CX Manual

Generated on Wed Jan 20 2016

Contents

1. Main Page

ofxCX (aka the C++ Experiment Development Library; 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, like psychology experiments. OpenFrameworks and CX are both free and open source, distributed under the MIT license.

You can always get the most recent version of CX from its GitHub repository here: `https://github.com/hardmanko/ofxCX`.

The best place to start with CX is the Getting Started page, which includes installation information. After that, there are a variety of topics to read about.

- Video
  - To learn about presenting visual stimuli, first read the Visual Stimuli tutorial.
  - For a variety of examples of drawing visual stimuli, see the `renderingTest` or `animation` examples or the `nBack` or `changeDetection` example experiments.
  - See the Video page for reference on visual stimulus presentation.
- Audio
  - To learn about playing or recording sounds, read the Audio Input and Output tutorial. See also the Sound page and the `soundBuffer` example.
  - For an example of synthesizing sounds, see the `modularSynth` example.
- To learn about collecting responses, read the Response Input tutorial.
- To learn how to store and output experiment data, read the Storing and Outputting Data tutorial. See also the Data page or see the `dataFrame` example.
- To learn about random number generation, see the Randomization page or the `changeDetection` or `nBack` examples.
- To learn about how CX logs errors and other runtime information, see the Message Logging page or the `logging` example.
- To learn about input and output timing, read the Timing Issues page.
- General information about the structure of CX, including informaton about some of the software used by CX internally, can be found on the Program Model page.
- Once you have completed developing an experiment, read the Deploying an Experiment page to learn how to deploy the experiment to the computers that will actually run the experiment.

2. Getting Started

In brief, you need a few things to use CX:

- A reasonably modern computer with Windows, Linux, or OSx.
- `openFrameworks`, which CX relies on.
- A C++ compiler/IDE to compile openFrameworks, CX, and your code.
- The CX code, which you can get from the github repository (`https://github.com/hardmanko/ofxCX`).

The sections below go into a more detail about each of these things.

System Requirements

The short version: Use a reasonably modern computer (made around 2010 or later) with Windows, Linux, or OSx.

The long(er) version:

Although openFrameworks works on a wide variety of hardware and software, CX does not support all of it. For example, CX does not support iPhones, although openFrameworks does. Windows is the best-supported OS as it is the OS that I use. With Linux, you get what you paid for, but I have gotten CX working well on Linux. OSx is ok, but I haven't used CX very much on that OS. Also, OSx is only an option if you use the latest version of openFrameworks (more below).

As far as hardware is concerned, the minimum requirements for openFrameworks and CX are low. However, if your video card is too old, you won't be able to use some types of graphical 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 is generally a good idea for psychology experiments, because one core can be hogged by CX while the operating system can use the other core for other things. Basically, use a computer made after 2010 and you will have no worries most of the time. However, CX has been found to work (with a lot of effect) 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.

Getting openFrameworks

In order to use CX, you must have openFrameworks installed. Currently version 0.8.4 of openFrameworks is stably supported by CX. Support for openFrameworks version 0.8.0 is being removed, but it may still work. Support for 0.9.0, which is the latest version of openFrameworks, is being added, but some features may not work on some platforms (there are audio problems on OSx).

I would recommend oF 0.8.4, unless you are on OSx, in which case you must use openFrameworks 0.9.0.

The latest version of openFrameworks can be downloaded from here and older versions from here. The main openFrameworks download page (http://openframeworks.cc/download/) has information about how to install openFrameworks, depending on what development environment you are using.

Compiler/IDE

You will need a C++ compiler/IDE with support for C++11, because CX uses C++11 features extensively. The openFrameworks download page lists the officially supported IDEs for the different platforms. You can probably make openFrameworks work with other compilers, but this is not recommended for beginners. As far as I know, all compilers/IDEs that support openFrameworks AND have c++11 support will work with CX.

For Windows, I recommend Visual Studio, which is well-supported by openFrameworks.

