Content

Introduction.....2
Intended Audience.....2
Compiler compatibility.....2
1 Introduction toYagl.....3
2 Yagl Design Overview.....3
3 Compiling and Using Yagl.....4
  3.1 Windows.....4
    3.1.1 Compiling the Library.....4
    3.1.2 Using Yagl.....4
  3.2 Linux.....5
    3.2.1 Compiling the Library.....5
    3.2.2 Using Yagl.....5
4 The Graphics Module.....6
  4.1 Overview.....6
  4.2 Initialization.....6
  4.3 Window Managment.....7
  4.4 Clearing the Screen.....7
  4.5 Used Coordinate System and Clipping.....7
  4.6 Drawing Primitives.....8
  4.7 Fonts and Printing.....8
  4.8 Bitmaps and Blitting.....10
  4.8 Presenting the Content of the Backbuffer on Screen.....14
  4.9 Notes on the OpenGL backend.....15
  4.10 Cleaning up.....15
5 The Input Module.....15
  5.1 Overview.....15
  5.2 Getting Input from the Keyboard.....15
  5.3 Getting Input from the Mouse.....16
  5.4Getting Input from Joysticks.....17
6 The Sound Module.....19
  6.1 Overview.....19
  6.2 Initialization.....19
  6.3 Used Coordinate System and the Listener.....19
  6.4 Soundbuffers and Soundsources.....20
    6.4.1 Soundbuffers.....20
    6.4.2 Soundsources.....22
  6.5 Soundstreams.....25
  6.6Notes on the OpenAL backend.....27
  6.7 Cleaning up.....28

Introduction

Yagl is a crossplattform gameprograming library providing usefull classes to cope with 2d graphics, 3d sound, input handling, threading and networking. The classes provide easy to use interfaces and try to hide lowlevel concepts from the user so that he can focus on the actual application he has in mind. Yagl also provides mechanisms such as logging and memory leak detection to ease the debugging process of your applications. Yagl is written in C++ and is known to work on Windows and Linux. Wrappers for other language bindings are currently in the making.

Intended Audience

This User's Guide is mainly focused at describing the different modules of Yagl and giving small examples of their usage. A complete Reference for all classes is available in Appendix A or in form of doxygen documentation within the header files. As Yagl is written in C++ so are the examples given here. The reader however only needs a limited basic understanding of C++.

Compiler compatibility

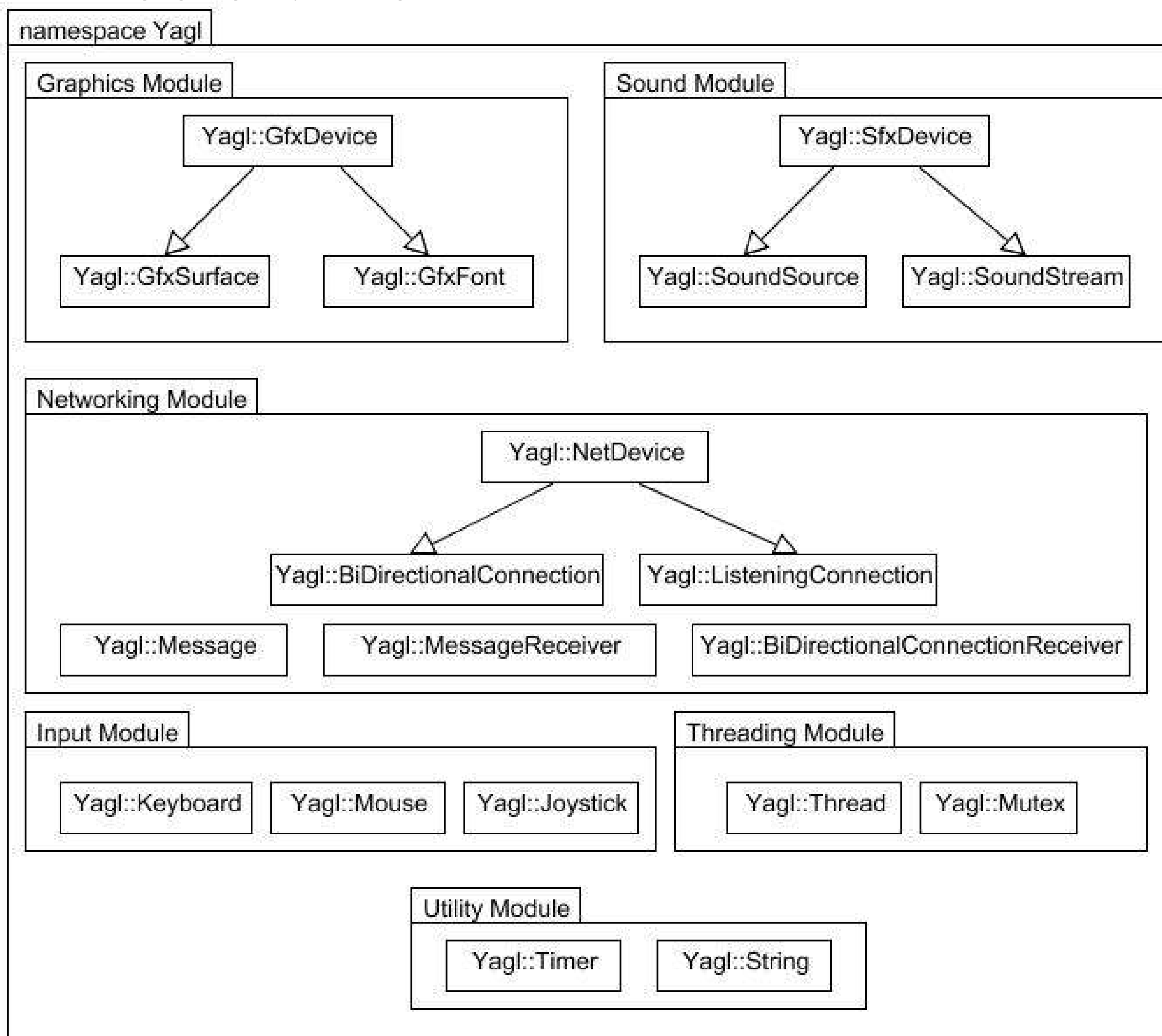
Yagl was written with GCC version 3.4.2 and is known to work with higher and lower versions. On Windows Yagl currently only supports the MinGW compiler suite altough this is a matter of change. Visual C++ 6/7/8 will be supported in future versions of Yagl, also cygwin will be targeted in the next release. Yagl does not make use of templated programming and only uses parts of the stl (namely std::list and std::vector), so it should be possible to use Yagl with other c++ compilers too.

1 Introduction toYagl

Yagl's design was created with ease of use in mind. Functionality is grouped in different classes each having certain features that let you accomplish a certain task, e.g. creating a soundstream, blitting a bitmap to the screen and so on. The 3 main modules of Yagl, audio, graphics and networking, each follow the samedesign principle: a so called device is representing the hardware it is related to (e.g. Yagl::GfxDevice represents your graphicscard) providing you with methods to create different objects that can be used in conjunction with this device (e.g. Yagl::GfxSurface representing a bitmap).The device is the only one capable of creating those objects, keeps track of all created objects and will destroy them on deinitialization of the device (or program exit). This makes it harder for the user to unintentionally introduce memory leaks or resource hogging into an application that uses Yagl. All Yagl specific class, constants and so on are organized within the Yagl namespace in order to avoid nameclashes.

2 Yagl Design Overview

The following figure gives you a rough overview of all classes and their relations to each other.



This are all classes you will ever have to deal with in Yagl. The figure illustrates the grouping of the classes in different modules. Lines originating from a class connected to another reflect that the connected classes can only be created and destroyed by the class at the connections origin. As one can see all creating classes have the word Device within their name. Devices in Yagl represent the underlying hardware. They are all

implemented as singletons and will keep track of all objects created via them. They will also guarantee that all objects created will be destroyed without the user having to destroy them manually, in fact it is manadatory to let the device destroy the object created by it. Devices can not be instantiated. Rather the user has to get a reference to the singleton instance of a device via methods provided via the yagl namespace (e.g. Yagl::getGfxDevice()). This methods will return the caller a reference to a device he can work with. Given this design it is possible to access any device from within any part of your code without having to pass around references to the device. The Yagl::get*Device() methods are public and globally available by including yagl.h into the sourcefile you want to access a certain device in. All the classes listed in the Input Module are also implemented as Singletons. The same rules apply to them as for the devices just mentioned. Here also methods are provided via the Yagl namespace to access the keyboard, mouse or joysticks available on your system (e.g. Yagl::getKeyboard()).

All modules of Yagl are independant of one another with the exception of the Yagl::Keyboard and Yagl::Mouse class that require a screenmode to be set with the Yagl::GfxDevice to work.

