CX

Generated by Doxygen 1.8.6

Tue Apr 29 2014 20:39:08

Contents

1 Main Page

ofxCX (aka the C++ Experiment System; hereafter referred to as CX) is a "total conversion mod" for openFrameworks (often abbreviated oF) that is designed to be used used for creating psychology experiments. OpenFrameworks and CX are based on C++, which is a very good programming languange for anything requiring a high degree of timing precision. OpenFrameworks and CX are both free and open source, distributed under the MIT license.

One of the features that CX has is the ability to run without a substantial installation process. When a CX program is

compiled and linked, the resulting artifact is an executable binary that can be run directly with needing another program to be installed. The collection of files needed to run a CX program is about 10 MB (possibly more, depending on stimuli). Installing the program just requires unzipping a file.

1.1 Installation

In order to use CX, you must have openFrameworks installed. Currently only version 0.8.0 of openFrameworks is supported by CX, which you can download from this page: http://openframeworks.cc/download/older.html The main openFrameworks download page (http://openframeworks.cc/download/) has information about how to install openFrameworks for use with some of the more popular development environments.

Once you have installed openFrameworks, you can install CX by putting the contents of the CX repository into a subdirectory under OFDIR/addons (typically OFDIR/addons/ofxCX), where OFDIR is where you put openFramworks when you installed it. To use CX in a project, use the openFrameworks project generator and select ofxCX as an addon (see the instructions for using the Examples and Tutorials for information).

1.2 System Requirements

openFrameworks works on a wide variety of hardware and software, some of which are not supported by CX. CX works on computers with certain versions of Windows, Linux, or OSx operating systems. Windows 7 and XP are both supported.

As far as hardware is concerned, the minimum requirements for CX are very low. However, if your video card is too old, you won't be able to use some types of rendering. Having a video card that supports OpenGL version 3.2 at least is good, although older ones will work, potentially with reduced functionality. Also, a 2+ core CPU helps with some things and is generally a good idea for psychology experiments, because one core can be hogged by CX and the operating system can use the other core for other things. Basically, use a computer made after 2010 and you will have no worries whatsoever. However, CX has been found to work with reduced functionality on computers from the mid 90's, so there is that option, although I cannot make any guarantees that it will work on any given computer of that vintage.

1.3 Examples and Tutorials

There are several examples of how to use CX. The example files can be found in the CX directory (see Installation) in subfolders with names beginning with "example-". Some of the examples are on a specific topic and others are sample experiments that integrate together different features of CX.

In order to use the examples and tutorials, do the following:

1. Use the oF project generator (OFDIR/projectGenerator/projectGeneratorSimple.exe) to create a new project that uses the ofxCX addon. The project generator asks you what to name your project, where to put it (defaults to OFDIR/apps/myApps/), and has the option of selecting addons. Click the "addons" button and check to box next to ofxCX. If ofxCX does not appear in the list of addons, you probably didn't put the ofxCX directory in the right place.

2. Go to the newly-created project directory (that you chose when creating the project in step 1) and go into the src subdirectory.

3. Delete all of the files in the src directory (main.cpp, testApp.h, and testApp.cpp).

4. Copy the example .cpp file into this src directory.

5. If the example has a data folder, copy the contents of that folder into yourProjectDirectory/bin/data. The bin/data folder probably won't exist at this point. You can create it.

6. This step depends on your compiler, but you'll need to tell it to use the example source file that you copied in step 4 when it compiles the project (and possibly to specifically not use the files you deleted from the src directory in step 3).
7. Compile and run the project.

Tutorials:

- soundBuffer - Tutorial covering a number of things that you can do with CX_SoundObjects, including loading sound files, combining sounds, and playing them.
- modularSynth - This tutorial demonstrates a number of ways to generate auditory stimuli using the synthesizer modules in the CX::Synth namespace.
- dataFrame - Tutorial covering use of CX_DataFrame, which is a container for storing data of various types that is collected in an experiment.
- logging - Tutoral explaining how the error logging system of CX works and how you can use it in your experiments.
- animation - A simple example of a simple way to draw moving things in CX. Also includes some mouse input handling: cursor movement, clicks, and scroll wheel activity.

Experiments:

- changeDetection - A very straightforward change-detection task demonstrating some of the features of CX like presentation of time-locked stimuli, keyboard response collection, and use of the CX_RandomNumberGenerator. There is also an advanced version of the changeDetection task that shows how to do data storage and output with a CX_DataFrame and how to use a custom coordinate system with visual stimuli so that you don't have to work in pixels.
- nBack - Demonstrates advanced use of CX_SlidePresenter in the implementation of an N-Back task. An advanced version of this example contrasts two methods of rendering stimuli with a CX_SlidePresenter, demonstrating the advantages of each.

Misc.:

- helloWorld - A very basic getting started program.
- renderingTest - Includes several examples of how to draw stuff using ofFbo (a kind of offscreen buffer), ofImage (for opening image files: .png, .jpg, etc.), a variety of basic oF drawing functions (ofCircle, ofRect, ofTriangle, etc.), and a number of CX drawing functions from the CX::Draw namespace that supplement openFramework's drawing capabilities.

1.4 Creating a Blank Experiment

To create your first experiment, follow the steps for using the examples in Examples and Tutorials up to and including step 3. Then create an empty .cpp file in the source directory of the project folder you made. In the file, you will need to include CX_EntryPoint.h and define runExperiment, like in the example below:

```
#include "CX_EntryPoint.h"

void runExperiment (void) {
    //Do everything you need to do for your experiment
}
```

That's all you need to do to get started. You should look at the Examples and Tutorials in order to learn more about how CX works. You should start with the helloWorld example.

Topics

The best way to get an overview of how CX works is to look at the Examples and Tutorials.

- To learn about presenting visual stimuli, go to the Video page or see the renderingTest or animation examples or the nBack or changeDetection example experiments.
- To learn about playing, recording, and generating sounds, go to the Sound page or see the soundObject or modularSynth examples.
- To learn how to store and output experiment data, see the Data page or see the dataFrame example.
- To learn about random number generation, see the Randomization page.
- To learn about how CX logs errors and other runtime information, see the Error Logging page.

You can look at the Modules page to see the other modules that CX has.

2 Blocking Code

Blocking code is code that either takes a long time to complete or that waits until some event occurs before allowing code execution to continue. An example of blocking code that waits is

```
do {
    Input.pollEvents();
} while (Input.Keyboard.availbleEvents() == 0);
```

This code waits until the keyboard has been used in some way. No code past it can be executed until the keyboard is used, which could take a long time. Any code that blocks while waiting for a human to do something is blocking.

An example of blocking code that takes a long time (or at least could take a long time) is

```
vector<double> d = CX::Util::sequence<double>(0, 1000000, .033);
```

which requires the allocation of about 300 MB of RAM. This code doesn't wait for anything to happen, it just takes a long time to execute.

Blocking code is potentially harmful because it prevents some parts of CX from working in some situations. It is not a cardinal sin and there are times when using blocking code is acceptable. However, blocking code should not be used when trying to present stimuli or when responses are being made. The reason for this is that CX expects to be able to repeatedly check information related to stimulus presentation and input at very short intervals (at least every millisecond), but that cannot happen if a piece of code is blocking. There is of course an exception to the "no blocking while waiting for responses" rule, which is when your blocking code is doing nothing but waiting for a response and constantly polling for user input. For example, the following code waits until any response is made:

```
while(!Input.pollEvents())
    ;
//Process the inputs.
```

3 Framebuffers and Buffer Swapping

Somes pieces of terminology that come up a lot in the documentation for CX are framebuffer, front buffer, back buffer, and swap buffers. What exactly are these things?

A framebuffer is fairly easy to explain in the rough by example. The contents of the screen of a computer are stored in a framebuffer. A framebuffer is essentially a rectangle of pixels where each pixel can be set to display any color. Framebuffers do not always have the same number of pixels as the screen: you can have framebuffers that are smaller or larger than the size of the screen. Framebuffers larger than the screen don't really do much for you as you cannot fit the whole thing on the screen. In CX, framebuffers are typically worked with through the abstraction of an ofFbo.

There are two special framebuffers: The front buffer and the back buffer. These are created by OpenGL automatically as part of starting OpenGL. The size of these special framebuffers is functionally the same as the size of the window (or the whole screen, if in full screen mode). The front buffer contains what is shown on the screen. The back buffer is not presented on the screen, so it can be rendered to at any time without affecting what is visible on the screen. Typically, when you render stuff in CX, you call CX::CX_Display::beginDrawingToBackBuffer() and CX::CX_Display::endDrawing-ToBackBuffer() around whatever you are rendering. This causes drawing that happens between the two function calls to be rendered to the back buffer.

What you have rendered to the back buffer has no effect on what you see on screen until you swap the contents of the front and back buffers. This isn't always a true swap, in that that the back buffer does not end up with the contents of the front buffer in it. On many systems, the back buffer is copied to the front buffer and is itself unchanged. This swap can be done by using different functions of the CX_Display: swapBuffers(), swapBuffersInThread(), or setAutomaticSwapping(). These functions are not interchangable, so make sure you are using the right one for your application.

Vertical Synchronization

Vertical synchronization (Vsync) is the process by which the swaps of the front and back are synchronized to the refreshes of the monitor in order to prevent vertical tearing. Vertical tearing happens when one part of a scene is being drawn onto the monitor and a different scene is copied into the front buffer, causing parts of both scenes to be drawn at once. The "tearing" happens on the monitor where one scene abruptly becomes the other. In order to to Vsync, there must be some control over when the front buffer is drawn to. The ideal process might be that when the user requests a buffer swap the video card waits until the next vertical blank to swap the buffers. Unfortunately, what actually happens is implementation dependent, which makes writing software that will always work properly difficult.

One problem that I have observed is that even with Vsync enabled if there have been no buffer swaps for some time (several screen refresh periods), buffer swaps can happen more quickly than expected. For example, if the buffers have not been swapped for 2.5 refresh periods and a buffer swap is requested, the buffer swap function can return immediately, not waiting until 3 refresh periods have passed to queue to swap. One process that could explain this is if when the user requests a buffer swap, if at least one vertical blank has passed since the last buffer swap, the buffers are swapped immediately. This can cause problems if the surrounding code is expecting the buffer swap to wait until the next refresh has occured to return. One possible solution to this is, after a buffer swap has been requested, to tell OpenGL to wait until all ongoing processes have completed before continuing. This can be done with glFinish() and results in a kind of "software" Vsync, as opposed to the "hardware" Vsync that is done by OpenGL internally. Calling the buffer swap function and then glFinish() works sometimes, but it isn't perfect. On some systems, this will result in a wait of two frame periods before continuing (don't ask me why). On other systems, it works just fine. You can turn on hardware or software Vsync with CX::CX_Display::setVSync().

So how do you know if you are having problems with video presentation that are related to Vsync? Probably the easiest way is to use a feature of CX::CX_SlidePresenter to learn about the timing of your stimuli. CX::CX_SlidePresenter::print-LastPresentationInformation() provides a lot of timing information related to slide presentation so that you can check for errors easily. The errors can take the form of incorrect slide durations or frame counts (depending on presentation mode). If slides are consistently not started at the intended start time but the copy to the back buffer is happening in time, the most likely culprit is that something strange is going on with Vsync. If you are experiencing problems in windowed mode but not in full screen mode, you shouldn't worry. Vsync does not work properly in windowed mode in most modern operating systems due to the way in which they do window compositing.

CX provides some functionality to help deal with the Vsync annoyances. CX::CX_Display::setVSync() can turn "hardware" and "software" Vsync on or off independently so that if you experience problems you can try different solutions. You can test different combinations of Vsync using CX::CX_Display::testBufferSwapping(). You can also try different buffer swapping modes of CX_SlidePresenter (see CX::CX_SlidePresenter::Configuration::SwappingMode). One of the

swapping modes (MULTI_CORE) swaps the buffers every frame in a secondary thread which avoids issues that arise from occasional buffer swapping. However, this mode can really only be used effectively with a 2+ core CPU, so if you are working with old computers, this may not be for you.

Another option to help deal with Vsync issues is to force Vsync on or off in your video card driver. Modern AMD and Nvidia drivers allow you to force Vsync on or off for specific applications or globally, which seems to be more reliable than turning Vsync on or off in software. If you force Vsync to a setting in the video card driver, the "hardware" Vsync setting of CX::CX_Display::setVSync() will probably not do anything, but the software setting probably would (although it is not clear that you would want to have both hardware and software Vsync working at the same time).

4 Modules

5 Program Model

Program Flow

One of the foundational aspects of CX is the design of the overall program flow, which includes things such as how responses are collected, how stimuli are drawn to the screen, and other similar concepts. The best way to learn about program flow is to examine the examples. The examples cover most of the critical topics and introduce the major components of CX.

The most important thing to understand is that in CX, nothing happens that your code does not explicitly ask for, with the exception of a small amount of setup, which is discussed below (see Pre-experiment Setup). For example, CX does not magically collect and timestamp user responses for you. Your code must poll for user input in order to get timestamps for input. This is explained more in the input section. In CX, there is no code running in the background that makes everything work out for your experiment, you have to design your experiment in such a way that you are covering all of your bases. That said, CX is specifically designed to make doing that as easy and painless as possible, while still giving you as much control over your experiment as is reasonably possible.

Internals/Attributions

CX is not a monolithic entity. It is based on a huge amount of open source software written by many different authors over the years. Clearly, CX is based on openFrameworks, but openFramworks itself is based on many different libraries. Window handling, which involves creating a window that can be rendered to and receiving user input events from the operating system, is managed by GLFW (http://www.glfw.org/). The actual rendering is visual stimuli is done using OpenGL (http://www.opengl.org/), which is wrapped by several openFrameworks abstractions (e.g. ofGLProgrammableRenderer at a lower level, e.g. ofPath at the level at which a typical user would use).

Audio is processed in different ways depending on the type of audio player used. CX_SoundBufferPlayer and CX_SoundBufferRecorder wrap CX_SoundStream which wraps RtAudio (https://www.music.mcgill.ca/~gary/rtaudio/). If you are using ofSoundPlayer, depending on your operating system it might eventually use FMOD on Windows or OSx (http://www.fmod.org/; although the openFrameworks maintainers are considering moving away from FMOD) or OpenAL on Linux (http://en.wikipedia.org/wiki/OpenAL). However, you should check that this information is correct.

There are other libraries that are a part of openFrameworks that I am not as familiar with, including Poco (http://pocoproject.org/), which provides a variety of very useful utility functions and networking, FreeType (http://www.freetype.org/) which does font rendering, and many others.

CX would not have been possible without the existence of these high-quality open-source projects.

Overriding openFrameworks

Although CX is technically an addon to openFrameworks, there are a number of ways in which CX hijacks normal oF functionality in order to work better. As such, you cannot assume that all oF functionality is available to you.

Generally, drawing visual stimuli using oF classes and functions is fully supported. See the renderingTest example to see a pletora of ways to put things on the screen.

Audio output using ofSoundPlayer is supported, although no timing guarantees are made. Prefer CX::CX_SoundBuffer-Player.

The input events (e.g. ofEvents().mousePressed) technically work, but with two serious limitations. 1) The events only fire when CX_InputManager::pollEvents() is called (which internally calls glfwPollEvents() to actually kick off the events firing). 2) The standard oF events do not have timestamps, which limits their usefulness.

The following functions' behavior is superseded by functionality provided by CX_Display (see also CX::Instances::-Display): ofGetFrameNum() is replaced by CX_Display::getFrameNumber() ofGetLastFrameTime() is replaced by CX_-Display::getLastSwapTime()

The following functions do nothing: ofGetFrameRate(), ofSetFrameRate(), ofGetTargetFrameRate()

A variety of behaviors related to ofBaseApp do not function because CX is not based on a class derived from ofBaseApp nor does it use ofRunApp() to begin the program. For example, a standard oF app class should have steup(), update(), and draw() functions that will be called by oF during program execution. CX has a different model that does not force object-orientation (at least at some levels).

Pre-experiment Setup

There is very little that CX does without you asking for it. The one major exception is pre-experiment setup, in which a number of basic operations are performed in order to set up a platform on which the rest of the experiment can run. The most significant step is to open a window and set up the OpenGL rendering environment. The main pseudorandom number generator (CX::Instances::RNG) is seeded. The logging system is prepared for use. The main clock (CX::-Instances::Clock) is prepared for use.

Input Timing

The way user input is handled by CX is easily explained by giving the process of receiving a mouse click. Assume that a CX program is running in a window. The user clicks inside of the window. At this point, the operating system (or at least the windowing subsystem of the operating system, but I will choose to conflate them) detects that the click has occured and notes that the location of the click is within the window. The operating system then attempts to tell the program that a mouse click has occured. In order to be notified about input events like mouse click, the program has previously set up a message queue for incoming messages from the operating system. The OS puts the mouse event into the message queue. In order for the program to find out about the message it needs to check to message queue. This is what happens when CX::CX_InputManager::pollEvents() is called: The message queue is checked and all messages in the queue are processed, given timestamps, and routed to the next queue (e.g. the message queue in CX::CX_Mouse that is accessed with CX::CX_Mouse::availableEvents() and CX::CX_Mouse::getNextEvent()). The timestamps are not given by the operating system$*$, so if pollEvents is not called regularly, input events will be received and everything will appear to be working correctly, but the timestamps will be wrong.

Of course, the actual process extends all the way back to the input device itself. The user presses the button and the microcontroller in the input device senses that a button has been pressed. It places this button press event into its outgoing message queue. At the next polling interval (typically 1 ms), the USB host controller on the computer polls the device for messages, discovers that a message is waiting and copies the message to the computer. At some point, the operating system checks to see if the USB host controller has received messages from any devices. It discovers the message and moves the message into the message queue of the program. At each step in which the message moves from one message queue to the next, the data contained in the message likely changes a little. At the start in the

mouse, the message might just be "button 1 pressed". At the next step in the USB host controller, the message might be "input device 1 (type is mouse) button 1 pressed". Once the operating system gets the message it might be "mouse button 1 pressed while cursor at (367, 200) relative to clicked window". Eventually, the message gets into the message queue that users of CX work with, in CX::CX_Mouse, for example.

This process sounds very long and complicated, suggesting that it might take a long time to complete, throwing off timing data. That is true: Input timing data collected by CX is not veridical, there are invariably delays, including non-systematic delays. However, there are several steps in the process that no experiment software can get around, so the problems with timing data are not unique to CX. It might be possible to write a custom driver for the mouse or keyboard that allows the software to bypass the operating system's message queue, but it is very difficult to avoid the USB hardware delays, which can be on the order of milliseconds for many kinds of standard input devices. The next layer of the problem is that we are really interested in response time to a specific stimulus, but the time at which the stimulus was actually presented may be misreported by audio or video hardware/software, so even if the response timestamp had no error whatsoever, when it is compared with the stimulus presentation time, the respones latency would be wrong due to errors in measures stimulus presentation time. Based on this large set of problems with collecting accurate response latency data, it is my firm belief that the only way to accurately measure response latency is with a button box that measures actual stimulus onset time with a light or sound sensor and also measures the time of a button press or other response. If you don't use such a system, the expectation is that you simply allow any error in response latencies to be dealt with statistically. Typically, any systematic error in response times will be subtracted out when conditions are compared with one another. Any random error will simply slightly inflate the estimated variance, but probably not to any meaningful extent.

If you would like to learn more about the internals of how input is handled in CX, you can see how GLFW and openFramworks manage input by examining the source code in the respective repositories.

∗Technically, on Windows the messages that are given to a program do have a timestamp. However, the documentation doesn't actually say what the timestamp represents. My searching turns up the suggestion that it is a timestamp in milliseconds from system boot, but that the timestamp is set using the GetTickCount function, which typically has worse than 10 ms precision. This makes the timestamp attached to the message of very little value. See this page for documentation of what information comes with a Windows message: http://msdn.microsoft.com/en-us/library/windows/desktop/ms644958%28v=vs.85%29.aspx. The only page on which I actually found a definition of what the time member stores is this page http://msdn.microsoft.com/en-us/library/aa929818.aspx, which gives information pertaining to Windows Mobile 6.5, which is an obsolete smartphone operating system.

Stimulus Timing

Although the kinds of error introduced into response time data can often be dealt with statistically, errors in stimulus presentation can be more serious. For example, if a visual stimulus is systematically presented for an extra frame throughout an experiment, then the method of the experiment has been altered without the experimenter learning about the alteration. Even if the extra frame does not always happen, on average participants are seeing more of that stimulus than they should be. An error on the magnitude of an extra frame is nearly impossible to detect by eye in most cases, so it is important that there is some way to detect errors in stimulus presentation. The primary method of presenting time-locked visual stimuli is the CX_SlidePresenter. It has built in error-detection features that pick up on certain kinds of errors. Information about presentation errors can be found by using CX::CX_SlidePresenter::checkForPresentationErrors().

Although it is nice to be made aware of errors when they occur, it is better to not have the errors happen in the first place. For this reason, stimulus presentation in CX is designed around avoiding errors. For visual stimuli, the CX_SlidePresenter provides a very easy-to-use way to present visual stimuli. Because the interface is so simple, user error is minimized. The backend code of the CX_SlidePresenter is designed to minimize the potential for timing errors by carefully tracking the passage of time, monitor refreshes, and timing of stimulus rendering.

On the audio front, CX provides the CX_SoundBufferPlayer, which plays CX::CX_SoundBuffer "CX_SoundBuffers". If several sounds are to be presented in a time-locked sequence, playing the sounds individually at their intended onset time can result in unequal startup delays for each sound, but if all of the sounds are combined together into a single audio buffer this possibility is eliminated. CX_SoundObjects are designed to make combining multiple sound stimuli together

easy, which helps to prevent timing errors that could have otherwise occurred between sounds. CX also includes CX-_SoundStream, which provides a method for directly acessing and manipulating the contents of audio buffers that are received from or sent to audio hardware.

6 license

7 Module Index

7.1 Modules

Here is a list of all modules:

8 Namespace Index

8.1 Namespace List

Here is a list of all documented namespaces with brief descriptions:

9 Hierarchical Index

9.1 Class Hierarchy

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

10 Class Index

10.1 Class List

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

11 File Index

11.1 File List

Here is a list of all documented files with brief descriptions:

12 Module Documentation

12.1 Data

Classes

- class CX::CX_DataFrame
- class CX::CX_DataFrameColumn

- class CX::CX_DataFrameRow
- class CX::CX_DataFrameCell

12.1.1 Detailed Description

This module is related to storing experimental data. CX_DataFrame is the most important class in this module.

12.2 Entry Point

Functions

- void runExperiment (void)

Variables

- CX_Display CX::Instances::Display
- CX_InputManager CX::Instances::Input
- CX_Logger CX::Instances::Log
- CX_RandomNumberGenerator CX::Instances::RNG

12.2.1 Detailed Description

The entry point provides access to a few instances of classes that can be used by user code. It also provides declarations (but not definitions) of a function which the user should define (see runExperiment()).

12.2.2 Function Documentation

12.2.2.1 runExperiment (void)

The user code should define a function with this name and type signature (takes no arguments and returns nothing). This function will be called once setup is done for CX. When runExperiment returns, the program will exit.

```
void runExperiment (void) {
    //Do your experiment.

    return; //Return when done to exit the program. You don't have to explicity return; you can just fall
       off the end of the function.
        //You can alternately call std::exit() or ofExit() at any point.
}
```

12.2.3 Variable Documentation

12.2.3.1 CX::CX_Display CX::Instances::Display

An instance of CX::CX_Display that is lightly hooked into the CX backend. setup() is called for Display before runExperiment() is called.

12.2.3.2 CX::CX_InputManager CX::Instances::Input

An instance of CX_InputManager that is very lightly hooked into the CX backend.

12.2.3.3 CX_Logger CX::Instances::Log

This is an instance of CX::CX_Logger that is hooked into the CX backend. All log messages generated by CX and openFrameworks go through this instance.

12.2.3.4 CX::CX_RandomNumberGenerator CX::Instances::RNG

An instance of CX_RandomNumberGenerator that is (lightly) hooked into the CX backend.

12.3 Error Logging

Classes

- class CX::CX_Logger

Enumerations

- enum CX::CX_LogLevel {
  LOG_ALL, **LOG_VERBOSE**, **LOG_NOTICE**, **LOG_WARNING**,
  LOG_ERROR, **LOG_FATAL_ERROR**, **LOG_NONE** }

12.3.1 Detailed Description

12.3.2 Enumeration Type Documentation

12.3.2.1 enum CX::CX_LogLevel [strong]

Log levels for log messages. Depending on the log level chosen, the name of the level will be printed before the message. Depending on the settings set using level(), levelForConsole(), or levelForFile(), if the log level of a message is below the level set for the module or logging target it will not be printed. For example, if LOG_ERROR is the level for the console and LOG_NOTICE is the level for the module "test", then messages logged to the "test" module will be completely ignored if at verbose level (because of the module setting) and will not be printed to the console if they are below the level of an error (because of the console setting).

12.4 Input Devices

Classes

- class CX::CX_InputManager
- class CX::CX_Joystick
- class CX::CX_Keyboard
- class CX::CX_Mouse

12.4.1 Detailed Description

There are a number of different classes that together perform the input handling functions of CX. Start by looking at CX::CX_InputManager and the instance of that class that is created for you: CX::Instances::Input.

For interfacing with serial ports, use ofSerial (http://www.openframeworks.cc/documentation/communication/ofSerial.html).

See Also

CX::CX_InputManager for the primary interface to input devices.
CX::CX_Keyboard for keyboard specific information.
CX::CX_Mouse for mouse specific information.
CX::CX_Joystick for joystick specific information.

12.5 Randomization

Classes

- class CX::CX_RandomNumberGenerator

12.5.1 Detailed Description

This module provides a class that is used for random number generation.

12.6 Sound

Namespaces

- CX::Synth

Classes

- class CX::CX_SoundBufferPlayer
- class CX::CX_SoundBufferRecorder
- class CX::CX_SoundBuffer
- class CX::CX_SoundStream

12.6.1 Detailed Description

There are a few different ways to deal with sounds in CX. The thing that most people want to do is to play sounds, which is done with the CX_SoundBufferPlayer. See the soundBuffer tutorial for information on how to do that.

If you want to record sound, use the CX_SoundBufferRecorder.

If you want to generate sound stimuli through sound synthesis, see the CX::Synth namespace and modularSynth tutorial.

Finally, if you want to have direct control of the data going to and from a sound device, see CX_SoundStream.

12.7 Timing

Classes

- class CX::CX_Clock
- class CX::CX_BaseClockImplementation
- class CX::CX_Time_t< T >
- class CX::Util::CX_LapTimer
- class CX::Util::CX_SegmentProfiler

Variables

- CX_Clock CX::Instances::Clock

12.7.1 Detailed Description

This module provides methods for timestamping events in experiments.

12.7.2 Variable Documentation

12.7.2.1 CX_Clock CX::Instances::Clock

An instance of CX::CX_Clock that is hooked into the CX backend. Anything in CX that requires timing information uses this instance. You should use this instance in your code and not make your own instance of CX_Clock. You should never need another instance. You should never use another instance, as the experiment start times will not agree between instances.

12.8 Utility

Namespaces

- CX::Util

Classes

- class CX::Util::CX_TrialController
- class CX::Util::CX_DegreeToPixelConverter
- class CX::Util::CX_LengthToPixelConverter
- class CX::Util::CX_CoordinateConverter

12.8.1 Detailed Description

12.9 Video

Namespaces

- CX::Draw

Classes

- class CX::CX_Display
- class CX::CX_SlidePresenter

12.9.1 Detailed Description

This module is related to creating and presenting visual stimuli.

The CX::Draw namespace contains some more complex drawing functions. However, almost all of the drawing of stimuli is done using openFrameworks functions. A lot of the common functions can be found in ofGraphics.h (http://www.openframeworks.cc/documentation/graphics/ofGraphics.html), but there are a lot of other ways to draw stimuli: see the graphics and 3d sections if this page: http://www.openframeworks.cc/documentation/.

13 Namespace Documentation

13.1 CX Namespace Reference

Namespaces

- Algo
- Draw
- Instances
- Private
- Synth
- Util

Classes

- class CX_Clock
- class CX_BaseClockImplementation
- class CX_DataFrame
- class CX_DataFrameColumn
- class CX_DataFrameRow
- class CX_DataFrameCell
- class CX_Display
- struct CX_WindowConfiguration_t
- class CX_InputManager
- class CX_Joystick
- class CX_Keyboard
- class CX_Logger
- class CX_Mouse
- class CX_RandomNumberGenerator
- class CX_SlidePresenter
- class CX_SoundBuffer
- class CX_SoundBufferPlayer
- class CX_SoundBufferRecorder
- class CX_SoundStream
- class CX_Time_t

Typedefs

- typedef int64_t **CX_RandomInt_t**
- typedef CX_Time_t< std::ratio< 3600, 1 > > **CX_Hours**
- typedef CX_Time_t< std::ratio< 60, 1 > > **CX_Minutes**
- typedef CX_Time_t< std::ratio< 1, 1 > > **CX_Seconds**
- typedef CX_Time_t< std::ratio< 1, 1000 > > **CX_Millis**
- typedef CX_Time_t< std::ratio< 1, 1000000 > > **CX_Micros**
- typedef CX_Time_t< std::ratio< 1, 1000000000 > > **CX_Nanos**

Enumerations

- enum CX_LogLevel {
  LOG_ALL, **LOG_VERBOSE**, **LOG_NOTICE**, **LOG_WARNING**,
  LOG_ERROR, **LOG_FATAL_ERROR**, **LOG_NONE** }
- enum **CX_MouseButtons** : int { **LEFT_MOUSE** = OF_MOUSE_BUTTON_LEFT, **MIDDLE_MOUSE** = OF_MOUSE_BUTTON_MIDDLE, **RIGHT_MOUSE** = OF_MOUSE_BUTTON_RIGHT }

Functions

- std::ostream & **operator**<< (std::ostream &os, const CX_DataFrameCell &cell)
- void reopenWindow (CX_WindowConfiguration_t config)
- std::ostream & **operator**<< (std::ostream &os, const CX_Joystick::Event &ev)
- std::istream & **operator**>> (std::istream &is, CX_Joystick::Event &ev)
- std::ostream & **operator**<< (std::ostream &os, const CX_Keyboard::Event &ev)
- std::istream & **operator**>> (std::istream &is, CX_Keyboard::Event &ev)
- std::ostream & **operator**<< (std::ostream &os, const CX_Mouse::Event &ev)
- std::istream & **operator**>> (std::istream &is, CX_Mouse::Event &ev)
- template<typename TimeUnit >
  std::ostream & operator<< (std::ostream &os, const CX_Time_t< TimeUnit > &t)
- template<typename TimeUnit >
  std::istream & **operator**>> (std::istream &is, CX_Time_t< TimeUnit > &t)

13.1.1 Detailed Description

This namespace contains all of the symbols related to CX, except for runExperiment(), which is not namespace qualified.

13.1.2 Function Documentation

13.1.2.1 template<typename TimeUnit > std::ostream& CX::operator<< (std::ostream & *os*, const CX_Time_t< TimeUnit > & *t*)

This is a standard stream operator for converting a CX_Time_t to a std::ostream.

Note

If operator<< and operator>> are used to convert to/from a stream representation, you MUST use the same time type on both ends of the conversion.

13.1.2.2 void CX::reopenWindow (CX_WindowConfiguration_t *config*)

This function opens a GLFW window that can be rendered to. If another window was already open by the application at the time this is called, that window will be closed. This is useful if you want to control some of the parameters of the window that cannot be changed after the window has been opened.