- If using openFrameworks 0.9.0, I recommend Visual Studio 2015 Community. It's free.
- If using openFrameworks 0.8.4, I recommend Visual Studio 2012. The Professional version of it costs money (unless you are a student, in which case it is free through Dreamspark: https://www.dreamspark.↩com/). If you don't want to buy Visual Studio just to try CX, you can use Visual Studio 2012 Express (http://www.microsoft.com/en-us/download/details.aspx?id=34673), which is free but does not have all of the functionality of the full version of Visual Studio (but it will work).

For Linux, it depends on the version of openFrameworks you're using:

- For openFrameworks 0.9.0, I recommend Qt Creator.
- For openFrameworks 0.8.4, I recommend Code::Blocks.
- For either, you can use makefiles, if you so desire.

For OSx, only openFrameworks 0.9.0 is supported by CX. You can either use Xcode (free from the app store) or Qt Creator (free under some license restrictions), either of which seem to be pretty good options.

Installing CX

Once you have installed openFrameworks, you can install CX. First, download CX from the releases area of the GitHub repository. Select a release that is appropriate for the version of openFrameworks you chose and click on the "Source code (zip)" link to download it (you don't need to download the manual separately, it will be in the `docs` subdirectory within the zip). Once the zip file is downloaded it should contain one folder with a name like "ofxCX-0.1.2". Put this folder into `OFDIR/addons`, where OFDIR is where you put openFrameworks when you installed it. The directory structure should be `OFDIR/addons/ofxCX-0.1.2`. Within the folder `ofxCX` there should be a number of folders (`docs`, examples, `libs`, `src`) plus license and readme files. If what you have matches this, you are now done installing things!

Creating Your First CX Project

1. Use the oF project generator to create a new project that uses the ofxCX addon. The help page for the project generator is here.
    - The project generator asks you what to name your project and allows you to change where to put it (defaults to `OFDIR/apps/myApps/myAppName`, where `myAppName` is the name you picked for your app).
    - Once you have selected a name and location for the project, in the addons area, select the version of "ofxCX" that you installed. If "ofxCX" does not appear in the list of addons, you probably didn't put the ofxCX directory in the right place when installing it.
    - Once ofxCX has been added as an addon, click on the "Generate" button to create the project. There are usually no errors when generating a project.
2. Go to the newly-created project directory (that you chose when creating the project in step 1; typically within `OFDIR/apps/myApps`) and go into the `src` subdirectory.
3. Delete all of the files in the `src` directory (main.cpp, testApp.h, and testApp.cpp). The project generator creates these files for normal openFrameworks apps, but you don't need them for CX apps.
4. Create a new .cpp file in the `src` subdirectory and give it a name, like "MyFirstExperiment.cpp". In the new file, you will need to include CX.h and define a function called `runExperiment`, just like in the example below:

    ```
    #include "CX.h"

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

    Including CX.h brings into your program all of the classes and functions from CX and openFrameworks so that you can use them. `runExperiment` is the CX version of a `main` function: It is called once, after CX has been set up, and the program closes after `runExperiment` returns.
5. Now you need to tell the compiler that it should compile the whole project, including openFrameworks, CX, and your new .cpp file. This step depends on your exact compiler and operating system, but I have provided information for two common configurations.
    - For Visual Studio (VS), you go to the root directory for your application (up one level from `src`) and open the file with the same name as your project with the `.sln` extension. This should open VS and your project.
        - On the left side of the VS window, there should be a pane called "Solution Explorer". Within the Solution Explorer, there should be a few items. One will be called "Solution 'APP_NAME' (2 projects)", which contains your project, called APP_NAME, and a project called `openframeworksLib`. You should expand your project until you can see a folder called `src`. It will have the same files as you deleted in step 3 listed there, so get rid of them by highlighting them and pressing the delete key (or right click on them and select "Exclude From Project").
        - Now right click on the `src` folder in VS and select "Add" -> "Existing item...". In the file selector that opens, navigate your way to the `src` folder in your project directory and select the .cpp file you made in step 4. You can alternately drag and drop your cpp file from the Explorer window into the `src` folder within VS.