3 Compiling and Using Yagl

3.1 Windows

3.1.1 Compiling the Library

Yagl currently supports only the MinGW compiler suite. In Order to compile Yagl on Windows you therefor have to have MinGW installed on your system and the mingw\bin folder in your PATH environment variable so that g++, ar, ld and make are available. In the src directory you find a makefile that you can call with different options enabled. Here are all compilation options for Yagl:

yagl\src\make TARGET=win32, this will compile yagl in release mode, that is no debugging information is given within the resulting library file.

yagl\src\make TARGET=win32 DEBUG=1, this will build yagl with logging and debugging information. Applications linked to the library resulting from this build type will output a logfile that will contain various logmessages yagl created during runtime. Also the memory leak detector will be enabled and report any memory leaks.

After compiling Yagl successfully you will find a file called libyagl.a in the lib\win32 folder that you can then use for linking. Note: Yagl is linked statically for now. This results in rather big executables (around 2mb for a barebone application). This might be a matter of change in future Yagl releases.

3.1.2 Using Yagl

As mentioned earlier Yagl currently only supports MinGW on Windows so all instructions given here reference the steps for MinGW. The first thing you will need to be able to compile an application using Yagl is the include files. You can find those in the inc\ folder of the distribution. Make sure your include path points to this directory or copy the files over to any other include file folder you want to use. At the linking stage you need to link several library files. Here are the libraries you will need on Windows given as gcc link flags:

-lyagl -lglfw -lgdi32 -lws2_32 -lwinmm -lopengl32 -lglu32

libyagl.a and libglfw.a can be found in the lib\win32 folder of the yagl distribution. You have to make sure

that those libraries are in your link path. Copying those two files to your mingw\lib folder is probably the easiest way to achieve this.

After you have successfully compiled your application the exe will need a few .dll files to work. You can find those dll files in the bin\ folder of the Yagl distribution. You have to ensure that those .dll files are either in your PATH environemnt variable or in the same directory as your executable.

3.2 Linux

3.2.1 Compiling the Library

3.2.2 Using Yagl

4 The Graphics Module

4.1 Overview

The graphics module of Yagl is (for now) based entirely on OpenGL and relies on glfw for initializing a window and creating an opengl context on both windows and linux. It mainly focuses on 2D Graphics but can be used in conjunction with normal OpenGL graphics. The main class of the graphics module is Yagl::GfxDevice which you can get a reference to via Yagl::getGfxDevice(). This will return you a reference to the singleton instance of the graphics device (see Yagl Design Overview for a description of the singleton design). The Device is capable of creating and destroying Yagl::GfxSurfaces and Yagl::GfxFonts which will be described in depth below. Yagl::GfxSurfaces represent a bitmap of certain colordepth and size, a Yagl::GfxFont represents a truetype font that can be loaded from any truetype file and then be used for printing text to screen.

4.2 Initialization

Before any methods of the Yagl::GfxDevice are called one has to set a screenmode. This is done by the Yagl::GfxDevice::setScreenMode( ) method:

```
bool Yagl::GfxDevice::setScreenMode( int width, int height, int bitdepth, bool
fullscreen )
```

The first two parameters **width** and **height** let you specify the dimension of the window you want to create in pixels. **bitdepth** specifies the desired number of bits used for one pixel. Currently only 16-, 24- and 32-bit are supported. When the 16-bit format is chosen a pixel is encoded with 5 bits for the red color, 6 bits for the green color and 5 bits for the blue color. In 24-bit each color is encoded with 8-bits. In 32-bit mode the same applies as in 24-bit modes except that an additional channel is introduced for the alpha value of a pixel. The alpha value of a pixel can be used to specify the tranparency of a pixel, ranging from 0 (completely transparent) to 255 (completely solid). Finally **fullscreen** indicates wheter the screenmode should be initialized fullscreen or windowed (true for fullscreen, false for windowed mode). The Yagl::GfxDevice will try to setup a screenmode that is equal to the parameters you specified. However in case of windowed mode the color depth will default to the desktops colordepth. The method will return true on success and false on failure. Note that all previously Yagl::GfxFont and Yagl::GfxSurface instances that have been created with this Yagl::GfxDevice will get destroyed. E.g. if you provide the possibility of changing the screenmode in your application you have to reload graphics related data you previously loaded. If the call was successfull the user will be left with a window, either fullscreen or windowed, that can now be used to draw too. Also a doublebuffering system is established by default, meaning that everything you draw with commands like Yagl::GfxDevice::line or Yagl::GfxDevice::blit will first be drawn to an offscreen surface and at a call to Yagl::GfxDevice::swapBuffers be copied to the actual screen.

This would setup a screenmode with 320x200 pixels and 32-bit per pixel in fullscreen mode.

```
if( Yagl::getGfxDevice().setScreenMode( 320, 200, 32, true ) == false )
{
      cout <<  ''couldn't initialize screenmode\n'';
}
```

Note: Yagl's graphics module is based on OpenGL. The above described method will additionally setup a z-buffer with 16-bit precision. In future releases this might be a matter of change and could possibly controlled by the user himself.

4.3 Window Managment

Yagl will be responsible for handling all window related business once a screenmode is set. It is responsible for showing or hiding the mouse and reporting when the user clicked on the close button of the window.

To hide and show the mouse use:

```
void Yagl::GfxDevice::hideMouseCursor( )
void Yagl::GfxDevice::ShowMouseCursor( )
```

this two methods will hide or show the mouse cursor respectively.

Another usefull method allows querying wheter the close button of the window was pressed:

```
bool Yagl::GfxDevice::wasWindowCloseButtonPressed( )
```

This will return true in case the user clicked on the close button or false otherwise. Note that the window will not get closed if the user clicks on the close button.

4.4 Clearing the Screen

After initialization of the screenmode you probably want to clear the entire screen to some color. The Yagl::GfxDevice offers a method called Yagl::GfxDevice::clear for this purpose:

```
void Yagl::GfxDevice::clear( int color )
```

color specifies the color that the screen should be cleared with. It has to be specified in 24-bit rgb format with the lowest byte representing the blue color channel, the second byte representing the green color channel and the third byte representing the red color channel. The 4th byte is ignored.

This would clear the screen with an intense red color:

```
Yagl::getGfxDevice().clear( 0xff0000 );
```

4.5 Used Coordinate System and Clipping

Yagl uses the cartesian coordinate system with the twist that the y-axis is flipped, that is it's positive halfspace is pointing downwards on screen. This is pretty much standard in 2D computer graphics due to the storage of bitmaps in ram. Given a window created with Yagl the origin of the coordinate system is located in the top left corner of the window. Note that coordinates are discrete, that is of integer type as one is dealing with pixel positions here which are discrete in nature.

Clipping is automatically enabled in Yagl. Every drawing operation will perform clipping, that is don't draw parts of the primitive or bitmap that are outside of the so called clipping region. The clipping region is of rectangular shape and defines the area of the window where drawing operations will have an actual effect, that is pixels can be drawn to. Initialy the clipping region is set to the area of the hole window. You can manipulate the clipping region via the Yagl::GfxDevice's method:

```
void Yagl::GfxDevice::setClippingRegion( int clip_min_x, int clip_min_y, int
clip_max_x, int clip_max_y )
```

clip_min_x and **clip_max_x** define the minimum and maximum x coordinates for a pixel to have to get plotted to the window. **clip_min_y** and **clip_max_y** describe the minimum and maximum y coordinate a pixel can have to get plotted to the window. Values passed that lie outside the window (< 0, $>$ width/height) will be ignored and clamped to the minimum value respectively.

In order to retrieve the currents clipping regions measurements you can use:

```
void Yagl::GfxDevice::getClippingRegion( int &clip_min_x, int &clip_min_y, int
&clip_max_x, int &clip_max_y )
```

This will fill the passed references with the currents clipping regions size.

4.6 Drawing Primitives

Drawing primitives are simple shapes like lines and rectangles. Currently Yagl::GfxDevice only offers methods to draw lines, rectangles and filled rectangles. This is due to the circumstance that it is based on OpenGL and drawing other shapes would possibly not be performant enough. This primitives are provided for quick prototyping mostly as the average user will probably rely on blitting bitmaps to the screen rather than composing images by using lines.

Drawing lines in Yagl is done by a call to the method:

```
void Yagl::GfxDevice::line( int x1, int y1, int x2, int y2, int color, float
blend_factor = 1.0 )
```

x1, **y1** define the starting point of the line on the screen, **x2**, **y2** define the endpoint of the line on the screen. **color** is given in 24-bit rgb format and will be converted to the appropriate color depth internally. Finally blend_factor specifies the transluency of the line from 0.0 (transparent) to 1.0 (solid).

Drawing rectangles with Yagl is done via:

```
void Yagl::GfxDevice::box( int x1, int y1, int x2, int y2, int color. float
blend_factor = 1.0 )
void Yagl::GfxDevice::solidBox( int x1, int y1, int x2, int y2, int color,
float blend_factor = 1.0 )
```

the first method will only draw the outlines of the rectangle whereas the second method draws a filled rectangles. **x1**, **y1** define the bottom left corner of the rectangle (note that the positive y-axis is pointing downwards), **x2** and **y2** defines the top right corner of the rectangle. If **x2** is smaller than **x1** then they are internally exchanged, the same applies to **y2** and **y1**. The **color** should be given in 24-bit rgb format and will be converted to the current screenmodes color depth internally. Finally **blend_factor** specifies the transluency of the text from 0.0 (transparent) to 1.0 (solid).

