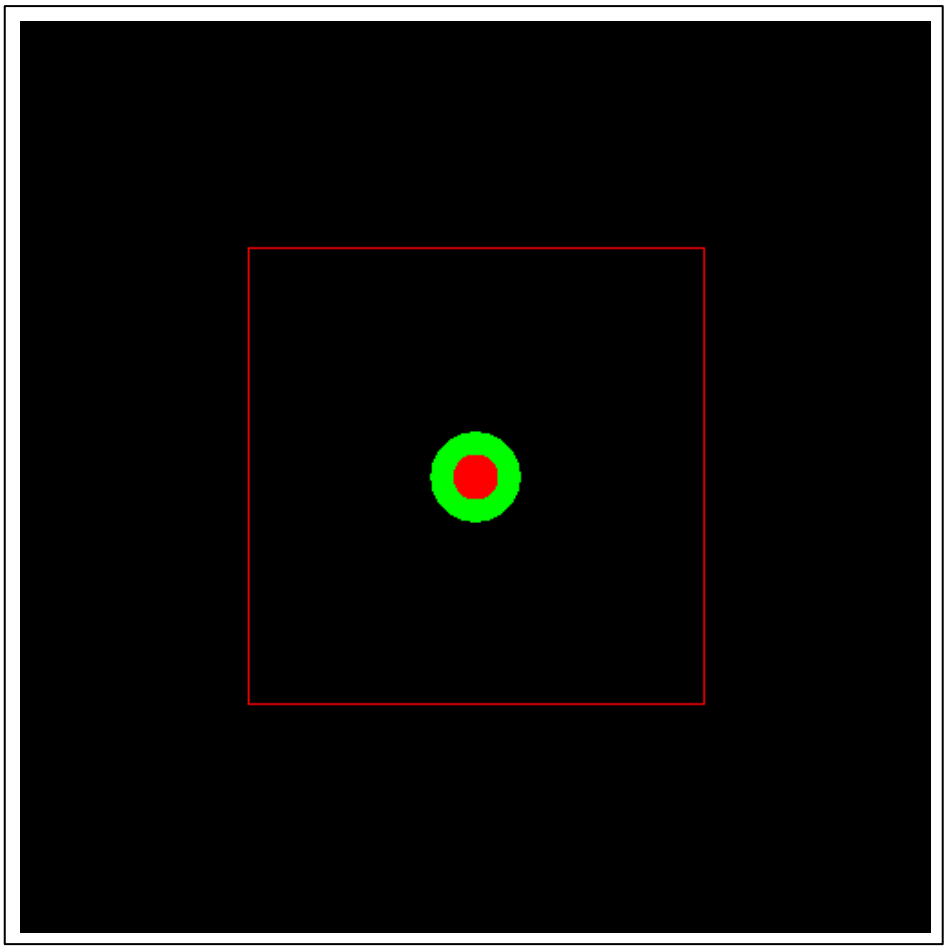

Figure 43 : Demo Application – listener and sound source at centered position

## 6.2. User Manual

The demo application is started by executing the "spatializedBergenTestOpenGL" application within the "clients" directory.

With program execution the listener and the sound source representations are positioned in the centre.

Both the listener and the sound source can be moved by using the keyboard.

The "L" and "S" keys are used to switch between listener ("L") movement and source ("S") movement.

The "l", "r", "u", "d" keys are used to move the active object ("l" to move left, "r" to move right, "u" to move up, "d" to move down).

Before closing the application window the audio objects have to be destroyed and audio

processing has to be stopped. By pressing the "Q" key audio processing is stopped. The application is finally closed by closing the window.

### 6.3. Experiencing Sound

With the demo application the simulation of several listening situations is possible. Moving the sound source to any position the functionality of the spatialized sound server can be demonstrated.

Sound sources moved within the boundaries of the physical audio reproduction environment are spatialized as well as attenuated according to their distance. (See figure 44)