13.2 CX::Algo Namespace Reference

Classes

- class LatinSquare
- class BlockSampler

Functions

- template<typename T >
  std::vector< T > generateSeparatedValues (int count, double minDistance, std::function< double(T, T)> distanceFunction, std::function< T(void)> randomDeviate, int maxSequentialFailures=200)
- template<typename T >
  std::vector< std::vector< T > > fullyCross (std::vector< std::vector< T > > factors)

13.2.1 Detailed Description

This namespace contains a few complex algorithms that can be difficult to properly implement or are very experiment-specific.

13.2.2 Function Documentation

13.2.2.1 template<typename T > std::vector< std::vector< T > > CX::Algo::fullyCross (std::vector< std::vector< T > > *factors*)

This function fully crosses the levels of the factors of a design. For example, for a 2X3 design, it would give you all 6 combinations of the levels of the design.

Parameters

factors	A vector of factors, each factor being a vector containing all the levels of that factor.

Returns

A vector of crossed factor levels. It's length is equal to the product of the levels of the factors. The length of each "row" is equal to the number of factors.

Example use:

```
std::vector< std::vector<int> > levels(2); //Two factors
levels[0].push_back(1); //The first factor has two levels (1 and 2)
levels[0].push_back(2);
levels[1].push_back(3); //The second factor has three levels (3, 4, and 5)
levels[1].push_back(4);
levels[1].push_back(5);
auto crossed = fullyCross(levels);
```

crossed should contain a vector with six subvectors with the contents:

```
{ {1,3}, {1,4}, {1,5}, {2,3}, {2,4}, {2,5} }
```

where

```
crossed[3][0] == 2
crossed[3][1] == 3
crossed[0][1] == 3
```

13.2.2.2 template<typename T > std::vector< T > CX::Algo::generateSeparatedValues (int *count*, double *minDistance*, std::function< double(T, T)> *distanceFunction*, std::function< T(void)> *randomDeviate*, int *maxSequentialFailures* = 200)

This algorithm is designed to deal with the situation in which a number of random values must be generated that are each at least some distance from every other random value. This is a very generic implementation of this algorithm.

It works by taking pointers to two functions that work on whatever type of data you are using. The first function is a distance function: it returns the distance between two values of the type. You can define distance in whatever way you would like. The second function generates random values of the type.

Template Parameters

<T>	The type of data you are working with.

Parameters

count	The number of values you want to be generated.
minDistance	The minimum distance between any two values. This will be compared to the result of distanceFunction.
distanceFunction	A function that computes the distance, in whatever units you want, between two values of type T.
randomDeviate	A function that generates random values of type T.
maxSequentialFailures	The maximum number of times in a row that a newly-generated value is less than minDistance from at least one other value. This essentially makes sure that if it is not possible to generate a random value that is at least some distance from the others, the algorithm will terminate.

Returns

A vector of values. If the function terminated prematurely due to maxSequentialFailures being reached, the returned vector will have 0 elements.

```
//This example function generates locCount points with both x and y values bounded by minimumValues and
       maximumValues that
//are at least minDistance pixels from each other.
std::vector<ofPoint> getObjectLocations(int locCount, double minDistance, ofPoint minimumValues, ofPoint
      maximumValues) {
    auto pointDistance = [](ofPoint a, ofPoint b) {
        return sqrt(pow(a.x - b.x, 2) + pow(a.y - b.y, 2));
    };

    auto randomPoint = [&]() {
        ofPoint rval;
        rval.x = RNG.randomInt(minimumValues.x, maximumValues.x);
        rval.y = RNG.randomInt(minimumValues.y, maximumValues.y);
        return rval;
    };

    return CX::Algo::generateSeparatedValues<ofPoint>(locCount, minDistance, pointDistance, randomPoint,
      1000);
}

//Call of example function
vector<ofPoint> v = getObjectLocations(5, 50, ofPoint(0, 0), ofPoint(400, 400));
```

13.3 CX::Draw Namespace Reference

Enumerations

- enum **LineCornerMode** { **OUTER_POINT**, **BEZIER_ARC**, **STRAIGHT_LINE**, **ARC** }

Functions

- ofPath squircleToPath (double radius, double amount)
- ofPath arrowToPath (float length, float headOffsets, float headSize, float lineWidth)
- std::vector< ofPoint > getStarVertices (unsigned int numberOfPoints, float innerRadius, float outerRadius, float rotationDeg)
- ofPath starToPath (unsigned int numberOfPoints, float innerRadius, float outerRadius)
- void star (ofPoint center, unsigned int numberOfPoints, float innerRadius, float outerRadius, ofColor fillColor, float rotationDeg)
- void centeredString (int x, int y, std::string s, ofTrueTypeFont &font)
- void centeredString (ofPoint center, std::string s, ofTrueTypeFont &font)

- ofPixels **greyscalePattern** (const CX_PatternProperties_t &properties)
- ofPixels **gaborToPixels** (const CX_GaborProperties_t &properties)
- ofTexture **gaborToTexture** (const CX_GaborProperties_t &properties)
- void **gabor** (ofPoint p, const CX_GaborProperties_t &properties)
- bool **isPointInRegion** (ofPoint p, ofPoint r1, ofPoint r2, float tolerance)
- bool **arePointsInLine** (ofPoint p1, ofPoint p2, ofPoint p3, float slopeTolerance=1e-3)
- ofPoint **findIntersectionOfLines** (LineSegment ls1, LineSegment ls2)
- std::vector< LineSegment > **getParallelLineSegments** (LineSegment ls, float distance)
- ofVec3f **getCornerOuterVector** (ofPoint p1, ofPoint p2, ofPoint p3, float vectorLength)
- ofPath **lines** (std::vector< ofPoint > points, ofColor color, float width, LineCornerMode cornerMode)
- void lines (std::vector< ofPoint > points, float lineWidth)
- void line (ofPoint p1, ofPoint p2, float width)
- void ring (ofPoint center, float radius, float width, unsigned int resolution)
- void arc (ofPoint center, float radiusX, float radiusY, float width, float angleBegin, float angleEnd, unsigned int resolution)
- void bezier (std::vector< ofPoint > controlPoints, float width, unsigned int resolution)
- void colorWheel (ofPoint center, vector< ofFloatColor > colors, float radius, float width, float angle)
- void colorArc (ofPoint center, vector< ofFloatColor > colors, float radiusX, float radiusY, float width, float angleBegin, float angleEnd)
- ofVbo **colorWheelToVbo** (ofPoint center, vector< ofFloatColor > colors, float radius, float width, float angle)
- ofVbo **colorArcToVbo** (ofPoint center, vector< ofFloatColor > colors, float radiusX, float radiusY, float width, float angleBegin, float angleEnd)
- std::vector< double > convertColors (std::string conversionFormula, double S1, double S2, double S3)
- ofFloatColor convertToRGB (std::string inputColorSpace, double S1, double S2, double S3)
- template<typename ofColorType >
  std::vector< ofColorType > getRGBSpectrum (unsigned int colorCount)
- ofVbo **colorWheelToVbo** (ofPoint center, std::vector< ofFloatColor > colors, float radius, float width, float angle)
- ofVbo **colorArcToVbo** (ofPoint center, std::vector< ofFloatColor > colors, float radiusX, float radiusY, float width, float angleBegin, float angleEnd)
- void **colorWheel** (ofPoint center, std::vector< ofFloatColor > colors, float radius, float width, float angle)
- void **colorArc** (ofPoint center, std::vector< ofFloatColor > colors, float radiusX, float radiusY, float width, float angleBegin, float angleEnd)

13.3.1 Detailed Description

This namespace contains functions for drawing certain complex stimuli. These functions are provided "as-is": If what they draw looks nice to you, great; however, there are no strong guarantees about what the output of the functions will look like.

13.3.2 Function Documentation

13.3.2.1 void CX::Draw::arc (ofPoint *center,* float *radiusX,* float *radiusY,* float *width,* float *angleBegin,* float *angleEnd,* unsigned int *resolution*)

Draw an arc around a central point. If radiusX and radiusY are equal, the arc will be like a section of a circle. If they are unequal, the arc will be a section of an ellipse.

Parameters

center	The point around which the arc will be drawn.
radiusX	The radius of the arc in the X-axis.
radiusY	The radius of the arc in the Y-axis.
width	The width of the arc, radially from the center.
angleBegin	The angle at which to begin the arc, in degrees.
angleEnd	The angle at which to end the arc, in degrees. If the arc goes in the "wrong" direction, try giving a negative value for `angleEnd`.
resolution	The resolution of the arc. The arc will be composed of `resolution` line segments.

Note

This uses an ofVbo internally. If VBOs are not supported by your video card, this may not work at all.

13.3.2.2 ofPath CX::Draw::arrowToPath (float *length,* float *headOffsets,* float *headSize,* float *lineWidth*)

Draws an arrow to an ofPath. The outline of the arrow is drawn with strokes, so you can have the path be filled to have a solid arrow, or you can use non-zero width strokes in order to have the outline of an arrow. The arrow points up by default but you can rotate it with ofPath::rotate().

Parameters

length	The length of the arrow in pixels.
headOffsets	The angle between the main arrow body and the two legs of the tip, in degrees.
headSize	The length of the legs of the head in pixels.
lineWidth	The width of the lines used to draw the arrow (i.e. the distance between parallel strokes).

Returns

An ofPath containing the arrow. The center of the arrow is at (0,0) in the ofPath.

13.3.2.3 void CX::Draw::bezier (std::vector< ofPoint > *controlPoints,* float *width,* unsigned int *resolution*)

Draws a bezier curve with an arbitrary number of control points. May become slow with a large number of control points. Uses de Casteljau's algorithm to calculate the curve points. See this awesome guide: `http://pomax.github.io/bezierinfo/`

Parameters

controlPoints	Control points for the bezier.
width	The width of the lines to be drawn. Uses CX::Draw::lines internally to draw the connecting lines.
resolution	Controls the approximation of the bezier curve. There will be `resolution` line segments drawn to complete the curve.

13.3.2.4 void CX::Draw::centeredString (int *x,* int *y,* std::string *s,* ofTrueTypeFont & *font*)

Equivalent to a call to CX::Draw::centeredString(ofPoint(x, y), s, font).

13.3.2.5 void CX::Draw::centeredString (ofPoint *center,* std::string *s,* ofTrueTypeFont & *font*)

Draws a string centered on a given location using the given font. Strings are normally drawn such that the x coordinate gives the left edge of the string and the y coordinate gives the line above which the letters will be drawn, where some characters (like y or g) can descend below the line.

Parameters

center	The coordinates of the center of the string.
s	The string to draw.
font	A font that has already been prepared for use.

13.3.2.6 void CX::Draw::colorArc (ofPoint *center,* vector< ofFloatColor > *colors,* float *radiusX,* float *radiusY,* float *width,* float *angleBegin,* float *angleEnd*)

Draws an arc with specified colors. The precision of the arc is controlled by how many colors are supplied.

Parameters

center	The center of the color wheel.
colors	The colors to use in the color arc.
radiusX	The radius of the color wheel in the X-axis.
radiusY	The radius of the color wheel in the Y-axis.
width	The width of the arc. The arc will extend half of the width in either direction from the radii.
angleBegin	The angle at which to begin the arc, in degrees.
angleEnd	The angle at which to end the arc, in degrees. If the arc goes in the "wrong" direction, try giving a negative value for `angleEnd`.

13.3.2.7 void CX::Draw::colorWheel (ofPoint *center,* vector< ofFloatColor > *colors,* float *radius,* float *width,* float *angle*)

Draws a color wheel with specified colors.

Parameters

center	The center of the color wheel.
colors	The colors to use in the color wheel.
radius	The radius of the color wheel.
width	The width of the color wheel. The color wheel will extend half of the width in either direction from the radius.
angle	The amount to rotate the color wheel.

13.3.2.8 std::vector< double > CX::Draw::convertColors (std::string *conversionFormula,* double *S1,* double *S2,* double *S3*)

Convert between two color spaces. This conversion uses this library internally: http://www.getreuer.-info/home/colorspace

Parameters

conversion-Formula	A formula of the format "SRC -> DEST", where SRC and DEST are valid color spaces. For example, if you wanted to convert from HSL to RGB, you would use "HSL -> RGB" as the formula. The whitespace is immaterial, but the arrow must exist (the arrow can point either direction). See this page for options for the color space: http://www.getreuer.-info/home/colorspace#TOC-MATLAB-Usage.

S1	Source coordinate 1. Corresponds to, e.g., R from RGB. S2 and S3 follow as expected.

Returns

An vector of length 3 containing the converted coordinates in the destination color space.

```
vector<double> hslValues = Draw::convertColors("XYZ -> HSL", .7, .4, .6);
hslValues[0]; //Access the hue value.
hslValues[2]; //Access the lightness value.
```

Note

The values returned by this function may not be in the allowed range for the destination color space. Make sure they are clamped to reasonable values if they are to be used directly.

See Also

CX::Draw::convertToRGB() is a convenience function for the most common conversion that will typically be done.

13.3.2.9 ofFloatColor CX::Draw::convertToRGB (std::string *inputColorSpace*, double *S1*, double *S2*, double *S3*)

This function converts from a color space to the RGB color space. This is convenient, because in order to draw stimuli with a color, you need to have the color in the RGB space.

Parameters

inputColorSpace	The color space to convert from. For example, if you wanted to convert from LAB coordinates, you would provde the string "LAB". See this page for more options for the color space: http://www.getreuer.info/home/colorspace#TOC-MATLAB-Usage (ignore the MATLAB title on that page).
S1	Source coordinate 1. Corresponds to, e.g., R from RGB. S2 and S3 follow as expected.

Returns

An ofFloatColor contaning the RGB coordinates.

```
//This code snippet draws an isluminant color wheel to the screen using color conversion from LAB to RGB.
void runExperiment(void) {

    vector<ofFloatColor> wheelColors(100);
    float L = 50; //Fix luminance value

    for (int i = 0; i < wheelColors.size(); i++) {
        //Take color values from a circle within the given luminance slice.
        float angle = (float)i / wheelColors.size() * 2 * PI;
        float A = sin(angle) * 30;
        float B = cos(angle) * 30;

        wheelColors[i] = Draw::convertToRGB("LAB", L, A, B); //Convert the L, A, and B
       components to the RGB color space.
    }

    Display.beginDrawingToBackBuffer();
    ofBackground(0);
    Draw::colorWheel(Display.getCenterOfDisplay(), wheelColors,
      200, 70, 0);
    Display.endDrawingToBackBuffer();
    Display.swapBuffers();

    Input.setup(true, false);
    while (!Input.pollEvents())
        ;
}
```

13.3.2.10 template<typename ofColorType > std::vector< ofColorType > CX::Draw::getRGBSpectrum (unsigned int *colorCount*)

Sample colors from the RGB spectrum with variable precision. Colors will be sampled beginning with red, continue through yellow, green, cyan, blue, violet, and almost, but not quite, back to red.

Template Parameters

ofColorType	An oF color type. One of: ofColor, ofFloatColor, or ofShortColor.

Parameters

colorCount	The number of colors to draw from the RGB spectrum, which will be rounded up to the next power of 6.

Returns

A vector containing the sampled colors with a number of colors equal to colorCount rounded up to the next power of 6.

13.3.2.11 std::vector< ofPoint > CX::Draw::getStarVertices (unsigned int *numberOfPoints*, float *innerRadius*, float *outerRadius*, float *rotationDeg*)

This function obtains the vertices needed to draw an N pointed star.

Parameters

numberOfPoints	The number of points in the star.
innerRadius	The distance from the center of the star at which the inner points of the star hit.
outerRadius	The distance from the center of the star to the outer points of the star.
rotationDeg	The number of degrees to rotate the star. 0 degrees has one point of the star pointing up. Positive values rotate the star counter-clockwise.

Returns

A vector of points defining the vertices needed to draw the star. There will be 2 $*$ numberOfPoints + 1 vertices with the last vertex equal to the first vertex. The vertices are centered on (0, 0).

13.3.2.12 void CX::Draw::line (ofPoint *p1*, ofPoint *p2*, float *width*)

This function draws a line from p1 to p2 with the given width.

Note

This function supersedes ofLine because the line width of the line drawn with ofLine cannot be set to a value greater than 1.

13.3.2.13 void CX::Draw::lines (std::vector< ofPoint > *points*, float *lineWidth*)

This function draws a series of line segments to connect the given points. At each point, the line segments are joined with a circle, which results in overdraw. As a result, this function does not work well with transparency.

Parameters

points	The points to connect with lines.
lineWidth	The width of the line.

Note

If the last point is the same as the first point, the final line segment junction will be joined with a circle.

13.3.2.14 void CX::Draw::ring (ofPoint *center,* float *radius,* float *width,* unsigned int *resolution*)

This function draws a ring, i.e. an unfilled circle. The filled area of the ring is between `radius + width/2` and `radius - width/2`.

Parameters

center	The center of the ring.
radius	The radius of the ring.
width	The radial width of the ring.
resolution	The ring will be approximated with a number of line segments, which is controlled with `resolution`.

Note

This function supersedes drawing rings with ofCircle with fill set to off because the line width of the unfilled circle cannot be set to a value greater than 1.

13.3.2.15 ofPath CX::Draw::squircleToPath (double *radius*, double *amount*)

This function draws an approximation of a squircle (http://en.wikipedia.org/wiki/Squircle) using Bezier curves. The squircle will be centered on (0,0) in the ofPath.

Parameters

radius	The radius of the largest circle that can be enclosed in the squircle.
amount	The "squircliness" of the squircle. The default (0.9) seems like a pretty good amount for a good approximation of a squircle, but different amounts can give different sorts of shapes.

Returns

An ofPath containing the squircle.

13.3.2.16 void CX::Draw::star (ofPoint *center*, unsigned int *numberOfPoints*, float *innerRadius*, float *outerRadius*, ofColor *fillColor*, float *rotationDeg*)

This draws an N-pointed star.

Parameters

center	The point at the center of the star.
numberOfPoints	The number of points in the star.
innerRadius	The distance from the center of the star to where the inner points of the star hit.
outerRadius	The distance from the center of the star to the outer points of the star.
fillColor	The color used to fill in the center of the star.
rotationDeg	The number of degrees to rotate the star. 0 degrees has one point of the star pointing up. Positive values rotate the star counter-clockwise.

13.3.2.17 ofPath CX::Draw::starToPath (unsigned int *numberOfPoints*, float *innerRadius*, float *outerRadius*)

This draws an N-pointed star to an ofPath. The star will be centered on (0,0) in the ofPath.

Parameters

numberOfPoints	The number of points in the star.
innerRadius	The distance from the center of the star at which the inner points of the star hit.

outerRadius	The distance from the center of the star to the outer points of the star.

Returns

An ofPath containing the star.

13.4 CX::Instances Namespace Reference

Variables

- CX_Clock Clock
- CX_Display Display
- CX_InputManager Input
- CX_Logger Log
- CX_RandomNumberGenerator RNG

13.4.1 Detailed Description

This namespace contains instances of some classes that are fundamental to the functioning of CX.

13.5 CX::Private Namespace Reference

Functions

- void **setupCX** (void)
- CX_Events & **getEvents** (void)
- void **learnOpenGLVersion** (void)
- CX_GLVersion **getOpenGLVersion** (void)
- CX_GLVersion **getGLSLVersion** (void)
- bool **glFenceSyncSupported** (void)
- bool **glVersionAtLeast** (int desiredMajor, int desiredMinor, int desiredRelease=0)
- int **glCompareVersions** (CX_GLVersion a, CX_GLVersion b)
- CX_GLVersion **getGLSLVersionFromGLVersion** (CX_GLVersion glVersion)
- template<typename tOut , typename tIn , typename resultT >
  resultT **convertTimeCount** (resultT countIn)
- template<>
  long long **convertTimeCount**< **std::nano, std::nano, long long** > (long long countIn)
- template<>
  long long **convertTimeCount**< **std::micro, std::micro, long long** > (long long countIn)
- template<>
  long long **convertTimeCount**< **std::milli, std::milli, long long** > (long long countIn)
- void **setMsaaSampleCount** (unsigned int count)

Variables

- GLFWwindow * **glfwContext** = NULL
- ofPtr< CX_AppWindow > **window**

13.5.1 Detailed Description

This namespace contains symbols that may be visible in user code but which should not be used by user code.

13.6 CX::Synth Namespace Reference

Classes

- class ModuleBase
- class AdditiveSynth
- class Adder
- class Clamper
- class Envelope
- class Filter
- class Mixer
- class Multiplier
- class Oscillator
- class Splitter
- class SoundBufferInput
- class StreamOutput
- class StereoStreamOutput
- class SoundBufferOutput
- class StereoSoundBufferOutput
- class FIRFilter

Functions

- double **sinc** (double x)
- double **relativeFrequency** (double f, double semitoneDifference)

13.6.1 Detailed Description

This namespace contains a number of classes that can be combined together to form a modular synthesizer that can be used to procedurally generate sound stimuli. There are methods for saving the sound stimuli to a file for later use or directly outputting the sounds to sound hardware. There is also a way to use the data from a CX_SoundBuffer as the input to the synth.

There are two types of oscillators (Oscillator and AdditiveSynth), an ADSR Envelope, two types of filters (Filter and FIRFilter), a Splitter and a Mixer, and some utility classes for adding, multiplying, and clamping values.

Making your own modules is simplified by the fact that all modules inherit from ModuleBase. You only need to overload one function from ModuleBase in order to have a functional module, although there are some other functions that can be overloaded for advanced uses.

13.7 CX::Util Namespace Reference

Classes

- class CX_LapTimer
- class CX_SegmentProfiler
- class CX_TrialController

- class CX_BaseUnitConverter
- class CX_DegreeToPixelConverter
- class CX_LengthToPixelConverter
- class CX_CoordinateConverter

Enumerations

- enum **CX_RoundingConfiguration** { **ROUND_TO_NEAREST**, **ROUND_UP**, **ROUND_DOWN**, **ROUND_TOWARD_ZERO** }

Functions

- float degreesToPixels (float degrees, float pixelsPerUnit, float viewingDistance)
- float pixelsToDegrees (float pixels, float pixelsPerUnit, float viewingDistance)
- unsigned int getMsaaSampleCount (void)
- bool checkOFVersion (int versionMajor, int versionMinor, int versionPatch)
- bool writeToFile (std::string filename, std::string data, bool append)
- double round (double d, int roundingPower, CX::Util::CX_RoundingConfiguration c)
- void saveFboToFile (ofFbo &fbo, std::string filename)
- std::map< std::string, std::string > readKeyValueFile (std::string filename, std::string delimiter, bool trimWhitespace, std::string commentString)
- float getAngleBetweenPoints (ofPoint p1, ofPoint p2)
- template<typename T >
  std::vector< T > arrayToVector (T arr[], unsigned int arraySize)
- template<typename T >
  std::vector< T > sequence (T start, T end, T stepSize)
- template<typename T >
  std::vector< T > sequenceSteps (T start, unsigned int steps, T stepSize)
- template<typename T >
  std::vector< T > sequenceAlong (T start, T end, unsigned int steps)
- template<typename T >
  std::vector< T > intVector (T start, T end)
- template<typename T >
  std::vector< T > repeat (T value, unsigned int times)
- template<typename T >
  std::vector< T > repeat (std::vector< T > values, unsigned int times, unsigned int each=1)
- template<typename T >
  std::vector< T > repeat (std::vector< T > values, std::vector< unsigned int > each, unsigned int times=1)
- template<typename T >
  std::string vectorToString (std::vector< T > values, std::string delimiter=",", int significantDigits=8)
- template<typename T >
  T clamp (T val, T minimum, T maximum)
- template<typename T >
  std::vector< T > **clamp** (std::vector< T > vals, T minimum, T maximum)
- template<typename T >
  std::vector< T > unique (std::vector< T > vals)
- template<typename T >
  std::vector< T > concatenate (const std::vector< T > &A, const std::vector< T > &B)
- template<typename T >
  T max (std::vector< T > vals)

- template<typename T >
  T min (std::vector< T > vals)
- template<typename T >
  T mean (std::vector< T > vals)
- template<typename T_OUT , typename T_IN >
  T_OUT mean (std::vector< T_IN > vals)
- template<typename T >
  T **var** (std::vector< T > vals)
- template<typename T_OUT , typename T_IN >
  T_OUT var (std::vector< T_IN > vals)

13.7.1 Detailed Description

This namespace contains a variety of utility functions.

13.7.2 Function Documentation

13.7.2.1 template<typename T > std::vector< T > CX::Util::arrayToVector (T *arr[],* unsigned int *arraySize*)

Copies arraySize elements of an array of T to a vector<T>.

Template Parameters

<T>	The type of the array. Is often inferred by the compiler.

Parameters

arr	The array of data to put into the vector.
arraySize	The length of the array, or the number of elements to copy from the array if not all of the elements are wanted.

Returns

The elements in a vector.

13.7.2.2 bool CX::Util::checkOFVersion (int *versionMajor,* int *versionMinor,* int *versionPatch*)

Checks that the version of oF that is used during compilation matches the requested version. If the desired version was 0.7.1, simply input (0, 7, 1) as the arguments. A warning will be logged if the versions don't match.

Returns

True if the versions match, false otherwise.

13.7.2.3 template<typename T > T CX::Util::clamp (T *val,* T *minimum,* T *maximum*)

Clamps a value (i.e. forces the value to be between two bounds). If the value is outside of the bounds, it is set to be equal to the nearest bound.

Parameters

val	The value to clamp.
minimum	The lower bound. Must be less than or equal to maximum.
maximum	The upper bound. Must be greater than or equal to minimum.

Returns

The clamped value.

13.7.2.4 template<typename T > std::vector< T > CX::Util::concatenate (const std::vector< T > & *A*, const std::vector< T > & *B*)

Concatenates together two vectors A and B.

Parameters

A	The first vector of values.
B	The second vector of values.

Returns

The concatenation of A and B, being a vector containing {A1, A2, ... An, B1, B2, ... Bn}.

13.7.2.5 float CX::Util::degreesToPixels (float *degrees*, float *pixelsPerUnit*, float *viewingDistance*)

Returns the number of pixels needed to subtend deg degrees of visual angle. You might want to round this if you want to align to pixel boundaries. However, if you are antialiasing your stimuli you might want to use floating point values to get precise subpixel rendering.

Parameters

degrees	Number of degrees.
pixelsPerUnit	The number of pixels per distance unit on the target monitor. You can pick any unit of distance, as long as `viewingDistance` has the same unit.
viewingDistance	The distance of the viewer from the monitor, with the same distance unit as `pixelsPerUnit`.

Returns

The number of pixels needed.

13.7.2.6 float CX::Util::getAngleBetweenPoints (ofPoint *p1*, ofPoint *p2*)

Returns the angle in degrees "between" p1 and p2. If you take the difference between p2 and p1, you get a resulting vector, V, that gives the displacement from p1 to p2. Imagine that you begin at (0, 0) and move to (0, abs(V.y)), creating a line segment. Now if you "rotate" this line segment clockwise until you reach V, the angle rotated through is the value returned by this function.

This is useful if you want to know, e.g., what angle you must travel on to get from some arbitrary point on screen to where the mouse cursor is.

Parameters

p1	The start point of the vector V.
p2	The end point of V. If p1 and p2 are reversed, the angle will be off by 180 degrees.

Returns

The angle between p1 and p2.

13.7.2.7 unsigned int CX::Util::getMsaaSampleCount (void)

This function retrieves the MSAA (http://en.wikipedia.org/wiki/Multisample_anti-aliasing) sample count. The sample count can be set by calling CX::relaunchWindow() with the desired sample count set in the argument to relaunchWindow().

13.7.2.8 template<typename T > std::vector< T > CX::Util::intVector (T *start*, T *end*)

Creates a vector of integers going from start to end. start may be greater than end, in which case the returned values will be in descending order. This is similar to using CX::sequence, but the step size is fixed to 1 and it works properly when trying to create a descending sequence of unsigned integers.

Returns

A vector of the values int the sequence.

13.7.2.9 template<typename T > T CX::Util::max (std::vector< T > *vals*)

Finds the maximum value in a vector of values.

Template Parameters

T	The type of data to be operated on. This type must have operator> defined.

Parameters

vals	The vector of values.

Returns

The maximum value in the vector.

13.7.2.10 template<typename T > T CX::Util::mean (std::vector< T > *vals*)

Calculates the mean value of a vector of values.

Template Parameters

T	The type of data to be operated on and returned. This type must have operator+(T) and operator/(unsigned int) defined.

Parameters

vals	The vector of values.

Returns

The mean of the vector.

13.7.2.11 template<typename T_OUT , typename T_IN > T_OUT CX::Util::mean (std::vector< T_IN > *vals*)

Calculates the mean value of a vector of values.

Template Parameters

T_OUT	The type of data to be returned. This type must have operator+(T_IN) and operator/(unsigned int) defined.
T_IN	The type of data to be operated on.

Parameters

vals	The vector of values.

Returns

The mean of the vector.

13.7.2.12 template<typename T > T CX::Util::min (std::vector< T > *vals*)

Finds the minimum value in a vector of values.

Template Parameters

T	The type of data to be operated on. This type must have operator< defined.

Parameters

vals	The vector of values.

Returns

The minimum value in the vector.

13.7.2.13 float CX::Util::pixelsToDegrees (float *pixels,* float *pixelsPerUnit,* float *viewingDistance*)

The inverse of CX::Util::degreesToPixels().

13.7.2.14 std::map< std::string, std::string > CX::Util::readKeyValueFile (std::string *filename,* std::string *delimiter,* bool *trimWhitespace,* std::string *commentString*)

This function reads in a file containing information stored as key-value pairs. A file of this kind could look like:

```
unleash_penguins=true
Key=Value
blue=0000FF
```

This type of file is often used for configuration of a program. This function simply provides a simple way to read in such data.

Parameters

filename	The name of the file containing key-value data.
delimiter	The string that separates the key from the value. In the example, it is "=".
trimWhitespace	If true, whitespace characters surrounding both the key and value will be removed.
commentString	If commentString is not the empty string (i.e. ""), everything on a line following the first instance of commentString will be ignored.

Returns

A map<string, string>, where the keys are the keys to the map.

13.7.2.15 template<typename T > std::vector< T > CX::Util::repeat (T *value,* unsigned int *times*)

Repeats value "times" times.

Parameters

value	The value to be repeated.
times	The number of times to repeat the value.

Returns

A vector containing times copies of the repeated value.

13.7.2.16 template<typename T > std::vector< T > CX::Util::repeat (std::vector< T > *values,* unsigned int *times,* unsigned int *each* = `1`)

Repeats the elements of values. Each element of values is repeated "each" times and then the process of repeating the elements is repeated "times" times.

Parameters

values	Vector of values to be repeated.
times	The number of times the process should be performed.
each	Number of times each element of values should be repeated.

Returns

A vector of the repeated values.

13.7.2.17 template<typename T > std::vector< T > CX::Util::repeat (std::vector< T > *values,* std::vector< unsigned int > *each,* unsigned int *times* = `1`)

Repeats the elements of values. Each element of values is repeated "each" times and then the process of repeating the elements is repeated "times" times.

Parameters

values	Vector of values to be repeated.
each	Number of times each element of values should be repeated. Must be the same length as values. If not, an error is logged and an empty vector is returned.

times	The number of times the process should be performed.

Returns

A vector of the repeated values.

13.7.2.18 double CX::Util::round (double *d,* int *roundingPower,* CX::Util::CX_RoundingConfiguration *c*)

Rounds the given double to the given power of 10.

Parameters

d	The number to be rounded.
roundingPower	The power of 10 to round d to. For example, if roundingPower is 0, d is rounded to the one's place (10^0 == 1). If roundingPower is -3, d is rounded to the thousandth's place (10^-3 = .001). If roundingPower is 1, d is rounded to the ten's place.
c	The type of rounding to do, from the CX::Util::CX_RoundingConfiguration enum. You can round up, down, to nearest, and toward zero.