Here is a simple example that will draw a line across the screen with the color blue, a rectangle outlining the windows bounds in the color green and a filled rectangle that is placed at some position with the color red at half intensity (alpha channel is set to 0x80).

```
Yagl::getGfxDevice().line( 0, 0, 640, 480, 0xff0000 );
Yagl::getGfxDevice().box( 0, 0, 639, 479, 0xff00 );
Yagl::getGfxDevice().solidBox( 100, 100, 200, 200, 0x800000ff );
```

4.7 Fonts and Printing

Yagl provides the user with facilities to load and print truetype fonts via Freetype 2. Only resizeable fonts can be used with Yagl.

The class responsible for all font operations is Yagl::GfxFont. It is one of two classes that can only be created and destroyed via the Yagl::GfxDevice. The Yagl::GfxDevice will keep track of all Yagl::GfxFont instances created and will destroy them either at program exit or if the user explicitely wishes to destroy them via Yagl::GfxDevice::destroyFont() and Yagl::GfxDevice::destroyAllFonts().

To create a Yagl::GfxFont instance one has to invoke:

```
Yagl::GfxFont* Yagl::GfxDevice::createFont( )
```

This will return a pointer to a Yagl::GfxFont instance that you can use for loading a truetype font and

printing text to screen with the loaded font. The Yagl::GfxDevice will internally keep track of the Yagl::GfxFont instance in a list so it can delete it at program exit or when the user wishes to do so explicitely. The method will return a null pointer in case a new Yagl::GfxFont could not be created (which is rather unlikely though).

After creating a Yagl::GfxFont one has to load a truetype font from a file. This is done via the method:

```
bool Yagl::GfxFont::loadFont( Yagl::String filename )
```

filename is the name of the truetype font file to be loaded. It is of type Yagl::String which will be described below in depth. For now it is sufficient to know that you can pass anything here from a char* pointer (this also includes string literals) to an stl string. The method will return true in case the font could be loaded successfully or false otherwise. Any previously loaded truetype font will be unloaded and replaced with the newly loaded font. The font is loaded with a default size of 16 pixels in height.

In orderto change the size of a Yagl::GfxFont one has to invoke:

```
void Yagl::GfxFont::setSize( unsigned int pt )
```

this method will of course only work if you previously loaded a truetype font successfully. Note that this method is extremely slow. Therefor it is recommended to set the sizes of all fonts you use before you enter your applications mainloop. Also, if you specify a very high size for the font it is possible that you will run out of memory. Note that Yagl::GfxFont::SetSize() is probably going to change in the next release, returning a boolean value indicating wheter the resizing was successfull or not.

If you successfully loaded and sized the font you can use it for printing text to the screen. Printing is done via:

```
void Yagl::GfxDevice::printAt( Yagl::String text, int x, int y, int color,
Yagl::GfxFont* font, float blend_factor = 1.0 )
```

text is a Yagl::String to the text you want to print on screen. Zero length strings are ignored. **x** and **y** specify the coordinates at which the text should be printed in the window. The coordinates relate to the top left corner of the imaginary rectangle that surrounds the text. **color** should be given in 32-bit argb format and specifies the color of the printed text. Note that the alpha channel is taken into account so one can have transluent text. This method will also recognize the escape sequence \n and insert a new line if it detects this in the passed string. **font** is a pointer to a previously created Yagl::GfxFont that you should have loaded a font to. If no font was loaded to the Yagl::GfxFont or one passes a null pointer the call is ignored. Finally **blend_factor** specifies the transluency of the text from 0.0 (transparent) to 1.0 (solid).

As stated above, there are methods to explicitely destroy one or more Yagl::GfxFont instance. To destroy a single Yagl::GfxFont object one has to invoke:

```
void Yagl::GfxDevice::destroyFont( Yagl::GfxFont* font )
```

font should be a pointer to a previously created Yagl::GfxFont instance. The Yagl::GfxDevice will then look up the font based on the pointer in a list of all created Yagl::GfxFonts and destroy the font if it can find it. If you pass a null pointer or the font is not in the list of the Yagl::GfxDevice this call is ignored.

You can also tell the Yagl::GfxDevice to destroy all currently in use Yagl::GfxFonts via:

```
void Yagl::GfxDevice::destroyAllFonts( )
```

This will destroy all Yagl::GfxFonts known by the Yagl::GfxDevice. Using the pointers formerly received by a call to Yagl::GfxDevice::createFont() will result in a segmentation violation and is not recommended.

Here's a small example demonstrating how to load, resize and print text with a Yagl::GfxFont.

```
Yagl::GfxFont* font = Yagl::getGfxDevice().createFont( );

if( font == 0 )
{
   cout << ''couldn't create font\n'';
}

if( font->loadFont( ''arial.ttf'' ) == false )
{
   cout << ''couldn't load font from file\n'';
}

font->resize( 22 );

Yagl::getGfxDevice( ''This is a test string'', 0, 0, 0xff, font );
```

4.8 Bitmaps and Blitting

Yagl's graphics module most usefull mechanism is using bitmaps for blitting. Blitting refers to drawing a certain part of a bitmap to the screen at a certain position. Yagl extends this by providing blitting methods that allow the user to scale and rotate the bitmap too. Bitmap in this context means a digital image of certain dimensions and with a certain colordepth.

A bitmap is represented as a Yagl::GfxSurface within Yagl. Like Yagl::GfxFonts a Yagl::GfxSurface can only be created and destroyed via the Yagl::GfxDevice. At the moment a Yagl::GfxSurface is in fact an OpenGL texture. In the next release it will be possible to get access to the OpenGL texture object handle in a transparent way.

To create a Yagl::GfxSurface one has to invoke:

```
Yagl::GfxSurface* Yagl::GfxDevice::createSurface( )
```

This will return a pointer to a Yagl::GfxSurface on success. As with Yagl::GfxFonts the created surfaces will be keept track of within the Yagl::GfxDevice so it is able to destroy it at program exit or if the user wishes to explicitely. If the method fails it will return a null pointer.

In order to make any use of the Yagl::GfxSurface one has to load an image from a file or memory location to it. Loading from a file is done via:

```
bool Yagl::GfxSurface::loadFile( Yagl::String filename )
```

filename specifies the name of the file to be loaded. Currently Yagl only supports 24- and 32-bit Windows Bitmaps and 24- and 32-bit PNG images to be loaded. This is going to change in the next release. There's also the possibility of writting your own Decoder and plug it into Yagl easily. This is described below in more detail. A previously loaded bitmap will be destroyed when calling this method. All loaded images are converted to 32-bit abgr format as used by OpenGL. Pixels with the color 0xff00ff (an intense pink) will have their alpha channel set to 0 having the effect that those pixels are not going to be blitted when using an alphamasked blit. This makes it possible to do color keying on images that don't have an alpha channel. In the next release yagl will provide a mechanism to specify what color is used as the color key. If the method was successfull true is returned, otherwise false

Loading from a memory location works via:

```
bool loadFromMemory( const unsigned char* data, unsigned int width, unsigned
int height, YAGL_BITMAP_FORMAT format )
```

data is a pointer to the location of the bitmaps pixels. They should be given in the same format as specified in the **format** parameter. The first pixel of the bitmap in the data corresponds to the pixel at coordinates 0,0

in the bitmap (check your graphics program of choice to see where this is, normally the top left corner of the image). **width** and **height** specify the dimensions of the image in pixels. **format** is an enumeration that specifies what colordepth the bitmap has and in what order the color channels are given for one pixel. **format** can be BITMAP_FORMAT_GRAYSCALE8 (each pixel is represented as an 8-bit grayscale value), BITMAP_FORMAT_RGB16 (each pixel is represented as a 16-bit rgb color value, with the lowest 5 bits being the blue channel the next 6 bits being the green channel and the highest 5 bits being the red channel), BITMAP_FORMAT_RGB24 (each pixel is represented by a 24-bit rgb color value, with the lowest byte representing the blue color channel, the next byte representing the green channel and the highest byte representing the red channel) and BITMAP_FORMAT_ARGB32 (each pixel is represented by a 32-bit argb color value, with the lowest byte representing the blue color channel, the next byte representing the green channel, the next byte representing the red channel and the highest byte representing the alpha channel). Use this method at your own risk and only if you know what you do. specifying contradicting values might even crash your application due to a segmentation violation or similar. The specified data will be converted to 32-bit abgr format as used by OpenGL and colorkeying is applied to as described above for Yagl::GfxSurface::loadFile(). If this method is successfull true is returned, false otherwise. A previously loaded bitmap will be destroyed when you call this.