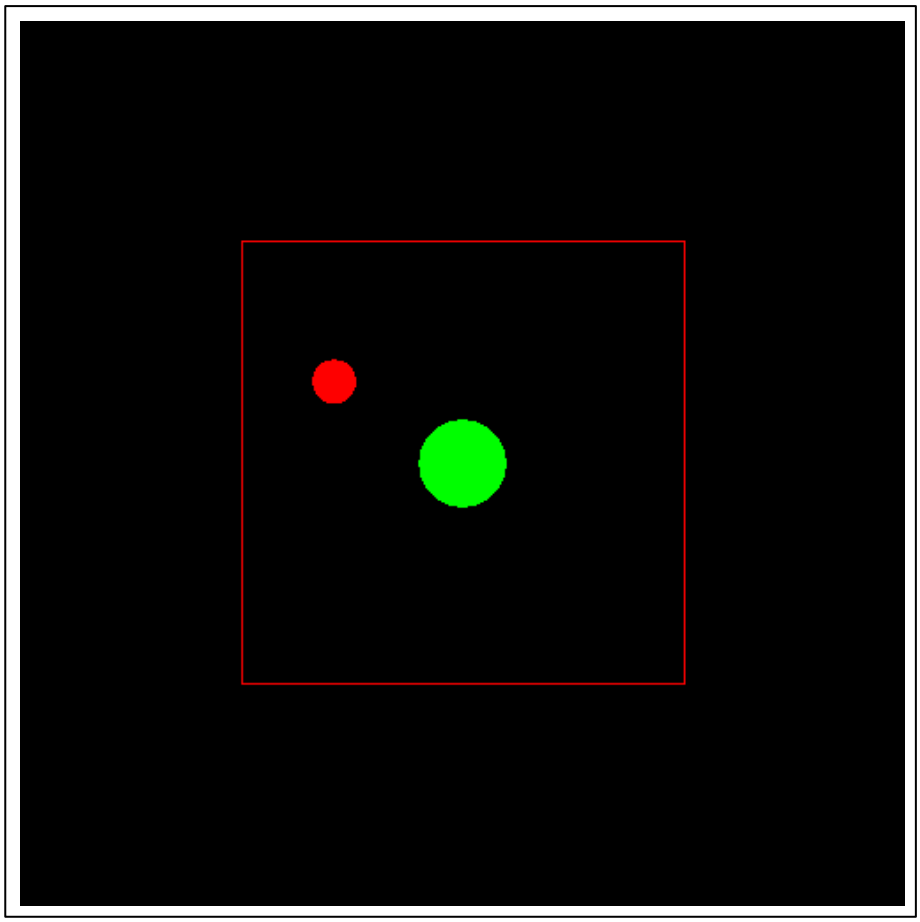

Figure 44 : Demo Application – sound source moved within the physical audio reproduction set-up's boundaries

Moving the sound source further away from the listener it's amplitude is further attenuated. Positing the sound source outside the physical audio reproduction set-up's boundaries is possible without any artifacts. (See figure 45)

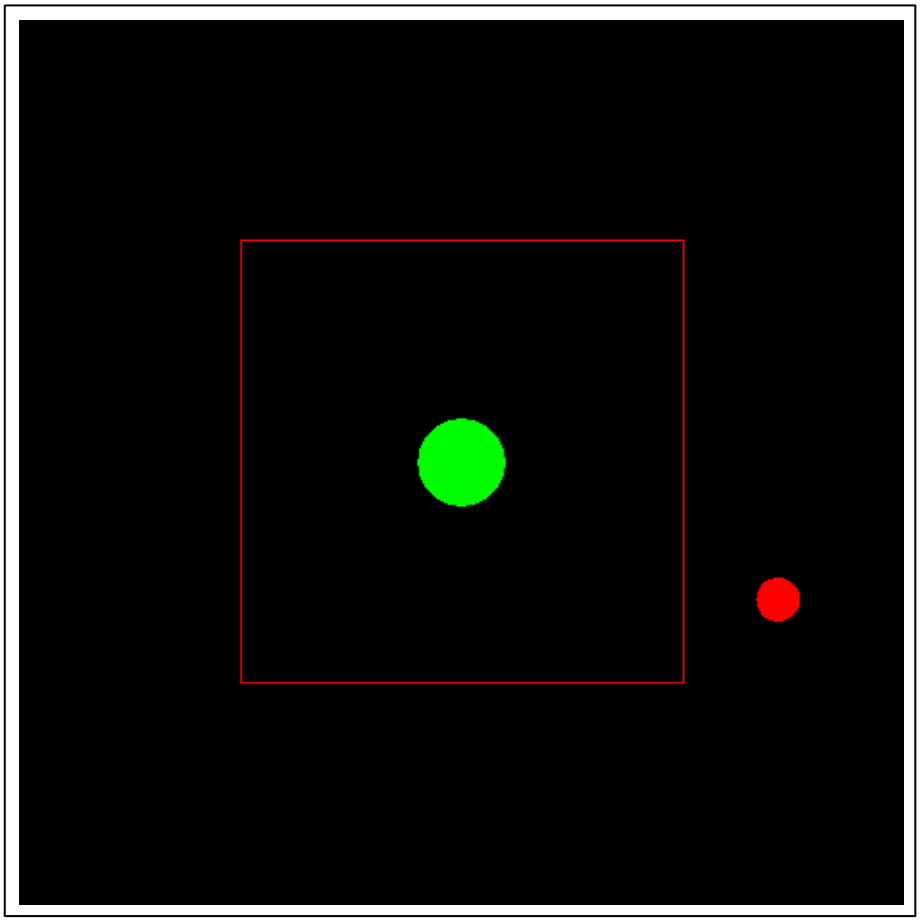

Figure 45 : Demo Application – sound source moved outside the physical audio reproduction set-up's boundaries