  - Now press F5, or select "Debug" -> "Start Debugging" from the menu bar at the top of the VS window. This will compile and run your project in debug mode. It will take a long time to compile the first time, because it has to compile all of openFrameworks and all of CX the first time. However, subsequent builds will only need to compile your code and will be much faster.

- On Linux, if you are using Code::Blocks, you don't need to tell Code::Blocks about the new file you made. The build process simply compiles everything in the `src` directory of your project.
  - Note that on Linux, you may need to explicitly enable C++11 features of the compiler before compiling. When the openFrameworks project generator creates a new project on Linux, it creates a file called config.make in the root directory of your project. Find the line in config.make that has "#↩ PROJECT_CFLAGS" on it and change that line to "PROJECT_CFLAGS = -std=c++11" (note that the # at the start of the line has been removed). This will enable C++11 features of the compiler. If the line is already there and lists a newer C++ standard (like C++14), you don't need to change anything.
  - After opening the Code::Blocks workspace file, you click on the Compile and Run button (looks like a yellow gear and a green play symbol) to compile and run the project.

That's all you need to do to get started with a blank experiment. However, you probably have no idea what to put into `runExperiment` at this point. There are two places to start. The first is to read some of the tutorials in this manual, which include Visual Stimuli, Audio Input and Output, Response Input, and Storing and Outputting Data. After reading those tutorials, you will have the basics down. The second is to look at the Examples, which are complete, runnable pieces of code with comments. The advantage of the tutorials is that they are presented in an easy-to-read style. The advantage of the examples is that you can run them and see their output.

2.1. Examples

There are several code examples of how to use CX. The example files can be found in the CX directory (`OF_DI↩ R/addons/ofxCX`) 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. You should start with the helloWorld example and go from there.

In order to use the examples, do everything for creating a new CX project (above) up until step 3. Then, instead of creating a new .cpp file in step 4, copy one of the example .cpp files from the example folders into the `src` directory. Then do step 5, telling the compiler about the .cpp file you just copied and compiling the example.

Some of the examples have data files that they need run. For example, the renderingTest example has a picture of some birds that it uses. If the example has data, in the example directory there will be a directory called `bin` with a directory under it called `data` containing the necessary files. These should be copied to `PROJECT_NA↩ ME/bin/data`. The `bin/data` folder in the project directory might not exist immediately after creating a new project. You can create it if it is not there.

Misc. examples:

- helloWorld - A very basic getting started program.
- animation - A simple example of a way to draw moving things in CX without using blocking code. Also includes some mouse input handling: cursor movement, clicks, and scroll wheel activity.
- renderingTest - Includes several examples of how to draw stuff using ofFbo (a kind of offscreen buffer), of↩ Image (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.

Experiments:

- flanker - A Flanker task in which letters are used as the stimuli. This is a good minimal experiment example (the other example experiments are substantially more complex).

- changeDetection - A very straightforward working memory 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.

Specific topics:

- 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.
- soundBuffer - Tutorial covering a number of things that you can do with CX_SoundBuffers, 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.

3. Visual Stimuli

This section will describe a number of important details about how CX handles drawing visual stimuli. It begins with a discussion of framebuffers, which are what visual stimuli are drawn into. Then some examples of how to draw stimuli are given. Then the preferred method of presenting time-locked stimuli in CX is explained. Finally, some more background on achieving accurate stimulus timing by using vertical synchronization is given.

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 buffer swapping.

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. All drawing in OpenGL and CX is done into a framebuffer.

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::endDrawingToBackBuffer() 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. In CX, this swap can be done by using different functions of CX_Display: CX::CX_Display::swapBuffers(), CX::CX_Display::swapBuffersInThread(), or CX::CX_Display::setAutomaticSwapping(). These functions are not interchangable, so make sure you are using the right one for your application. If there is any doubt, start with swapBuffers(), because it will work 100% of the time, with the others serving as ways to optimize the presentation of stimuli later.