As soon as we've loaded a bitmap to the Yagl::GfxSurface we can use it to blit it to screen. There's 4 different blitting routines. Let us first review those 4 shortly before we investigate the effects they have. The first and easiest to use one is Yagl::GfxDevice::blit(), which will just blit the image to the screen. The next method is Yagl::GfxDevice::blitScaled() which will scale the image on both axis according to the parameters you specified. The third method is Yagl::GfxDevice::blitRotated() which will blit the bitmap rotated around it's center. The last method is Yagl::GfxDevice::blitRotatedScaled(). This method will first rotate the image around it's center and then scale it according to the parameters you specified.

All those 4 methods come in 2 flavors either unranged or ranged. Unranged means that the complete bitmap is taken into account. Ranged means that only parts of the bitmap are used for blitting. You specificy the partial bitmap by passing the top left and bottom right pixel coordinate of the rectangular part of the bitmap that should get blitted. Those coordinates are denoted as scr_min_x, src_min_y, src_max_x and src_max_y in the argument list of the blitter methods.

Besides the blitting unranged and ranged you can also specifiy a blitting mode which tells yagl wheter the alpha channel of a bitmap should be taken into account or not. This is reflected in the mode argument for all the blitter methods. Valid values are Yagl::BLIT_SOLID, Yagl::BLIT_ALPHAMASKED and Yagl::BLIT_ALPHAMASKED_AND_BLENDFACTOR. The first blitting mode will blit all the pixels of the bitmap at full intensity to the screen, so the alpha channel is completely ignored. The second mode takes the alpha channel of the bitmap into account. The third method takes the alpha channel and a blendfactor into account which is another parameter all blitter methods have in common. This blendfactor defines the transluency of the complete bitmap. This can be used for effects like outfading explosions, where you need an alphamask to define only those parts of the bitmap that are part of the actual explosion and the blendfactor to slowly fade the explosion out (decreasing the blendfactor until it reaches 0). blendfactors are given between 0.0 (transparent) and 1.0 (solid). by default the YAGL::BLIT_SOLID mode is used if you don't sepcify a blit mode.

The next paragraphs describe the different blitter methods in detail.

The average user will mostly use the following two methods to blit a Yagl::GfxSurface to screen:

```
void Yagl::GfxDevice::blit( int x, int y, Yagl::GfxSurface *surface,
YAGL_BLIT_MODE mode = BLIT_SOLID, float blend_factor = 1.0 )

void Yagl::GfxDevice::blit( int x, int y, int src_min_x, int src_min_y, int
src_max_x, int src_max_y, Yagl::GfxSurface *surface, YAGL_BLIT_MODE mode =
BLIT_SOLID, float blend_factor = 1.0 )
```

x and **y** specify the location where the Yagl::GfxSurface should get blitted to. The coordinates correspond to where the top left pixel of the Yagl::GfxSurface will be located. **surface** is a pointer to a previously created

Yagl::GfxSurface. If the surface does not contain a bitmap, that is one didn't load a file or from memory then the call gets ignored. The same applies to providing a null pointer. **mode** specifies how the Yagl::GfxSurface should be blitted. By default BLIT_SOLID is used, meaning that all the pixels will be blitted and the alpha channel is not taken into account. Other modes are BLIT_ALPHAMASKED and BLIT_ALPHAMASKED_AND_BLENDFACTOR. In the last case the parameter **blend_factor** is also taken into account and blends the complete Yagl::GfxSurface. If you only want to blit a part of the Yagl::GfxSurface you haev to use the second method. You can specify a rectangle in pixel coordinates. All pixels within this rectangle on the Yagl::GfxSurface will be used for blitting, everything outside that rectangle is ignored. **src_min_x** and **src_min_y** correspond to the top left pixel of the area of the Yagl::GfxSurface that gets blitted**.** **src_max_x** and **src_max_y** correspond to the bottom right pixel. In case the coordinates passed are outside the Yagl::GfxSurface's bitmap area they get clamped to the minimum value. Also if **src_min_x** > **src_max_x** they will get swapped, the same applies to **src_min_y** and **src_max_y**. All things said about this and the mode parameter apply to all other blitter methods and will not be explained again.

Yagl also provides the user with a method to blit a Yagl::GfxSurface scaled:

```
void Yagl::GfxDevice::blitScaled( int x, int y, float scale_x, float scale_y,
Yagl::GfxSurface *surface, YAGL_BLIT_MODE mode = BLIT_SOLID, float
blend_factor = 1.0 )

void Yagl::GfxDevice::blitScaled( int x, int y, float scale_x, float scale_y,
int src_min_x, int src_min_y, int src_max_x, int src_max_y, Yagl::GfxSurface
*surface, YAGL_BLIT_MODE mode = BLIT_SOLID, float blend_factor = 1.0 )
```

we'll only discuss the new parameters as the rest is the same as for Yagl::GfxDevice::blit(). **scale_x** and **scale_y** give you a possibilty to scale the Surface along the x and y axis of the bitmap. Passing 1.0 for both parameters is equal to blitting the Yagl::GfxSurface with Yagl::GfxDevice::blit(). For downsizing the Yagl::GfxSurface specify a value smaller than 1.0, for increasing the size pass a value greater than 1.0. negative values will result in flipping the image.

Besides scaling a Yagl::GfxSurface you can also rotate it:

```
void Yagl::GfxDevice::blitRotated( int x, int y, float angle, Yagl::GfxSurface
*surface, YAGL_BLIT_MODE mode = BLIT_SOLID, float blend_factor = 1.0 )

void Yagl::GfxDevice::blitrotated( int x, int y, float angle, int src_min_x,
int src_min_y, int src_max_x, int src_max_y, Yagl::GfxSurface *surface,
YAGL_BLIT_MODE mode = BLIT_SOLID, float blend_factor = 1.0 )
```

Only the parameter **angle** is of interest here as the other parameters function the same as with a call to Yagl::GfxDevice::blit(). **angle** specifies the rotation angle about which the image should be rotated in degrees. The center of the rotation is the center of the Yagl::GfxSurface. Also **x** and **y** don't correspond to the top left corner of the Yagl::GfxSurface but to the center of it.

Finally there's one method that lets you rotate and scale a Yagl::GfxSurface all at once

```
void Yagl::GfxDevice::blitRotatedScaled( int x, int y, float angle, float
scale_x, float scale_y, Yagl::GfxSurface *surface, YAGL_BLIT_MODE mode =
BLIT_SOLID, float blend_factor = 1.0 )

void Yagl::GfxDevice::blitRotatedScaled( int x, int y, float angle, float
scale_x, float scale_y, int src_min_x, int src_min_y, int src_max_x, int
src_max_y, Yagl::GfxSurface *surface, YAGL_BLIT_MODE mode = BLIT_SOLID, float
blend_factor = 1.0 )
```

x and **y** again correlate to the center of the Yagl::GfxSurface. The rest of the parameters works as with the previously described blitters. Note that Yagl::GfxSurface is first scaled and then rotated.

To explicitely destoy a certain Yagl::GfxSurface one can use:

```
void Yagl::GfxDevice::destroySurface( Yagl::GfxSurface* surface )
```

surface is a pointer to a Yagl::GfxSurface previously created. All memory associated with the Yagl::GfxSurface will be freed. Don't use the pointer after a call to this.

If you want to destroy all the Yagl::GfxSurfaces currently in the system use:

```
void Yagl::GfxDevice::destroyAllSurfaces( )
```

this will destroy all Yagl::GfxSurfaces in the system explicitely. All the memory associated with the Yagl::GfxSurfaces is freed. Do no use any pointers to Yagl::GfxSurfaces after a call to this as they are all invalid.

Note that the System will automatically clean up any still existing Yagl::GfxSurfaces in the system at programexit or a call to Yagl::GfxDevice::setScreenMode(). This also applies to all Yagl::GfxFonts.

Here's a small example that shows how to create a Yagl::GfxSurface and blit it with the different blitter methods:

```
Yagl::GfxSurface* surface = Yagl::getGfxDevice().createSurface();
if( surface == 0 )
{
   cout << ''couldn't create a surface'';
   return 0;
}

if( !surface->loadFile( ''myimage.bmp'' ) )
{
   cout << ''couldn't load file 'myimage.bmp' to surface'';
   return 0;
}

Yagl::getGfxDevice().clear( 0 );

//
// this will blit all the pixels of the surface to screen
//
Yagl::getGfxDevice().blit( 0, 0, surface );

//
// this will only blit the pixels inside the specified rectangle on the
surface to the screen ( 32*32 pixels )
//
Yagl::getGfxDevice().blit( 0, 0, 0, 0, 31, 31, surface );

//
// this will blit the surface to screen with it's width scaled to 50%
//
Yagl::getGfxDevice().blitScaled( 0, 0, 0.5, 1.0, surface );

//
// this will blit the surface rotated about 45 degree to the screen with it's
center at 100, 100
//
Yagl::getGfxDevice().blitRotated( 100, 100, 45, surface );

//
// this will blit the surface with it's height scaled to 50% and rotated about
45 degree to the screen with it's
// center at 100, 100 and with alphamasking turned on
//
Yagl::getGfxDevice().blitRotatedScaled( 100, 100, 45, 1.0, 0.5, surface,
Yagl::BLIT_ALPHAMASKED );

//
// present the frame on screen
//
Yagl::getGfxDevice().swapBuffers();
```

As the current Yagl graphics module is based on OpenGL it is suggested to use power of 2 bitmaps for best memory usage and performance. Bitmaps that are not a power of two dimension wise will be extended to the next power of two, e.g. a bitmap of 300x300 pixels in size is converted to a 512x512 bitmap. Also the maximum size for a bitmap is dependand on your graphics card. All OpenGL implementations have to guarantee that 256x256 pixel wide bitmaps are supported as textures, so this marks the lower end. The next release of Yagl will have methods that let you query the maximum bitmap size.