Returns

The rounded value.

13.7.2.19 void CX::Util::saveFboToFile (ofFbo & *fbo,* std::string *filename*)

Saves the contents of an ofFbo to a file. The file type is hinted by the file extension you provide as part of the file name.

Parameters

fbo	The framebuffer to save.
filename	The path of the file to save. The file extension determines the type of file that is saved. Many standard file types are supported: png, bmp, jpg, gif, etc. However, if the fbo has an alpha channel, only png works properly (at least of those I have tested).

13.7.2.20 template<typename T > std::vector< T > CX::Util::sequence (T *start,* T *end,* T *stepSize*)

Creates a sequence of numbers from start to end by steps of size stepSize. start may be geater than end, but only if stepSize is less than 0. If start is less than end, stepSize must be greater than 0.

Example call: `sequence<double>(1, 3.3, 2)` results in a vector containing {1, 3}

Parameters

start	The start of the sequence. You are guaranteed to get this value in the sequence.
end	The number past which the sequence should end. You are not guaranteed to get this value.
stepSize	A nonzero number.

Returns

A vector containing the sequence.

13.7.2.21 template<typename T > std::vector< T > CX::Util::sequenceAlong (T *start,* T *end,* unsigned int *outputLength*)

Creates a sequence from start to end, where the size of each step is chosen so that the length of the sequence if equal to outputLength.

Parameters

start	The value at which to start the sequence.
end	The value to which to end the sequence.
outputLength	The number of elements in the returned sequence.

Returns

A vector containing the sequence.

13.7.2.22 template<typename T > std::vector< T > CX::Util::sequenceSteps (T *start*, unsigned int *steps*, T *stepSize*)

Make a sequence starting from start and taking steps steps of stepSize.

sequenceSteps(1.5, 4, 2.5);

Creates the sequence {1.5, 4, 6.5, 9, 11.5}

Parameters

start	Value from which to start.
steps	The number of steps to take.
stepSize	The size of each step.

Returns

A vector containing the sequence.

13.7.2.23 template<typename T > std::vector< T > CX::Util::unique (std::vector< T > *vals*)

Uses std::unique to find all of the unique values in `vals` and return copies of those values.

Parameters

vals	Th vector of values to find unique values in.

Returns

A vector containing the unique values in `vals`.

13.7.2.24 template<typename T_OUT , typename T_IN > T_OUT CX::Util::var (std::vector< T_IN > *vals*)

Calculates the sample variance of a vector of values.

Template Parameters

T_OUT	The type of data to be returned.
T_IN	The type of data to be operated on.

Parameters

vals	The vector of values.

Returns

The mean of the vector.

13.7.2.25 template<typename T > std::string CX::Util::vectorToString (std::vector< T > *values*, std::string *delimiter* = " , ", int *significantDigits* = 8)

This function converts a vector of values to a string representation of the values.

Parameters

values	The vector of values to convert.
delimiter	A string that is used to separate the elements of `value` in the final string.
significantDigits	Only for floating point types. The number of significant digits in the value.

Returns

A string containing a representation of the vector of values.

13.7.2.26 bool CX::Util::writeToFile (std::string *filename*, std::string *data*, bool *append*)

Writes data to a file, either appending the data to an existing file or creating a new file, overwriting any existing file with the given filename.

Parameters

filename	Name of the file to write to. If it is a relative file name, it will be placed relative the the data directory.
data	The data to write
append	If true, data will be appended to an existing file, if it exists. If append is false, any existing file will be overwritten and a warning will be logged. If no file exists, a new one will be created.

Returns

True if an error was encountered while writing the file, true otherwise. If there was an error, an error message will be logged.

14 Class Documentation

14.1 CX::Synth::Adder Class Reference

`#include <CX_ModularSynth.h>`

Inherits CX::Synth::ModuleBase.

Public Member Functions

- double getNextSample (void) override

Public Attributes

- ModuleParameter **amount**

Additional Inherited Members

14.1.1 Detailed Description

This class simply takes an input and adds an `amount` to it. The `amount` can be negative, in which case this class is a subtracter. If there is no input to this module, it behaves as though the input is 0, so the output value will be equal to `amount`. Thus, it can also behave as a numerical constant.

14.1.2 Member Function Documentation

14.1.2.1 double Adder::getNextSample (void) [override], [virtual]

This function should be overloaded for any derived class that can be used as the input for another module.

Reimplemented from CX::Synth::ModuleBase.

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

- CX_ModularSynth.h
- CX_ModularSynth.cpp

14.2 CX::Synth::AdditiveSynth Class Reference

`#include <CX_ModularSynth.h>`

Inherits CX::Synth::ModuleBase.

Public Types

- enum **HarmonicSeriesType** { **HS_MULTIPLE**, **HS_SEMITONE**, **HS_USER_FUNCTION** }
- enum HarmonicAmplitudeType { **SINE**, **SQUARE**, **SAW**, **TRIANGLE** }
- typedef float **wavePos_t**
- typedef float **amplitude_t**

Public Member Functions

- void **configure** (unsigned int harmonicCount, HarmonicSeriesType hs, HarmonicAmplitudeType aType)
- void **setFundamentalFrequency** (double f)
- void setStandardHarmonicSeries (unsigned int harmonicCount)
- void setHarmonicSeries (unsigned int harmonicCount, HarmonicSeriesType type, double controlParameter)
- void setHarmonicSeries (unsigned int harmonicCount, std::function< double(unsigned int)> userFunction)
- void setAmplitudes (HarmonicAmplitudeType type)
- void setAmplitudes (HarmonicAmplitudeType t1, HarmonicAmplitudeType t2, double mixture)
- void setAmplitudes (std::vector< amplitude_t > amps)
- std::vector< amplitude_t > **calculateAmplitudes** (HarmonicAmplitudeType type, unsigned int count)
- void pruneLowAmplitudeHarmonics (double tol)
- double getNextSample (void) override

Additional Inherited Members

14.2.1 Detailed Description

This class is an implementation of an additive synthesizer. Additive synthesizers are essentially an inverse fourier transform. You specify at which frequencies you want to have a sine wave and the amplitudes of those waves, and they are combined together into a single waveform.

The frequencies are refered to as harmonics, due to the fact that typical audio applications of additive synths use the standard harmonic series (f(i) = f_fundamental $*$ i). However, setting the harmonics to values not found in the standard harmonic series can result in really unusual and interesting sounds.

The output of the additive synth is not easily bounded between -1 and 1 due to various oddities of additive synthesis. For example, although in the limit as the number of harmonics goes to infinity square and sawtooth waves made with additive synthesis are bounded between -1 and 1, with smaller numbers of harmonics the amplitudes actually overshoot these bounds slightly. Of course, if an unusual harmonic series is used with arbitrary amplitudes, it can be hard to know if the output of the synth will be within the bounds. A Synth::Multiplier can help deal with this.

14.2.2 Member Enumeration Documentation

14.2.2.1 enum CX::Synth::AdditiveSynth::HarmonicAmplitudeType

Assuming that the standard harmonic series is being used, the values in this enum, when passed to setAmplitudes(), cause the amplitudes of the harmonics to be set in such a way as to produce the desired waveform.

14.2.3 Member Function Documentation

14.2.3.1 double AdditiveSynth::getNextSample (void) [override], [virtual]

This function should be overloaded for any derived class that can be used as the input for another module.

Reimplemented from CX::Synth::ModuleBase.

14.2.3.2 void AdditiveSynth::pruneLowAmplitudeHarmonics (double *tol*)

This function removes all harmonics that have an amplitude that is less than or equal to a tolerance times the amplitude of the frequency with the greatest absolute amplitude.

The result of this pruning is that the synthesizer will be more computationally efficient but provide a possibly worse approximation of the desired waveform.

Parameters

tol	`tol` is interpreted differently depending on its value. If `tol` is greater than or equal to 0, it is treated as a proportion of the amplitude of the frequency with the greatest amplitude. If `tol` is less than 0, it is treated as the difference in decibels between the frequency with the greatest amplitude and the tolerance.

Note

Because harmonics with an amplitude equal to the tolerance times an amplitude, setting `tol` to 0 will remove harmonics with 0 amplitude, but no others.

14.2.3.3 void AdditiveSynth::setAmplitudes (HarmonicAmplitudeType *type*)

This function sets the amplitudes of the harmonics based on the chosen type. The resulting waveform will only be correct if the harmonic series is the standard harmonic series (see setStandardHarmonicSeries()).

Parameters

type	The type of wave calculate amplitudes for.

14.2.3.4 void AdditiveSynth::setAmplitudes (HarmonicAmplitudeType *t1*, HarmonicAmplitudeType *t2*, double *mixture*)

This function sets the amplitudes of the harmonics based on a mixture of the chosen types. The resulting waveform will only be correct if the harmonic series is the standard harmonic series (see setStandardHarmonicSeries()). This is a convenient way to morph between waveforms.

14.2.3.5 void AdditiveSynth::setAmplitudes (std::vector< amplitude_t > *amps*)

This function sets the amplitudes of the harmonics to arbitrary values as specified in `amps`.

Parameters

amps	The amplitudes of the harmonics. If this vector does not contain as many values as there are harmonics, the unspecified amplitudes will be set to 0.

14.2.3.6 void AdditiveSynth::setHarmonicSeries (unsigned int *harmonicCount,* HarmonicSeriesType *type,* double *controlParameter*)

Parameters

type	The type of harmonic series to generate. Can be either HS_MULTIPLE or HS_SEMITONE. For HS_MULTIPLE, each harmonic's frequency will be some multiple of the fundamental frequency, depending on the harmonic number and controlParameter. For HS_SEMITONE, each harmonic's frequency will be some number of semitones above the previous frequency, based on controlParameter (specifying the number of semitones).
controlParameter	If type == HS_MULTIPLE, the frequency for harmonic i will be i $*$ controlParameter, where the fundamental gives the value 1 for i. If type == HS_SEMITONE, the frequency for harmonic i will be `pow(2, (i - 1) * controlParameter/12)`, where the fundamental gives the value 1 for i.

Note

If `type == HS_MULTIPLE` and `controlParameter == 1`, then the standard harmonic series will be generated.
If `type == HS_SEMITONE`, `controlParameter` does not need to be an integer.

14.2.3.7 void AdditiveSynth::setHarmonicSeries (unsigned int *harmonicCount,* std::function< double(unsigned int)> *userFunction*)

This function calculates the harmonic series from a function supplied by the user.

Parameters

harmonicCount	The number of harmonics to generate.
userFunction	The user function takes an integer representing the harmonic number, where the fundamental has the value 1, and returns the frequency that should be used for that harmonic.

14.2.3.8 void AdditiveSynth::setStandardHarmonicSeries (unsigned int *harmonicCount*)

The standard harmonic series begins with the fundamental frequency f1 and each seccuessive harmonic has a frequency equal to f1 $*$ n, where n is the harmonic number for the harmonic. This is the natural harmonic series, one that occurs, e.g., in a vibrating string.

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

- CX_ModularSynth.h
- CX_ModularSynth.cpp

14.3 CX::Algo::BlockSampler< T > Class Template Reference

`#include <CX_Algorithm.h>`

Public Member Functions

- **BlockSampler** (CX_RandomNumberGenerator *rng, const std::vector< T > &values)
- T **getNextValue** (void)
- void **resetBlocks** (void)
- unsigned int **getBlockNumber** (void)
- unsigned int **getBlockPosition** (void)

14.3.1 Detailed Description

template<typename T>class CX::Algo::BlockSampler< T >

This class helps with the case where a set of V values must be sampled randomly with the constraint that each block of V samples should have each value in the set. For example, if you want to present a number of trials in four different conditions, where the conditions are intermixed, but you want to observe all four trial types every four trials, you would use this class.

```
Algo::BlockSampler<int> bs(&RNG, Util::intVector(1, 4));

cout << "Block, Position, Value" << endl;
while (bs.getBlockNumber() < 4) { //Generate 4 blocks of values
    cout << bs.getBlockNumber() << ", " << bs.getBlockPosition() << ", ";
    cout << bs.getNextValue() << endl;
}
```

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

- CX_Algorithm.h

14.4 CX::Synth::Clamper Class Reference

`#include <CX_ModularSynth.h>`

Inherits CX::Synth::ModuleBase.

Public Member Functions

- double getNextSample (void) override

Public Attributes

- ModuleParameter **low**
- ModuleParameter **high**

Additional Inherited Members

14.4.1 Detailed Description

This class clamps inputs to be in the interval [`low`, `high`], where `low` and `high` are the members of this class.

14.4.2 Member Function Documentation

14.4.2.1 double Clamper::getNextSample (void) [override], [virtual]

This function should be overloaded for any derived class that can be used as the input for another module.

Reimplemented from CX::Synth::ModuleBase.

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

- CX_ModularSynth.h
- CX_ModularSynth.cpp

14.5 CX::CX_SlidePresenter::Configuration Struct Reference

`#include <CX_SlidePresenter.h>`

Public Types

- enum SwappingMode { **SINGLE_CORE_BLOCKING_SWAPS**, **MULTI_CORE** }

Public Attributes

- CX_Display * display

  A pointer to the display to use.
- std::function< void(CX_SlidePresenter::FinalSlideFunctionArgs &)> finalSlideCallback

  A pointer to a user function that will be called as soon as the final slide is presented.
- CX_SlidePresenter::ErrorMode **errorMode**
- bool deallocateCompletedSlides

  If true, once a slide has been presented, its framebuffer will be deallocated to conserve memory.
- CX_Millis preSwapCPUHoggingDuration

  Only used if swappingMode is a single core mode. The amount of time, before a slide is swapped from the back buffer to the front buffer, that the CPU is put into a spinloop waiting for the buffers to swap.
- enum CX::CX_SlidePresenter::Configuration::SwappingMode **swappingMode**
- bool useFenceSync

  Hint that fence sync should be used to check that slides are fully copied to the back buffer before they are swapped in.
- bool waitUntilFenceSyncComplete

  If useFenceSync is false, this is also forced to false. If this is true, new slides will not be swapped in until there is confirmation that the slide has been fully copied into the back buffer. This prevents vertical tearing, but may cause slides to be swapped in late if the copy confirmation is delayed but the copy has actually occurred. Does nothing if swappingMode is MULTI_CORE.

14.5.1 Detailed Description

This struct is used for configuring a CX_SlidePresenter.

14.5.2 Member Enumeration Documentation

14.5.2.1 enum CX::CX_SlidePresenter::Configuration::SwappingMode

The method used by the slide presenter to swap stimuli that have been drawn to the back buffer to the front buffer. MULTI_CORE is the best method, but only really works properly if you have at least a 2 core CPU. It uses a secondary thread to constantly swap the front and back buffers, which allows each frame to be counted. This results in really good synchronization between the copies if data to the back buffer and the swaps of the front and back buffers. In the SINGLE_CORE_BLOCKING_SWAPS mode, after a stimulus has been copied to the front buffer, the next stimulus is immediately drawn to the back buffer. After the correct amount of time minus preSwapCPUHoggingDuration, the buffers are swapped. The main problem with this mode is that the buffer swapping in this mode blocks in the main thread while waiting for the swap.

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

- CX_SlidePresenter.h

14.6 CX::CX_SoundStream::Configuration Struct Reference

`#include <CX_SoundStream.h>`

Public Attributes

- int inputChannels

  The number of input (e.g. microphone) channels to use. If 0, no input will be used.
- int outputChannels

  The number of output channels to use. Currently only stereo and mono are well-supported. If 0, no output will be used.
- int sampleRate
- unsigned int bufferSize
- RtAudio::Api api
- RtAudio::StreamOptions streamOptions
- int inputDeviceId

  The ID of the desired input device. A value of -1 will cause the system default input device to be used.
- int outputDeviceId

  The ID of the desired output device. A value of -1 will cause the system default output device to be used.

14.6.1 Detailed Description

This struct controls the configuration of the CX_SoundStream.

14.6.2 Member Data Documentation

14.6.2.1 RtAudio::Api CX::CX_SoundStream::Configuration::api

This argument depends on your operating system. Using RtAudio::Api::UNSPECIFIED will pick an available API for your system (if any; see the links below). The API means the type of software interface to use. For example, on Windows, you can choose from Windows Direct Sound (DS) and ASIO. ASIO is commonly used with audio recording equipment because it has lower latency whereas DS is more of a consumer-grade interface. The choice of API does not affect how you use this class, but it may affect the performance of sound playback.

See http://www.music.mcgill.ca/~gary/rtaudio/classRtAudio.html#ac9b6f625da88249d08a8409a9db for a listing of the APIs. See http://www.music.mcgill.ca/~gary/rtaudio/classRtAudio.-html#afd0bfa26deae9804e18faff59d0273d9 for the default ordering of the APIs if RtAudio::Api::UNSPECIFIED is used.

14.6.2.2 unsigned int CX::CX_SoundStream::Configuration::bufferSize

The size of the audio data buffer to use, in sample frames. A larger buffer size means more latency but also a greater potential for audio glitches (clicks and pops). Buffer size is per channel (i.e. if there are two channels and buffer size is set to 256, the actual buffer size will be 512 samples).

14.6.2.3 int CX::CX_SoundStream::Configuration::sampleRate

The requested sample rate for the input and output channels. If, for the selected device(s), this sample cannot be used, the nearest greater sample rate will be chosen. If there is no greater sample rate, the next lower sample rate will be used.

14.6.2.4 RtAudio::StreamOptions CX::CX_SoundStream::Configuration::streamOptions

See http://www.music.mcgill.ca/~gary/rtaudio/structRtAudio_1_1StreamOptions.-html for more information.

`flags` must not include RTAUDIO_NONINTERLEAVED: The audio data used by CX is interleaved.

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

- CX_SoundStream.h

14.7 CX::CX_BaseClockImplementation Class Reference

`#include <CX_Clock.h>`

Inherited by CX::CX_StdClockWrapper< stdClock >.

Public Member Functions

- virtual long long **nanos** (void)=0
- virtual void **resetStartTime** (void)=0
- virtual std::string **getName** (void)

14.7.1 Detailed Description

CX_Clock uses classes that are derived from this class for timing. See CX::CX_Clock::setImplementation().

nanos() should return the current time in nanoseconds. If the implementation does not have nanosecond precision, it should still return time in nanoseconds, which might just involve a multiplication (e.g. clock ticks are in microseconds, so multiply by 1000 to make each value equal to a nanosecond).

It is assumed that the implementation has some way to subtract off a start time so that nanos() counts up from 0 and that resetStartTime() can reset the start time so that the clock counts up from 0 after resetStartTime() is called.

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

- CX_Clock.h

14.8 CX::Util::CX_BaseUnitConverter Class Reference

`#include <CX_UnitConversion.h>`

Inherited by CX::Util::CX_DegreeToPixelConverter, and CX::Util::CX_LengthToPixelConverter.

Public Member Functions

- virtual float **operator()** (float x)
- virtual float **inverse** (float y)

14.8.1 Detailed Description

This class should be inherited from by any unit converters. You should override both operator() and inverse(). inverse() should perform the mathematical inverse of the operation performed by operator().

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

- CX_UnitConversion.h

14.9 CX::CX_Clock Class Reference

`#include <CX_Clock.h>`

Public Member Functions

- void setImplementation (CX::CX_BaseClockImplementation *impl)
- void precisionTest (unsigned int iterations)
- CX_Millis now (void)
- void sleep (CX_Millis t)
- void delay (CX_Millis t)
- void resetExperimentStartTime (void)
- std::string getExperimentStartDateTimeString (std::string format="%Y-%b-%e %h-%M-%S %a")

Static Public Member Functions

- static std::string getDateTimeString (std::string format="%Y-%b-%e %h-%M-%S %a")

14.9.1 Detailed Description

This class is responsible for getting timestamps for anything requiring timestamps. The way to get timing information is the function now(). It returns the current time relative to the start of the experiment in microseconds (on most systems, see getTickPeriod() to check the actual precision).

An instance of this class is preinstantiated for you. See CX::Instances::Clock.

14.9.2 Member Function Documentation

14.9.2.1 void CX_Clock::delay (CX_Millis *t*)

This functions blocks for the requested period of time. This is likely more precise than CX_Clock::sleep() because it does not give up control to the operating system, but it wastes resources because it just sits in a spinloop for the requested duration. This is functionally a static function.

14.9.2.2 std::string CX_Clock::getDateTimeString (std::string *format* = "%Y-%b-%e %h-%M-%S %a") [static]

This function returns a string containing the local time encoded according to some format.

Parameters

format	See http://pocoproject.org/docs/Poco.DateTimeFormatter.html#4684 for documentation of the format. E.g. "%Y/%m/%d %H:%M:%S" gives "year/month/day 24HourClock:minute:second" with some zero-padding for most things. The default "%Y-%b-%e %h-%M-%S %a" is "yearWithCentury-abbreviatedMonthName-nonZeroPaddedDay 12HourClock-minuteZeroPadded-secondZeroPadded am/pm".

14.9.2.3 std::string CX_Clock::getExperimentStartDateTimeString (std::string *format* = "%Y-%b-%e %h-%M-%S %a")

Get a string representing the date/time of the start of the experiment encoded according to a format.

Parameters

format	See getDateTimeString() for the definition of the format.

14.9.2.4 CX_Millis CX_Clock::now (void)

This function returns the current time relative to the start of the experiment in milliseconds. The start of the experiment is defined by default as when the CX_Clock instance named Clock (instantiated in this file) is constructed (typically the beginning of program execution).

Returns

A CX_Millis object containing the time.

Note

This cannot be converted to current date/time in any meaningful way. Use getDateTimeString() for that.

14.9.2.5 void CX_Clock::precisionTest (unsigned int *iterations*)

This function tests the precision of the clock used by CX. The results are computer-specific. If the precision of the clock is worse than microsecond accuracy, a warning is logged including information about the actual precision of the clock.

Depending on the number of iterations, this function may be considered blocking. See Blocking Code.

Parameters

iterations	Number of time duration samples to take. More iterations should give a better estimate.

14.9.2.6 void CX_Clock::resetExperimentStartTime (void)

If for some reason you have a long setup period before the experiment proper starts, you could call this function so that the values returned by CX_Clock::now() will count up from 0 starting from when this function was called. This function

also resets the experiment start date/time (see getExperimentStartDateTimeString()).

14.9.2.7 void CX_Clock::setImplementation (CX::CX_BaseClockImplementation * *impl*)

Set the underlying clock implementation used by this instance of CX_Clock. You would use this function if the default clock implementation used by CX_Clock has insufficient precision on your system. You can use CX::CX_StdClock-Wrapper to wrap any of the clocks from the std::chrono namespace or any clock that conforms to the standard of those clocks.

Parameters

impl	A pointer to an instance of a class derived from CX::CX_BaseClockImplementation.

14.9.2.8 void CX_Clock::sleep (CX_Millis *t*)

This functions sleeps for the requested period of time. This can be somewhat imprecise because it requests a specific sleep duration from the operating system, but the operating system may not provide the exact sleep time.

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

- CX_Clock.h
- CX_Clock.cpp

14.10 CX::Util::CX_CoordinateConverter Class Reference

`#include <CX_UnitConversion.h>`

Public Member Functions

- CX_CoordinateConverter (ofPoint origin, bool invertX, bool invertY, bool invertZ=false)
- ofPoint operator() (ofPoint p)
- ofPoint operator() (float x, float y, float z=0)
- ofPoint inverse (ofPoint p)
- ofPoint inverse (float x, float y, float z=0)
- void setAxisInversion (bool invertX, bool invertY, bool invertZ=false)
- void setOrigin (ofPoint newOrigin)
- void setMultiplier (float multiplier)
- void setUnitConverter (CX_BaseUnitConverter *converter)

14.10.1 Detailed Description

This helper class is used for converting from a somewhat user-defined coordinate system into the standard computer monitor coordinate system. When user coordinates are input into this class, they will be converted into the standard monitor coordinate system. This lets you use this class to allow you to use coordinates in your own system and convert those coordinates into the standard coordinates that are used by the drawing functions of openFrameworks.

See setUnitConverter() for a way to do change the units of the coordinate system to, for example, inches or degrees of visual angle.

Example use:

```
CX_CoordinateConverter conv(Display.
      getCenterOfDisplay(), false, true); //Make the center of the display the origin and
       invert
//the Y-axis. This makes positive x values go to the right and positive y values go up from the center of
```

```
       the display.
ofSetColor(255, 0, 0); //Draw a red circle in the center of the display.
ofCircle(conv(0, 0), 20);
ofSetColor(0, 255, 0); //Draw a green circle 100 pixels to the right of the center.
ofCircle(conv(100, 0), 20);
ofSetColor(0, 0, 255); //Draw a blue circle 100 pixels above the center (inverted y-axis).
ofCircle(conv(0, 100), 20);
```

Another example can be found in the advancedChangeDetection example experiment.

14.10.2 Constructor & Destructor Documentation

14.10.2.1 CX::Util::CX_CoordinateConverter::CX_CoordinateConverter (ofPoint *origin,* bool *invertX,* bool *invertY,* bool *invertZ =* `false`)

Constructs a coordinate converter with the given settings.

Parameters

origin	The location within the standard coordinate system at which the origin (the point at which the x, y, and z values are 0) of the user-defined coordinate system is located. If, for example, you want the center of the display to be the origin within your user-defined coordinate system, you could use CX_Display::getCenterOfDisplay() as the value for this argument.
invertX	Invert the x-axis from the default, which is that x increases to the right.
invertY	Invert the y-axis from the default, which is that y increases downward.
invertZ	Invert the z-axis from the default, which is that z increases toward the user (i.e. pointing out of the front of the screen).

14.10.3 Member Function Documentation

14.10.3.1 ofPoint CX::Util::CX_CoordinateConverter::inverse (ofPoint *p*)

Performs the inverse of operator(), i.e. converts from standard coordinates to user coordinates.

Parameters

p	A point in standard coordinates.

Returns

A point in user coordinates.

14.10.3.2 ofPoint CX::Util::CX_CoordinateConverter::inverse (float *x,* float *y,* float *z =* `0`)

Equivalent to `inverse(ofPoint(x,y,z));`

14.10.3.3 ofPoint CX::Util::CX_CoordinateConverter::operator() (ofPoint *p*)

The primary method of conversion between coordinate systems. You supply a point in user coordinates and get in return a point in standard coordinates.

Example use:

```
CX_CoordinateConverter cc(ofPoint(200,200), false, true);
ofPoint p(-50, 100); //P is in user-defined coordinates, 50 units left and 100 units above the origin.
ofPoint res = cc(p); //Use operator() to convert from the user system to the standard system.
//res should contain (150, 100) due to the inverted y axis.
```

Parameters

p	The point in user coordinates that should be converted to standard coordinates.

Returns

The point in standard coordinates.

14.10.3.4 ofPoint CX::Util::CX_CoordinateConverter::operator() (float *x*, float *y*, float *z* = `0`)

Equivalent to a call to `operator()(ofPoint(x, y, z));`.

14.10.3.5 void CX::Util::CX_CoordinateConverter::setAxisInversion (bool *invertX*, bool *invertY*, bool *invertZ* = `false`)

Sets whether each axis within the user-defined system is inverted from the standard coordinate system.

Parameters

invertX	Invert the x-axis from the default, which is that x increases to the right.
invertY	Invert the y-axis from the default, which is that y increases downward.
invertZ	Invert the z-axis from the default, which is that z increases toward the viewer (i.e. pointing out of the front of the screen).

14.10.3.6 void CX::Util::CX_CoordinateConverter::setMultiplier (float *multiplier*)

This function sets the amount by which user coordinates are multiplied before they are converted to standard coordinates. This allows you to easily scale stimuli. The multiplier is 1 by default.

Parameters

multiplier	The amount to multiply user coordinates by.

14.10.3.7 void CX::Util::CX_CoordinateConverter::setOrigin (ofPoint *newOrigin*)

Sets the location within the standard coordinate system at which the origin of the user-defined coordinate system is located.

Parameters

newOrigin	The location within the standard coordinate system at which the origin (the point at which the x, y, and z values are 0) of the user-defined coordinate system is located. If, for example, you want the center of the display to be the origin within your user-defined coordinate system, you could use CX_Display::getCenterOfDisplay() as the value for this argument.

14.10.3.8 void CX::Util::CX_CoordinateConverter::setUnitConverter (CX_BaseUnitConverter * *converter*)

Sets the unit converter that will be used when converting the coordinate system. In this way you can convert both the coordinate system in use and the units used by the coordinate system in one step. See CX_DegreeToPixelConverter and CX_LengthToPixelConverter for examples of the converters that can be used.

Example use:

```
//At global scope:
CX_CoordinateConverter conv(ofPoint(0,0), false, true); //The origin will be set to a
      proper value later.
CX_DegreeToPixelConverter d2p(35, 70);

//During setup:
conv.setOrigin(Display.getCenterOfDisplay());
```

```
conv.setUnitConverter(&d2p); //Use degrees of visual angle as the units of the user coordinate system.

//Draw a blue circle 2 degrees of visual angle to the left of the origin and 3 degrees above (inverted
       y-axis) the origin.
ofSetColor(0, 0, 255);
ofCircle(conv(-2, 3), 20);
```

Parameters

converter	A pointer to an instance of a class that is a CX_BaseUnitConverter or which has inherited from that class. See CX_UnitConversion.h/cpp for the implementation of CX_LengthToPixelConverter to see an example of how to create you own converter.

Note

The origin of the coordinate converter must be in the units that result from the unit conversion. E.g. if you are converting the units from degrees to pixels, the origin must be in pixels. See setOrigin().
The unit converter passed to this function must continue to exist throughout the lifetime of the coordinate converter. It is not copied.

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

- CX_UnitConversion.h
- CX_UnitConversion.cpp

14.11 CX::CX_DataFrame Class Reference

`#include <CX_DataFrame.h>`

Public Types

- typedef std::vector < CX_DataFrameCell > ::size_type **rowIndex_t**

Public Member Functions

- CX_DataFrame & operator= (CX_DataFrame &df)
- CX_DataFrameCell operator() (std::string column, rowIndex_t row)
- CX_DataFrameCell **operator()** (rowIndex_t row, std::string column)
- CX_DataFrameCell at (rowIndex_t row, std::string column)
- CX_DataFrameCell **at** (std::string column, rowIndex_t row)
- CX_DataFrameColumn **operator[]** (std::string column)
- CX_DataFrameRow **operator[]** (rowIndex_t row)
- void appendRow (CX_DataFrameRow row)
- void setRowCount (rowIndex_t rowCount)
- void addColumn (std::string columnName)
- std::string print (std::string delimiter="\t", bool printRowNumbers=true)
- std::string print (const std::set< std::string > &columns, std::string delimiter="\t", bool printRowNumbers=true)
- std::string print (const std::vector< rowIndex_t > &rows, std::string delimiter="\t", bool printRowNumbers=true)
- std::string print (const std::set< std::string > &columns, const std::vector< rowIndex_t > &rows, std::string delimiter="\t", bool printRowNumbers=true)

- bool printToFile (std::string filename, std::string delimiter="\t", bool printRowNumbers=true)
- bool printToFile (std::string filename, const std::set< std::string > &columns, std::string delimiter="\t", bool printRowNumbers=true)
- bool printToFile (std::string filename, const std::vector< rowIndex_t > &rows, std::string delimiter="\t", bool printRowNumbers=true)
- bool printToFile (std::string filename, const std::set< std::string > &columns, const std::vector< rowIndex_t > &rows, std::string delimiter="\t", bool printRowNumbers=true)
- bool readFromFile (std::string filename, std::string cellDelimiter="\t", std::string vectorEncloser="\"")
- void clear (void)
- bool deleteColumn (std::string columnName)
- bool deleteRow (rowIndex_t row)
- std::vector< std::string > columnNames (void)
- rowIndex_t getRowCount (void)

  Returns the number of rows in the data frame.
- bool reorderRows (const vector< CX_DataFrame::rowIndex_t > &newOrder)
- CX_DataFrame copyRows (vector< CX_DataFrame::rowIndex_t > rowOrder)
- CX_DataFrame copyColumns (vector< std::string > columns)
- void shuffleRows (void)
- void shuffleRows (CX_RandomNumberGenerator &rng)
- template<typename T >
  std::vector< T > copyColumn (std::string column)

Protected Member Functions

- void **_resizeToFit** (std::string column, rowIndex_t row)
- void **_resizeToFit** (rowIndex_t row)
- void **_resizeToFit** (std::string column)
- void **_equalizeRowLengths** (void)

Protected Attributes

- std::map< std::string, vector
  < CX_DataFrameCell > > **_data**
- rowIndex_t **_rowCount**

Friends

- class **CX_DataFrameRow**
- class **CX_DataFrameColumn**

14.11.1 Detailed Description

This class provides and easy way to store data from an experiment and output that data to a file at the end of the experiment. A CX_DataFrame is a square two-dimensional array of cells, but each cell is capable of holding a vector of data. Each cell is indexed with a column name (a string) and a row number. Cells can store many different kinds of data and the data can be inserted or extracted easily. The standard method of storing data is to use operator(), which dynamically resizes the data frame. When an experimental session is complete, the data can be written to a file using printToFile().