Other than the back buffer, you can also make other offscreen buffers. You do this with an `ofFbo`, which is an openFrameworks class. These offscreen buffers can be drawn into just like the back buffer, so you can use them to store part of a scene or a whole scene in an ofFbo. The contents of an ofFbo can be drawn onto the back buffer at a later time.

Drawing Visual Stimuli

Now that framebuffers have been explained, we can talk about how to draw stimuli into them. Assume for the sake of example that we want to draw a red circle and a green rectangle on a black background. We might write this little test program:

```
#include "CX.h"

void runExperiment(void) {
    Disp.beginDrawingToBackBuffer();

    ofBackground(ofColor::black);

    ofSetColor(ofColor::red);
    ofCircle(200, 300, 100);

    ofSetColor(0, 255, 0);
    ofRect(400, 200, 200, 100);

    Disp.endDrawingToBackBuffer();

    Disp.swapBuffers();

    Input.Keyboard.waitForKeypress(-1);
}
```

Let's break this down. As always with a CX program, we include CX.h in order to access the functionality of C↩ X. We also define the `runExperiment` function, in which we use the global display object `Disp`, which is a CX::CX_Display. We start with

```
Disp.beginDrawingToBackBuffer();
```

to say that we want to draw our stimuli to the back buffer. After saying that we want to draw into the back buffer, we can start drawing things. We do all of the drawing in this example with openFrameworks drawing functions. Like all functions in openFrameworks, they are prefixed with the abbreviation "of". With the call

```
ofBackground(ofColor::black);
```

we set the background color to black. There is nothing fancy here like a special background color layer: this is just filling the entire back buffer with black. If you draw some stimuli and then call `ofBackground`, you will overwrite your stimuli. The way that we specify the color is by using a named constant color that is a static member of the `ofColor` class, by using double-colon to access the static member. We then draw the circle with

```
ofSetColor(ofColor::red);
ofCircle(200, 300, 100); //x position, y position, radius
```

The way that drawing is set up in openFrameworks is that for a lot of things that you draw, you first set the color that it will be drawn with, then you draw the thing itself. Here, we call `ofSetColor` to set the drawing color to red before drawing the circle. `ofCircle` draws a circle at the specified x and y coordinates with the given radius. All of the values are in pixels. By default, the coordinate system is set up so that the point (0,0) is in the upper-left corner of the screen. The x values increase to the right and the y values increase downwards. If you don't like the fact that y values increase downwards, you can call CX::CX_Display::setYIncreasesUpwards() with `true` as the argument at the beginning of the experiment. It is possible that not everything properly accounts for the change in the y-axis direction, so some graphical bugs are possible if the y-values increase upwards. However, the vast majority of things work just fine.

Now that the circle has been drawn, we draw the rectangle with

```
ofSetColor(0, 255, 0); //red, green, blue (amounts out of 255)
ofRect(400, 200, 200, 100); //x position, y position, width, height
```

As before, we set the color before drawing the object. However, in this case, the color is specified with RGB coordinates. Values for the RGB coordinates of colors (at least 24-bit colors) go from 0 to 255 and are given in order. Calling `ofSetColor(0, 255, 0)` sets the drawing color to have no red (0), maximum green (255), and no blue (0). With the color set, `ofRect` draws a rectangle at the given x and y coordinates with a width and height specified in the last two arguments.

Now that we have drawn everything we wanted to, we need to say that we are done drawing into the back buffer and ask for the back buffer to be swapped to the front buffer so that it is actually visible, which is done with

```
Disp.endDrawingToBackBuffer();

Disp.swapBuffers();
```

With the first line of code, we tell the display that we are done drawing to the back buffer. By calling `swapBuffers`, we tell the display to swap the front and back buffers. Just after `swapBuffers()` is called, the objects should appear on screen. The final line of code before `runExperiment` returns is

```
Input.Keyboard.waitForKeypress(-1);
```

which just says to wait until any key has been pressed. Once a key has been pressed, control flow will fall off the end of `runExperiment`, causing it to implicitly return, after which the program will exit.