4.8 Presenting the Content of the Backbuffer on Screen

As mentioned in chapter 4.2, Yagl sets up a double buffered system by default. This means that all drawing is done on a invisible buffer that has to be copied to the onscreen buffer when the current frame is done drawing. To do this one has to invoke:

```
void Yagl::GfxDevice::swapBuffers( )
```

You will have to call this method when you have finished drawing all the content you want to see on screen. In it's current design this method will also do something else i want to explain shortly.

4.9 Notes on the OpenGL backend

As Yagl is based on OpenGL there's certain things one has to keep in mind when using OpenGL as a 2D backend. First some matrices have to be manipulated, the texture unit has to be initialized to a certain state and a couple of other attributes have to be set accordingly. Yagl preserves all the states a user might have set for say his 3D engine outside of Yagl. This way it is possible to use Yagl seamlessly with an OpenGL based 3D engine for example. No special care has to be taken as Yagl will preserv all states it changes.

Yagl doesn't draw and blit immediatly if the user calls one of those methods. The commands issued are really stored in an internal list that can hold up to 10000 commands. If this command buffer is full Yagl will execute every command in it (that is draw and blit) and empty the buffer again. This will also happen if one calls Yagl::GfxDevice::swapBuffers(). This implies two things users of Yagl have to keep in mind:

1) The user is NOT allowed to delete any Yagl::GfxSurface or Yagl::GfxFont he used in a Yagl::GfxDevice::printAt() or Yagl::GfxDevice::blit() call before he called Yagl::GfxDevice::swapBuffers(). If the user doesn't follow this rule a segmentation violation will happen as the commands in the command list store pointers to Yagl::GfxSurfaces and Yagl::GfxFonts passed to the drawing methods that will be invalid after deletion.
2) As Yagl::GfxDevice::swapBuffers() is the last call issued before the current frame is presented on screen and since this is the time when Yagl actually draws all the issued commands it keeps in it's command buffer all the drawn things will appear on top of anything drawn without Yagl. For example if the user renders a 3D object with normal OpenGL calls and issues Yagl drawing methods inbetween the graphics drawn via Yagl will be drawn on top of the 3D object no matter the order of operations.

4.10 Cleaning up

The graphics module does not need any explicit cleanup. The Cleanup is performed at program exit automatically. This also means that any not yet deleted Yagl::GfxSurfaces and Yagl::GfxFonts are destroyed at program exit.

5 The Input Module

5.1 Overview

The Input Module offers the user possibilities to query the state of the keyboard, the mouse and up to 16 joysticks connected to the system. For each input device there's a class representing it namely Yagl::Keyboard, Yagl::Mouse and Yagl::Joystick. Each of this classes are again implemented as singletons and can thus be accessed globally in a convenient manner. The namespace Yagl offers 3 methods to get a reference to each of the singleton instances. Those are Yagl::getKeyboard(), Yagl::getMouse() and Yagl::getJoystick(). Yagl::getJoystick() is special as you have to specify the number of the joystick you want to get a reference to. Joysticks are numbered from 0 to 15.

5.2 Getting Input from the Keyboard

The Yagl::Keyboard class is very simple for now. It only supports the latin-1 character set (see http://en.wikipedia.org/wiki/Latin-1) and only let's you check wheter a certain key is pressed or not. Support for unicode and buffered keyboard input might be included in the next release.

To check wheter a certain key is pressed one has to invoke:

```
bool Yagl::Keyboard::isKeyPressed( int scancode )
```

scancode can either be any printable character of the the character set defined in the latin-1 set or one of the following special keycodes:

YAGL_KEY_ESCAPE	YAGL_KEY_RIGHT
YAGL_KEY_BACKSPACE	YAGL_KEY_DOWN
YAGL_KEY_TAB	YAGL_KEY_END
YAGL_KEY_ENTER	YAGL_KEY_INSERT
YAGL_KEY_LCTRL	YAGL_KEY_DELETE
YAGL_KEY_RCTRL	YAGL_KEY_F1
YAGL_KEY_LSHIFT	YAGL_KEY_F2
YAGL_KEY_RSHIFT	YAGL_KEY_F3
YAGL_KEY_LALT	YAGL_KEY_F4
YAGL_KEY_RALT	YAGL_KEY_F5
YAGL_KEY_SPACE	YAGL_KEY_F6
YAGL_KEY_HOME	YAGL_KEY_F7
YAGL_KEY_PAGEUP	YAGL_KEY_F8
YAGL_KEY_PAGEDOWN	YAGL_KEY_F9
YAGL_KEY_UP	YAGL_KEY_F10
YAGL_KEY_LEFT	YAGL_KEY_F11

Note that the keyboard's state will only be updated if you call Yagl::GfxDevice::swapBuffers() and a window was perviously created via Yagl::GfxDevice::setScreenMode(). A call to this will effectively poll the keyboard's state and update it. This might be a matter of change in future releases.

Here's a short example that shows how to use the Yagl::Keyboard::isKeyPressed() method in a mainloop

```
while( Yagl::getKeyboard().isKeyPressed( YAGL_KEY_ESCAPE) ||
       Yagl::getKeyboard().isKeyPressed( 'q' ) )
{
   // do some drawing here...
   Yagl::getGfxDevice().swapBuffers();
}
```

for things like text input boxes Yagl provides you with a method that works like Qbasic's inkey$. This method is buffering keypresses as you are used to it when writting text in notepad or emacs. The method is called:

```
int Yagl::Keyboard::getKey( )
```

it will return either a printable character of the latin-1 character set or one of the above special keys. In case no key was pressed it will return YAGL_KEY_NONE. Note that all the special keys have a value < 0 so you can easily distinguish between a printable key and a special key when using this function.

5.3 Getting Input from the Mouse

The mouse is represented via the Yagl::Mouse class within Yagl. As already mentioned it is implemented as

a singleton you can get a reference to via Yagl::getMouse(). Yagl's mouse implementation supports quering up to 3 mouse buttons (left, right, middle) and 3 axis (x, y and mousewheel).

to query for the position on one of the axes use:

```
int Yagl::Mouse::getX()
int Yagl::Mouse::getY()
int Yagl::Mouse::getZ()
```

Yagl::Mouse::getX() will return the mouse's current position on the x axis within the window, Yagl::Mouse::getY() will return the mouse's current position on the y axis within the window (see chapter 4.4 for a description of the used coordinate system). Yagl::Mouse::getZ() returns the current position of the mousewheel. If the mouse is outside the windows area -1 will be returned for any of the above methods. Also, if no mousewheel is available -1 will be returned too.

To query the state of a certain mouse button Yagl::Mouse offers:

```
bool Yagl::Mouse::isLeftButtonPressed( )
bool Yagl::Mouse::isRightButtonPressed( )
bool Yagl::Mouse::isMiddleButtonPressed( )
```

each of the methods will return true in case the button in question is pressed or false otherwise.

Note that the mouse's state will only be updated if you call Yagl::GfxDevice::swapBuffers() and a window was perviously created via Yagl::GfxDevice::setScreenMode(). A call to this will effectively poll the mouse's state and update it. This might be a matter of change in future releases.

In the rare event that you have to set the mouse position there's:

```
void Yagl::Mouse::setPosition( int x, int y )
```

this method lets you set the mouse position relative to the window origin (top left corner) in pixels

5.4Getting Input from Joysticks

Yagl supports up to 16 connected joysticks at once. A Joystick is represented via the Yagl::Joystick class. Yagl is querying the number of joysticks connected and will assign a number to each connected joystick starting at 0. So if you have 2 joysticks connected to your system then the first joystick will have the index 0 and the second joystick will have the index 1.

Yagl's joystick implementation allows the user to query for 6 joystick axis (x, y, z, r, u, v) and up to 32 joystick buttons. All the axis coordinates returned are given in a normalized fashion between -1.0 and 1.0 where a value of 0.0 means that the joystick is centered on that axis.