See the example dataFrame.cpp for thorough examples of how to use a CX_DataFrame.

Several of the member functions of this class could be blocking if the amount of data in the data frame is large enough.

14.11.2 Member Function Documentation

14.11.2.1 void CX_DataFrame::addColumn (std::string *columnName*)

Adds a column to the data frame.

Parameters

columnName	The name of the column to add. If a column with that name already exists in the data frame, a warning will be logged.

14.11.2.2 void CX_DataFrame::appendRow (CX_DataFrameRow *row*)

Appends the row to the end of the data frame.

Parameters

row	The row to add.

Note

If the row has columns that do not exist in the data frame, those columns will be added to the data frame.

14.11.2.3 CX_DataFrameCell CX_DataFrame::at (rowIndex_t *row,* std::string *column*)

Access the cell at the given row and column with bounds checking. Throws a std::out_of_range exception and logs an error if either the row or column is out of bounds.

Parameters

row	The row number.
column	The column name.

Returns

A CX_DataFrameCell that can be read from or written to.

14.11.2.4 void CX_DataFrame::clear (void)

Deletes the contents of the data frame. Resizes the data frame to have no rows and no columns.

14.11.2.5 std::vector< std::string > CX_DataFrame::columnNames (void)

Returns a vector containing the names of the columns in the data frame.

Returns

Vector of strings with the column names.

14.11.2.6 template<typename T > std::vector< T > CX::CX_DataFrame::copyColumn (std::string *column*)

Makes a copy of the data contained in the named column, converting it to the specified type (such a conversion must be possible).

Parameters

column	The name of the column to copy data from.

Returns

A vector containing the copied data.

14.11.2.7 CX_DataFrame CX_DataFrame::copyColumns (vector< std::string > *columns*)

Copies the specified columns into a new data frame.

Parameters

columns	A vector of column names to copy out. If a requested column is not found, a warning will be logged, but the function will otherwise complete successfully.

Returns

A CX_DataFrame containing the specified columns.

Note

This function may be Blocking Code if the amount of copied data is large.

14.11.2.8 CX_DataFrame CX_DataFrame::copyRows (vector< CX_DataFrame::rowIndex_t > *rowOrder*)

Creates CX_DataFrame containing a copy of the rows specified in rowOrder. The new data frame is not linked to the existing data frame.

Parameters

rowOrder	A vector of CX_DataFrame::rowIndex_t containing the rows from this data frame to be copied out. The indices in rowOrder may be in any order: They don't need to be ascending. Additionally, the same row to be copied may be specified multiple times.

Returns

A CX_DataFrame containing the rows specified in rowOrder.

Note

This function may be Blocking Code if the amount of copied data is large.

14.11.2.9 bool CX_DataFrame::deleteColumn (std::string *columnName*)

Deletes the given column of the data frame.

Parameters

columnName	The name of the column to delete. If the column is not in the data frame, a warning will be logged.

Returns

True if the column was found and deleted, false if it was not found.

14.11.2.10 bool CX_DataFrame::deleteRow (rowIndex_t *row*)

Deletes the given row of the data frame.

Parameters

row	The row to delete (0 indexed). If row is greater than or equal to the number of rows in the data frame, a warning will be logged.

Returns

True if the row was in bounds and was deleted, false if the row was out of bounds.

14.11.2.11 CX_DataFrameCell CX_DataFrame::operator() (std::string *column*, rowIndex_t *row*)

Access the cell at the given row and column. If the row or column is out of bounds, the data frame will be dynamically resized in order to fit the row or column.

Parameters

row	The row number.
column	The column name.

Returns

A CX_DataFrameCell that can be read from or written to.

14.11.2.12 CX_DataFrame & CX_DataFrame::operator= (CX_DataFrame & *df*)

Copy the contents of another CX_DataFrame to this data frame. Because this is a copy operation, this may be Blocking Code if the copied data frame is large enough.

Parameters

df	The data frame to copy.

Returns

A reference to this data frame.

Note

The contents of this data frame are deleted during the copy.

14.11.2.13 std::string CX_DataFrame::print (std::string *delimiter* = "\t", bool *printRowNumbers* = true)

Reduced argument version of print(). Prints all rows and columns.

14.11.2.14 std::string CX_DataFrame::print (const std::set< std::string > & *columns*, std::string *delimiter* = "\t", bool *printRowNumbers* = true)

Reduced argument version of print(). Prints all rows and the selected columns.

14.11.2.15 std::string CX_DataFrame::print (const std::vector< rowIndex_t > & *rows*, std::string *delimiter* = "\t", bool *printRowNumbers* = true)

Reduced argument version of print(). Prints all columns and the selected rows.

14.11.2.16 std::string CX_DataFrame::print (const std::set< std::string > & *columns*, const std::vector< rowIndex_t > & *rows*, std::string *delimiter* = "\t", bool *printRowNumbers* = true)

Prints the selected rows and columns of the data frame to a string. Each cell of the data frame will be separated with the selected delimiter.

Each row of the data frame will be ended with a new line (whatever std::endl evaluates to, typically "\r\n").

Parameters

columns	Columns to print. Column names not found in the data frame will be ignored with a warning.
rows	Rows to print. Row indices not found in the data frame will be ignored with a warning.
delimiter	Delimiter to be used between cells of the data frame. Using comma or semicolon for the delimiter is not recommended because semicolons are used as element delimiters in the string-encoded vectors stored in the data frame and commas are used for element delimiters within each element of the string-encoded vectors.
printRowNumbers	If true, a column will be printed with the header "rowNumber" with the contents of the column being the selected row indices. If false, no row numbers will be printed.

Returns

A string containing the printed version of the data frame.

Note

This function may be Blocking Code if the data frame is large enough.

14.11.2.17 bool CX_DataFrame::printToFile (std::string *filename*, std::string *delimiter* = "\t", bool *printRowNumbers* = true)

Reduced argument version of printToFile(). Prints all rows and columns.

14.11.2.18 bool CX_DataFrame::printToFile (std::string *filename*, const std::set< std::string > & *columns*, std::string *delimiter* = "\t", bool *printRowNumbers* = true)

Reduced argument version of printToFile(). Prints all rows and the selected columns.

14.11.2.19 bool CX_DataFrame::printToFile (std::string *filename*, const std::vector< rowIndex_t > & *rows*, std::string *delimiter* = "\t", bool *printRowNumbers* = true)

Reduced argument version of printToFile(). Prints all columns and the selected rows.

14.11.2.20 bool CX_DataFrame::printToFile (std::string *filename*, const std::set< std::string > & *columns*, const std::vector< rowIndex_t > & *rows*, std::string *delimiter* = "\t", bool *printRowNumbers* = true)

This function is equivalent in behavior to print() except that instead of returning a string containing the printed contents of the data frame, the string is printed to a file. If the file exists, it will be overwritten.

Parameters

filename	Name of the file to print to. If it is an absolute path, the file will be put there. If it is a local path, the file will be placed relative to the data directory of the project.

Returns

True for success, false if there was some problem writing to the file (insufficient permissions, etc.)

14.11.2.21 bool CX_DataFrame::readFromFile (std::string *filename,* std::string *cellDelimiter* = "\t", std::string *vectorEncloser* = "\"")

Reads data from the given file into the data frame. This function assumes that there will be a row of column names as the first row of the file. It does not treat consecutive delimiters as a single delimiter.

Parameters

filename	The name of the file to read data from. If it is a relative path, the file will be read relative to the data directory.
cellDelimiter	A string containing the delimiter between cells of the data frame.
vectorEncloser	A string containing the characters that surround cell that contain a vector of data. By default, vectors are enclosed in double quotes. This indicates to most software that it should treat the contents of the quotes "as-is", i.e. if it finds a delimiter within the quotes, it should not split there, but wait until out of the quotes.

Returns

False if an error occurred, true otherwise.

Note

The contents of the data frame will be deleted before attempting to read in the file.
If the data is read in from a file written with a row numbers column, that column will be read into the data frame. You can remove it using deleteColumn("rowNumber").
This function may be Blocking Code if the read in data frame is large enough.

14.11.2.22 bool CX_DataFrame::reorderRows (const vector< CX_DataFrame::rowIndex_t > & *newOrder*)

Re-orders the rows in the data frame.

Parameters

newOrder	Vector of row indices. newOrder.size() must equal this->getRowCount(). newOrder must not contain any out-of-range indices (i.e. they must be < getRowCount()). Both of these error conditions are checked for in the function call and errors are logged.

Returns

true if all of the conditions of newOrder are met, false otherwise.

14.11.2.23 void CX_DataFrame::setRowCount (rowIndex_t *rowCount*)

Sets the number of rows in the data frame.

Parameters

rowCount	The new number of rows in the data frame.

Note

If the row count is less than the number of rows already in the data frame, it will delete those rows with a warning.

14.11.2.24 void CX_DataFrame::shuffleRows (void)

Randomly re-orders the rows of the data frame using CX::Instances::RNG as the random number generator for the shuffling.

Note

This function may be Blocking Code if the data frame is large.

14.11.2.25 void CX_DataFrame::shuffleRows (CX_RandomNumberGenerator & *rng*)

Randomly re-orders the rows of the data frame.

Parameters

rng	Reference to a CX_RandomNumberGenerator to be used for the shuffling.

Note

This function may be Blocking Code if the data frame is large.

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

- CX_DataFrame.h
- CX_DataFrame.cpp

14.12 CX::CX_DataFrameCell Class Reference

`#include <CX_DataFrameCell.h>`

Public Member Functions

- CX_DataFrameCell (const char *c)
- template<typename T >
  CX_DataFrameCell (const T &value)

  Construct the cell, assigning the value to it.
- template<typename T >
  CX_DataFrameCell (const std::vector< T > &values)

  Construct the cell, assigning the values to it.
- CX_DataFrameCell & operator= (const char *c)
- template<typename T >
  CX_DataFrameCell & operator= (const T &value)

  Assigns a value to the cell.
- template<typename T >
  CX_DataFrameCell & operator= (const std::vector< T > &values)

  Assigns a vector of values to the cell.
- template<typename T >
  operator T (void) const

  Attempts to convert the contents of the cell to T using to().
- template<typename T >
  operator std::vector< T > (void) const

  Attempts to convert the contents of the cell to vector<T> using toVector<T>().
- template<typename T >
  void store (const T &value)
- template<typename T >
  T to (void) const
- std::string toString (void) const

  Returns a copy of the stored data in the internal string representation. Type checking is not done because this is a lossless operation.
- bool toBool (void) const

  Returns a copy of the stored data converted to bool. Equivalent to to<bool>().
- int toInt (void) const

  Returns a copy of the stored data converted to int. Equivalent to to<int>().

- double toDouble (void) const

  Returns a copy of the stored data converted to double. Equivalent to to<double>().
- template<typename T >
  std::vector< T > toVector (void) const
- template<typename T >
  void storeVector (std::vector< T > values)
- void copyCellTo (CX_DataFrameCell *targetCell)
- std::string getStoredType (void) const
- void **deleteStoredType** (void)
- template<>
  std::string to (void) const

14.12.1 Detailed Description

This class manages the contents of a single cell in a CX_DataFrame. It handles all of the type conversion nonsense that goes on when data is inserted into or extracted from a data frame. It tracks the type of the data that is inserted or extracted and logs warnings if the inserted type does not match the extracted type, with a few exceptions (see notes).

Note

There are a few exceptions to the type tracking. If the inserted type is const char*, it is treated as a string. Additionally, you can extract anything as string without a warning. This is because the data is stored as a string internally so extracting the data as a string is a lossless operation.

14.12.2 Constructor & Destructor Documentation

14.12.2.1 CX_DataFrameCell::CX_DataFrameCell (const char * *c*)

Constructs the cell with a string literal, treating it as a std::string.

14.12.3 Member Function Documentation

14.12.3.1 void CX_DataFrameCell::copyCellTo (CX_DataFrameCell * *targetCell*)

Copies the contents of this cell to targetCell, including type information.

Parameters

targetCell	A pointer to the cell to copy data to.

14.12.3.2 std::string CX_DataFrameCell::getStoredType (void) const

Gets a string representing the type of data stored within the cell. This string is implementation-defined (which is the C++ standards committee way of saying "It can be anything at all"). It is only guranteed to be the same for the same type, but not neccessarily be different for different types.

Returns

A string containing the name of the stored type as given by typeid(typename).name().

14.12.3.3 CX_DataFrameCell & CX_DataFrameCell::operator= (const char * *c*)

Assigns a string literal to the cell, treating it as a std::string.

14.12.3.4 template<typename T > void CX::CX_DataFrameCell::store (const T & *value*)

Stores the given value with the given type. This function is a good way to explicitly state the type of the data you are storing into the cell if, for example, it is a literal.

Template Parameters

<T>	The type to store the value as. If T is not specified, this function is essentially equivalent to using operator=.

Parameters

value	The value to store.

14.12.3.5 template<typename T > void CX::CX_DataFrameCell::storeVector (std::vector< T > *values*)

Stores a vector of data in the cell. The data is stored as a string with each element delimited by a semicolon. If the data to be stored are strings containing semicolons, the data will not be extracted properly.

Parameters

values	A vector of values to store.

14.12.3.6 template<typename T > T CX::CX_DataFrameCell::to (void) const

Attempts to convert the contents of the cell to type T. There are a variety of reasons why this conversion can fail and they all center on the user inserting data of one type and then attempting to extract data of a different type. Regardless of whether the conversion is possible, if you try to extract a type that is different from the type that is stored in the cell, a warning will be logged.

Template Parameters

<T>	The type to convert to.

Returns

The data in the cell converted to T.

14.12.3.7 std::string CX::CX_DataFrameCell::to (void) const

Equivalent to a call to toString(). This is specialized because it skips the type checks of to<T>.

Returns

A copy of the stored data encoded as a string.

14.12.3.8 template<typename T > std::vector< T > CX::CX_DataFrameCell::toVector (void) const

Returns a copy of the contents of the cell converted to a vector of the given type. If the type of data stored in the cell was not a vector of the given type or the type does match but it was a scalar that is stored, the logs a warning but attempts the conversion anyway.

Template Parameters

<T>	The type of the elements of the returned vector.

Returns

A vector containing the converted data.

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

- CX_DataFrameCell.h
- CX_DataFrameCell.cpp

14.13 CX::CX_DataFrameColumn Class Reference

Public Member Functions

- CX_DataFrameCell **operator[]** (CX_DataFrame::rowIndex_t row)
- CX_DataFrame::rowIndex_t **size** (void)

Friends

- class **CX_DataFrame**

14.13.1 Detailed Description

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

- CX_DataFrame.h
- CX_DataFrame.cpp

14.14 CX::CX_DataFrameRow Class Reference

Public Member Functions

- CX_DataFrameCell **operator[]** (std::string column)
- vector< std::string > **names** (void)
- void **clear** (void)

Friends

- class **CX_DataFrame**

14.14.1 Detailed Description

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

- CX_DataFrame.h
- CX_DataFrame.cpp

14.15 CX::Util::CX_DegreeToPixelConverter Class Reference

`#include <CX_UnitConversion.h>`

Inherits CX::Util::CX_BaseUnitConverter.

Public Member Functions

- CX_DegreeToPixelConverter (float pixelsPerUnit, float viewingDistance, bool roundResult=false)
- float operator() (float degrees) override
- float inverse (float pixels) override

14.15.1 Detailed Description

This simple utility class is used for converting degrees of visual angle to pixels on a monitor. This class uses CX::Util::degreesToPixels() internally. See also CX::Util::CX_CoordinateConverter for a way to also convert from one coordinate system to another.

Example use:

```
CX_DegreeToPixelConverter d2p(34, 60); //34 pixels per unit length (e.g. cm) on
       the target monitor, user is 60 length units from monitor.
ofLine( 200, 100, 200 + d2p(1), 100 + d2p(2) ); //Draw a line from (200, 100) (in pixel coordinates) to 1
       degree
//to the right and 2 degrees below that point.
```

14.15.2 Constructor & Destructor Documentation

14.15.2.1 CX::Util::CX_DegreeToPixelConverter::CX_DegreeToPixelConverter (float *pixelsPerUnit*, float *viewingDistance*, bool *roundResult* = `false`)

Constructs an instance of a CX_DegreeToPixelConverter using the given settings.

Parameters

pixelsPerUnit	The number of pixels within one length unit (e.g. inches, centimeters). This can be measured by drawing a ~100-1000 pixel square on the screen and measuring the length of a side and dividing the number of pixels by the total length measured.
viewingDistance	The distance from the monitor that the participant will be viewing the screen from.
roundResult	If true, the result of conversions will be rounded to the nearest integer (i.e. pixel). For drawing certain kinds of stimuli (especially text) it can be helpful to draw on pixel boundaries.

14.15.3 Member Function Documentation

14.15.3.1 float CX::Util::CX_DegreeToPixelConverter::inverse (float *pixels*) [override], [virtual]

Performs the inverse of the operation performed by operator(), i.e. converts pixels to degrees.

Parameters

pixels	The number of pixels to convert to degrees.

Returns

The number of degrees of visual angle subtended by the given number of pixels.

Reimplemented from CX::Util::CX_BaseUnitConverter.

14.15.3.2 float CX::Util::CX_DegreeToPixelConverter::operator() (float *degrees*) [override], [virtual]

Converts the degrees to pixels based on the settings given during construction.

Parameters

degrees	The number of degrees of visual angle to convert to pixels.

Returns

The number of pixels corresponding to the number of degrees of visual angle.

Reimplemented from CX::Util::CX_BaseUnitConverter.

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

- CX_UnitConversion.h
- CX_UnitConversion.cpp

14.16 CX::CX_Display Class Reference

`#include <CX_Display.h>`

Public Member Functions

- void setup (void)
- void configureFromFile (std::string filename, std::string delimiter="=", bool trimWhitespace=true, std::string commentString="//")
- void setFullScreen (bool fullScreen)
- bool isFullscreen (void)

  Returns true if the display is in full screen mode.
- void useHardwareVSync (bool b)
- void useSoftwareVSync (bool b)
- void beginDrawingToBackBuffer (void)
- void endDrawingToBackBuffer (void)
- void swapBuffers (void)
- void swapBuffersInThread (void)
- void setAutomaticSwapping (bool autoSwap)
- bool isAutomaticallySwapping (void)
- bool hasSwappedSinceLastCheck (void)
- CX_Millis getLastSwapTime (void)
- CX_Millis estimateNextSwapTime (void)
- uint64_t getFrameNumber (void)
- void estimateFramePeriod (CX_Millis estimationInterval)
- CX_Millis getFramePeriod (void)
- CX_Millis getFramePeriodStandardDeviation (void)
- void setWindowResolution (int width, int height)
- void setWindowTitle (std::string title)
- ofRectangle getResolution (void)
- ofPoint getCenterOfDisplay (void)
- void waitForOpenGL (void)
- CX_DataFrame testBufferSwapping (CX_Millis desiredTestDuration, bool testSecondaryThread)
- void copyFboToBackBuffer (ofFbo &fbo)
- void copyFboToBackBuffer (ofFbo &fbo, ofPoint destination)
- void copyFboToBackBuffer (ofFbo &fbo, ofRectangle source, ofPoint destination)

14.16.1 Detailed Description

This class represents an abstract visual display surface, which is my way of saying that it doesn't necessarily represent a monitor. The display surface can either be a window or, if full screen, the whole monitor. It is also a bit abstract in that it does not draw anything, but only creates an context in which things can be drawn.

14.16.2 Member Function Documentation

14.16.2.1 void CX_Display::beginDrawingToBackBuffer (void)

Prepares a rendering context for using drawing functions. Must be paired with a call to endDrawingToBackBuffer().

14.16.2.2 void CX_Display::configureFromFile (std::string *filename*, std::string *delimiter* = "=", bool *trimWhitespace* = true, std::string *commentString* = "//")

This function exists to serve a per-computer configuration function that is otherwise difficult to provide due to the fact that C++ programs are compiled to binaries and cannot be easily edited on the computer on which they are running. This function takes the file name of a specially constructed configuration file and reads the key-value pairs in that file in order to configure the CX_Display. The format of the file is provided in the example code below.

Sample configuration file:

```
display.windowWidth = 600
display.windowHeight = 300
display.windowTitle = My Neat Name
display.fullscreen = false
display.hardwareVSync = true
//display.softwareVSync = false //Commented out: no change
//display.swapAutomatically = false //Commented out: no change
```

All of the configuration keys are used in this example. Configuration options can be ignored. Ignored options result in no change in the configuration of the CX_Display. Note that the "display" prefix allows this configuration to be embedded in a file that also performs other configuration functions.

Because this function uses CX::Util::readKeyValueFile() internally, it has the same arguments.

Parameters

filename	The name of the file containing configuration data.
delimiter	The string that separates the key from the value. In the example, it is "=", but can be other values.
trimWhitespace	If true, whitespace characters surrounding both the key and value will be removed. This is a good idea to do.
commentString	If commentString is not the empty string (i.e. ""), everything on a line following the first instance of commentString will be ignored.

14.16.2.3 void CX_Display::copyFboToBackBuffer (ofFbo & *fbo*)

Copies an ofFbo to the back buffer using an efficient blitting operation. This overwrites the contents of the back buffer, it does not draw over them. For this reason, transparaency is ignored.

Parameters

fbo	The framebuffer to copy. It will be drawn starting from (0, 0) and will be drawn at the full dimensions of the fbo (whatever size was chosen at allocation of the fbo).

14.16.2.4 void CX_Display::copyFboToBackBuffer (ofFbo & *fbo*, ofPoint *destination*)

Copies an ofFbo to the back buffer using an efficient blitting operation.

Parameters

fbo	The framebuffer to copy.
destination	The point on the back buffer where the fbo will be placed.

14.16.2.5 void CX_Display::copyFboToBackBuffer (ofFbo & *fbo,* ofRectangle *source,* ofPoint *destination*)

Copies an ofFbo to the back buffer using an efficient blitting operation.

Parameters

fbo	The framebuffer to copy.
source	A rectangle giving an area of the fbo to copy.
destination	The point on the back buffer where the area of the fbo will be placed.

If this function does not provide enough flexibility, you can always draw ofFbo's using the following technique, which allows for transparency:

```
Display.beginDrawingToBackBuffer();
ofSetColor( 255 );
fbo.draw( x, y, width, height ); //Replace these variables with the destination location (x,y) and
      dimensions of the FBO.
Display.endDrawingToBackBuffer();
```

14.16.2.6 void CX_Display::endDrawingToBackBuffer (void)

Finish rendering to the back buffer. Must be paired with a call to beginDrawingToBackBuffer().

14.16.2.7 void CX_Display::estimateFramePeriod (CX_Millis *estimationInterval*)

This function estimates the typical period of the display refresh. This function blocks for estimationInterval while the swapping thread swaps in the background (see Blocking Code). This function is called with an argument of 300 ms during construction of this class, so there will always be some information about the frame period. If more precision of the estimate is desired, this function can be called again with a longer wait duration.

Parameters

estimationInterval	The length of time to spend estimating the frame period.

14.16.2.8 CX_Millis CX_Display::estimateNextSwapTime (void)

Get an estimate of the next time the front and back buffers will be swapped. This function depends on the precision of the frame period as estimated using estimateFramePeriod(). If the front and back buffers are not swapped every frame, the result of this function is meaningless because it uses the last buffer swap time as a reference.

Returns

A time value that can be compared to CX::Instances::Clock.now().

14.16.2.9 ofPoint CX_Display::getCenterOfDisplay (void)

Returns an ofPoint representing the center of the display. Works in either windowed or full screen mode.

14.16.2.10 uint64_t CX_Display::getFrameNumber (void)

This function returns the number of the last frame presented, as determined by number of front and back buffer swaps. It tracks buffer swaps that result from 1) the front and back buffer swapping automatically (as a result of setAutomaticSwapping() with true as the argument) and 2) manual swaps resulting from a call to swapBuffers() or swapBuffersInThread().

Returns

The number of the last frame. This value can only be compared with other values returned by this function.

14.16.2.11 CX_Millis CX_Display::getFramePeriod (void)

Gets the estimate of the frame period calculated with estimateFramePeriod().

14.16.2.12 CX_Millis CX_Display::getFramePeriodStandardDeviation (void)

Gets the estimate of the standard deviation of the frame period calculated with estimateFramePeriod().

14.16.2.13 CX_Millis CX_Display::getLastSwapTime (void)

Get the last time at which the front and back buffers were swapped.

Returns

A time value that can be compared with CX::Instances::Clock.now().

14.16.2.14 ofRectangle CX_Display::getResolution (void)

Returns the resolution of the current display area. If in windowed mode, this will return the resolution of the window. If in full screen mode, this will return the resolution of the monitor.

Returns

An ofRectangle containing the resolution. The width in pixels is stored in both the width and x members and the height in pixles is stored in both the height and y members, so you can use whichever makes the most sense to you.

14.16.2.15 bool CX_Display::hasSwappedSinceLastCheck (void)

Check to see if the display has swapped the front and back buffers since the last call to this function. This is generally used in conjuction with automatic swapping of the buffers (setAutomaticSwapping()) or with an individual threaded swap of the buffers (swapBuffersInThread()). This technically works with swapBuffers(), but given that that function only returns once the buffers have swapped, using this function to check that the buffers have swapped is redundant.

Returns

True if a swap has been made since the last call to this function, false otherwise.

14.16.2.16 bool CX_Display::isAutomaticallySwapping (void)

Determine whether the display is configured to automatically swap the front and back buffers every frame. See setAutomaticSwapping for more information.

14.16.2.17 void CX_Display::setAutomaticSwapping (bool *autoSwap*)

Set whether the front and buffers of the display will swap automatically every frame or not. You can check to see if a swap has occured by calling hasSwappedSinceLastCheck(). You can check to see if the display is automatically swapping by calling isAutomaticallySwapping().

Parameters

autoSwap	If true, the front and back buffer will swap automatically every frame.

Note

This function may block to due the requirement that it synchronize with the thread. See Blocking Code.

14.16.2.18 void CX_Display::setFullScreen (bool *fullScreen*)

Set whether the display is full screen or not. If the display is set to full screen, the resolution may not be the same as the resolution of display in windowed mode, and vice versa.

14.16.2.19 void CX_Display::setup (void)

Set up the display. Must be called for the display to function correctly.

14.16.2.20 void CX_Display::setWindowResolution (int *width,* int *height*)

Sets the resolution of the window. Has no effect if called while in full screen mode.

Parameters

width	The desired width of the window, in pixels.
height	The desired height of the window, in pixels.

14.16.2.21 void CX_Display::setWindowTitle (std::string *title*)

Sets the title of the experiment window.

Parameters

title	The new window title.

14.16.2.22 void CX_Display::swapBuffers (void)

This function queues up a swap of the front and back buffers then blocks until the swap occurs. This usually should not be used if `isAutomaticallySwapping() == true`. If it is, a warning will be logged.

See Also

Blocking Code

14.16.2.23 void CX_Display::swapBuffersInThread (void)

This function cues a swap of the front and back buffers. It avoids blocking (like swapBuffers()) by spawning a thread in which the swap is waited for. This does not make it obviously better than swapBuffers(), because spawning a thread has a cost and may introduce synchronization problems. Also, because this function does not block, in order to know when the buffer swap took place, you need to check hasSwappedSinceLastCheck() in order to know when the buffer swap has taken place.

14.16.2.24 CX_DataFrame CX_Display::testBufferSwapping (CX_Millis *desiredTestDuration,* bool *testSecondaryThread*)

This function tests buffer swapping under various combinations of Vsync setting and whether the swaps are requested in the main thread or in a secondary thread. The tests combine visual inspection and automated time measurement. The visual inspection is important because what the computer is told to put on the screen and what is actually drawn

on the screen are not always the same. It is best to run the tests in full screen mode, although that is not enforced. At the end of the tests, the results of the tests are provided to you to interpret based on the guidelines described here. The outcome of the test will usually be that there are some modes that don't work correctly and some that work well for the tested computer.

In the resulting CX_DataFrame, there are three columns that give the test conditions. "thread" indicates whether the main thread or a secondary thread was used. "hardVSync" and "softVSync" indicate whether hardware or software Vsync were enabled for the test (see useHardwareVSync() and useSoftwareVSync()). Other columns, giving data from the tests, are explained below. Whatever combination of Vsync works best can be chosen for use in experiments using useHardwareVSync() and useSoftwareVSync(). The threading mode is primarily used by CX_SlidePresenter with the CX::CX_SlidePresenter::Configuration::SwappingMode setting.

Continuous swapping test

This test examines the case of constantly swapping the fron and back buffers. It measures the amount of time between swaps, which should always approximately equal the frame period. The data from this test are found in columns of the data frame beginning with "cs": "csDurations" gives the raw between-swap durations, and "csDurationMean" and "csDurationStdDev" give the mean and standard deviation of the durations. If the swapping durations are not very consistent, which can be determined by examination or by looking at the standard deviation, then there is a problem with the configuration. If the mean duration is different from the monitor's actual refresh period, then there is a serious problem with the configuration.

During this test, you should see the screen very rapidly flickering between black and white. If you see slow flickering or a constant color, that is an error.

Wait swap test

One case that this function checks for is what happens if a swap is requested after a long period of no swaps being requested. In particular, this function swaps, waits for 2.5 swap periods and then swaps twice in a row. The idea is that there is a long delay between the first swap (the "long" swap) and the second swap (the "short" swap), followed by a standard delay before the third swap (the "normal" swap).

There are graded levels of success in this test. Complete success is when the duration of the first swap is 3P, where P is the standard swap period, and the duration of both of the second two swaps is 1P. Partial success is if the duration of the long swap is $\sim$2.5P, the duration of the short swap is $\sim$.5P, and the duration of the normal swap is 1P. In this case, the short swap at least gets things back on the right track. Failure occurs if the short swap duration is $\sim$0P. Mega failure occurs if the normal swap duration is $\sim$0P. In this case, it is taking multiple repeated swaps in order to regain vertical synchronization, which is unacceptable behavior.

You can visually check these results. During this test, an attempt is made to draw three bars on the left, middle, and right of the screen. The left bar is drawn for the long duration, the middle bar for the short duration, and the right bar for the normal duration. Complete success results in all three bars flickering (although you still need to check the timing data). Partial success results in only the left and right bars flickering with the middle bar location flat black. For the partial success case, the middle bar is never visible because at the time at which it is swapped in, the screen is in the middle of a refresh cycle. When the next refresh cycle starts, then the middle bar can start to be drawn to the screen. However, before it has a chance to be drawn, the right rectangle is drawn to the back buffer, overwriting the middle bar (or at least, this is my best explanation for why it isn't visible).