This example shows a number of basic things about how to draw stimuli in CX: 1) Use of the `Disp` object to control the rendering and framebuffer environment and 2) The basics of drawing specific stimuli using openFrameworks' drawing functions. OpenFrameworks has many different kinds of drawing functions for a wide variety of stimuli. 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 with openFrameworks: See the graphics and 3d sections of this page: http://www.openframeworks.cc/documentation/. In addition to the many drawing functions of openFrameworks, CX provides a number of drawing functions in the CX::Draw namespace. The renderingTest example contains samples of many of the different kinds of stimuli that can be drawn with CX and openFrameworks.

Time-Locked Visual Stimuli: CX_SlidePresenter

Typically, stimuli should be presented at specific times. CX provides a helpful class that controls stimulus timing for you, called the CX_SlidePresenter. Examples of the use of a CX_SlidePresenter can be found in the nBack and changeDetection examples. In particular, the nBack example goes into some depth with advanced features of the CX_SlidePresenter. However, we will start with examples of basic use of the slide presenter. In the example, we will present the same circle and rectangle that we drew above, but this time, we will present them in a time-locked sequence. The full example:

```
#include "CX.h"

CX_SlidePresenter slidePresenter;

void runExperiment(void) {

    slidePresenter.setup(&Disp);

    slidePresenter.beginDrawingNextSlide(3000);
        ofBackground(ofColor::black);
        ofSetColor(ofColor::red);
        ofCircle(200, 300, 100);
    slidePresenter.endDrawingCurrentSlide();

    slidePresenter.beginDrawingNextSlide(1500);
        ofBackground(ofColor::black);
        ofSetColor(0, 255, 0);
        ofRect(400, 200, 200, 200);
    slidePresenter.endDrawingCurrentSlide();

    slidePresenter.beginDrawingNextSlide(1);
        ofBackground(ofColor::black);
        ofSetColor(ofColor::red);
        ofCircle(200, 300, 100);
        ofSetColor(0, 255, 0);
        ofRect(400, 200, 200, 200);
    slidePresenter.endDrawingCurrentSlide();
```

```
    slidePresenter.startSlidePresentation();
    while (slidePresenter.isPresentingSlides()) {
        slidePresenter.update();
    }

    Input.Keyboard.waitForKeypress(-1);
}
```

On the third line, we instantiate a CX_SlidePresenter and call it `slidePresenter`. Within `runExperiment` we set up `slidePresenter` by giving it a pointer to `Disp` by calling

```
slidePresenter.setup(&Disp);
```

Using `&Disp` means to get a pointer to `Disp`. There is another setup function for CX_SlidePresenter that takes a CX_SlidePresenter::Configuration `struct`, which allows you to configure the CX_SlidePresenter more thoroughly. For now, we will just use the basic setup function. The slide presenter needs to know what CX_Display to use because it uses a variety of functions of the display to present your stimuli.

With the following code, we will create a new slide in `slidePresenter` and draw stimuli into the slide.

```
slidePresenter.beginDrawingNextSlide(3000, "circle"); //slide duration, name
    ofBackground(ofColor::black);
    ofSetColor(ofColor::red);
    ofCircle(200, 300, 100);
slidePresenter.endDrawingCurrentSlide();
```

You should recogize the stimulus drawing functions from before. By calling `slidePresenter.begin↩DrawingNextSlide(3000, "circle")`, we are saying to create a new slide with a duration of 3000 ms and name "circle". The name of a slide is optional and purely to make it easy for you to identify the slide later. Everything that is drawn before `slidePresenter.endDrawingCurrentSlide()` is called will be drawn into the slide. It works this way because for each new slide, the slide presenter makes an offscreen framebuffer (an ofFbo) and sets it up so that everything will be drawn into that framebuffer until `endDrawingCurrentSlide()` is called. Calling `endDrawingCurrentSlide()` is optional in that if you forget to do it, it will be done for you.