Before you want to retrieve information of a certain joystick you want to know wheter that joystick is connected. This can be done via:

```
bool Yagl::Joystick::isConnected( unsigned int id )
bool Yagl::Joystick::isConnected( )
```

the first method is a static method, meaning that you can call it from anywhere. It takes a parameter **id** which identifies the joystick you want to check. In case the specified id is smaller than 0 or bigger than 15 or the joystick is not connected the method will return false. The second method is an instance method you can use if you are using an instance of Yagl::Joystick. All methods of Yagl::Joystick come in this two flavors.

In case you don't want to use the indexing method you can get a reference to a Yagl::Joystick instance for a joystick with a certain id. This can be done via:

```
Yagl::Joystick& Yagl::Joystick::getInstance( unsigned int id )
```

id specifies the number of the joystick you want to get a reference to. If **id** is smaller than 0 or bigger than 15 this will return a reference to a null joystick, meaning that it is a virtual non connected joystick which will always return 0 for the axis and false for the button states.

To query the position of the joystick on a certain axis use:

```
float Yagl::Joystick::getX( unsigned int id )
float Yagl::Joystick::getY( unsigned int id )
float Yagl::Joystick::getZ( unsigned int id )
float Yagl::Joystick::getR( unsigned int id )
float Yagl::Joystick::getU( unsigned int id )
float Yagl::Joystick::getV( unsigned int id )
float Yagl::Joystick::getX( )
float Yagl::Joystick::getY( )
float Yagl::Joystick::getZ( )
float Yagl::Joystick::getR( )
float Yagl::Joystick::getU( )
float Yagl::Joystick::getV( )
```

id specifies the number of the joystick you want to query. If **id** is smaller than 0 or bigger than 15 0 is returned. The coordinates returned are within the range [-1.0, 1.0], where a coordinate of value 0.0 means that the joystick is centered on that axis.

To query the state of a joysticks buttons use:

```
int Yagl::Joystick::getButtons( unsigned int id )
int Yagl::Joystick::getButtons( )
```

id specifies the number of the joystick you want to query. If **id** is smaller than 0 or bigger than 15 this will return 0. otherwise one each bit within the returned integer represents the state of a button of the joystick where a set bit means that the button is pressed. You can easily test for a pressed button by bitwise „anding“ the returned bitfield with 2 powered by the buttons number.

Here's a small example demonstrating the use of the Yagl::Joystick class. It checks wheter two joysticks are connected and then uses the global and instance versions to query information of the joystick.

```
Yagl::Joystick& joystick = Yagl::Joystick::getInstance( 1 );
//
// check if joystick 0 and 1 are connected
//
if( Yagl::Joystick::isConnected( 0 ) != true )
{
      cout << ''joystick 0 is not connected'';
}

if( joystick.isConnected() != true )
{
      cout << ''joystick 1 is not connected
}

cout << ''Joystick 0: '' << Yagl::Joystick::getX(0) << Yagl::Joystick::getY
(0);
cout << ''\nJoystick 1: '' << joystick.getX() << joystick.getY();
```

Note that the Yagl::Joystick class does not depend on a call to Yagl::GfxDevice::swapBuffers() to be up to date unlike the Yagl::Keyboard and Yagl::Mouse.

6 The Sound Module

6.1 Overview

The sound module of Yagl is based on OpenAL which is a 3D audio library very similar to OpenGL interface wise. Benefiting from OpenAL's functionality Yagl can provide real 3D sound meaning that the user can play sounds at certain positions in 3D space.

The class Yagl::SfxDevice represents the underlying audio hardware and gives the user methods to load sound samples from files, play them once with certain attributes, create Yagl::SoundSources and Yagl::SoundStreams and position the so called Listener. The Listener represents the ears that hear the sound within the 3D space, thus the listener has a position and orientation given in 3D space. Additionaly the listener has a velocity that is used when simulating the doppler effect is wanted.

Yagl::SoundSources and Yagl::SoundStreams are essentially the same except that they use different sources for the audio data. Yagl::SoundSources use previously loaded soundbuffers which are managed by the Yagl::SfxDevice as an audio data source. Those soundbuffers mostly contain audio data that is small like a soundsample for a gunshot or an explosion. Yagl::SoundStreams use a file to stream audio data from like a music file for background music in some format. Aside from this Yagl::SoundSources and Yagl::SoundStreams are very similar in natur and represent an object in 3D space emitting sound. Both classes have a position in 3D space and optionally a velocity if the user wants to simulate the doppler effect (see http://en.wikipedia.org/wiki/Doppler_effect).

6.2 Initialization

Before one can use the sound module one has to initialize the Yagl::SfxDevice. This will prepare the underlying hardware for audio output and is done via:

```
bool Yagl::SfxDevice::initialize( )
```

this will initialize the hardware and return true on success or false otherwise. It will also set the listener to the position (0, 0, 0), his velocity to (0, 0, 0) and his orientation to (0, 1, 0) which is the default position for the listener. More information on the listener can be found below.

6.3 Used Coordinate System and the Listener

As stated previously, Yagl's sound module is a 3D sound module. Thus sounds have to be positioned in 3D space. Being based on OpenAL Yagl uses the same coordinate system as OpenAL: the positive x-axis points to the right, the positive y-axis points upwards and the negative z-axis points away from the listener. This is in accordance to the OpenGL coordinate system.

In analogy to a viewer in OpenGL there's a so called listener in OpenAL and Yagl's sound module respectively. As the viewer represents the eyes in 3D space the listener represents the ears in 3D space. The listener has 3 properties: position, orientation and velocity where the last one is only taken into account if the doppler effect is being simulated. Initially the user is positioned at the origin of the coordinate system with his velocity set to 0 and his orientation being (0, 1, 0). the orientation is sometimes also called up vector and represents the y-axis of the listeners own coordinate system.

To manipulate the listeners attributes the Yagl::SfxDevice offers 3 methods:

```
void Yagl::SfxDevice::setListenerPosition( float x, float y, float z )
void Yagl::SfxDevice::setListenerOrientation( float x, float y, float z )
void Yagl::SfxDevice::setListenerVelocity( float x, float y, float z )
```

the first method let's you specify the listeners position in the 3D space. The second method allows you to specify the listeners orientation in 3d space (imagine this as a vector going upwards from his head). the last method can be used to set a velocity for the listener which is taken into account when the doppler effect is calculated.

In order to get to know the listeners current attributes one can use:

```
void Yagl::SfxDevice::getListenerPosition( float &x, float &y, float &z )
void Yagl::SfxDevice::getListenerOrientation( float &x, float &y, float &z )
void Yagl::SfxDevice::getListenerVelocity( float &x, float &y, float &z )
```

this methods will each store the listeners current attributes in the passed references.

Note that for a simple 2D game you will most likely ignore the 3D capabilities of Yagl's sfx module. You can safely ignore all the passed and following references to parameters used for 3D sound calculations if you simply want to play sound samples or streams.

6.4 Soundbuffers and Soundsources

In Soundprogramming 2 different approaches can be taken,one being loading a complete audio sample to ram and play it from there and the other being streaming (and sometimes also decoding) audio data from a file on the fly. The first approach is suited for things like soundsamples for gunshots or explosions where the later approach is perfectly fine for use with background music. One might ask why the first approach can't be used for bigger samples like music too. The answer is: memory constraints. For example a simple ogg file might be 3mb in size on your harddisk, however this file is compressed and would take up around 30mb of ram when loaded entirely to memory. It is therefor less resource wasting to stream and decode bigger audiosamples on the fly. This section describes how Yagl can be used to implement the first approach in your application.

6.4.1 Soundbuffers

Soundbuffers are basically memory locations managed by the Yagl::SfxDevice that hold audio data and have an identification number. There's no special class for soundbuffers, all you will have to deal with is a handle (aka identification number) to a soundbuffer. Those soundbuffers are audio data sources for Yagl::SoundSources we'll examine later but can also be played independantly of a Yagl::SoundSource. A soundbuffer can be used by several Yagl::SoundSources simultaniously and can also be played multiple times in parallel.

To create a soundbuffer invoke:

```
bool Yagl::SfxDevice::createSoundBufferFromFile( const Yagl::String filename,
unsigned int *buffer_handle )
```

filename is the name of the file to be loaded into the buffer, **buffer_handle** is a pointer to an integer that will be assigned the handle to the buffer on success. Currently this method can load oggs, wavs, aus and a couple of other wavelet fileformats. See http://www.mega-nerd.com/libsndfile/ for a complete list of all supported wavelet formats. In case the method was successfull it will return true and set buffer_handle to the handle of the soundbuffer. In case the method failed it will return false and buffer_handle is undefined.

After creating a soundbuffer you can use it in 2 ways.