The timing data from the wait swap test can be found in columns of the data frame beginning with "ws". "wsLongMean", "wsShortMean", and "wsNormalMean" are the averages of the long, short, and normal swap durations, respectively. "wsTotalMean" is the sum of wsLongMean, wsShortMean, and wsNormalMean. But also be sure to check the raw data in "wsDurations", which goes along with the duration type in "wsTypes".

The wait swap test is not performed for the secondary thread, because the assumption is that if the secondary thread is used, in that thread the front and back buffers will be swapped constantly so there will be no wait swaps.

Parameters

desiredTest-Duration	An approximate amount of time to spend performing the tests.
testSecondary-Thread	If true, buffer swapping from within a secondary thread will be tested. If false, only swapping from within the main thread will be tested.

Returns

A CX_DataFrame containing timing results from the tests.

Note

This function blocks for approximately `desiredTestDuration` or more. See Blocking Code.

14.16.2.25 void CX_Display::useHardwareVSync (bool *b*)

Sets whether the display is using hardware VSync to control frame presentation. Without some form of Vsync, vertical tearing can occur.

Parameters

b	If true, hardware VSync will be enabled in the video card driver. If false, it will be disabled.

Note

This may not work, depending on your video card settings. Modern video card drivers allow you to control whether Vsync is used for all applications or not, or whether the applications are allowed to choose from themselves whether to use Vsync. If your drivers are set to force Vsync to a particular setting, this function is unlikely to have an effect. Even when the drivers allow applications to choose a Vsync setting, it is still possible that this function will have not have the expected effect. OpenGL seems to struggle with VSync.

See Also

See Framebuffers and Buffer Swapping for information on what VSync is.

14.16.2.26 void CX_Display::useSoftwareVSync (bool *b*)

Sets whether the display is using software VSync to control frame presentation. Without some form of Vsync, vertical tearing can occur. Hardware VSync, if available, is generally preferable to software VSync, so see useHardwareVSync() as well. However, software and hardware VSync are not mutally exclusive, sometimes using both together works better than only using one.

Parameters

b	If true, the display will attempt to do VSync in software.

See Also

See Framebuffers and Buffer Swapping for information on what Vsync is.

14.16.2.27 void CX_Display::waitForOpenGL (void)

This function blocks until all OpenGL instructions that were given before this was called to complete. This can be useful if you are trying to determine how long a set of rendering commands takes or need to make sure that all rendering is complete before moving on.

See Also

Blocking Code

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

- CX_Display.h
- CX_Display.cpp

14.17 CX::CX_InputManager Class Reference

`#include <CX_InputManager.h>`

Public Member Functions

- bool setup (bool useKeyboard, bool useMouse, int joystickIndex=-1)
- bool pollEvents (void)

Public Attributes

- CX_Keyboard Keyboard

  An instance of CX::CX_Keyboard. Enabled or disabled with CX::CX_InputManager::setup().
- CX_Mouse Mouse

  An instance of CX::CX_Mouse. Enabled or disabled with CX::CX_InputManager::setup().
- CX_Joystick Joystick

  An instance of CX::CX_Joystick. Enabled or disabled with CX::CX_InputManager::setup().

14.17.1 Detailed Description

This class is responsible for managing three basic input devices: a keyboard, mouse, and joystick. You access each of these devices with the corresponding member class: Keyboard, Mouse, and Joystick. See CX::CX_Keyboard, CX::CX_Mouse, and CX::CX_Joystick for more information about each specific device.

By default, all three input devices are disabled. Call setup() to enable specific devices.

14.17.2 Member Function Documentation

14.17.2.1 bool CX_InputManager::pollEvents (void)

It is not typically neccessary for the user to call this function directly, although there is no harm in doing so. This function polls for new events on all of the configured input devices (see setup()). After a call to this function, new events for the input devices can be found by checking the availableEvents() function for each device.

Returns

True if there are any events available for enabled devices, false otherwise. The events do not neccessarily need to be new events. If there are events that were already stored in Mouse, Keyboard, or Joystick but had not been processed by user code, this function will return true.

14.17.2.2 bool CX_InputManager::setup (bool *useKeyboard,* bool *useMouse,* int *joystickIndex* = -1)

Setup the input manager to use the requested devices. You may call this function multiple times if you want to change the configuration over the course of the experiment. Every time this function is called, all input device events are cleared.

Parameters

useKeyboard	Enable or disable the keyboard.
useMouse	Enable or disable the mouse.
joystickIndex	Optional. If >= 0, an attempt will be made to set up the joystick at that index. If < 0, no attempt will be made to set up the joystick.

Returns

False if the requested joystick could not be set up correctly, true otherwise.

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

- CX_InputManager.h
- CX_InputManager.cpp

14.18 CX::CX_Joystick Class Reference

`#include <CX_Joystick.h>`

Classes

- struct Event

Public Member Functions

- bool setup (int joystickIndex)
- std::string getJoystickName (void)
- bool pollEvents (void)
- int availableEvents (void)
- CX_Joystick::Event getNextEvent (void)
- void clearEvents (void)
- std::vector< float > getAxisPositions (void)
- std::vector< unsigned char > getButtonStates (void)

14.18.1 Detailed Description

This class manages a joystick that is attached to the system (if any). If more than one joystick is needed for the experiment, you can create more instances of CX_Joystick other than the one in CX::Instances::Input.

14.18.2 Member Function Documentation

14.18.2.1 int CX_Joystick::availableEvents (void)

Get the number of new events available for this input device.

14.18.2.2 void CX_Joystick::clearEvents (void)

Clear (delete) all events from this input device.

Note

This function only clears already existing events from the device, which means that responses made between a call to CX_InputManager::pollEvents() and a subsequent call to clearEvents() will not be removed by calling clearEvents().

14.18.2.3 vector< float > CX_Joystick::getAxisPositions (void)

This function is to be used for direct access to the axis positions of the joystick. It does not generate events (i.e. CX_Joystick::Event), nor does it do any timestamping. If timestamps and uncertainies are desired, you MUST use pollEvents() and the associated event functions (e.g. getNextEvent()).

14.18.2.4 vector< unsigned char > CX_Joystick::getButtonStates (void)

This function is to be used for direct access to the button states of the joystick. It does not generate events (i.e. CX_Joystick::Event), nor does it do any timestamping. If timestamps and uncertainies are desired, you MUST use pollEvents() and the associated event functions (e.g. getNextEvent()).

14.18.2.5 std::string CX_Joystick::getJoystickName (void)

Get the name of the joystick, presumably as set by the joystick driver. The name may not be very meaningful.

14.18.2.6 CX_Joystick::Event CX_Joystick::getNextEvent (void)

Get the next event available for this input device. This is a destructive operation: the returned event is deleted from the input device.

14.18.2.7 bool CX_Joystick::pollEvents (void)

Check to see if there are any new joystick events. If there are new events, they can be accessed with availableEvents() and getNextEvent().

Returns

True if there are new events.

14.18.2.8 bool CX_Joystick::setup (int *joystickIndex*)

Set up the joystick by attempting to initialize the joystick at the given index. If the joystick is present on the system, it will be initialized and its name can be accessed by calling getJoystickName().

If the set up is successful (i.e. if the selected joystick is present on the system), this function will return true. If the joystick is not present, it will return false.

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

- CX_Joystick.h
- CX_Joystick.cpp

14.19 CX::CX_Keyboard Class Reference

`#include <CX_Keyboard.h>`

Classes

- struct Event

Public Member Functions

- int availableEvents (void)
- CX_Keyboard::Event getNextEvent (void)
- void clearEvents (void)
- bool isKeyPressed (int key)

Friends

- class **CX_InputManager**

14.19.1 Detailed Description

This class is responsible for managing the mouse.

14.19.2 Member Function Documentation

14.19.2.1 int CX_Keyboard::availableEvents (void)

Get the number of new events available for this input device.

14.19.2.2 void CX_Keyboard::clearEvents (void)

Clear (delete) all events from this input device.

Note

This function only clears already existing events from the device, which means that responses made between a call to CX_InputManager::pollEvents() and a subsequent call to clearEvents() will not be removed by calling clearEvents().

14.19.2.3 CX_Keyboard::Event CX_Keyboard::getNextEvent (void)

Get the next event available for this input device. This is a destructive operation: the returned event is deleted from the input device.

14.19.2.4 bool CX_Keyboard::isKeyPressed (int *key*)

This function checks to see if the given key is pressed.

Parameters

key	The key code or character for the key you are interested in. See the documentation for CX_-Keyboard::Event::key for more information about this value.

Returns

True iff the given key is held.

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

- CX_Keyboard.h
- CX_Keyboard.cpp

14.20 CX::Util::CX_LapTimer Class Reference

`#include <CX_TimeUtilities.h>`

Public Member Functions

- void setup (CX_Clock *clock, unsigned int logSamples)
- void **reset** (void)
- void **takeSample** (void)
- std::vector< double >::size_type **collectedSamples** (void)
- CX_Millis **mean** (void)
- CX_Millis **min** (void)
- CX_Millis **max** (void)
- CX_Millis **stdDev** (void)
- std::string **getStatString** (void)

14.20.1 Detailed Description

This class can be used for profiling event loops. It measures the amount of time that has elapsed between subsequent calls to takeSample().

```
//Set up collection:
CX_LapTimer lt;
lt.setup(&Clock, 1000); //Every 1000 samples, the results of those samples will be logged.

//In the loop:
while (whatever) {
//other code...
lt.takeSample();
//other code...
}
Log.flush(); //Check the results of the profiling.
```

14.20.2 Member Function Documentation

14.20.2.1 void CX::Util::CX_LapTimer::setup (CX_Clock * *clock*, unsigned int *logSamples*)

Set up the

Parameters

clock	The instance of CX_Clock to use.

logSamples	If this is not 0, then every `logSamples` samples, the stats string will be logged.

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

- CX_TimeUtilities.h
- CX_TimeUtilities.cpp

14.21 CX::Util::CX_LengthToPixelConverter Class Reference

`#include <CX_UnitConversion.h>`

Inherits CX::Util::CX_BaseUnitConverter.

Public Member Functions

- CX_LengthToPixelConverter (float pixelsPerUnit, bool roundResult=false)
- float operator() (float length) override
- float inverse (float pixels) override

14.21.1 Detailed Description

This simple utility class is used for converting lengths (perhaps of objects drawn on the monitor) to pixels on a monitor. See also CX::Util::CX_CoordinateConverter for a way to also convert from one coordinate system to another. This assumes that pixels are square, which may not be true, especially if you are using a resolution that is not the native resolution of the monitor.

Example use:

```
CX_LengthToPixelConverter l2p(75); //75 pixels per unit length (e.g. inch) on the
      target monitor.
ofLine( 200, 100, 200 + l2p(1), 100 + l2p(2) ); //Draw a line from (200, 100) (in pixel coordinates) to 1
      unit
//horizontally and 2 units vertically from that point.
```

14.21.2 Constructor & Destructor Documentation

14.21.2.1 CX::Util::CX_LengthToPixelConverter::CX_LengthToPixelConverter (float *pixelsPerUnit*, bool *roundResult* = `false`)

Constructs a CX_LengthToPixelConverter with the given configuration.

Parameters

pixelsPerUnit	The number of pixels per one length unit. This can be measured by drawing a ~100-1000 pixel square on the screen and measuring the length of a side and dividing the number of pixels by the total length measured.
roundResult	If true, the result of conversions will be rounded to the nearest integer (i.e. pixel). For drawing certain kinds of stimuli (especially text) it can be helpful to draw on pixel boundaries.

14.21.3 Member Function Documentation

14.21.3.1 float CX::Util::CX_LengthToPixelConverter::inverse (float *pixels*) `[override], [virtual]`

Performs to inverse of operator(), i.e. converts pixels to length.

Parameters

pixels	The number of pixels to convert to a length.

Returns

The length of the given number of pixels.

Reimplemented from CX::Util::CX_BaseUnitConverter.

14.21.3.2 float CX::Util::CX_LengthToPixelConverter::operator() (float *length*) [override], [virtual]

Converts the length to pixels based on the settings given during construction.

Parameters

length	The length to convert to pixels.

Returns

The number of pixels corresponding to the length.

Reimplemented from CX::Util::CX_BaseUnitConverter.

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

- CX_UnitConversion.h
- CX_UnitConversion.cpp

14.22 CX::CX_Logger Class Reference

`#include <CX_Logger.h>`

Public Member Functions

- std::stringstream & log (CX_LogLevel level, std::string module="")
- std::stringstream & verbose (std::string module="")
- std::stringstream & notice (std::string module="")
- std::stringstream & warning (std::string module="")
- std::stringstream & error (std::string module="")
- std::stringstream & fatalError (std::string module="")
- void level (CX_LogLevel level, std::string module="")
- void levelForConsole (CX_LogLevel level)
- void levelForFile (CX_LogLevel level, std::string filename="CX_DEFERRED_LOGGER_DEFAULT")
- void levelForAllModules (CX_LogLevel level)
- CX_LogLevel getModuleLevel (std::string module)
- void flush (void)
- void timestamps (bool logTimestamps, std::string format="%H:%M:%S.%i")
- void setMessageFlushCallback (std::function< void(CX_MessageFlushData &)> f)
- void captureOFLogMessages (void)

14.22.1 Detailed Description

This class is used for logging messages throughout the CX backend code. It can also be used in user code to log messages. Rather than instantiating your own copy of CX_Logger, it is probably better to use the preinstantiated CX::-Instances::Log.

There is an example showing a number of the features of CX_Logger named example-logging.

14.22.2 Member Function Documentation

14.22.2.1 void CX_Logger::captureOFLogMessages (void)

Set this instance of CX_Logger to be the target of any messages created by oF logging functions.

14.22.2.2 std::stringstream & CX_Logger::error (std::string *module* = " ")

This function is equivalent to a call to log(CX_LogLevel::LOG_ERROR, module).

14.22.2.3 std::stringstream & CX_Logger::fatalError (std::string *module* = " ")

This function is equivalent to a call to log(CX_LogLevel::LOG_FATAL_ERROR, module).

14.22.2.4 void CX_Logger::flush (void)

Log all of the messages stored since the last call to flush() to the selected logging targets. This is a blocking operation, because it may take quite a while to output all log messages to various targets (see Blocking Code). This function is not thread-safe: Only call it from the main thread.

14.22.2.5 CX_LogLevel CX_Logger::getModuleLevel (std::string *module*)

Gets the log level in use by the given module.

Parameters

module	The name of the module.

Returns

The CX_LogLevel for `module`.

14.22.2.6 void CX_Logger::level (CX_LogLevel *level*, std::string *module* = " ")

Sets the log level for the given module. Messages from that module that are at a lower level than level will be ignored.

Parameters

level	See the CX::CX_LogLevel enum for valid values.
module	A string representing one of the modules from which log messages are generated.

14.22.2.7 void CX_Logger::levelForAllModules (CX_LogLevel *level*)

Set the log level for all modules. This works both retroactively and proactively: All currently known modules are given the log level and the default log level for new modules as set to the level.

14.22.2.8 void CX_Logger::levelForConsole (CX_LogLevel *level*)

Set the log level for messages to be printed to the console.

14.22.2.9 void CX_Logger::levelForFile (CX_LogLevel *level*, std::string *filename* = "CX_DEFERRED_LOGGER_-DEFAULT")

Sets the log level for the file with given file name. If the file does not exist, it will be created. If the file does exist, it will be overwritten with a warning logged to cerr.

Parameters

level	See the CX_LogLevel enum for valid values.
filename	Optional. If no file name is given, a file with name generated from a date/time from the start time of the experiment will be used.

14.22.2.10 std::stringstream & CX_Logger::log (CX_LogLevel *level*, std::string *module* = "")

This is the basic logging function for this class. Example use:

Log.log(CX_LogLevel::LOG_WARNING, "myModule") << "My message number " << 20;

Possible output: "[warning] <myModule> My message number 20"

Parameters

level	Log level for this message. This has implications for message filtering. See level(). This should not be LOG_ALL or LOG_NONE, because that would be weird, wouldn't it?
module	Name of the module that this log message is related to. This has implications for message filtering. See level().

Returns

A reference to a std::stringstream that the log message data should be streamed into.

Note

This function and all of the trivial wrappers of this function (verbose(), notice(), warning(), error(), fatalError()) are thread-safe.

14.22.2.11 std::stringstream & CX_Logger::notice (std::string *module* = "")

This function is equivalent to a call to log(CX_LogLevel::LOG_NOTICE, module).

14.22.2.12 void CX_Logger::setMessageFlushCallback (std::function< void(CX_MessageFlushData &)> *f*)

Sets the user function that will be called on each message flush event. For every message that has been logged, the user function will be called. No filtering is performed: All messages regardless of the module log level will be sent to the user function.

Parameters

f	A pointer to a user function that takes a reference to a CX_MessageFlushData struct and returns nothing.

14.22.2.13 void CX_Logger::timestamps (bool *logTimestamps*, std::string *format* = "%H:%M:%S.%i")

Set whether or not to log timestamps and the format for the timestamps.

Parameters

logTimestamps	Does what it says.
format	Timestamp format string. See http://pocoproject.org/docs/Poco.DateTimeFormatter.html#4684 for documentation of the format. Defaults to H:M:S.i (24-hour clock with milliseconds at the end).

14.22.2.14 std::stringstream & CX_Logger::verbose (std::string *module* = " ")

This function is equivalent to a call to log(CX_LogLevel::LOG_VERBOSE, module).

14.22.2.15 std::stringstream & CX_Logger::warning (std::string *module* = " ")

This function is equivalent to a call to log(CX_LogLevel::LOG_WARNING, module).

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

- CX_Logger.h
- CX_Logger.cpp

14.23 CX::CX_Mouse Class Reference

`#include <CX_Mouse.h>`

Classes

- struct Event

Public Member Functions

- int availableEvents (void)
- CX_Mouse::Event getNextEvent (void)
- void clearEvents (void)
- void showCursor (bool show)
- void setCursorPosition (ofPoint pos)
- ofPoint getCursorPosition (void)

Friends

- class **CX_InputManager**

14.23.1 Detailed Description

This class is responsible for managing the mouse.

14.23.2 Member Function Documentation

14.23.2.1 int CX_Mouse::availableEvents (void)

Get the number of new events available for this input device.

14.23.2.2 void CX_Mouse::clearEvents (void)

Clear (delete) all events from this input device.

Note

This function only clears already existing events from the device, which means that responses made between a call to CX_InputManager::pollEvents() and a subsequent call to clearEvents() will not be removed by calling clearEvents().

14.23.2.3 ofPoint CX_Mouse::getCursorPosition (void)

Get the cursor position within the program window. If the mouse has left the window, this will return the last known position of the cursor within the window.

Returns

An ofPoint with the last cursor position.

14.23.2.4 CX_Mouse::Event CX_Mouse::getNextEvent (void)

Get the next event available for this input device. This is a destructive operation: the returned event is deleted from the input device.

14.23.2.5 void CX_Mouse::setCursorPosition (ofPoint *pos*)

Sets the position of the cursor, relative to the program the window. The window must be focused.

Parameters

pos	The location within the window to set the cursor.

14.23.2.6 void CX_Mouse::showCursor (bool *show*)

Show or hide the mouse cursor within the program window. If in windowed mode, the cursor will be visible outside of the window.

Parameters

show	If true, the cursor will be shown, if false it will not be shown.

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

- CX_Mouse.h
- CX_Mouse.cpp

14.24 CX::CX_RandomNumberGenerator Class Reference

`#include <CX_RandomNumberGenerator.h>`

Public Member Functions

- CX_RandomNumberGenerator (void)
- void setSeed (unsigned long seed)
- unsigned long getSeed (void)

- CX_RandomInt_t getMinimumRandomInt (void)
- CX_RandomInt_t getMaximumRandomInt (void)
- CX_RandomInt_t randomInt (void)
- CX_RandomInt_t randomInt (CX_RandomInt_t rangeLower, CX_RandomInt_t rangeUpper)
- double randomDouble (double lowerBound_closed, double upperBound_open)
- template<typename T >
  void shuffleVector (std::vector< T > *v)
- template<typename T >
  std::vector< T > shuffleVector (std::vector< T > v)
- template<typename T >
  T sample (const std::vector< T > &values)
- template<typename T >
  std::vector< T > sample (unsigned int count, const std::vector< T > &source, bool withReplacement)
- std::vector< int > sample (unsigned int count, int lowerBound, int upperBound, bool withReplacement)
- template<typename T >
  T sampleExclusive (const std::vector< T > &values, const T &exclude)
- template<typename T >
  T sampleExclusive (const std::vector< T > &values, const std::vector< T > &exclude)
- template<typename T >
  std::vector< T > sampleBlocks (const std::vector< T > &values, unsigned int blocksToSample)
- template<typename stdDist >
  std::vector< typename
  stdDist::result_type > sampleRealizations (unsigned int count, stdDist dist)
- std::vector< double > sampleUniformRealizations (unsigned int count, double lowerBound_closed, double upperBound_open)
- std::vector< unsigned int > sampleBinomialRealizations (unsigned int count, unsigned int trials, double probSuccess)
- std::vector< double > sampleNormalRealizations (unsigned int count, double mean, double standardDeviation)
- std::mt19937_64 & getGenerator (void)

14.24.1 Detailed Description

This class is used for generating random values from a pseudo-random number generator. If uses a version of the Mersenne Twister algorithm, in particular std::mt19937_64 (see http://en.cppreference.com/w/cpp/numeric/random/mersenne_twister_engine for the parameters used with this algorithm).

The monolithic structure of CX_RandomNumberGenerator provides a certain important feature that a collection of loose function does not have, which is the ability to trivially track the random seed being used for the random number generator. The function CX_RandomNumberGenerator::setSeed() sets the seed for all random number generation tasks performed by this class. Likewise, CX_RandomNumberGenerator::getSeed() allows you to easily find the seed that is being used for random number generation. Due to this structure, you can easily save the seed that was used for each participant, which allows you to repeat the exact randomizations used for that participant (unless random number generation varies as a function of the responses given by a participant).

An instance of this class is preinstantiated for you. See CX::Instances::RNG for information about the instance with that name.

Because the underlying C++ std library random number generators are not thread safe, CX_RandomNumberGenerator is not thread safe. If you want to use a CX_RandomNumberGenerator in a thread, that thread should have its own CX_RandomNumberGenerator. You may seed the thread's CX_RandomNumberGenerator with CX::Instances::RNG.

14.24.2 Constructor & Destructor Documentation

14.24.2.1 CX_RandomNumberGenerator::CX_RandomNumberGenerator (void)

Constructs an instance of a CX_RandomNumberGenerator. Seeds the CX_RandomNumberGenerator using a std::random_device.

By the C++11 specification, std::random_device is supposed to be a non-deterministic (hardware) RNG. However, from http://en.cppreference.com/w/cpp/numeric/random/random_device: "Note that std::random_device may be implemented in terms of a pseudo-random number engine if a non-deterministic source (e.g. a hardware device) is not available to the implementation." According to a Stack Overflow comment, Microsoft's implementation of std::random_device is based on a ton of stuff, which should result in a fairly random result to be used as a seed for our Mersenne Twister. See the comment: http://stackoverflow.com/questions/9549357/the-implementation-of-random-device-in-vs2010/9575747#9575747 Although this data should have high entropy, it is not a hardware RNG. The random_device is only used to seed the Mersenne Twister, so as long as the initial value is random enough, it should be fine.

14.24.3 Member Function Documentation

14.24.3.1 std::mt19937_64 & CX_RandomNumberGenerator::getGenerator (void)

This function returns a reference to the standard library PRNG used by the CX_RandomNumberGenerator. This can be used for various things, including sampling from some of the other distributions provided by the standard library: http://en.cppreference.com/w/cpp/numeric/random

```
std::poisson_distribution<int> pois(4);
int deviate = pois(RNG.getGenerator());
```

14.24.3.2 CX_RandomInt_t CX_RandomNumberGenerator::getMaximumRandomInt (void)

Get the maximum possible value that can be returned by randomInt().

Returns

The maximum value.

14.24.3.3 CX_RandomInt_t CX_RandomNumberGenerator::getMinimumRandomInt (void)

Get the minimum value that can be returned by randomInt().

Returns

The minimum value.

14.24.3.4 unsigned long CX_RandomNumberGenerator::getSeed (void)

Get the seed used to seed the random number generator.

Returns

The seed. May have been set by the user with setSeed() or during construction of the CX_RandomNumberGenerator.

14.24.3.5 double CX_RandomNumberGenerator::randomDouble (double *lowerBound_closed*, double *upperBound_open*)

Samples a realization from a uniform distribution with the range [lowerBound_closed, upperBound_open).

Parameters

lowerBound_closed	The lower bound of the distribution. This bound is closed, meaning that you can observe this value.
upperBound_open	The upper bound of the distribution. This bound is open, meaning that you cannot observe this value.

Returns

The realization.

14.24.3.6 CX_RandomInt_t CX_RandomNumberGenerator::randomInt (void)

Get a random integer in the range getMinimumRandomInt(), getMaximumRandomInt(), inclusive.

Returns

The int.

14.24.3.7 CX_RandomInt_t CX_RandomNumberGenerator::randomInt (CX_RandomInt_t *min*, CX_RandomInt_t *max*)

This function returns an integer from the range [rangeLower, rangeUpper]. The minimum and maximum values for the int returned from this function are given by getMinimumRandomInt() and getMaximumRandomInt().

If rangeLower > rangeUpper, the lower and upper ranges are swapped. If rangeLower == rangeUpper, it returns rangeLower.

14.24.3.8 template<typename T > T CX::CX_RandomNumberGenerator::sample (const std::vector< T > & *values*)

Returns a single value sampled randomly from values.

Returns

The sampled value.

Note

If values.size() == 0, an error will be logged and T() will be returned.

14.24.3.9 template<typename T > std::vector< T > CX::CX_RandomNumberGenerator::sample (unsigned int *count*, const std::vector< T > & *source*, bool *withReplacement*)

Returns a vector of count values drawn randomly from source, with or without replacement. The returned values are in a random order.

Parameters

count	The number of samples to draw.
source	A vector to be sampled from.
withReplacement	Sample with or without replacement.

Returns

A vector of the sampled values.

Note

If (count > source.size() && withReplacement == false), an empty vector is returned.

14.24.3.10 std::vector< int > CX_RandomNumberGenerator::sample (unsigned int *count,* int *lowerBound,* int *upperBound,* bool *withReplacement*)

Returns a vector of count integers drawn randomly from the range [lowerBound, upperBound] with or without replacement.

Parameters

count	The number of samples to draw.
lowerBound	The lower bound of the range to sample from. It is possible to sample this value.
upperBound	The upper bound of the range to sample from. It is possible to sample this value.
withReplacement	Sample with or without replacement.

Returns

A vector of the samples.

14.24.3.11 std::vector< unsigned int > CX_RandomNumberGenerator::sampleBinomialRealizations (unsigned int *count,* unsigned int *trials,* double *probSuccess*)

Samples count realizations from a binomial distribution with the given number of trials and probability of success on each trial.

Parameters

count	The number of deviates to generate.
trials	The number of trials. Must be a non-negative integer.
probSuccess	The probability of a success on a given trial, where a success is the value 1.

Returns

A vector of the realizations.

14.24.3.12 template<typename T > std::vector< T > CX::CX_RandomNumberGenerator::sampleBlocks (const std::vector< T > & *values,* unsigned int *blocksToSample*)

This function helps with the case where a set of V values must be sampled randomly with the constraint that each block of V samples should have each value in the set. For example, if you want to present a number of trials in four different conditions, where the conditions are intermixed, but you want to observe all four trial types in every block of four trials, you would use this function.

Parameters

values	The set of values to sample from.
blocksToSample	The number of blocks to sample.

Returns

A vector with `valueSize.size() * blocksToSample` elements.

14.24.3.13 template<typename T > T CX::CX_RandomNumberGenerator::sampleExclusive (const std::vector< T > & *values,* const T & *exclude*)

Sample a random value from a vector, without the possibility of getting the excluded value.

Parameters

values	The vectors of values to sample from.
exclude	The value to exclude from sampling.

Returns

The sampled value.

Note

If all of the values are excluded, an error will be logged and T() will be returned.

14.24.3.14 template<typename T > T CX::CX_RandomNumberGenerator::sampleExclusive (const std::vector< T > & *values*, const std::vector< T > & *exclude*)

Sample a random value from a vector without the possibility of getting any of the excluded values.

Parameters

values	The vector of values to sample from.
exclude	The vector of values to exclude from sampling.

Returns

The sampled value.

Note

If all of the values are excluded, an error will be logged and T() will be returned.

14.24.3.15 std::vector< double > CX_RandomNumberGenerator::sampleNormalRealizations (unsigned int *count*, double *mean*, double *standardDeviation*)

Samples count realizations from a normal distribution with the given mean and standard deviation.

Parameters

count	The number of deviates to generate.
mean	The mean of the distribution.
standard-Deviation	The standard deviation of the distribution.

Returns

A vector of the realizations.

14.24.3.16 template<typename stdDist > std::vector< typename stdDist::result_type > CX::CX_RandomNumberGenerator-::sampleRealizations (unsigned int *count*, stdDist *dist*)

Draws `count` samples from a distribution `dist` that is provided by the user.

Parameters

count	The number of samples to take.
dist	A configured instance of a distribution class that has operator()(Generator& g), where Generator is a random number generator that has operator() that returns a random value. Basically, just look at this page: http://en.cppreference.com/w/cpp/numeric/random and pick one of the random number distributions.

Returns

A vector of stdDist::result_type, where stdDist::result_type is the type of data that is returned by the distribution (e.g. int, double, etc.). You can usually set this when creating the distribution object.

```
//Take 100 samples from a poisson distribution with lamda (mean result value) of 4.2.
//stdDist::result_type is unsigned int in this example.
vector<unsigned int> rpois = RNG.sampleFrom(100, std::poisson_distribution<unsigned int>(4.2));
```

14.24.3.17 std::vector< double > CX_RandomNumberGenerator::sampleUniformRealizations (unsigned int *count*, double *lowerBound_closed*, double *upperBound_open*)

Samples count deviates from a uniform distribution with the range [lowerBound_closed, upperBound_open).

Parameters

count	The number of deviates to generate.
lowerBound_closed	The lower bound of the distribution. This bound is closed, meaning that you can observe deviates with this value.
upperBound_open	The upper bound of the distribution. This bound is open, meaning that you cannot observe deviates with this value.

Returns

A vector of the realizations.

14.24.3.18 void CX_RandomNumberGenerator::setSeed (unsigned long *seed*)

Set the seed for the random number generator. You can retrieve the seed with getSeed().

Parameters

seed	The new seed.

14.24.3.19 template<typename T > void CX::CX_RandomNumberGenerator::shuffleVector (std::vector< T > * *v*)

Randomizes the order of the given vector.

Parameters

v	A pointer to the vector to be shuffled.

14.24.3.20 template<typename T > std::vector< T > CX::CX_RandomNumberGenerator::shuffleVector (std::vector< T > *v*)

Makes a copy of the given vector, randomizes the order of its elements, and returns the shuffled copy.

Parameters

v	The vector to be operated on.

Returns

A shuffled copy of v.