Moving the listener around a sound source which is positioned within the audio reproduction set-up's boundaries is also possible resulting in the desired spatialization effects. (See figure 46)

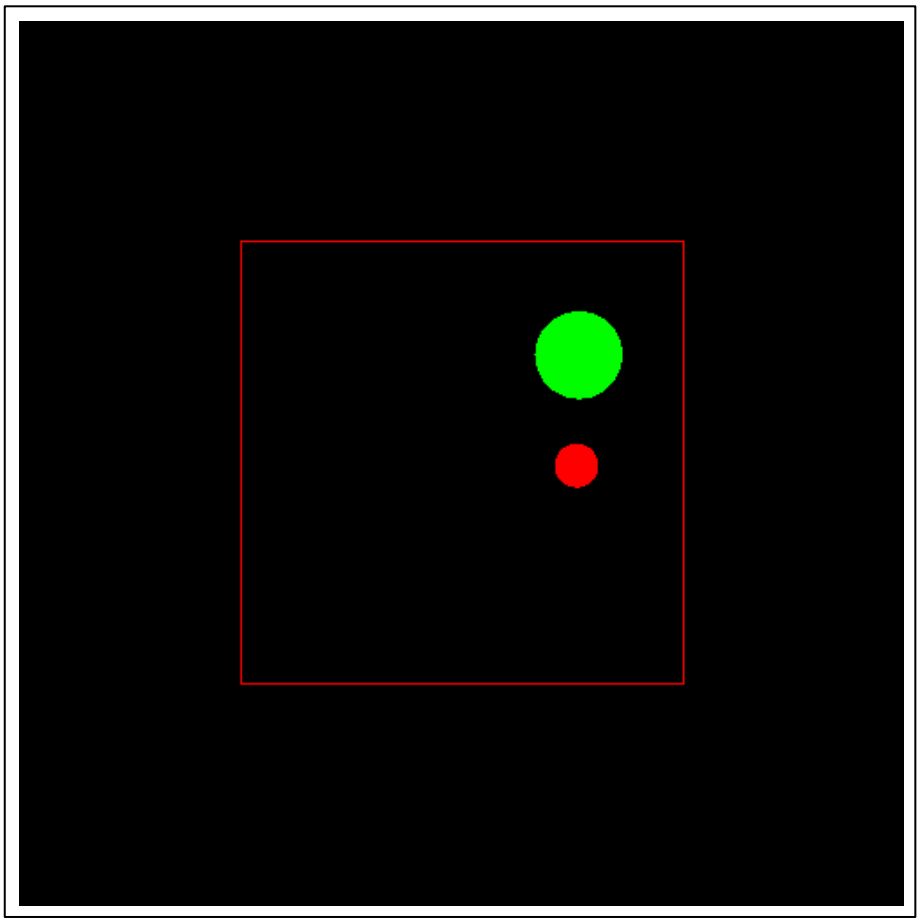

Figure 46 : Demo Application – listener moved within the audio reproduction set-up's boundaries around a sound source within the physical audio reproduction set-up's boundaries

Moving the listener outside the set-up's boundaries results in spatialization failures as mentioned within the spatialization technique section.[22] (See figure 47)