The first one is to simply play it back with certain attributes. This can be done via:

```
bool Yagl::SfxDevice::playSoundBuffer( unsigned int buffer_handle, bool
relative = true, float *position = 0, float *velocity = 0, float pitch = 1.0f,
float gain = 1.0f, float roll_off = 0.0 )
```

buffer_handle is the handle to a soundbuffer you previously created. If you only want to play the sound at full volume and centered (stereo audio wise) that's the only parameter you have to specifiy. This is extremely well suited for situations where you don't care for spatial positioning of a sound for example in a

simple 2D game. **relative** specifies wheter the sound should be played relative to the listener. This means that no matter where the listener is the sound will be played relative to him. In case you don't specify a position the sound will be played at the same position the listener is at resulting in the same effect a call to this only passing **buffer_handle** would have. **position** is a pointer to an array of floats that represent the coordinates the sound should be played at or the distance to the listener if relative is set to true. The array should hold 3 floats specifying the x, y and z coordinate in that order. **velocity** is also a pointer to an array of 3 floats that specify the velocity on each axis. This will be used for calculating the doppler effect. You can pass 0 for both parameters **position** and **velocity** which will be equal to positioning the sound at (0, 0, 0) with a zero velocity. **pitch** specifies the factor the frequencies of the sound should be multiplied with. This can be used to play a sound at a slower or higher rate resulting in lower and higher sounds. 1.0 is the standard value and tells the method to play the sound at normal speed. **gain** is the volume of the sound where 1.0 means standard volume. **roll_off** is a factor that specifies how much the sound will decrease in volume the more distant it is to the listener. A roll_off of 0.0 means that the distance to the listener is not taken into account and the sound will be played at full volume no matter it's position. A roll_off of 1.0 is equal to the physical decrease of sound volume in reallife.

You can call this method for one soundbuffer as often as you want, it will be played in parallel. Note that there's a limit of how many sounds you can play in parallel in general no matter wheter they are soundbuffers, Yagl::SoundSources or Yagl::SoundStreams. More on this below.

To explicitely destroy a soundbuffer you can use:

```
void Yagl::SfxDevice::destroySoundBuffer( unsigned int buffer_handle )
```

buffer_handle speficies the handle to the soundbuffer that should be destroyed. This will cause the memory of the soundbuffer to be deallocated and free and other resources possibly attached to it. Any Yagl::SoundSources that use this soundbuffer will stop playing and any attempt to play the Yagl::SoundSource will fail unless a new existant buffer is attached to the Yagl::SoundSource.

For deleting all soundbuffers currently in use invoke:

```
void Yagl::SfxDevice::destroyAllSoundBuffer( )
```

this will delete all soundbuffers currently existant and free any resources attached to it. Any Yagl::SoundSources will stop playing and any attempt to play the a Yagl::SoundSource will fail unless a new existant buffer is attached to the Yagl::SoundSource.

Note that you don't have to explicitely delete soundbuffers as the Yagl::SfxDevice will do this for you at program exit or a call to Yagl::SfxDevice::deinitialize().

Here's a short example that loads a soundbuffer with a wav file and plays it 2 times.

```
unsigned int buffer_handle = 0;
float left_to_listener[] = { -1.0f, 0.0f, 0.0f };

if( Yagl::SfxDevice::initialize() == false )
{
   cout << ''couldn't initialize sfx device'';
   return 0;
}

if( Yagl::SfxDevice::createSoundBufferFromFile( ''mywave.wav'', &buffer_handle
) == false )
{
   cout << ''couldn't load soundbuffer'';
   return 0;
}

Yagl::SfxDevice::playSoundBuffer( buffer_handle );
Yagl::SfxDevice::playSoundBuffer( buffer_handle, true, left_to_listener );
```

6.4.2 Soundsources

Playing a soundbuffer in the above presented way is limiting. The sounds attributes can not be changed while it plays, so the method above is a fire and forget way to play a sound. For more control of the attributes of a played sound while it is playing one can use the class Yagl::SoundSource.

A Yagl::SoundSource represents a sound emitting object in 3D space. A Yagl::SoundSource uses a soundbuffer as a source for the audio data it emits. It has attributes that specify it's position, velocity, pitch, gain and roll off factor.

Yagl::SoundSources are created and managed by the Yagl::SfxDevice. A Yagl::SoundSource can be created via a call to:

```
Yagl::SoundSource* Yagl::SfxDevice::createSoundSource( unsigned int
buffer_handle )
```

buffer_handle is a handle to a previously created soundbuffer. This soundbuffer will be assigned to the Yagl::SoundSource. In case the soundbuffer described by the handle does not exist the method will return 0 indicating that an error occured. In case of a successfull creation the method returns a pointer to a Yagl::SoundSource. Note that this Yagl::SoundSource is managed wihin the Yagl::SfxDevice and will be destroyed implicitely at a call to Yagl::SfxDevice::deinitialize() or at program exit. The concept is similar to Yagl::GfxDevice and Yagl::GfxSurfaces. Also, the current signature of the method above might change in the next release of Yagl so it is not necessary to provide a handle to a soundbuffer. The soundbuffer used by the Yagl::SoundSource can be set after creation too so this is a bit of a bad design. All the Yagl::SoundSource's attributes are set to default values. The position will be set to (0, 0, 0) the same applies to the velocity. the gain, pitch and roll off factor are all set to 1.0 and the Yagl::SoundSource will be relative to the coordinate system's origin rather than to the listener.

To destroy a specific Yagl::SoundSource one can use:

```
void Yagl::SfxDevice::destroySoundSource( Yagl::SoundSource* source )
```

this will stop the soundsource. **source** is the pointer to a previously created Yagl::SoundStream

to destroy all Yagl::SoundStreams currently in the system use:

```
void Yagl::SfxDevice::destroyAllSoundSources( )
```

this will stop all the soundstreams from playing and free up the channels assoicated with them.

To let the Yagl::SoundSource use another soundbuffer as audio data source you can use:

```
bool Yagl::SoundSource::setSoundBuffer( unsigned int buffer_handle )
```

buffer_handle is a handle to a previously created soundbuffer. In case the Yagl::SoundSource was playing it will stop playing. If the specified soundbuffer handle refers to a non existant soundbuffer this method will return false and the Yagl::SoundSource will not be able to play a sound as not audio data source is attached to it. Otherwise the Yagl::SoundSource will play the audio data of the specified buffer_handle on the next call to Yagl::SoundSource::play(). This method will most likely be chosen to let the soundsource play a certain sample at a specific event. Imagine some kind of first person shooter where an enemy has a Yagl::SoundSource attached to it. In case the enemy fires the weapon one will set the Yagl::SoundSource's attached soundbuffer to a sample that holds the audio data for a gunshot. When the enemy dies you most likely want to play a dying sound and so on.

To play a sound you can use:

```
void Yagl::SoundSource::play( )
```

if the soundbuffer attached to the Yagl::SoundSource is valid the SoundSource will play the content of the buffer at it's position with the attributes previously set. If the Yagl::SoundSource was already playing this call will have no effect.

There's another method that allows you to play a Yagl::SoundSource looped:

```
void Yagl::SoundSource::playLooped( )
```

This is working the same as Yagl::SoundSource::play expect that it will play the sound looped until the Yagl::SoundSource is stopped.

Stopping a Yagl::SoundSource can be achieved via:

```
void Yagl::SoundSource::stop( )
```

this will stop the Yagl::SoundSource if it was playing. Otherwise this will have no effect.

Pausing a Yagl::SoundSource is done by calling:

```
void Yagl::SoundSource::pause( )
```

In case the Yagl::SoundSource was playing this will pause it otherwise the call is ignored. To unpause the Yagl::SoundSource invoke Yagl::SoundSource::play() again.

To query the current playback state of a Yagl::SoundSource you can use:

```
bool Yagl::SoundSource::isPlaying( )
bool Yagl::SoundSource::isStopped( )
bool Yagl::SoundSource::isPaused( )
```

each of the methods will report wheter the Yagl::SoundSource is in the state in question by returning true or false.

The Yagl::SoundSource class also provides the user with methods to set and get Attributes related to the playback:
the first method allows you to set the volume of the Yagl::SoundSource by passing the **gain** that has to be between 0.0 and 1.0 (full volume). The **pitch** of the Yagl::SoundSource can be set via the second method.

```
void Yagl::SoundSource::setGain( float gain )
void Yagl::SoundSource::setPitch( float pitch )
void Yagl::SoundSource::setRollOff( float roll_off )
```

The pitch is a factor multiplied with the current frequency of the audiodata and allows you to slow down or speed up the playback speed. 1.0 stands for normal playback speed, values smaller 1.0 will slow the playback down, values bigger than 1.0 will speed it up. Finally the thrid method allows you to set the **roll_off** which is a factor indicating what effect the distance between the soundsource and the listener should have on the audio data. A value of 1.0 will reflect the conditions you find in real life, a value of 0.0 means that the distance has no effect on the audio playback (the gain stays the same no matter the distance).

to retrieve the current values of this attributes use the following methods

```
float Yagl::SoundSource::setGain( )
float Yagl::SoundSource::setPitch( )
float Yagl::SoundSource::setRollOff( )
```

The Yagl::SoundSource class also allows you to set and get the Position and velocity of the soundsource.