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

- CX_RandomNumberGenerator.h
- CX_RandomNumberGenerator.cpp

14.25 CX::Util::CX_SegmentProfiler Class Reference

`#include <CX_TimeUtilities.h>`

Public Member Functions

- **CX_SegmentProfiler** (CX_Clock *clock)
- void setup (CX_Clock *clock)
- void **t1** (void)
- void **t2** (void)
- std::vector< double >::size_type **collectedSamples** (void)
- void **reset** (void)
- std::string **getStatString** (void)
- CX_Millis **mean** (void)
- CX_Millis **min** (void)
- CX_Millis **max** (void)
- CX_Millis **stdDev** (void)

14.25.1 Detailed Description

This class is used for profiling small segments of code embedded within other code.

```
//During setup
CX::Util::CX_SegmentProfiler profiler(&
      CX::Instances::Clock);

//In main code somewhere you have a process that is repeated some number
//of times that has the code of interest embedded in it.
for (int i = 0; i < 100; i++) {
//Some code you aren't interested in profiling...

profiler.t1();
//Code you are interested in profiling.
profiler.t2();

//Other code you aren't interested in profiling...
}

//Once the process has been performed some number of times,
//check out the statistics for the code segment that was profiled.
std::cout << profiler.getStatString() << std::endl;
```

14.25.2 Member Function Documentation

14.25.2.1 void CX::Util::CX_SegmentProfiler::setup (CX_Clock * *clock*)

Set up the CX_SegmentProfiler with the given CX_Clock as the source for timing measurements.

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

- CX_TimeUtilities.h
- CX_TimeUtilities.cpp

14.26 CX::CX_SlidePresenter Class Reference

`#include <CX_SlidePresenter.h>`

Classes

- struct Configuration
- struct FinalSlideFunctionArgs
- struct PresentationErrorInfo
- struct Slide
- struct SlideTimingInfo

Public Types

- enum ErrorMode { **PROPAGATE_DELAYS** }

Public Member Functions

- bool setup (CX_Display *display)
- bool setup (const CX_SlidePresenter::Configuration &config)
- virtual void update (void)
- void appendSlide (CX_SlidePresenter::Slide slide)
- void appendSlideFunction (std::function< void(void)> drawingFunction, CX_Millis slideDuration, std::string slideName="")
- void beginDrawingNextSlide (CX_Millis slideDuration, std::string slideName="")
- void endDrawingCurrentSlide (void)
- bool startSlidePresentation (void)
- void stopSlidePresentation (void)

  Stops slide presentation.
- bool isPresentingSlides (void) const

  Returns true if slide presentation is in progress, even if the first slide has not yet been presented.
- void clearSlides (void)
- std::vector < CX_SlidePresenter::Slide > & getSlides (void)
- std::vector< CX_Millis > getActualPresentationDurations (void)
- std::vector< unsigned int > getActualFrameCounts (void)
- CX_SlidePresenter::PresentationErrorInfo checkForPresentationErrors (void) const
- std::string printLastPresentationInformation (void) const

Protected Member Functions

- unsigned int **_calculateFrameCount** (CX_Millis duration)
- void **_singleCoreBlockingUpdate** (void)
- void **_singleCoreThreadedUpdate** (void)
- void **_multiCoreUpdate** (void)
- void **_renderCurrentSlide** (void)
- void **_waitSyncCheck** (void)
- void **_finishPreviousSlide** (void)
- void **_handleFinalSlide** (void)
- void **_prepareNextSlide** (void)

Protected Attributes

- CX_SlidePresenter::Configuration **_config**
- CX_Millis **_hoggingStartTime**
- bool **_presentingSlides**
- bool **_synchronizing**
- unsigned int **_currentSlide**
- std::vector < CX_SlidePresenter::Slide > **_slides**
- std::vector< ExtraSlideInfo > **_slideInfo**
- bool **_lastFramebufferActive**
- bool **_useFenceSync**

14.26.1 Detailed Description

This class is a very useful abstraction that presents slides (typically a full display) of visual stimuli for fixed durations. See the changeDetection and nBack examples for the usage of this class.

A brief example:

```
CX_SlidePresenter slidePresenter;
slidePresenter.setup(&Display);

slidePresenter.beginDrawingNextSlide(2000 * 1000, "circle");
ofBackground(50);
ofSetColor(ofColor::red);
ofCircle(Display.getCenterOfDisplay(), 40);

slidePresenter.beginDrawingNextSlide(1000 * 1000, "rectangle");
ofBackground(50);
ofSetColor(ofColor::green);
ofRect(Display.getCenterOfDisplay() - ofPoint(100, 100), 200, 200);

slidePresenter.beginDrawingNextSlide(1, "off");
ofBackground(50);
slidePresenter.endDrawingCurrentSlide();

slidePresenter.startSlidePresentation();

//Update the slide presenter while waiting for slide presentation to complete
while (slidePresenter.isPresentingSlides()) {
    slidePresenter.update(); //You must remember to call update() regularly while slides are being
       presented!
}
```

14.26.2 Member Enumeration Documentation

14.26.2.1 enum CX::CX_SlidePresenter::ErrorMode [strong]

The settings in this enum are related to what a CX_SlidePresenter does when it encounters a timing error. Timing errors are probably almost exclusively related to one slide being presented for too long.

The PROPAGATE_DELAYS setting causes the slide presenter to handle these errors by moving the start time of all future stimuli back by the amount of extra time (or frames) used to the erroneous slide. This makes the durations of all future stimuli correct, so that there is only an error in the duration of one slide. If a slide's presentation start time is early, the intended start time is used (i.e. only delays, not early arrivals, are propogated).

Other alternatizes are being developed.

14.26.3 Member Function Documentation

14.26.3.1 void CX_SlidePresenter::appendSlide (CX_SlidePresenter::Slide *slide*)

Add a fully configured slide to the end of the list of slides. The user code must configure several components of the slide:

- If the framebuffer will be used, the framebuffer must be allocated and drawn to.
- If the drawing function will be used, a valid function pointer must be given. A check is made that either the drawing function is set or the framebuffer is allocated and an error is logged if neither is configured.
- The intended duration must be set.
- The name may be set (optional).

Parameters

slide	The slide to append.

14.26.3.2 void CX_SlidePresenter::appendSlideFunction (std::function< void(void)> *drawingFunction*, CX_Millis *slideDuration*, std::string *slideName* = " ")

Appends a slide to the slide presenter that will call the given drawing function when it comes time to render the slide to the back buffer. This approach has the advantage over using framebuffers that it takes essentially zero time to append a function to the list of slides, whereas a framebuffer must be allocated, which takes time. Additionally, because framebuffers must be allocated, they use video memory, so if you are using a very large number of slides, you could potentially run out of video memory. Also, when it comes time to draw the slide to the back buffer, it may be faster to draw directly to the back buffer than to copy an FBO to the fack buffer (although this depends on various factors).

Parameters

drawingFunction	A pointer to a function that will draw the slide to the back buffer. The contents of the back buffer are not cleared before this function is called, so the function must clear the background to the desired color.
slideDuration	The amount of time to present the slide for. If this is less than or equal to 0, the slide will be ignored.

slideName	The name of the slide. This can be anything and is purely for the user to use to help identify the slide.

Note

See Framebuffers and Buffer Swapping for more information about framebuffers.

One of the most tedius parts of using drawing functions is the fact that they can take no arguments. Here are two ways to get around that limitation using std::bind and function objects (functors):

```
#include "CX_EntryPoint.h"

CX_SlidePresenter SlidePresenter;

//This is the function we want to use to draw a stimulus, but it takes two arguments
void drawRectangle(ofRectangle r, ofColor col) {
    ofBackground(0);
    ofSetColor(col);
    ofRect(r);
}

struct rectFunctor {
    ofRectangle position;
    ofColor color;
    void operator() (void) {
        drawRectangle(position, color);
    }
};

void runExperiment(void) {

    SlidePresenter.setup(&Display);

    ofRectangle rectPos(100, 50, 100, 30);
    ofColor rectColor(255, 255, 0);

    //We can use std::bind to "bake in" the arguments rectPos and rectColor to drawRectangle. Because all
       of the
    //arguments for drawRectangle have been bound to it, it no longer takes any arguments and is a valid
    //drawing function to give to appendSlideFunction.
    SlidePresenter.appendSlideFunction(std::bind(drawRectangle, rectPos, rectColor), 2000.0, "bind rect");

    //We can also use a functor to sort of shift around where the arguments to the function come from. With
       a
    //functor, like rectFunctor, you can define an operator() that takes no arguments directly, but gets
       its
    //data from members of the structure like `position` and `color`. Because rectFunctor has operator(),
       it looks
    //like a function and can be called like a function, so you can use instances of it as drawing
       functions.
    rectFunctor rf;
    rf.position = ofRectangle(100, 100, 50, 80);
    rf.color = ofColor(0, 255, 0);
    SlidePresenter.appendSlideFunction(rf, 2000.0, "functor rect");

    SlidePresenter.startSlidePresentation();
    while (SlidePresenter.isPresentingSlides()) {
        SlidePresenter.update();
    }
}
```

14.26.3.3 void CX_SlidePresenter::beginDrawingNextSlide (CX_Millis *slideDuration*, std::string *slideName* = " ")

Prepares the framebuffer of the next slide for drawing so that any drawing commands given between a call to beginDrawingNextSlide() and endDrawingCurrentSlide() will cause stimuli to be drawn to the framebuffer of the slide.

Parameters

slideDuration	The amount of time to present the slide for. If this is less than or equal to 0, the slide will be ignored.
slideName	The name of the slide. This can be anything and is purely for the user to use to help identify the slide.

```
CX_SlidePresenter sp; //Assume that this has been set up.

sp.beginDrawingNextSlide(2000, "circles");
ofBackground(50);
ofSetColor(255, 0, 0);
ofCirlce(100, 100, 30);
ofCircle(210, 50, 20);
sp.endDrawingCurrentSlide();
```

14.26.3.4 CX_SlidePresenter::PresentationErrorInfo CX_SlidePresenter::checkForPresentationErrors (void) const

Checks the timing data from the last presentation of slides for presentation errors. Currently it checks to see if the intended frame count matches the actual frame count of each slide, which indicates if the duration was correct. It also checks to make sure that the framebuffer was copied to the back buffer before the onset of the slide. If not, vertical tearing might have occurred when the back buffer, containing a partially copied slide, was swapped in.

Returns

A struct with information about the errors that occurred on the last presentation of slides.

Note

If clearSlides() has been called since the end of the presentation, this does nothing as its data has been cleared. If this function is called during slide presentation, the returned struct will have the presentationErrorsSuccessfully-Checked member set to false and an error will be logged.

14.26.3.5 void CX_SlidePresenter::clearSlides (void)

Clears (deletes) all of the slides contained in the slide presenter and stops presentation, if it was in progress.

14.26.3.6 void CX_SlidePresenter::endDrawingCurrentSlide (void)

Ends drawing to the framebuffer of the slide that is currently being drawn to. See beginDrawingNextSlide().

14.26.3.7 std::vector< unsigned int > CX_SlidePresenter::getActualFrameCounts (void)

Gets a vector containing the number of frames that each of the slides from the last presentation of slides was presented for. Note that these frame counts may be wrong. If checkForPresentationErrors() not detect any errors, the frame counts are likely to be right, but there is no guarantee.

Returns

A vector containing the frame counts. The frame count corresponding to the first slide added to the slide presenter will be at index 0.

Note

The frame count of the last slide is meaningless. As far as the slide presenter is concerned, as soon as the last slide is put on the screen, it is done presenting the slides. Because the slide presenter is not responsible for removing the last slide from the screen, it has no idea about the duration of that slide.

14.26.3.8 std::vector< CX_Millis > CX_SlidePresenter::getActualPresentationDurations (void)

Gets a vector containing the durations of the slides from the last presentation of slides. Note that these durations may be wrong. If checkForPresentationErrors() does not detect any errors, the durations are likely to be right, but there is no guarantee.

Returns

A vector containing the durations. The duration corresponding to the first slide added to the slide presenter will be at index 0.

Note

The duration of the last slide is meaningless. As far as the slide presenter is concerned, as soon as the last slide is put on the screen, it is done presenting the slides. Because the slide presenter is not responsible for removing the last slide from the screen, it has no idea about the duration of that slide.

14.26.3.9 std::vector< CX_SlidePresenter::Slide > & CX_SlidePresenter::getSlides (void)

Get a reference to the vector of slides held by the slide presenter. If you modify any of the memebers of any of the slides, you do so at your own risk. This data is mostly useful in a read-only sort of way (when was that slide presented?).

Returns

A reference to the vector of slides.

14.26.3.10 std::string CX_SlidePresenter::printLastPresentationInformation (void) const

This function prints a ton of data relating to the last presentation of slides. It prints the total number of errors and the types of the errors. For each slide, it prints the slide index and name, and various information about the slide presentation timing. All of the printed information can also be accessed programmatically by using getSlides().

Returns

A string containing formatted presentation information.

14.26.3.11 bool CX_SlidePresenter::setup (CX_Display * *display*)

Set up the slide presenter with the given CX_Display as the display.

Parameters

display	Pointer to the display to use.

Returns

False if there was an error during setup, in which case a message will be logged.

14.26.3.12 bool CX_SlidePresenter::setup (const CX_SlidePresenter::Configuration & *config*)

Set up the slide presenter using the given configuration.

Parameters

config	The configuration to use.

Returns

False if there was an error during setup, in which case a message will be logged.

14.26.3.13 bool CX_SlidePresenter::startSlidePresentation (void)

Start presenting the slides that are stored in the slide presenter. After this function is called, calls to update() will advance the state of the slide presentation. If you do not call update(), nothing will be presented.

Returns

False if an error was encountered while starting presentation, in which case messages will be logged, true otherwise.

14.26.3.14 void CX_SlidePresenter::update (void) [virtual]

Updates the state of the slide presenter. If the slide presenter is presenting stimuli, update() must be called very regularly (at least once per millisecond) in order for the slide presenter to function. If slide presentation is stopped, you do not need to call update()

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

- CX_SlidePresenter.h
- CX_SlidePresenter.cpp

14.27 CX::CX_SoundBuffer Class Reference

`#include <CX_SoundBuffer.h>`

Public Member Functions

- bool loadFile (std::string fileName)
- bool addSound (std::string fileName, CX_Millis timeOffset)
- bool addSound (CX_SoundBuffer so, CX_Millis timeOffset)
- bool setFromVector (const std::vector< float > &data, int channels, float sampleRate)
- void clear (void)
- bool isReadyToPlay (void)
- bool isLoadedSuccessfully (void)
- bool applyGain (float gain, int channel=-1)
- bool multiplyAmplitudeBy (float amount, int channel=-1)
- void normalize (float amount=1.0)
- float getPositivePeak (void)
- float getNegativePeak (void)
- void setLength (CX_Millis length)
- CX_Millis getLength (void)
- void stripLeadingSilence (float tolerance)
- void addSilence (CX_Millis duration, bool atBeginning)
- void deleteAmount (CX_Millis duration, bool fromBeginning)

- void reverse (void)
- void multiplySpeed (float speedMultiplier)
- void resample (float newSampleRate)
- float getSampleRate (void)

  Returns the sample rate of the sound data stored in this CX_SoundBuffer.
- bool setChannelCount (int channels)
- int getChannelCount (void)

  Returns the number of channels in the sound data stored in this CX_SoundBuffer.
- uint64_t getTotalSampleCount (void)
- uint64_t getSampleFrameCount (void)
- std::vector< float > & getRawDataReference (void)
- bool writeToFile (std::string path)

Public Attributes

- std::string name

  This stores the name of the file from which data was read, if any. It can be set by the user with no side effects.

14.27.1 Detailed Description

This class is a container for a sound. It can load sound files, manipulate the contents of the sound data, add other sounds to an existing sound at specified offsets.

In order to play a CX_SoundBuffer, you use a CX::CX_SoundBufferPlayer. See the soundBuffer tutorial for an introduction on how to use this class along with a CX_SoundBufferPlayer.

To record from a microphone to a CX_SoundBuffer, you use a CX::CX_SoundBufferRecorder.

Note

Nearly all functions of this class should be considered Blocking Code. Many of the operations can take quite a while to complete because they are performed on a potentially large vector of sound samples.

14.27.2 Member Function Documentation

14.27.2.1 void CX_SoundBuffer::addSilence (CX_Millis *duration,* bool *atBeginning*)

Adds the specified amount of silence to the CX_SoundBuffer at either the beginning or end.

Parameters

duration	Duration of added silence in microseconds. Dependent on the sample rate of the sound. If the sample rate changes, so does the duration of silence.
atBeginning	If true, silence is added at the beginning of the CX_SoundBuffer. If false, the silence is added at the end.

14.27.2.2 bool CX_SoundBuffer::addSound (std::string *fileName,* CX_Millis *timeOffset*)

Uses loadFile(string) and addSound(CX_SoundBuffer, uint64_t) to add the given file to the current CX_SoundBuffer at the given time offset (in microseconds). See those functions for more information.

Parameters

fileName	Name of the sound file to load.
timeOffset	Time at which to add the new sound.

Returns

Returns true if the new sound was added sucessfully, false otherwise.

14.27.2.3 bool CX_SoundBuffer::addSound (CX_SoundBuffer *nsb,* CX_Millis *timeOffset*)

Adds the sound data in `nsb` at the time offset. If the sample rates of the sounds differ, nsb will be resampled to the sample rate of this CX_SoundBuffer. If the number of channels of nsb does not equal the number of channels of this, an attempt will be made to set the number of channels of nsb equal to the number of channels of this CX_SoundBuffer. The data from `nsb` and this CX_SoundBuffer are merged by adding the amplitudes of the sounds. The result of the addition is clamped between -1 and 1.

Parameters

nsb	A CX_SoundBuffer. Must be successfully loaded.
timeOffset	Time at which to add the new sound data in microseconds. Dependent on sample rate.

Returns

True if nsb was successfully added to this CX_SoundBuffer, false otherwise.

14.27.2.4 bool CX_SoundBuffer::applyGain (float *decibels,* int *channel* = `-1`)

Apply gain to the channel in terms of decibels.

Parameters

decibels	Gain to apply. 0 does nothing. Positive values increase volume, negative values decrease volume. Negative infinity is essentially mute, although see multiplyAmplitudeBy() for a more obvious way to do that same operation.
channel	The channel that the gain should be applied to. If channel is less than 0, the gain is applied to all channels.

14.27.2.5 void CX_SoundBuffer::clear (void)

Clears all data stored in the sound buffer and returns it to an uninitialized state.

14.27.2.6 void CX_SoundBuffer::deleteAmount (CX_Millis *duration,* bool *fromBeginning*)

Deletes the specified amount of sound from the CX_SoundBuffer from either the beginning or end.

Parameters

duration	Duration of removed sound in microseconds. If this is greater than the duration of the sound, the whole sound is deleted.
fromBeginning	If true, sound is deleted from the beginning of the CX_SoundBuffer's buffer. If false, the sound is deleted from the end, toward the beginning.

14.27.2.7 CX_Millis CX_SoundBuffer::getLength (void)

Gets the length, in time, of the data stored in the sound buffer. This depends on the sample rate of the sound.

Returns

The length.

14.27.2.8 float CX_SoundBuffer::getNegativePeak (void)

Finds the minimum amplitude in the sound buffer.

Returns

The minimum amplitude.

Note

Amplitudes are between -1 and 1, inclusive.

14.27.2.9 float CX_SoundBuffer::getPositivePeak (void)

Finds the maximum amplitude in the sound buffer.

Returns

The maximum amplitude.

Note

Amplitudes are between -1 and 1, inclusive.

14.27.2.10 std::vector<float>& CX::CX_SoundBuffer::getRawDataReference (void) [inline]

This function returns a reference to the raw data underlying the CX_SoundBuffer.

Returns

A reference to the data. Modify at your own risk!

14.27.2.11 uint64_t CX::CX_SoundBuffer::getSampleFrameCount (void) [inline]

This function returns the number of sample frames in the sound data held by the CX_SoundBuffer, which is equal to the total number of samples divided by the number of channels.

14.27.2.12 uint64_t CX::CX_SoundBuffer::getTotalSampleCount (void) [inline]

This function returns the total number of samples in the sound data held by the CX_SoundBuffer, which is equal to the number of sample frames times the number of channels.

14.27.2.13 bool CX::CX_SoundBuffer::isLoadedSuccessfully (void) [inline]

Checks to see if sound data has been successfully loaded into this CX_SoundBuffer from a file.

14.27.2.14 bool CX_SoundBuffer::isReadyToPlay (void)

Checks to see if the CX_SoundBuffer is ready to play. It basically just checks if there is sound data available and that the number of channels is set to a sane value.

14.27.2.15 bool CX_SoundBuffer::loadFile (std::string *fileName*)

Loads a sound file with the given file name into the CX_SoundBuffer. Any pre-existing data in the CX_SoundBuffer is deleted. Some sound file types are supported. Others are not. In the limited testing, mp3 and wav files seem to work well. If the file cannot be loaded, descriptive error messages will be logged.

Parameters

fileName	Name of the sound file to load.

Returns

True if the sound given in the fileName was loaded succesffuly, false otherwise.

14.27.2.16 bool CX_SoundBuffer::multiplyAmplitudeBy (float *amount*, int *channel* = -1)

Apply gain to the sound. The original value is simply multiplied by the amount and then clamped to be within [-1, 1].

Parameters

amount	The gain that should be applied. A value of 0 mutes the channel. 1 does nothing. 2 doubles the amplitude. -1 inverts the waveform.
channel	The channel that the given multiplier should be applied to. If channel is less than 0, the amplitude multiplier is applied to all channels.

14.27.2.17 void CX_SoundBuffer::multiplySpeed (float *speedMultiplier*)

This function changes the speed of the sound by some multiple.

Parameters

speedMultiplier	Amount to multiply the speed by. Must be greater than 0.

Note

If you would like to use a negative value to reverse the direction of playback, see reverse().

14.27.2.18 void CX_SoundBuffer::normalize (float *amount* = 1.0)

Normalizes the contents of the sound buffer.

Parameters

amount	Must be in the interval [0,1]. If 1, normalize will normalize in the standard way: The peak with the greatest magnitude will be set to +/-1 and everything else will be scaled relative to the peak. If amount is less than 1, the greatest peak will be set to that value.

14.27.2.19 void CX_SoundBuffer::resample (float *newSampleRate*)

Resamples the audio data stored in the CX_SoundBuffer by linear interpolation. Linear interpolation is not the ideal way to resample audio data; some audio fidelity is lost, more so than with other resampling techinques. It is, however, very fast compared to higher-quality methods both in terms of run time and programming time. It has acceptable results, at least when the new sample rate is similar to the old sample rate.

Parameters

newSampleRate	The requested sample rate.

14.27.2.20 void CX_SoundBuffer::reverse (void)

This function reverses the sound data stored in the CX_SoundBuffer so that if it is played, it will play in reverse.

14.27.2.21 bool CX_SoundBuffer::setChannelCount (int *newChannelCount*)

Sets the number of channels of the sound. Depending on the old number of channels (N) and the new number of channels (M), the conversion is performed in different ways. If N == M, nothing happens. If N == 1, each of the M new channels is set equal to the value of the single old channel. If M == 1, the new channel is set equal to the arithmetic average of the N old channels. If (N != 1 && M != 1 && M > N), the first N channels are preserved unchanged and the M - N new channels are set to the arithmetic average of the N old channels. Any other combination of M and N is an error condition.

Parameters

newChannel-Count	The number of channels the CX_SoundBuffer will have after the conversion.

Returns

True if the conversion was successful, false if the attempted conversion is unsupported.

14.27.2.22 bool CX_SoundBuffer::setFromVector (const std::vector< float > & *data,* int *channels,* float *sampleRate*)

Set the contents of the sound buffer from a vector of float data.

Parameters

data	A vector of sound samples. These values should go from -1 to 1. This requirement is not checked for. If there is more than once channel of data, the data must be interleaved. This means that if, for example, there are two channels, the ordering of the samples is 12121212... where 1 represents a sample for channel 1 and 2 represents a sample for channel 2. This requirement is not checked for. The number of samples in this vector must be evenly divisible by the number of channels set with the `channels` argument.
channels	The number of channels worth of data that is stored in `data`.
sampleRate	The sample rate of the samples. If `data` contains, for example, a sine wave, that wave was sampled at some rate (e.g. 48000 samples per second of waveform). `sampleRate` should be that rate. return True in all cases. No checking is done on any of the arguments.

14.27.2.23 void CX_SoundBuffer::setLength (CX_Millis *length*)

Set the length of the sound to the specified length in microseconds. If the new length is longer than the old length, the new data is zeroed (i.e. set to silence).

14.27.2.24 void CX_SoundBuffer::stripLeadingSilence (float *tolerance*)

Removes leading "silence" from the sound, where silence is defined by the given tolerance. It is unlikely that the beginning of a sound, even if perceived as silent relative to the rest of the sound, has an amplitude of 0. Therefore, a tolerance of 0 is unlikely to prove useful. Using getPositivePeak() and/or getNegativePeak() can help to give a reference amplitude of which some small fraction is perceived as "silent".

Parameters

tolerance	All sound data up to and including the first instance of a sample with an amplitude with an absolute value greater than or equal to tolerance is removed from the sound.

14.27.2.25 bool CX_SoundBuffer::writeToFile (std::string *filename*)

Writes the contents of the sound buffer to a file with the given file name. The data will be encoded as 16-bit PCM. The sample rate is determined by the sample rate of the sound buffer.

Parameters

filename	The name of the file to save the sound data to. `filename` should have a .wav extension. If it does not, ".wav" will be appended to the file name and a warning will be logged.

Returns

False if there was an error while opening the file. If so, an error will be logged.

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

- CX_SoundBuffer.h
- CX_SoundBuffer.cpp

14.28 CX::CX_SoundBufferPlayer Class Reference

`#include <CX_SoundBufferPlayer.h>`

Public Types

- typedef
  CX_SoundStream::Configuration Configuration

  This is typedef'ed to CX::CX_SoundStream::Configuration.

Public Member Functions

- bool setup (Configuration config)
- bool play (void)
- bool startPlayingAt (CX_Millis experimentTime, CX_Millis offset)
- bool stop (void)
- bool isPlaying (void)

  Check if the sound is currently playing.
- bool isQueuedToStart (void)

  Check if the sound is queued to play.
- Configuration getConfiguration (void)

  Returns the configuration used for this CX_SoundBufferPlayer.
- bool setSoundBuffer (CX_SoundBuffer *sound)
- CX_SoundBuffer * getSoundBuffer (void)
- CX_SoundStream & getSoundStream (void)
- void setTime (CX_Millis time)

14.28.1 Detailed Description

This class is used for playing CX_SoundBuffers. See the soundBuffer tutorial for an example of how to use this class.

14.28.2 Member Function Documentation

14.28.2.1 CX_SoundBuffer * CX_SoundBufferPlayer::getSoundBuffer (void)

This function provides direct access to the CX_SoundBuffer that is in use by the CX_SoundBufferPlayer.

14.28.2.2 CX_SoundStream& CX::CX_SoundBufferPlayer::getSoundStream (void) [inline]

This function provides direct access to the CX_SoundStream used by the CX_SoundBufferRecorder.

14.28.2.3 bool CX_SoundBufferPlayer::play (void)

Attempts to start playing the current CX_SoundBuffer associated with the player.

Returns

True if the sound buffer associated with the player isReadyToPlay(), false otherwise.

14.28.2.4 bool CX_SoundBufferPlayer::setSoundBuffer (CX_SoundBuffer * *sound*)

This function is potentially blocking because the sample rate and number of channels of sound are changed to those of the currently open stream if they do not already match (see Blocking Code).

Parameters

sound	A pointer to a CX_SoundBuffer that will be set as the current sound for the CX_SoundBufferPlayer. There are a variety of reasons why the sound could fail to be set as the current sound for the player. If sound was not loaded successfully, this function call fails and an error is logged. If it is not possible to convert the number of channels of sound to the number of channels that the CX_SoundBufferPlayer is configured to use, this function call fails and an error is logged.

This function call is not blocking if the same rate and channel count of the CX_SoundBuffer are the same as those in use by the CX_SoundBufferPlayer. See Blocking Code for more information.

Returns

True if sound was successfully set to be the current sound, false otherwise.

14.28.2.5 void CX_SoundBufferPlayer::setTime (CX_Millis *time*)

Set the current time in the active sound. When playback starts, it will begin from that time in the sound. If the sound buffer is currently playing, this will jump to that point in the sound.

Parameters

time	The time in the sound to seek to.

14.28.2.6 bool CX_SoundBufferPlayer::setup (Configuration *config*)

Configures the CX_SoundBufferPlayer with the given configuration.

14.28.2.7 bool CX_SoundBufferPlayer::startPlayingAt (CX_Millis *experimentTime,* CX_Millis *latencyOffset*)

Queue the start time of the sound in experiment time with an offset to account for latency.

Parameters

experimentTime	The desired experiment time at which the sound should start playing. This time plus the offset should be in the future. If it is not, the sound will start playing immediately.

latencyOffset	An offset that accounts for latency. If, for example, you called this function with an offset of 0 and discovered that the sound played 200 ms later than you were expecting it to, you would set offset to -200 in order to queue the start time 200 ms earlier than the desired experiment time.

Returns

False if the start time plus the offset is in the past. True otherwise.

Note

See setTime() for a way to choose the current time point within the sound.

14.28.2.8 bool CX_SoundBufferPlayer::stop (void)

Stop the currently playing sound buffer, or, if a playback start was cued, cancel the cued playback.

Returns

Always returns true currently.

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

- CX_SoundBufferPlayer.h
- CX_SoundBufferPlayer.cpp

14.29 CX::CX_SoundBufferRecorder Class Reference

`#include <CX_SoundBufferRecorder.h>`

Public Types

- typedef CX_SoundStream::Configuration Configuration

  This is typedef'ed to CX::CX_SoundStream::Configuration.

Public Member Functions

- bool setup (Configuration &config)
- Configuration getConfiguration (void)

  Returns the configuration used for the CX_SoundBufferRecorder.
- CX_SoundStream & getSoundStream (void)

  This function provides direct access to the CX_SoundStream used by the CX_SoundBufferRecorder.
- void setSoundBuffer (CX_SoundBuffer *so)
- CX_SoundBuffer * getSoundBuffer (void)
- void startRecording (bool clearExistingData=false)
- void stopRecording (void)

14.29.1 Detailed Description

This class is used for recording audio data from, e.g., a microphone. The recorded data is stored in a CX_SoundBuffer for further use.

```
CX_SoundBufferRecorder recorder;

CX_SoundBufferRecorder::Configuration recorderConfig;
recorderConfig.inputChannels = 1;
//You will probably need to configure more than just the number of input channels.
recorder.setup(recorderConfig);

CX_SoundBuffer recording;
recorder.setSoundBuffer(&recording); //Associate a CX_SoundBuffer with the recorder so that the buffer can
       be recorded to.

//Record for 5 seconds
recorder.startRecording();
Clock.sleep(CX_Seconds(5));
recorder.stopRecording();

//Write the recording to a file
recording.writeToFile("recording.wav");
```

14.29.2 Member Function Documentation

14.29.2.1 CX_SoundBuffer * CX_SoundBufferRecorder::getSoundBuffer (void)

This function returns a pointer to the CX_SoundBuffer that is currently in use by the CX_SoundBufferRecorder.

14.29.2.2 void CX_SoundBufferRecorder::setSoundBuffer (CX_SoundBuffer * *so*)

This function associates a CX_SoundBuffer with the CX_SoundBufferRecorder. The CX_SoundBuffer will be recorded to when startRecording() is called.

Parameters

so	The CX_SoundBuffer to associate with the CX_SoundBufferRecorder. The sound buffer will be cleared and it will be configured to have the same number of channels and sample rate that the CX_SoundBufferRecorder was configured to use.

14.29.2.3 bool CX_SoundBufferRecorder::setup (CX_SoundBufferRecorder::Configuration & *config*)

This function sets up the CX_SoundStream that CX_SoundBufferRecorder uses to record audio data.

Returns

True if configuration of the CX_SoundStream was successful, false otherwise.