[22] cp.: Section 3.1

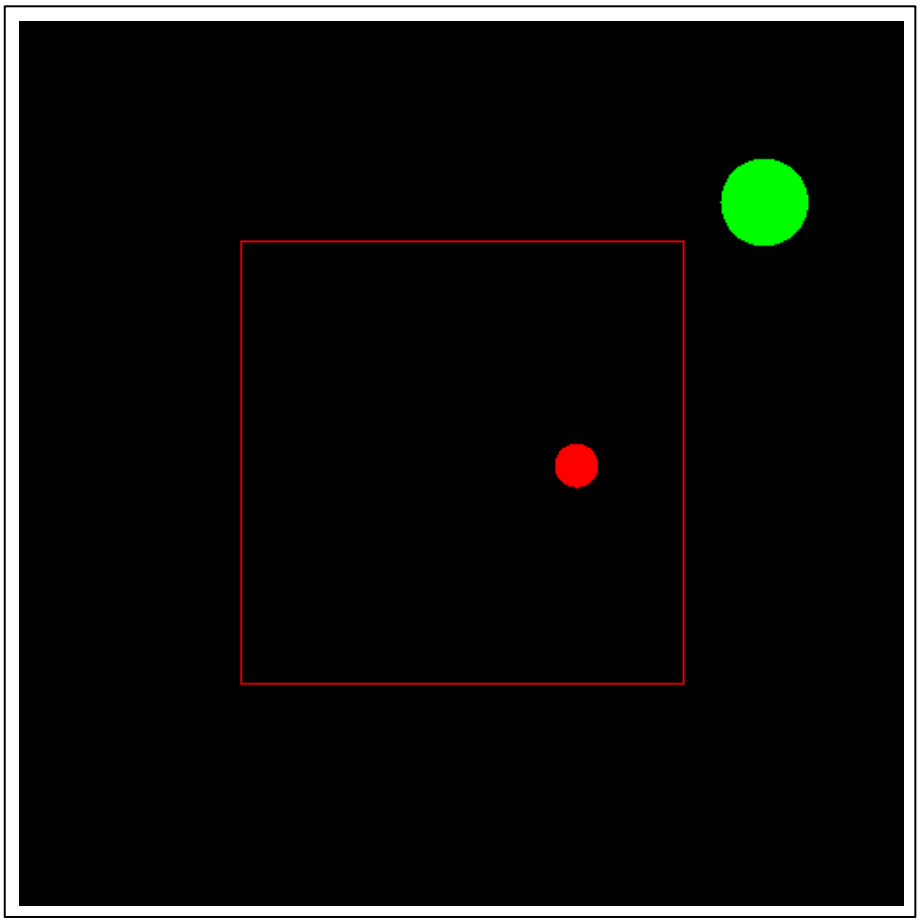

Figure 47 : Demo Application – listener moved outside the audio reproduction set-up's boundaries around a sound source within the physical audio reproduction set-up's boundaries

# 7. Conclusion and Future Work

Building a spatialized sound server on top of the "Bergen Server" implementation led to a stable implementation of a sound server being capable of reproducing a spatial impressions of sound sources being positioned anywhere around the listener.
For sound spatialization several different loudspeaker based techniques are used using one to four channels for sound reproduction.
Introducing new data structures based on abstract models for sound sources and physical listener representation, the exact calculation of spatialized sound characteristics is possible. Adjusting the attributes taken into account within the sound and listener models the boundary conditions are dynamically influenceable. Thus the sound representation is adjustable to a listener moving within the audio reproduction's set-up.
Although the final implementation of the spatialized sound server offers a realistic reproduction of spatial sound experiences and performs stable on different systems a variety of limitations arose during implementation of the server itself, the implementation of the demo application and within the testing phase.
The limitations are based on weaknesses within the assumed models for sound source and listener representation as well as weaknesses within the sound server implementation.

## 7.1. Sound Server Limitations

With the "Bergen Server" implementation a client – server communication model was implemented based on a UDP socket interface. Although UDP offers a direct way to send and receive datagrams over IP networks it does only provide a few error recovery services. Messages send within application flow may be lost resulting in unsynchronized conditions on the client and server side. Further limitations arise from the library the networking interface is based on. With the "Bergen Server" implementation the network interface is based on a platform dependent library forbidding a cross platform client server communication. Thus a client – server model exploiting advantages of two different operating systems on either one of the two sides is not possible.

Exchanging the networking library to a platform independent solution will open up the client – server communication and enabling the use of different operating systems and their advantages within the client – server model.

In addition to being limited to one common operating system within the client server communication the "Bergen Server" is only executable on Irix and Linux operating systems excluding the advantages of further multimedia specialized operating systems. Further following the idea of platform independency using a platform independent implementation of an audio library would make the sound server accessible for an enlarged user group. A stable and performant audio library has to be investigated and implemented within a next version of the spatialized sound server.

Tests have shown failure in audio reproduction when using a "SampleFile" sound object. Using a sound sample consisting of less samples than defined as the sound server's buffersize unpredictable values are reproduced for the missing samples resulting in disturbance. Adjusting the reproduction algorithm to ignore missing samples is removing the disturbing phenomenon.

The current techniques used for sound spatialization may not be applicable for all possible situations. Further techniques widening the currently available techniques to use one to four loudspeakers may extend the sound server's usability. Therefore additional techniques for sound spatialization must be implemented in future spatialized sound server implementations.

Changing the audio reproduction set-up while testing the spatialized sound server implementation made it's use uncomfortable. With every change in the loudspeaker set-up the "spatializedSnerdServer" applications had to be recompiled after having changed the loudspeaker positions.[23] Starting a different application for each server implementation made the sound server's use even more confusing.