Setting those attributes is done via:

```
void Yagl::SoundSource::setPosition( float x, float y, float z )
void Yagl::SoundSource::setVelocity( float x, float y, float z )
```

x, **y** and **z** represent the vectorial components of the position and velocity respectively.

To retrieve the position and velocity use:

```
void Yagl::SoundSource::getPosition( float *x, float *y, float *z )
void Yagl::SoundSource::getVelocity( float *x, float *y, float *z )
```

x, **y** and **z** again represent the vectorial components of the position and velocity of the Yagl::Soundsource this time however the values get stored in the passed pointers. Note that in future releases this method might change and use references instead of pointers.

Finally a Yagl::SoundSource can be positioned relative to the Listener via:

```
void Yagl::SoundSource::setRelativeToListener( bool relative )
```

relative defines wheter or not the source is relative to the listener. Passing true means that the position is given relative to the Listener, passing false means that the position of the Yagl::SoundSource is measured relative to the 3D space's origin.

To check wheter a Yagl::SoundSource is positioned relative to the Listener use:

```
bool Yagl::SoundSource::isRelativeToListener( )
```

if this method returns null then the Yagl::SoundSource is positioned relative to the Listener, otherwise it is not.

Here's a small exmaple showing how to use a soundsource and manipulate it while it plays:

```
Yagl::SoundSource *source = 0;
unsigned int buffer_handle = 0;

if( !Yagl::getSfxDevice().createSoundBufferFromFile( ''ping.wav'',
&buffer_handle )
{
   cout << ''couldn't create soundbuffer from file ping.wav\n'';
   return 0;
}

source = Yagl::getSfxDevice().createSoundSource( buffer_handle );
if( source == 0 )
{
   cout << ''couldn't create soundsource\n'';
   return 0;
}

source->setGain( 0.5f );
source->setPosition( -1.0f, 0.0f, 0.0f );
source->play( );

while( source->isPlaying( ) );
```

6.5 Soundstreams

Soundbuffers and Soundstreams are only suited for small samples due to the memory footprint bigger soundfiles have. Yagl allows the user to use a file as an audio data source to overcome this problem. The class representing an object in 3D space that streams from a file is called Yagl::SoundStream. It is equal in functionality to the Yagl::SoundSource except that it works with files instead of soundbuffers and it does not allow to play back looped (yet).

to create a Yagl::SoundStream one has to invoke:

```
Yagl::SoundStream* Yagl::SfxDevice::createSoundStream( Yagl::String filename )
```

filename is the name of the file the Yagl::SoundStream should stream from. The method will return a valid pointer to a Yagl::SoundStream on success or a null pointer on failure. Note that this Yagl::SoundStream is managed wihin the Yagl::SfxDevice and will be destroyed implicitely at a call to Yagl::SfxDevice::deinitialize() or at program exit. The concept is similar to Yagl::GfxDevice and Yagl::GfxSurfaces. Also, the current signature of the method above might change in the next release of Yagl so it is not necessary to provide a filename. The file used by the Yagl::SoundStream can be set after creation too so this is a bit of a bad design. All the Yagl::SoundStream's attributes are set to default values. The position will be set to (0, 0, 0) the same applies to the velocity. the gain, pitch and roll off factor are all set to 1.0 and the Yagl::SoundStream will be relative to the coordinate system's origin rather than to the listener.

To destroy a specific Yagl::SoundStream one can use:

```
void Yagl::SfxDevice::destroySoundStream( Yagl::SoundStream* stream )
```

this will stop the soundstream from playing and free up the channel it uses (see more on this in chapter 6.6). **stream** is the pointer to a previously created Yagl::SoundStream

to destroy all Yagl::SoundStreams currently in the system use:

```
void Yagl::SfxDevice::destroyAllSoundStreams( )
```

this will stop all the soundstreams from playing and free up the channels assoicated with them.

To set the file the Yagl::SoundStream should stream from use:

```
bool Yagl::SoundStream::setStreamedFile( Yagl::String filename )
```

filename specifies the file to be streamed from. The same formats are supported as are with soundbuffers. If the Yagl::SoundStream was already assigned a file and playing then it will be stopped, the stream will be closed and the new stream will be set. If the stream could not be set, for example when the format of the file is unknown then this will return false otherwise true is returned.

To start the Yagl::SoundStream you can use:

```
void Yagl::SoundStream::play( )
```

if the a valid file is set as the streams source it will play the stream at it's position with the attributes previously set. If the Yagl::SoundStream was already playing this call will have no effect.

Stopping a Yagl::SoundStream can be achieved via:

```
void Yagl::SoundStreams::stop( )
```

this will stop the Yagl::SoundStream and close the stream if it was playing. Otherwise this will have no effect. Note that a stopped stream needs to be reassigned a file to stream from.

Pausing a Yagl::SoundStream is done by calling:

```
void Yagl::SoundStream::pause( )
```

In case the Yagl::SoundStream was playing this will pause it otherwise the call is ignored. To unpause the Yagl::SoundStream invoke Yagl::SoundStream::play() again.

To query the current playback state of a Yagl::SoundStream you can use:

```
bool Yagl::SoundStream::isPlaying( )
bool Yagl::SoundStream::isStopped( )
bool Yagl::SoundStream::isPaused( )
```

each of the methods will report wheter the Yagl::SoundStream is in the state in question by returning true or false.

The Yagl::SoundStream class also provides the user with methods to set and get Attributes related to the playback:

```
void Yagl::SoundStream::setGain( float gain )
void Yagl::SoundStream::setPitch( float pitch )
void Yagl::SoundStream::setRollOff( float roll_off )
```

the first method allows you to set the volume of the Yagl::SoundStream by passing the **gain** that has to be between 0.0 and 1.0 (full volume). The **pitch** of the Yagl::SoundStream can be set via the second method. The pitch is a factor multiplied with the current frequency of the audiodata and allows you to slow down or speed up the playback speed. 1.0 stands for normal playback speed, values smaller 1.0 will slow the playback down, values bigger than 1.0 will speed it up. Finally the third method allows you to set the **roll_off** which is a factor indicating what effect the distance between the sound stream and the listener should have on the audio data. A value of 1.0 will reflect the conditions you find in real life, a value of 0.0 means that the distance has no effect on the audio playback (the gain stays the same no matter the distance).

to retrieve the current values of this attributes use the following methods

```
float Yagl::SoundStream::setGain( )
float Yagl::SoundStream::setPitch( )
float Yagl::SoundStream::setRollOff( )
```

The Yagl::SoundStream class also allows you to set and get the Position and velocity of the soundstream.

Setting those attributes is done via:

```
void Yagl::SoundStream::setPosition( float x, float y, float z )
void Yagl::SoundStream::setVelocity( float x, float y, float z )
```

x, **y** and **z** represent the vectorial components of the position and velocity respectively.

To retrieve the position and velocity use:

```
void Yagl::SoundStream::getPosition( float *x, float *y, float *z )
void Yagl::SoundStream::getVelocity( float *x, float *y, float *z )
```

x, **y** and **z** again represent the vectorial components of the position and velocity of the Yagl::SoundStream this time however the values get stored in the passed pointers. Note that in future releases this method might change and use references instead of pointers.

Finally a Yagl::SoundStream can be positioned relative to the Listener via:

```
void Yagl::SoundStream::setRelativeToListener( bool relative )
```

relative defines wheter or not the source is relative to the listener. Passing true means that the position is given relative to the Listener, passing false means that the position of the Yagl::SoundStream is measured relative to the 3D space's origin.

Here's a small example of how to playback a Yagl::SoundStream:

```
Yagl::SoundStream* stream = 0;

stream = Yagl::getSfxDevice().createSoundStream( ''mybackgroundmusic.ogg'' );
if( stream == 0 )
{
   cout << ''couldn't create soundstream from file\n'';
}

stream->setGain( 0.5 )
stream->setPosition( 1.0, 0.0, 0.0 )
stream->play( )

while( stream->isPlaying() );
```

6.6 Notes on the OpenAL backend

Yagl's sound module is based on OpenAL, a free 3D sound library very similar to OpenGL. OpenAL is directly using the hardware, which also has some drawbacks. The number of sounds that can be played in parallel is highly dependant on the hardware used, some cards support only 16 so called channels other might have 32 or 64. Yagl's sound module therefor tries to minimize the use of those channels as much as possible. Soundbuffers and SoundSources that are played back will temporarily receive a free channel while they are played back which will get freed immediatly after the sound is done playing. This guarantees that no channel will ever be blocked when a sound is not playing. Soundstreams get a permanent channel assigned. This means that if you have say 2 soundstreams in your application they will own 2 channels on your hardware that can not be used for anything else other than the soundstreams.

It is therefor recommended to use as little soundstreams as possible to increase the number of channels available for playing back soundbuffers and soundsources. This should not really be a problem in any situation as you mostly will only have on stream for you background music.

6.7 Cleaning up

Yagl's sound module does not need explicit cleanup. As it keeps track of all allocated resources it can and will destroy all soundbuffers, Yagl::SoundSources and Yagl::SoundStreams at program exit.