14.29.2.4 void CX_SoundBufferRecorder::startRecording (bool *clearExistingData* = `false`)

Begins recording data to the CX_SoundBuffer that was associated with this CX_SoundBufferRecorder with setSoundBuffer().

Parameters

clearExistingData	If true, any data in the CX_SoundBuffer will be deleted before recording starts.

14.29.2.5 void CX_SoundBufferRecorder::stopRecording (void)

Stop recording sound data.

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

- CX_SoundBufferRecorder.h
- CX_SoundBufferRecorder.cpp

14.30 CX::CX_SoundStream Class Reference

`#include <CX_SoundStream.h>`

Classes

- struct Configuration
- struct InputEventArgs
- struct OutputEventArgs

Public Member Functions

- bool setup (CX_SoundStream::Configuration &config)
- bool closeStream (void)
- bool start (void)
- bool stop (void)
- bool isStreamRunning (void) const
- const CX_SoundStream::Configuration & getConfiguration (void) const
- uint64_t getSampleFrameNumber (void) const
- CX_Millis getStreamLatency (void)
- bool hasSwappedSinceLastCheck (void)
- CX_Millis getLastSwapTime (void)
- CX_Millis estimateNextSwapTime (void)
- RtAudio * getRtAudioInstance (void)

Static Public Member Functions

- static std::vector< RtAudio::Api > getCompiledApis (void)
- static std::vector< std::string > convertApisToStrings (vector< RtAudio::Api > apis)
- static std::string convertApisToString (vector< RtAudio::Api > apis, std::string delim="\r\n")
- static std::string convertApiToString (RtAudio::Api api)
- static RtAudio::Api convertStringToApi (std::string apiString)
- static std::vector< std::string > formatsToStrings (RtAudioFormat formats)
- static std::string formatsToString (RtAudioFormat formats, std::string delim="\r\n")
- static std::vector < RtAudio::DeviceInfo > getDeviceList (RtAudio::Api api)
- static std::string listDevices (RtAudio::Api api)
- static CX_SoundStream::Configuration readConfigurationFromFile (std::string filename, std::string delimiter="=", bool trimWhitespace=true, std::string commentStr="//")

Public Attributes

- ofEvent < CX_SoundStream::OutputEventArgs > outputEvent

  This event is triggered every time the CX_SoundStream needs to feed more data to the output buffer of the sound card.
- ofEvent < CX_SoundStream::InputEventArgs > inputEvent

  This event is triggered every time the CX_SoundStream hsa gotten some data from the input buffer of the sound card.

14.30.1 Detailed Description

This class provides a method for directly accessing and manipulating sound data that is sent/received from sound hardware. To use this class, you should set up the stream (see setup()), set a user function that will be called when either the outputEvent or inputEvent is triggered, and start the stream with start().

If the stream in configured for output, the output event will be triggered whenever the sound card needs more sound data. If the stream is configured for input, the input event will be triggered whenever some amount of sound data has been recorded.

CX_SoundStream uses RtAudio internally, so you are having problems, you might be able to figure out what is going wrong by checking out the page for RtAudio: http://www.music.mcgill.ca/~gary/rtaudio/index.-html

14.30.2 Member Function Documentation

14.30.2.1 bool CX_SoundStream::closeStream (void)

Closes the sound stream.

Returns

False if an error was encountered while closing the stream, true otherwise.

14.30.2.2 std::string CX_SoundStream::convertApisToString (vector< RtAudio::Api > *apis*, std::string *delim* = "\r\n") [static]

This helper function converts a vector of RtAudio::Api to a string, with the specified delimiter between API names.

Parameters

apis	The vector of RtAudio::Api to convert to string.
delim	The delimiter between elements of the string.

Returns

A string containing the names of the APIs.

14.30.2.3 std::vector< std::string > CX_SoundStream::convertApisToStrings (vector< RtAudio::Api > *apis*) [static]

This helper function converts a vector of RtAudio::Api to a vector of strings, using convertApiToString() for the conversion.

Parameters

apis	A vector of apis to convert to strings.

Returns

A vector of string names of the apis.

14.30.2.4 std::string CX_SoundStream::convertApiToString (RtAudio::Api *api*) [static]

This helper function converts an RtAudio::Api to a string.

Parameters

api	The api to get a string of.

Returns

A string of the api name.

14.30.2.5 RtAudio::Api CX_SoundStream::convertStringToApi (std::string *apiString*) [static]

Converts a string name of an RtAudio API to an RtAudio::Api enum constant.

Parameters

apiString	The name of the API as a string. Should be one of the following, with no surrounding whitespace- : UNSPECIFIED, LINUX_ALSA, LINUX_PULSE, LINUX_OSS, UNIX_JACK, MACOSX_CORE, WINDOWS_ASIO, WINDOWS_DS, RTAUDIO_DUMMY

Returns

The RtAudio::Api corresponding to the provided string. If the string is not one of the above values, RtAudio::Api::- UNSPECIFIED is returned.

14.30.2.6 CX_Millis CX_SoundStream::estimateNextSwapTime (void)

Estimate the time at which the next buffer swap will occur.

Returns

The estimated time of next swap. This value can be compared with the result of CX::Instances::Clock.now().

14.30.2.7 std::string CX_SoundStream::formatsToString (RtAudioFormat *formats*, std::string *delim* = "\r\n") [static]

Converts a bitmask of audio formats to a string, with each format delimited by `delim`.

Parameters

formats	The bitmask of audio formats.
delim	The delimiter.

Returns

A string containing string representations of the valid formats in `formats`.

14.30.2.8 std::vector< std::string > CX_SoundStream::formatsToStrings (RtAudioFormat *formats*) [static]

Converts a bitmask of audio formats to a vector of strings.

Parameters

formats	The bitmask of audio formats.

Returns

A vector of strings, one string for each bit set in formats for which there is a corresponding valid audio format that RtAudio supports.

14.30.2.9 std::vector< RtAudio::Api > CX_SoundStream::getCompiledApis (void) [static]

Get a vector containing a list of all of the APIs for which the RtAudio driver has been compiled to use. If the API you want is not available, you might be able to get it by using a different version of RtAudio.

14.30.2.10 const CX_SoundStream::Configuration& CX::CX_SoundStream::getConfiguration (void) const [inline]

Gets the configuration that was used on the last call to open(). Because some of the configuration options are only suggestions, this function allows you to check what the actual configuration was.

Returns

A const reference to the configuration struct.

14.30.2.11 std::vector< RtAudio::DeviceInfo > CX_SoundStream::getDeviceList (RtAudio::Api *api*) [static]

For the given api, lists all of the devices on the system that support that api.

Parameters

api	Devices that support this API are scanned.

Returns

A machine-readable list of information. See http://www.music.mcgill.ca/~gary/rtaudio/struct-RtAudio_1_1DeviceInfo.html for information about the members of the RtAudio::DeviceInfo struct.

14.30.2.12 CX_Millis CX::CX_SoundStream::getLastSwapTime (void) [inline]

Gets the time at which the last buffer swap occurred.

Returns

This time value can be compared with the result of CX::CX_Clock::now().

14.30.2.13 RtAudio * CX_SoundStream::getRtAudioInstance (void)

This function returns a pointer to the RtAudio instance that this CX_SoundStream is using. This should not be needed most of the time, but there may be cases in which you need to directly access RtAudio. Here is the documentation for RtAudio: https://www.music.mcgill.ca/~gary/rtaudio/

14.30.2.14 uint64_t CX::CX_SoundStream::getSampleFrameNumber (void) const [inline]

Returns the number of the sample frame that is about to be loaded into the stream buffer on the next buffer swap.

14.30.2.15 CX_Millis CX_SoundStream::getStreamLatency (void)

This function gets an estimate of the stream latency. However, it should not be relied on as it is based on what the sound card driver reports, which is often false.

Returns

The stream latency in microseconds.

14.30.2.16 bool CX_SoundStream::hasSwappedSinceLastCheck (void)

This function checks to see if the audio buffers have been swapped since the last time this function was called.

Returns

True if at least one audio buffer has been swapped out, false if no buffers have been swapped.

14.30.2.17 bool CX_SoundStream::isStreamRunning (void) const

Check whether the sound stream is running.

Returns

false if the stream is not open or not running or if RtAudio has not been initialized. Returns true if the stream is running.

14.30.2.18 std::string CX_SoundStream::listDevices (RtAudio::Api *api*) [static]

For the given api, lists all of the devices on the system that support that api. Lots of information about each device is given, like supported sample rates, number of input and output channels, etc.

Parameters

api	Devices that support this API are scanned.

Returns

A human-readable formatted string containing the scanned information. Can be printed directly to std::cout or elsewhere.

14.30.2.19 CX_SoundStream::Configuration CX_SoundStream::readConfigurationFromFile (std::string *filename*, std::string *delimiter* = "=", bool *trimWhitespace* = true, std::string *commentString* = "//") [static]

This function exists to serve a per-computer configuration function that is otherwise difficult to provide due to the fact that C++ programs are compiled to binaries and cannot be easily edited on the computer on which they are running. This function takes the file name of a specially constructed configuration file and reads the key-value pairs in that file in order to fill a CX_SoundStream::Configuration struct. The format of the file is provided in the example code below.

Sample configuration file:

```
ss.api = WINDOWS_DS //See convertStringToApi() for valid API names.
ss.sampleRate = 44100
ss.bufferSize = 512
ss.inputChannels = 0
//ss.inputDeviceId not used because no channels are used
ss.outputChannels = 2
ss.outputDeviceId = 0
ss.streamOptions.numberOfBuffers = 4
ss.streamOptions.flags = RTAUDIO_SCHEDULE_REALTIME | RTAUDIO_MINIMIZE_LATENCY //The | is not needed,
//but it matches the way these flags are used in code.
//ss.streamOptions.priority is not used.
```

All of the configuration keys are used in this example. Any values in the CX_SoundStream::Configuration struct that do not have values provided in the configuration file will be left at default values. Note that the "ss" prefix allows this configuration to be embedded in a file that also performs other configuration functions. Note that the names of the data members match the names used in the CX_SoundStream::Configuration struct and have a 1-to-1 relationship with those values.

Because this function uses CX::Util::readKeyValueFile() internally, it has the same arguments.

Parameters

filename	The name of the file containing configuration data.
delimiter	The string that separates the key from the value. In the example, it is "=", but can be other values.
trimWhitespace	If true, whitespace characters surrounding both the key and value will be removed. This is a good idea to do.
commentString	If commentString is not the empty string (i.e. ""), everything on a line following the first instance of commentString will be ignored.

14.30.2.20 bool CX_SoundStream::setup (CX_SoundStream::Configuration & *config*)

Opens the sound stream with the specified configuration. If there was an error during configuration, messages will be logged.

Parameters

config	The configuration settings that are desired. Some of the configuration options are only suggestions, so some of the values that are used may differ from the values that are chosen. In those cases, config is updated based on the actually used settings. You can check the configuration later using getConfiguration().

Returns

True if configuration appeared to be successful, false otherwise.

Note

Opening the stream does not start it. See start().

14.30.2.21 bool CX_SoundStream::start (void)

Starts the sound stream. The stream must already be open().

Returns

False if the stream was not started, true if the stream was started or if it was already running.

14.30.2.22 bool CX_SoundStream::stop (void)

Stop the stream, if is running. If there is an error, a message will be logged.

Returns

False if there was an error, true otherwise.

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

- CX_SoundStream.h
- CX_SoundStream.cpp

14.31 CX::CX_Time_t< T > Class Template Reference

`#include <CX_Time_t.h>`

Public Member Functions

- PartitionedTime getPartitionedTime (void)
- **CX_Time_t** (double t)
- **CX_Time_t** (int t)
- **CX_Time_t** (long long t)
- template<typename tArg >
  CX_Time_t (const CX_Time_t< tArg > &t)
- double value (void) const
- double hours (void) const

  Get the time stored by this CX_Time_t in hours, including fractional hours.
- double minutes (void) const

  Get the time stored by this CX_Time_t in minutes, including fractional minutes.
- double seconds (void) const

  Get the time stored by this CX_Time_t in seconds, including fractional seconds.
- double millis (void) const

  Get the time stored by this CX_Time_t in milliseconds, including fractional milliseconds.
- double micros (void) const

  Get the time stored by this CX_Time_t in microseconds, including fractional microseconds.
- long long nanos (void) const

  Get the time stored by this CX_Time_t in nanoseconds.
- template<typename RT >
  CX_Time_t< TimeUnit > **operator+** (const CX_Time_t< RT > &rhs) const
- template<typename RT >
  CX_Time_t< TimeUnit > **operator-** (const CX_Time_t< RT > &rhs) const
- template<typename RT >
  double operator/ (const CX_Time_t< RT > &rhs) const

  Divides a CX_Time_t by another CX_Time_t, resulting in a unitless ratio.
- CX_Time_t< TimeUnit > operator/ (double rhs) const

  Divides a CX_Time_t by a unitless value, resulting in a CX_Time_t of the same type.
- CX_Time_t< TimeUnit > operator* (double rhs) const

  Multiplies a CX_Time_t by a unitless value, resulting in a CX_Time_t of the same type. You cannot multiply a time by another time because that would result in units of time squared.
- CX_Time_t< TimeUnit > & operator*= (double rhs)

  Multiplies a CX_Time_t by a unitless value, storing the result in the CX_Time_t. You cannot multiply a time by another time because that would result in units of time squared.
- template<typename RT >
  CX_Time_t< TimeUnit > & **operator+=** (const CX_Time_t< RT > &rhs)
- template<typename RT >
  CX_Time_t< TimeUnit > & **operator-=** (const CX_Time_t< RT > &rhs)
- template<typename RT >
  bool **operator<** (const CX_Time_t< RT > &rhs) const
- template<typename RT >
  bool **operator<=** (const CX_Time_t< RT > &rhs) const
- template<typename RT >
  bool **operator>** (const CX_Time_t< RT > &rhs) const

- template<typename RT >
  bool **operator>=** (const CX_Time_t< RT > &rhs) const
- template<typename RT >
  bool **operator==** (const CX_Time_t< RT > &rhs) const
- template<typename RT >
  bool **operator!=** (const CX_Time_t< RT > &rhs) const

Static Public Member Functions

- static CX_Time_t< TimeUnit > **min** (void)
- static CX_Time_t< TimeUnit > **max** (void)
- static CX_Time_t< TimeUnit > standardDeviation (std::vector< CX_Time_t< TimeUnit >> vals)

14.31.1 Detailed Description

template<typename T>class CX::CX_Time_t< T >

This class provides a convenient way to deal with time in various units. The upside of this system is that although all functions in CX that take time can take time values in a variety of units. For example, CX_Clock::wait() takes CX_Millis as the time type so if you were to do

```
Clock.wait(20);
```

it would attempt to wait for 20 milliseconds. However, you could do

```
Clock.wait(CX_Seconds(.5));
```

to wait for half of a second, if units of seconds are easier to think in for the given situation.

CX_Time_t has at most nanosecond accuracy. The contents of any of the templated versions of CX_Time_t are all stored in nanoseconds, so conversion between time types is lossless.

See this example for a varity of things you can do with this class.

```
CX_Millis mil = 100;
CX_Micros mic = mil; //mic now contains 100000 microseconds == 100 milliseconds.
//Really, they both contain 100,000,000 nanoseconds.

//You can add times together.
CX_Seconds sec = CX_Minutes(.1) + CX_Millis(100); //sec contains 6.1 seconds.

//You can take the ratio of times.
double secondsPerMinute = CX_Seconds(1)/CX_Minutes(1);

//You can compare times using the standard comparison operators (==, !=, <, >, <=, >=).
if (CX_Minutes(60) == CX_Hours(1)) {
cout << "There are 60 minutes in an hour." << endl;
}

if (CX_Millis(12.3456) == CX_Micros(12345.6)) {
cout << "Time can be represented as a floating point value with sub-time-unit precision." << endl;
}

//If you want to be explicit about what time unit you want out, you can use the seconds(), millis(), etc.,
       functions:
sec = CX_Seconds(6);
cout << "In " << sec.seconds() << " seconds there are " << sec.millis() << " milliseconds and " << sec.
      minutes() << " minutes." << endl;

//You can alternately do a typecast if you're about to print the result:
cout << "In " << sec << " seconds there are " << (CX_Millis)sec << " milliseconds and " << (CX_Minutes)sec
      << " minutes." << endl;
```

```
//The difference between the above examples is the resulting type.
//double minutes = (CX_Minutes)sec; //This does not work: A CX_Minutes cannot be assigned to a double
double minutes = sec.minutes(); //minutes() returns a double.

//You can construct a time with the result of the construction of a time object with a different time unit.
CX_Minutes min = CX_Hours(.05); //3 minutes

//You can get the whole number amounts of different time units.
CX_Seconds longTime = CX_Hours(2) + CX_Minutes(16) + CX_Seconds(40) + CX_Millis(123) + CX_Micros(456) +
      CX_Nanos(1);
CX_Seconds::PartitionedTime parts = longTime.getPartitionedTime();
```

14.31.2 Member Function Documentation

14.31.2.1 template<typename T> PartitionedTime CX::CX_Time_t< T >::getPartitionedTime (void) [inline]

Partitions a CX_Time_t into component parts containing the number of whole time units that are stored in the CX_Time_t. This is different from seconds(), millis(), etc., because those functions return the fractional part (e.g. 5.340 seconds) whereas this returns only whole numbers (e.g. 5 seconds and 340 milliseconds).

14.31.2.2 template<typename T> static CX_Time_t<TimeUnit> CX::CX_Time_t< T >::standardDeviation (std::vector< CX_Time_t< TimeUnit >> *vals*) [inline], [static]

This function calculates the sample standard deviation for a vector of time values.

14.31.2.3 template<typename T> double CX::CX_Time_t< T >::value (void) const [inline]

Get the numerical value of the time in units of the time type. For example, if you are using an instance of CX_Seconds, this will return the time value in seconds, including fractional seconds.

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

- CX_Time_t.h

14.32 CX::Util::CX_TrialController Class Reference

`#include <CX_TrialController.h>`

Public Member Functions

- int update (void)
- void start (void)

  "Arm" the trial controller. Before this is called, update() will do nothing.
- void stop (void)

  "Disarm" the trial controller. After this is called, update() will do nothing.
- bool isActive (void)

  Check to see if the trial controller is active. See start() and stop().
- void appendFunction (std::function< int(void)> userFunction)
- void reset (void)
- bool setCurrentFunction (int currentFunction)
- int getCurrentFunction (void)

  Get the index of the current function (i.e. the function that will be called the next time update() is called).
- unsigned int getFunctionCount (void)

  Get the number of user functions stored by this trial controller.

14.32.1 Detailed Description

This class is used to help with the fact that most psychology experiments are by nature more or less linear, but that CX works better if code does not block (see Blocking Code).

The way this works is that segments of user code, each representing one part of a trial, are put into functions. Those functions are added to the trial controller with appendFunction(), which puts the function at the end of the list of functions. User functions take no arguments and return an int.

When you want to use the trial controller, call start() and it will be "armed". When it is armed and update() is called, it will call the current function in the list of user functions. If the user function is done with whatever it needs to do, it should return 1. This will cause the trial controller to move on to the next user function. If the user function is not done with its task, it should return 0. If it returns 0, it will be called again the next time update() is called.

14.32.2 Member Function Documentation

14.32.2.1 void CX_TrialController::appendFunction (std::function< int(void)> *userFunction*)

Adds a user function to the end of the list of functions to be called by the trial controller.

Parameters

userFunction	Typically a pointer to a function that takes no arguments and returns an int. Because it is a std::function, it can also be a lamda.

14.32.2.2 void CX_TrialController::reset (void)

Clear the user functions and otherwise reset to default state.

Note

This function stops the trial controller (isActive() will return false).

14.32.2.3 bool CX_TrialController::setCurrentFunction (int *currentFunction*)

Sets the current user function by index, which allows you to skip over functions or go back to a previous function.

Parameters

currentFunction	The new current function index. If this is out of range, an error will be logged and the function will return false.

Returns

False if the index was out of range, true otherwise.

Note

If this is called from within a user function that has been called from this instance of the CX_TrialController, that function should return 0. If it does not, setCurrentFunction() will set the function index and then that index will be incremented after the user function completes. However, if 0 is returned, the function index is not incremented.

14.32.2.4 int CX_TrialController::update (void)

Updates the trial controller state. Each time this function is called, the user function at the current function index is called. If that function returns a nonzero value, the trial controller will increment the current function index and that

function will be called the next time update() is called. If the current function index is incremented past the end of the list of functions, it will wrap around to the beginning of the list.

Returns

The value returned by the user function that was called.

Note

This function should probably be called every time updateExperiment() is called, although there are other use cases.
If not isActive(), this function does nothing and returns 0.

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

- CX_TrialController.h
- CX_TrialController.cpp

14.33 CX::CX_WindowConfiguration_t Struct Reference

`#include <CX_EntryPoint.h>`

Public Attributes

- ofWindowMode **mode**
- int **width**
- int **height**
- unsigned int **msaaSampleCount**
- ofPtr< ofBaseGLRenderer > **desiredRenderer**
- Private::CX_GLVersion **desiredOpenGLVersion**
- std::string **windowTitle**

14.33.1 Detailed Description

This structure is used to configure windows opened with CX::reopenWindow().

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

- CX_EntryPoint.h

14.34 CX::Synth::Envelope Class Reference

`#include <CX_ModularSynth.h>`

Inherits CX::Synth::ModuleBase.

Public Member Functions

- double getNextSample (void) override
- void **attack** (void)
- void **release** (void)

Public Attributes

- ModuleParameter **gateInput**
- double **a**
- double **d**
- double **s**
- double **r**

Additional Inherited Members

14.34.1 Detailed Description

This class is a standard ADSR envelope: http://en.wikipedia.org/wiki/Synthesizer#ADSR_envelope. `s` should be in the interval [0,1]. `a`, `d`, and `r` are expressed in seconds. Call attack() to start the envelope. Once the attack and decay are finished, the envelope will stay at the sustain level until release() is called.

The output values produced start at 0, rise to 1 during the attack, drop to the sustain level (`s`) during the decay, and drop from `s` to 0 during the release.

14.34.2 Member Function Documentation

14.34.2.1 double Envelope::getNextSample (void) [override], [virtual]

This function should be overloaded for any derived class that can be used as the input for another module.

Reimplemented from CX::Synth::ModuleBase.

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

- CX_ModularSynth.h
- CX_ModularSynth.cpp

14.35 CX::CX_Mouse::Event Struct Reference

`#include <CX_Mouse.h>`

Public Types

- enum MouseEventType {
  MOVED, PRESSED, RELEASED, DRAGGED,
  SCROLLED }

Public Attributes

- int button

  The relevant mouse button if the eventType is PRESSED, RELEASED, or DRAGGED. Can be compared with elements of enum CX_MouseButtons to find out about the primary buttons.
- int x

  The x position of the cursor at the time of the event, or the change in the x-axis scroll if the eventType is SCROLLED.
- int y

  The y position of the cursor at the time of the event, or the change in the y-axis scroll if the eventType is SCROLLED.

- CX_Millis eventTime

  The time at which the event was registered. Can be compared to the result of CX::Clock::now().
- CX_Millis uncertainty

  The uncertainty in eventTime. The event occured some time between eventTime and eventTime minus uncertainty.
- enum CX::CX_Mouse::Event::MouseEventType eventType

  The type of the event.

14.35.1 Detailed Description

This struct contains the results of a mouse event, which is any type of interaction with the mouse, be it simply movement, a button press or release, a drag event (mouse button held while mouse is moved), or movement of the scroll wheel.

14.35.2 Member Enumeration Documentation

14.35.2.1 enum CX::CX_Mouse::Event::MouseEventType

Enumerator

MOVED The mouse has been moved without a button being held. button should be -1 (meaningless).

PRESSED A mouse button has been pressed. Check button for the button index and x and y for the location.

RELEASED A mouse button has been released. Check button for the button index and x and y for the location.

DRAGGED can be changed during a drag, or multiple buttons may be held at once during a drag. The mouse has been moved while at least one button was held. button may not be meaningful because the held button

SCROLLED mouse has a wheel that can move horizontally. The mouse wheel has been scrolled. Check y to get the change in the standard mouse wheel, or x if your

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

- CX_Mouse.h

14.36 CX::CX_Joystick::Event Struct Reference

`#include <CX_Joystick.h>`

Public Types

- enum JoystickEventType { BUTTON_PRESS, BUTTON_RELEASE, AXIS_POSITION_CHANGE }

Public Attributes

- int buttonIndex

  If eventType is BUTTON_PRESS or BUTTON_RELEASE, this contains the index of the button that was changed.
- unsigned char buttonState

  If eventType is BUTTON_PRESS or BUTTON_RELEASE, this contains the current state of the button.
- int axisIndex

  If eventType is AXIS_POSITION_CHANGE, this contains the index of the axis which changed.
- float axisPosition

If eventType is AXIS_POSITION_CHANGE, this contains the amount by which the axis changed.

- CX_Millis eventTime

  The time at which the event was registered. Can be compared to the result of CX::CX_Clock::now().

- CX_Millis uncertainty

  The uncertainty in eventTime. The event occured some time between eventTime and eventTime minus uncertainty.

- enum CX::CX_Joystick::Event::JoystickEventType eventType

  The type of the event.

14.36.1 Detailed Description

This struct contains information about joystick events. Joystick events are either a button press or release or a change in the axes of the joystick.

14.36.2 Member Enumeration Documentation

14.36.2.1 enum CX::CX_Joystick::Event::JoystickEventType

Enumerator

BUTTON_PRESS A button on the joystick has been pressed. See buttonIndex and buttonState for the event data.

BUTTON_RELEASE A button on the joystick has been released. See buttonIndex and buttonState for the event data.

AXIS_POSITION_CHANGE The joystick has been moved in one of its axes. See axisIndex and axisPosition for the event data.

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

- CX_Joystick.h

14.37 CX::CX_Keyboard::Event Struct Reference

`#include <CX_Keyboard.h>`

Public Types

- enum KeyboardEventType { PRESSED, RELEASED, REPEAT }

Public Attributes

- int key
- CX_Millis eventTime

  The time at which the event was registered. Can be compared to the result of CX::CX_Clock::now().

- CX_Millis uncertainty

  The uncertainty in eventTime. The event occured some time between eventTime and eventTime minus uncertainty.

- enum CX::CX_Keyboard::Event::KeyboardEventType eventType

  The type of the event.

14.37.1 Detailed Description

This struct contains the results of a keyboard event, whether it be a key press or release, or key repeat.

14.37.2 Member Enumeration Documentation

14.37.2.1 enum CX::CX_Keyboard::Event::KeyboardEventType

Enumerator

PRESSED A key has been pressed.

RELEASED A key has been released.

REPEAT A key has been held for some time and automatic key repeat has kicked in, causing multiple keypresses to be rapidly sent. This event is one of the many repeats.

14.37.3 Member Data Documentation

14.37.3.1 int CX::CX_Keyboard::Event::key

The value created by the key involved in this event. The value of this can be compared with character literals for many of the standard keyboard keys. For example, you could use '(myKeyEvent.key == 'e')' to test if the key was the E key.

For special keys, `key` can be compared with the key constant values defined in ofConstants.h (e.g. `OF_KEY_ESC`).

Note that the modifier keys (shift, ctrl, alt, and super) are treated a little unusually. For those keys, you can check for a specific key using, for example, `OF_KEY_RIGHT_CONTROL` or `OF_KEY_LEFT_CONTROL`. However, you can alternately check to see if `key` is either of the control keys by performing a bitwise AND (`&`) with `OF_KEY_CONTROL` and checking that the result of the AND is still `OF_KEY_CONTROL`. For example:

```
if ((myKeyEvent.key & OF_KEY_CONTROL) == OF_KEY_CONTROL) {
//...
}
```

This works the same way for all of the modifier keys.

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

- CX_Keyboard.h

14.38 CX::Synth::Filter Class Reference

`#include <CX_ModularSynth.h>`

Inherits CX::Synth::ModuleBase.

Public Types

- enum FilterType { **LOW_PASS**, **HIGH_PASS**, **BAND_PASS**, **NOTCH** }

Public Member Functions

- void setType (FilterType type)

  Set the type of filter to use, from the Filter::FilterType enum.
- double getNextSample (void) override

Public Attributes

- ModuleParameter cutoff
- ModuleParameter bandwidth

Additional Inherited Members

14.38.1 Detailed Description

This class provides a basic way to filter waveforms as part of subtractive synthesis or other audio manipulation.

This class is based on simple IIR filters. They may not be stable at all frequencies. They are computationally very efficient. They are not highly configurable. They may be chained for sharper frequency response. This class is based on this chapter: http://www.dspguide.com/ch19.htm.

14.38.2 Member Enumeration Documentation

14.38.2.1 enum CX::Synth::Filter::FilterType

The type of filter to use.

14.38.3 Member Function Documentation

14.38.3.1 double Filter::getNextSample (void) [override], [virtual]

This function should be overloaded for any derived class that can be used as the input for another module.

Reimplemented from CX::Synth::ModuleBase.

14.38.4 Member Data Documentation

14.38.4.1 ModuleParameter CX::Synth::Filter::bandwidth

Only used for BAND_PASS and NOTCH FilterTypes. Sets the width (in frequency domain) of the stop or pass band at which the amplitude is equal to sin(PI/4) (i.e. .707). So, for example, if you wanted the frequencies 100 Hz above and below the breakpoint to be at .707 of the maximum amplitude, set bw to 100. Of course, past those frequencies the attenuation continues. Larger values result in a less pointy band.

14.38.4.2 ModuleParameter CX::Synth::Filter::cutoff

The cutoff frequency of the filter.

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

- CX_ModularSynth.h
- CX_ModularSynth.cpp

14.39 CX::CX_SlidePresenter::FinalSlideFunctionArgs Struct Reference

#include <CX_SlidePresenter.h>

Public Attributes

- CX_SlidePresenter * instance

  A pointer to the CX_SlidePresenter that called the user function.
- unsigned int currentSlideIndex

  The index of the slide that is currently being presented.

14.39.1 Detailed Description

The final slide function takes a reference to a struct of this type.

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

- CX_SlidePresenter.h

14.40 CX::Synth::FIRFilter Class Reference

`#include <CX_ModularSynth.h>`

Inherits CX::Synth::ModuleBase.

Public Types

- enum FilterType { **LOW_PASS**, **HIGH_PASS**, **FIR_USER_DEFINED** }
- enum **WindowType** { **RECTANGULAR**, **HANNING**, **BLACKMAN** }

Public Member Functions

- void **setup** (FilterType filterType, unsigned int coefficientCount)
- void **setup** (std::vector< double > coefficients)
- void **setCutoff** (double cutoff)
- double getNextSample (void)

Additional Inherited Members

14.40.1 Detailed Description

This class is a start at implementing a Finite Impulse Response filter (http://en.wikipedia.org/wiki/-Finite_impulse_response). You can use it as a basic low-pass or high-pass filter, or, if you supply your own coefficients, which cause the filter to do filtering in whatever way you want. See the "signal" package for R for a method of constructing your own coefficients.

14.40.2 Member Enumeration Documentation

14.40.2.1 enum CX::Synth::FIRFilter::FilterType [strong]

The type of filter to use.

14.40.3 Member Function Documentation

14.40.3.1 double CX::Synth::FIRFilter::getNextSample (void) [inline], [virtual]

This function should be overloaded for any derived class that can be used as the input for another module.

Reimplemented from CX::Synth::ModuleBase.