[23] cp.: Section 5.2

A simple configuration file containing the current configuration that is parsed by a single sound server application which is would ease the use of the server. Thus the implementation of a configuration routine must be a task for future implementations.

## 7.2. Sound Source Model Limitations

The current spatialized sound server implementation is based on simple sound source model. The sound source is desribed by it's position and the essential attributes needed for distance attenuation.
Further information about the sound source and functionality to generate a more realistic sound impression are not implemented yet.
Currently only omnidirectional sound sources are assumed within the sound model. Since omnidirectional sound sources barely appear in reality the implementation of directional sound sources must be performed in the future. The sound source model must be widened to enable the definition of a sound shape, a sound emitting behavior as well as a sound sources volume.
Even more realistic sound characteristic can be expected taken into account the geometry of the surroundings. The geometry can be used to calculate reflections, obstruction and other physical effects within sound wave propagation. Performant ways to implement the necessary complex algorithms have to be investigated, developed and implemented within future more realistic server implementations.

Defining an ambient sound within an application is possible through positioning a sound at the listener's position and following the listener's position with the sound source. Such an implementation is resulting in an overhead in communication since every listener movement leads to a movement of each ambient sound source.
Individually defining the spatialization technique and distance attenuation for each sound source would delete this problem. Turning off the spatialization technique as well as the technique for distance attenuation would result in a sound source constantly played with a given amplitude.
Since performance is a critical problem within audio processing tasks reducing overhead is of first priority in the next implementation.

### 7.3. Listener Model Limitations

For the initial spatialized sound server implementation not only a simple model is chosen for sound source implementation but also for the physical listener representation. Testing the current sound server implementation failures in spatialization arose with the listener moving away from the audio reproduction set-up's hotspot.
The change in distance to each of the loudspeakers resulting in a change of perceived intensity is simply ignored.

With small reproduction environments this leads to only a small failure in spatialization but may lead to more critical sound experiences for a bigger set-up.
Further problems exist for listener movement outside the audio reproduction set-up's boundaries. Although the techniques used for spatialization are not defined for listener positions outside the stereophonic area implementing an algorithm to adjust to the undefined situation would provide confusing sound characteristics.
If further spatialization techniques are implemented in the future the physical listener's position within the audio reproduction environment is not adequate for a realistic sound characteristic calculation.
The problems originating from the simple physical listener model can be resolved by widening the model by including the listener's orientation within the audio reproduction environment as well as more exactly exploiting the available information to adjust the calculation of the spatial sound impression.

## References

Begault, Durand R. : 3-D sound For Virtual Reality And Multimedia, Moffet Field, NASA Ames Research Center, Academics Press Professional, 1994, Reprint 2000

Begault, Durand R. : Head-Up Auditory Displays for Traffic Collision Avoidance System Advisories, Moffet Field, NASA Ames Research Center, The Human Factors and Ergonomics Society Inc., 1993

Begault, Durand R. : Preferred Sound Intensity Increase For Sensation Of Half Distance, Moffet Field, AES 93rd Convention San Francisco, 1992

Creek Patricia a.o. : IRIS Digital Media Programming Guide, Silicon Graphics, Inc., 1994

Davis, Elisabeth T. a.o. : Can Audio Enhance Visual Perception and Performance in a Virtual Environment, Atlanta, Georgia Institute of Technology, 1999

Gardner, William G.: 3-D Audio Using Loudspeakers, Boston, Massachusetts Institute of Technology, Kluwer Academic Publishers, Norwell, Massachusetts, 1998

Hill, Alex : Ygdrasil Documentation, Electronic Visualization Laboratory – Universit of Illinois at Chicago, http://www.evl.uic.edu/yg/about.html , Chicago, IL, USA, 2002

Kendall, Gary : A 3-D sound Primer, http://www.northwestern.edu/musicschool/classes/3D/pages/3DsoundPrimer.html, 08/02/2002

Pape, Dave : Bergen Sound Server & Library, Version 0.4.1, Electronic Visualization Laboratory - Universit of Illinois at Chicago, http://www.evl.uic.edu/pape/sw/bergen/, 11/10/2002

Todt, Severin S., Integration of Multichannel Sound in Virtual Environments, University of Applied Sciences – Wedel, Wedel, Germany , February 2002

Tonnesen, Cindy; Steinmetz, Joe : 3-D sound Synthesis, Washington, The Encyclopedia of Virtual Environments, http://www.hitl.washington.edu/scivw/EVE/I.B.1.3DSoundSynthesis.html, 1993