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

- CX_ModularSynth.h

14.41 CX::CX_SoundStream::InputEventArgs Struct Reference

`#include <CX_SoundStream.h>`

Public Attributes

- bool bufferOverflow

  This is set to true if there was a buffer overflow, which means that the sound hardware recorded data that was not processed.
- float * inputBuffer

  A pointer to an array of sound data that should be processed by the event handler function.
- unsigned int bufferSize

  The number of sample frames that are in `inputBuffer`. The total number of samples is `bufferSize * inputChannels`.
- int inputChannels

  The number of channels worth of data in `inputBuffer`.
- CX_SoundStream * instance

  A pointer to the CX_SoundStream instance that notified this input event.

14.41.1 Detailed Description

The audio input event of the CX_SoundStream sends a copy of this structure with the fields filled out when the event is called.

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

- CX_SoundStream.h

14.42 CX::Algo::LatinSquare Class Reference

`#include <CX_Algorithm.h>`

Public Member Functions

- **LatinSquare** (unsigned int dimensions)
- void generate (unsigned int dimensions)
- void reorderRight (void)
- void reorderLeft (void)

- void **reorderUp** (void)
- void **reorderDown** (void)
- void reverseColumns (void)
- void reverseRows (void)
- void swapColumns (unsigned int c1, unsigned int c2)
- void swapRows (unsigned int r1, unsigned int r2)
- bool appendRight (const LatinSquare &ls)
- bool appendBelow (const LatinSquare &ls)
- LatinSquare & operator+= (unsigned int value)
- std::string print (std::string delim=",")
- bool validate (void) const
- unsigned int columns (void) const
- unsigned int rows (void) const
- std::vector< unsigned int > getColumn (unsigned int col) const
- std::vector< unsigned int > getRow (unsigned int row) const

Public Attributes

- std::vector< std::vector < unsigned int > > **square**

14.42.1 Detailed Description

This class provides a way to work with Latin squares in a relatively easy way.

```
Algo::LatinSquare ls(4);
cout << "This latin square has " << ls.rows() << " rows and " << ls.columns() << " columns." << endl;
cout << ls.print() << endl;

ls.reverseColumns();
cout << "Reverse the columns: " << endl << ls.print() << endl;

ls.swapRows(0, 2);
cout << "Swap rows 0 and 2: " << endl << ls.print() << endl;

if (ls.validate()) {
    cout << "The latin square is still a valid latin square." << endl;
}

cout << "Let's copy, reverse, and append a latin square." << endl;
Algo::LatinSquare sq = ls;
sq.reverseColumns();
ls.appendBelow(sq);

cout << ls.print() << endl;
if (!ls.validate()) {
    cout << "The latin square is no longer valid, but it is still useful (8 counterbalancing conditions,
       both forward and backward ordering)." << endl;
}
```

14.42.2 Member Function Documentation

14.42.2.1 bool LatinSquare::appendBelow (const LatinSquare & *ls*)

Appends another LatinSquare (ls) below of this one. If the number of columns of both latin squares is not equal, this has no effect and returns false.

14.42.2.2 bool LatinSquare::appendRight (const LatinSquare & *ls*)

Appends another LatinSquare (ls) to the right of this one. If the number of rows of both latin squares is not equal, this has no effect and returns false.

14.42.2.3 unsigned int LatinSquare::columns (void) const

Returns the number of columns.

14.42.2.4 void LatinSquare::generate (unsigned int *dimensions*)

Generate a latin square with the given dimensions. The generated square is the very basic latin square that, for dimension 3, has {0,1,2} on the first row, {1,2,0} on the middle row, and {2,0,1} on the last row.

Note

This deletes any previous contents of the latin square.

14.42.2.5 std::vector< unsigned int > LatinSquare::getColumn (unsigned int *col*) const

Returns a copy of the given column. Throws std::out_of_range if the column is out of range.

14.42.2.6 std::vector< unsigned int > LatinSquare::getRow (unsigned int *row*) const

Returns a copy of the given row. Throws std::out_of_range if the row is out of range.

14.42.2.7 LatinSquare & LatinSquare::operator+= (unsigned int *value*)

Adds the given value to all of the values in the latin square.

14.42.2.8 std::string LatinSquare::print (std::string *delim* = " , ")

Prints the contents of the latin square to a string with the given delimiter between elements of the latin square.

14.42.2.9 void LatinSquare::reorderLeft (void)

This function shifts the columns to the left and the first column is moved to be the last column.

14.42.2.10 void LatinSquare::reorderRight (void)

This function shifts the columns to the right and the last column is moved to be the first column.

14.42.2.11 void LatinSquare::reverseColumns (void)

Reverses the order of the columns in the latin square.

14.42.2.12 void LatinSquare::reverseRows (void)

Reverses the order of the rows in the latin square.

14.42.2.13 unsigned int LatinSquare::rows (void) const

Returns the number of rows.

14.42.2.14 void LatinSquare::swapColumns (unsigned int *c1*, unsigned int *c2*)

Swap the given columns. If either column is out of range, this function has no effect.

14.42.2.15 void LatinSquare::swapRows (unsigned int *r1*, unsigned int *r2*)

Swap the given rows. If either row is out of range, this function has no effect.

14.42.2.16 bool LatinSquare::validate (void) const

Checks to make sure that the latin square held by this instance is a valid latin square.

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

- CX_Algorithm.h
- CX_Algorithm.cpp

14.43 CX::Synth::Mixer Class Reference

`#include <CX_ModularSynth.h>`

Inherits CX::Synth::ModuleBase.

Public Member Functions

- double getNextSample (void) override

Additional Inherited Members

14.43.1 Detailed Description

This class mixes together a number of inputs. It does no mixing in the usual sense of setting levels of the inputs. Use Multipliers on the inputs for that. This class simply adds together all of the inputs with no amplitude correction, so it is possible for the output of the mixer to have very large amplitudes.

This class is special in that it can have more than one input.

14.43.2 Member Function Documentation

14.43.2.1 double Mixer::getNextSample (void) [override], [virtual]

This function should be overloaded for any derived class that can be used as the input for another module.

Reimplemented from CX::Synth::ModuleBase.

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

- CX_ModularSynth.h
- CX_ModularSynth.cpp

14.44 CX::Synth::ModuleBase Class Reference

`#include <CX_ModularSynth.h>`

Inherited by CX::Synth::Adder, CX::Synth::AdditiveSynth, CX::Synth::Clamper, CX::Synth::Envelope, CX::Synth::Filter, CX::Synth::FIRFilter, CX::Synth::GenericOutput, CX::Synth::Mixer, CX::Synth::Multiplier, CX::Synth::Oscillator, CX::Synth::SoundBufferInput, CX::Synth::SoundBufferOutput, CX::Synth::Splitter, CX::Synth::StreamOutput, and CX::Synth::TrivialGenerator.

Public Member Functions

- virtual double getNextSample (void)
- void setData (ModuleControlData_t d)
- ModuleControlData_t **getData** (void)
- void disconnectInput (ModuleBase *in)
- void disconnectOutput (ModuleBase *out)

Protected Member Functions

- virtual void **_dataSetEvent** (void)
- void **_dataSet** (ModuleBase *caller)
- void **_setDataIfNotSet** (ModuleBase *target)
- void **_registerParameter** (ModuleParameter *p)
- virtual void **_assignInput** (ModuleBase *in)
- virtual void **_assignOutput** (ModuleBase *out)
- virtual int **_maxInputs** (void)
- virtual int **_maxOutputs** (void)
- virtual void **_inputAssignedEvent** (ModuleBase *in)
- virtual void **_outputAssignedEvent** (ModuleBase *out)

Protected Attributes

- vector< ModuleBase * > **_inputs**
- vector< ModuleBase * > **_outputs**
- vector< ModuleParameter * > **_parameters**
- ModuleControlData_t * **_data**

Friends

- ModuleBase & operator>> (ModuleBase &l, ModuleBase &r)

14.44.1 Detailed Description

All modules of the modular synth inherit from this class.

14.44.2 Member Function Documentation

14.44.2.1 void ModuleBase::disconnectInput (ModuleBase * *in*)

This is a reciprocal operation: This module's input is disconnected and `in`'s output to this module is disconnected.

14.44.2.2 void ModuleBase::disconnectOutput (ModuleBase * *out*)

This is a reciprocal operation: This module's output is disconnected and `out`'s input from this module is disconnected.

14.44.2.3 virtual double CX::Synth::ModuleBase::getNextSample (void) [inline], [virtual]

This function should be overloaded for any derived class that can be used as the input for another module.

Reimplemented in CX::Synth::FIRFilter, CX::Synth::SoundBufferInput, CX::Synth::Splitter, CX::Synth::Oscillator, CX::Synth::Multiplier, CX::Synth::Mixer, CX::Synth::Filter, CX::Synth::Envelope, CX::Synth::Clamper, CX::Synth::Adder, and CX::Synth::AdditiveSynth.

14.44.2.4 void CX::Synth::ModuleBase::setData (ModuleControlData_t *d*) [inline]

This function sets the data needed by this module in order to function properly. Many modules need this data, specifically the sample rate that the synth using. If several modules are connected together, you will only need to set the data for one module and the change will propagate to the other connected modules automatically.

This function does not usually need to be called driectly by the user. If an appropriate input or output is connected, the data will be set from that module. However, there are some cases where a pattern of reconnecting previously used modules may result in inappropriate sample rates being set. For that reason, if you are having a problem with seeing the correct sample rate after reconnecting some modules, try manually calling setData().

Parameters

d	The data to set.

14.44.3 Friends And Related Function Documentation

14.44.3.1 ModuleBase& operator>> (ModuleBase & *l*, ModuleBase & *r*) [friend]

This operator is used to connect modules together. `l` is set as the input for `r`.

```
Oscillator osc;
StreamOutput out;
osc >> out; //Connect osc as the input for out.
```

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

- CX_ModularSynth.h
- CX_ModularSynth.cpp

14.45 CX::Synth::Multiplier Class Reference

`#include <CX_ModularSynth.h>`

Inherits CX::Synth::ModuleBase.

Public Member Functions

- double getNextSample (void) override
- void setGain (double decibels)

Public Attributes

- ModuleParameter **amount**

Additional Inherited Members

14.45.1 Detailed Description

This class multiplies an input by an `amount`. You can set the amount in terms of decibels of gain by using the setGain() function. If there is no input to this module, it behaves as though the input was 0 and consequently outputs 0.

14.45.2 Member Function Documentation

14.45.2.1 double Multiplier::getNextSample (void) [override], [virtual]

This function should be overloaded for any derived class that can be used as the input for another module.

Reimplemented from CX::Synth::ModuleBase.

14.45.2.2 void Multiplier::setGain (double *decibels*)

Sets the `amount` of the multiplier based on gain in decibels.

Parameters

decibels	The gain to apply. If greater than 0, `amount` will be greater than 1. If less than 0, `amount` will be less than 1. After calling this function, `amount` will never be negative.

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

- CX_ModularSynth.h
- CX_ModularSynth.cpp

14.46 CX::Synth::Oscillator Class Reference

`#include <CX_ModularSynth.h>`

Inherits CX::Synth::ModuleBase.

Public Member Functions

- double getNextSample (void) override
- void setGeneratorFunction (std::function< double(double)> f)

Static Public Member Functions

- static double **saw** (double wp)
- static double **sine** (double wp)
- static double **square** (double wp)
- static double **triangle** (double wp)
- static double **whiteNoise** (double wp)

Public Attributes

- ModuleParameter **frequency**

Additional Inherited Members

14.46.1 Detailed Description

This class provides one of the simplest ways of generating waveforms. The output from an Oscillator can be filtered with a CX::Synth::Filter or used in other ways.

```
using namespace CX::Synth;
//Configure the oscillator to produce a square wave with a fundamental frequency of 200 Hz.
Oscillator osc;
osc.frequency = 200; //200 Hz
osc.setGeneratorFunction(Oscillator::square); //Produce a square wave
```

14.46.2 Member Function Documentation

14.46.2.1 double Oscillator::getNextSample (void) [override], [virtual]

This function should be overloaded for any derived class that can be used as the input for another module.

Reimplemented from CX::Synth::ModuleBase.

14.46.2.2 void Oscillator::setGeneratorFunction (std::function< double(double) > *f*)

It is very easy to make your own waveform generating functions to be used with an Oscillator. A waveform generating function takes a value that represents the location in the waveform at the current point in time. These values are in the interval [0,1).

The waveform generating function should return a double representing the amplitude of the wave at the given waveform position.

To put this all together, a sine wave generator looks like this:

```
double sineWaveGeneratorFunction(double waveformPosition) {
    return sin(2 * PI * waveformPosition); //The argument for sin() is in radians. 1 cycle is 2*PI radians.
}
```

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

- CX_ModularSynth.h
- CX_ModularSynth.cpp

14.47 CX::CX_SoundStream::OutputEventArgs Struct Reference

`#include <CX_SoundStream.h>`

Public Attributes

- bool bufferUnderflow

  This is set to true if there was a buffer underflow, which means that the sound hardware ran out of data to output.
- float * outputBuffer

  A pointer to an array that should be filled with sound data.
- unsigned int bufferSize

  The number of sample frames that are in `outputBuffer`. The total number of samples is `bufferSize * outputChannels`.

- int outputChannels

  The number of channels worth of data in `outputBuffer`.
- CX_SoundStream * instance

  A pointer to the CX_SoundStream instance that notified this output event.

14.47.1 Detailed Description

The audio output event of the CX_SoundStream sends a copy of this structure with the fields filled out when the event is called.

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

- CX_SoundStream.h

14.48 CX::CX_SlidePresenter::PresentationErrorInfo Struct Reference

`#include <CX_SlidePresenter.h>`

Public Member Functions

- unsigned int totalErrors (void)

  Returns the sum of the different types of errors that are measured.

Public Attributes

- bool presentationErrorsSuccessfullyChecked

  True if presentation errors were successfully checked for. This does not mean that there were no presentation errors, but that there were no presentation error checking errors.
- unsigned int incorrectFrameCounts
- unsigned int lateCopiesToBackBuffer

  The number of slides for which the time at which the slide finished being copied to the back buffer was after the actual start time of the slide.

14.48.1 Detailed Description

This struct contains information about errors that were detected during slide presentation. See CX_SlidePresenter::checkForPresentationErrors().

14.48.2 Member Data Documentation

14.48.2.1 unsigned int CX::CX_SlidePresenter::PresentationErrorInfo::incorrectFrameCounts

The number of slides for which the actual and intended frame counts did not match, indicating that the slide was presented for too many or too few frames.

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

- CX_SlidePresenter.h

14.49 CX::CX_SlidePresenter::Slide Struct Reference

`#include <CX_SlidePresenter.h>`

Public Types

- enum {
  NOT_STARTED, **COPY_TO_BACK_BUFFER_PENDING**, **SWAP_PENDING**, **IN_PROGRESS**, **FINISHED** }

  Status of the current slide vis a vis presentation. This should not be modified by the user.

Public Attributes

- std::string slideName

  The name of the slide. Set by the user during slide creation.
- ofFbo framebuffer

  A framebuffer containing image data that will be drawn to the screen during this slide's presentation. If drawingFunction points to a user function, framebuffer will not be drawn.
- std::function< void(void)> drawingFunction

  Pointer to a user function that will be called to draw the slide. If this points to a user function, it overrides `framebuffer`. The drawing function is not required to call ofBackground() or otherwise clear the display before drawing, which allows you to do what is essentially single-buffering using the back buffer as the framebuffer. However, if you want a blank framebuffer, you will have to clear it manually.
- enum
  CX::CX_SlidePresenter::Slide:: { ... } slideStatus

  Status of the current slide vis a vis presentation. This should not be modified by the user.
- SlideTimingInfo intended

  The intended timing parameters (i.e. what should have happened if there were no presentation errors).
- SlideTimingInfo actual

  The actual timing parameters.
- CX_Millis copyToBackBufferCompleteTime

  The time at which the drawing operations for this slide finished. This is pretty useful to determine if there was an error on the trial (e.g. framebuffer was copied late). If this is greater than actual.startTime, the slide may not have been fully drawn at the time the front and back buffers swapped.

14.49.1 Detailed Description

This struct contains information related to slide presentation using CX_SlidePresenter.

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

- CX_SlidePresenter.h

14.50 CX::CX_SlidePresenter::SlideTimingInfo Struct Reference

`#include <CX_SlidePresenter.h>`

Public Attributes

- uint32_t startFrame

  The frame on which the slide started/should have started. Can be compared with the value given by Display.getFrameNumber().
- uint32_t frameCount

  The number of frames the slide was/should be presented for.
- CX_Millis startTime

  The time at which the slide was/should have been started. Can be compared with values from CX::CX_Clock::now().
- CX_Millis duration

  The amount of time the slide was/should have been presented for.

14.50.1 Detailed Description

Contains information about the presentation timing of the slide.

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

- CX_SlidePresenter.h

14.51 CX::Synth::SoundBufferInput Class Reference

`#include <CX_ModularSynth.h>`

Inherits CX::Synth::ModuleBase.

Public Member Functions

- double getNextSample (void) override
- void setSoundBuffer (CX::CX_SoundBuffer *sb, unsigned int channel=0)
- void setTime (CX_Millis t)
- bool canPlay (void)

Additional Inherited Members

14.51.1 Detailed Description

This class allows you to use a CX_SoundBuffer as the input for the modular synth. It is strictly monophonic, so when you associate a CX_SoundBuffer with this class, you must pick one channel of the sound to use. You can use multiple SoundBufferInputs to play multiple channels from the same CX_SoundBuffer.

14.51.2 Member Function Documentation

14.51.2.1 bool SoundBufferInput::canPlay (void)

Checks to see if the CX_SoundBuffer that is associated with this SoundBufferInput is able to play. It is unable to play if CX_SoundBuffer::isReadyToPlay() is false or if the whole sound has been played.

14.51.2.2 double SoundBufferInput::getNextSample (void) [override], [virtual]

This function should be overloaded for any derived class that can be used as the input for another module.

Reimplemented from CX::Synth::ModuleBase.

14.51.2.3 void SoundBufferInput::setSoundBuffer (CX::CX_SoundBuffer * *sb*, unsigned int *channel* = 0)

This function sets the CX_SoundBuffer from which data will be drawn. Because the SoundBufferInput is monophonic, you must pick one channel of the CX_SoundBuffer to use.

Parameters

sb	The CX_SoundBuffer to use. Because this CX_SoundBuffer is taken as a pointer and is not copied, you should make sure that `sb` remains in existence and unmodified while the SoundBufferInput is in use.
channel	The channel of the CX_SoundBuffer to use.

14.51.2.4 void SoundBufferInput::setTime (CX::CX_Millis *t*)

Set the playback time of the current CX_SoundBuffer. When playback starts, it will start from this time. If playback is in progress, playback will skip to the selected time.

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

- CX_ModularSynth.h
- CX_ModularSynth.cpp

14.52 CX::Synth::SoundBufferOutput Class Reference

`#include <CX_ModularSynth.h>`

Inherits CX::Synth::ModuleBase.

Public Member Functions

- void setup (float sampleRate)
- void sampleData (CX_Millis t)

Public Attributes

- CX::CX_SoundBuffer **sb**

Additional Inherited Members

14.52.1 Detailed Description

This class provides a method of capturing the output of a modular synth and storing it in a CX_SoundBuffer for later use.

14.52.2 Member Function Documentation

14.52.2.1 void SoundBufferOutput::sampleData (CX::CX_Millis *t*)

This function samples `t` milliseconds of data at the sample rate given in setup(). The result is stored in the `sb` member of this class. If `sb` is not empty when this function is called, the data is appended to `sb`.

14.52.2.2 void SoundBufferOutput::setup (float *sampleRate*)

Configure the output to use a particular sample rate. If this function is not called, the sample rate of the modular synth may be undefined.

Parameters

sampleRate	The sample rate in Hz.

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

- CX_ModularSynth.h
- CX_ModularSynth.cpp

14.53 CX::Synth::Splitter Class Reference

`#include <CX_ModularSynth.h>`

Inherits CX::Synth::ModuleBase.

Public Member Functions

- double getNextSample (void) override

Additional Inherited Members

14.53.1 Detailed Description

This class splits a signal and sends that signal to multiple outputs. This can be used for panning effects, for example.

This class is special because it allows multiple outputs.

```
using namespace CX::Synth;
Splitter sp;
Oscillator osc;
Multiplier m1;
Multiplier m2;
StereoStreamOutput out;

osc >> sp;
sp >> m1 >> out.left;
sp >> m2 >> out.right;
```

14.53.2 Member Function Documentation

14.53.2.1 double Splitter::getNextSample (void) [override], [virtual]

This function should be overloaded for any derived class that can be used as the input for another module.

Reimplemented from CX::Synth::ModuleBase.

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

- CX_ModularSynth.h
- CX_ModularSynth.cpp

14.54 CX::Synth::StereoSoundBufferOutput Class Reference

`#include <CX_ModularSynth.h>`

Public Member Functions

- void setup (float sampleRate)
- void sampleData (CX_Millis t)

Public Attributes

- GenericOutput **left**
- GenericOutput **right**
- CX::CX_SoundBuffer **sb**

14.54.1 Detailed Description

This class provides a method of capturing the output of a modular synth and storing it in a CX_SoundBuffer for later use. This captures stereo audio by taking the output of different streams of data into either the `left` or `right` modules that this class has. See the example code.

```
using namespace CX::Synth;
StereoSoundBufferOutput sout;
sout.setup(44100);

Splitter sp;
Oscillator osc;
Multiplier leftM;
Multiplier rightM;

leftM.amount = .1;
rightM.amount = .01;

osc >> sp;
sp >> leftM >> sout.left;
sp >> rightM >> sout.right;

sout.sampleData(CX_Seconds(2)); //Sample 2 seconds worth of data on both channels.
```

14.54.2 Member Function Documentation

14.54.2.1 void StereoSoundBufferOutput::sampleData (CX::CX_Millis *t*)

This function samples `t` milliseconds of data at the sample rate given in setup(). The result is stored in the `sb` member of this class. If `sb` is not empty when this function is called, the data is appended to `sb`.

14.54.2.2 void StereoSoundBufferOutput::setup (float *sampleRate*)

Configure the output to use a particular sample rate. If this function is not called, the sample rate of the modular synth may be undefined.

Parameters

sampleRate	The sample rate in Hz.

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

- CX_ModularSynth.h
- CX_ModularSynth.cpp

14.55 CX::Synth::StereoStreamOutput Class Reference

`#include <CX_ModularSynth.h>`

Public Member Functions

- void **setOuputStream** (CX::CX_SoundStream &stream)

Public Attributes

- GenericOutput **left**
- GenericOutput **right**

14.55.1 Detailed Description

This class is much like StreamOutput except in stereo. This captures stereo audio by taking the output of different streams of data into either the `left` or `right` modules that this class has. See the example code for CX::Synth::StereoSoundBufferOutput and CX::Synth::StreamOutput for ideas on how to use this class.

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

- CX_ModularSynth.h
- CX_ModularSynth.cpp

14.56 CX::Synth::StreamOutput Class Reference

`#include <CX_ModularSynth.h>`

Inherits CX::Synth::ModuleBase.

Public Member Functions

- void **setOuputStream** (CX::CX_SoundStream &stream)

Additional Inherited Members

14.56.1 Detailed Description

This class provides a method of playing the output of a modular synth using a CX_SoundStream. In order to use this class, you need to configure a CX_SoundStream for use. See the soundBuffer example and the CX::CX_SoundStream class for more information.

```
using namespace CX::Synth;
//Assume that both osc and ss have been configured and the sound stream has been started.
CX_SoundStream ss;
Oscillator osc;

Synth::StreamOutput output;
output.setOutputStream(ss);

osc >> output; //Sound should be playing past this point.
```

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

- CX_ModularSynth.h
- CX_ModularSynth.cpp

15 File Documentation

15.1 advancedChangeDetection.cpp File Reference

```
#include "CX_EntryPoint.h"
```

Functions

- void **drawStimuli** (void)
- void **presentStimuli** (void)
- void **getResponse** (void)
- void **generateTrials** (int trialCount)
- void **updateExperiment** (void)
- void **drawFixation** (void)
- void **drawBlank** (void)
- void **drawSampleArray** (void)
- void **drawTestArray** (void)
- ofColor **backgroundColor** (50)
- void runExperiment (void)

Variables

- CX_SlidePresenter **SlidePresenter**
- CX_DataFrame **trialDf**
- int **trialIndex** = 0
- int **circleRadius** = 0

15.1.1 Detailed Description

This example is a more advanced version of the change detection task presented in the basicChangeDetection example. It is not "advanced" because it is more complex, but because it uses more features of CX. It actually ends up being simpler because of how it uses features of CX.

Items that are commented are new, although not all new stuff will neccessarily be commented. The main feature that as demonstrated as the CX_DataFrame, which is a way to store and output experimental data. Using custom units and a custom coordinate system is shown with CX_CoordinateConverter and CX_DegreeToPixelConverter as the unit converter.

15.2 advancedNBack.cpp File Reference

```
#include "CX_EntryPoint.h"
```

Functions

- ofColor **backgroundColor** (50)
- ofColor **textColor** (255)
- void **finalSlideFunction** (CX_SlidePresenter::FinalSlideFunctionArgs &info)
- void **drawStimuliToFramebuffers** (CX_SlidePresenter &sp, int trialIndex)
- void **appendDrawingFunctions** (CX_SlidePresenter &sp, int trialIndex)
- void **drawStimulus** (string letter, bool showInstructions)
- void **drawBlank** (void)
- void **drawFixationSlide** (int remainingTime)
- void **generateTrials** (int numberOfTrials)
- void runExperiment (void)

Variables

- CX_DataFrame **df**
- CX_DataFrame::rowIndex_t **trialNumber** = 0
- int **trialCount** = 40
- int **lastSlideIndex** = 0
- int **nBack** = 2
- ofTrueTypeFont **letterFont**
- ofTrueTypeFont **instructionFont**
- char **targetKey** = 'f'
- char **nonTargetKey** = 'j'
- string **keyReminderInstructions**
- CX_Millis **stimulusPresentationDuration** = 1000
- CX_Millis **interStimulusInterval** = 800
- CX_SlidePresenter **SlidePresenter**
- bool **useFramebuffersForStimuli** = false
- vector< stimulusFunctor > **stimulusFunctors**

15.2.1 Detailed Description

In this example, we are going to consider two different ways of using the slide presenter to present stimuli. One method is the one used in the change detection example: Draw the stimuli to framebuffers that are managed by the SlidePresenter. This approach is fairly easy to do, but it had a time cost, because allocating and copying all of the framebuffers takes time. This example builds on the nBack example.

In this example, we are going to contrast using the framebuffer approach with the slide presenter with the use of drawing functions. The standard framebuffer approach has the following major steps:

1) Allocate the framebuffer 2) Draw your stimuli to the famebuffer 3) Copy the framebuffer to the back buffer 4) Swap front and back buffers

Using drawing functions, we avoid steps 1 and 3, which are generally both very costly in terms of time. Step 2 becomes "Draw your stimuli to the back buffer directly". For an N-back task, there are no indefinitely-long pauses where you can

prepare stimuli: you have to get the next sitmulus ready within a fixed interval, so using drawing functions may be the best approach. We are going to examine that issue with this example.

The way in which you use a drawing function with a CX_SlidePresenter is with the appendSlideFunction() function. This function takes three arguments: the drawing function, the duration for which the stimuli drawn by that drawing function should be presented, and the name of the slide (optional). When the stimuli specified by the drawing function are ready to be presented, the drawing function will be called. In the drawing function, you do not need to call Display.-beginDrawingToBackBuffer(); the slide presenter does that for you. See drawStimulus() below for an example of what a drawing function might look like.

We can compare the performance of the framebuffer and functor approaches by examining how much time is used for various processes. One thing to consider is the amount of time it takes to allocate the framebuffers and render your stimuli to the framebuffers.

Another consideration is the amount of time it takes to copy a framebuffer to the back buffer (step 3 above). This may take longer than drawing a small amount of stimuli directly to the back buffer. The framebuffer approach provides a nice security blanket: "I know that regardless of whatever is the the framebuffer, it will be copied in the same amount of time as any other framebuffer." However, if that copying time is longer than any of the stimulus drawing times, then using framebuffers is less efficient than drawing directly to the back buffer.

We will be using a special kind of function object, known as a functor (http://www.cprogramming.-com/tutorial/functors-function-objects-in-c++.html) to do our drawing. A functor is basically a structure that can be called like a function using operator(). But unlike a normal function, a functor carries data along with it that can be used in the function call. Thus, a fuctor is a way to have a function without certain arguments for which you can still specify those arguments (in a sense) by setting data members of the functor. See struct stimulusFunctor below for an implementation.

15.3 basicChangeDetection.cpp File Reference

```
#include "CX_EntryPoint.h"
```

Functions

- void **updateExperiment** (void)
- vector< TrialData_t > **generateTrials** (int trialCount)
- void **outputData** (void)
- void **drawStimuli** (void)
- void **presentStimuli** (void)
- void **getResponse** (void)
- void **drawFixation** (void)
- void **drawBlank** (void)
- void **drawSampleArray** (const TrialData_t &tr)
- void **drawTestArray** (const TrialData_t &tr)
- ofColor **backgroundColor** (50)
- void runExperiment (void)

Variables

- CX_SlidePresenter **SlidePresenter**
- vector< TrialData_t > **trials**
- int **trialIndex** = 0
- int **circleRadius** = 30

15.3.1 Detailed Description

This example shows how to do a simple change-detection experiment using CX. The stimuli are colored circles which are presented in a 3X3 matrix.

Press the S key to indicate that you think the test array is the same or the D key to indicate that you think the test array is different.

15.4 basicNBack.cpp File Reference

```
#include "CX_EntryPoint.h"
```

Functions

- ofColor **backgroundColor** (50)
- ofColor **textColor** (255)
- void **finalSlideFunction** (CX_SlidePresenter::FinalSlideFunctionArgs &info)
- void **drawStimulusForTrial** (unsigned int trial, bool showInstructions)
- void **generateTrials** (int numberOfTrials)
- void runExperiment (void)

Variables

- CX_DataFrame **df**
- CX_DataFrame::rowIndex_t **trialNumber** = 0
- int **trialCount** = 40
- int **lastSlideIndex** = 0
- int **nBack** = 2
- ofTrueTypeFont **letterFont**
- ofTrueTypeFont **instructionFont**
- char **targetKey** = 'f'
- char **nonTargetKey** = 'j'
- string **keyReminderInstructions**
- CX_Millis **stimulusPresentationDuration** = 1000
- CX_Millis **interStimulusInterval** = 1000
- CX_SlidePresenter **SlidePresenter**

15.4.1 Detailed Description

This example shows how to implement an N-Back task using an advanced feature of the CX_SlidePresenter (SP). There is a feature of the SP that allows you to give it a pointer to a function that will be called every time the SP has just presented the final slide that it currently has. In your function, you can add more slides to the SP, which will allow it to continue presenting slides. If you don't add any more slides, slide presentation will stop with the currently presented slide. The applicability of this feature to an N-Back task should be fairly clear.

For this N-Back task, the presentation of stimuli will follow the pattern stimulus-blank-stimulus-blank etc. The idea is that you will load up the SP with the first few stimuli and blanks. The SP will be started and will present the first few stimuli. When it runs out of stimuli, the last slide user function will be called. In this function, we will check for any responses that have been made since the last time the function was called and draw the next stimulus-blank pair. See the definition of finalSlideFunction and setupExperiment for the implementation of these ideas.

Once you understand this example, please also see the advancedNBack example. It has some important improvements that make an N-Back task much more reliable in terms of visual stimulus timing.

15.5 modularSynth.cpp File Reference

```
#include "CX_EntryPoint.h"
```

Functions

- void **drawInformation** (void)
- void **modularSynthInternals** (void)
- void runExperiment (void)

15.5.1 Detailed Description

This example shows some of the ways in which a modular synthesizer can be constructed using modules provided in the CX::Synth namespace.

Index