We do the same thing to draw the rectangle.

```
slidePresenter.beginDrawingNextSlide(1500, "rectangle");
    ofBackground(ofColor::black);
    ofSetColor(0, 255, 0);
    ofRect(400, 200, 200, 200);
slidePresenter.endDrawingCurrentSlide();
```

Notice that we need to call `ofBackground()` for every slide, because there is no default background color for newly created slides. Also, we made the duration of this slide to be 1500 ms.

Finally, we make the final slide, which has both objects in it.

```
slidePresenter.beginDrawingNextSlide(1);
    ofBackground(ofColor::black);
    ofSetColor(ofColor::red);
    ofCircle(200, 300, 100);
    ofSetColor(0, 255, 0);
    ofRect(400, 200, 200, 200);
slidePresenter.endDrawingCurrentSlide();
```

Notice that the duration of the final slide is set to 1 ms, yet when the example is run, the final slide stays on sreen indefinitely. This is correct behavior: The final slide that is created is the finishing point for the slide presenter. As soon as the last slide is presented, the slide presentation is done. The logic of it is that it is not possible for the slide presenter to know what should be presented after the last slide. Should the screen turn black? Should a test pattern be presented? There is no obvious default. For this reason, after the last slide is presented, the slide presenter can't sensibly replace that slide with anything else, so it remains on screen until other code draws something else. Thus, the duration of the last slide is ignored and as soon as the last slide is presented, the slide presenter is done (although note that the duration must be > 0, because all slides with duration 0 are never presented by the slide presenter).

Once all of the slides have been drawn, to present the slides, the following code is used

```
slidePresenter.startSlidePresentation();
while (slidePresenter.isPresentingSlides()) {
    slidePresenter.update();
}
```

On the first line, the slide presentation is started. While a slide presentation is in progress, the slide presenter needs to be updated regularly. The reason for this is that every time the slide presenter is updated, it checks to see if the next stimulus should be drawn to the screen. If it is not updated regularly, it could miss a stimulus start time. For this reason, in the while loop, all we do is update the slide presenter with `slidePresenter.update();`. The condition in the loop is just checking to see if the slide presentation is in progress. If so, it keeps looping. At some point, all of the slides will have been presented and the loop will end.

While the slide presentation is in progress, we may want to do other things as well. We can do so by adding these other tasks to the `while` loop. During the slide presentation loop, you can check to see what slide was the last slide to be presented with CX_SlidePresenter::getLastPresentedSlideName(). Using that function, you can do some kinds of synchronization, such as synchronizing sound stimuli with the visual stimuli. Additionally, we could insert

```
Input.pollEvents();
```

into the loop to check for new input regularly. If you call CX::CX_SlidePresenter::presentSlides(), it does a standard slide presentation, including polling for input continuously.

At the end of the example, we wait for any key press before exiting.

```
Input.Keyboard.waitForKeypress(-1);
```

The CX_SlidePresenter is a very useful class that does away with most of the difficult aspects of presenting time-locked visual stimuli. You should probably never present time-locked visual stimuli without using a CX_Slide↩ Presenter or another similar mechanism. The one exception is if you are presenting a long animation sequence in which the scene changes on every frame, especially if the change occurs in response to user input. In that case, using a slide presenter is unweildly and you should probably just write a loop in which the animation is updated every frame. See the animation example for one way of doing an animation without blocking in the main thread.

Vertical Synchronization

An aspect of visual stimulus presentation that is important is vertical synchronization. Vertical synchronization (Vsync) is the process by which the swaps of the front and back buffers 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 while 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 use 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 the 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 CX::CX_Display::waitForOpenGL() 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 CX::CX_Display↩ ::waitForOpenGL() 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. Other times, it does nothing to fix the problem. You can turn on hardware or software Vsync with CX::CX_Display::useHardwareVSync() and CX::CX_Display::useSoftwareVSync().

How do you check to see if you are having problems with video presentation that are related to Vsync? One way to check is to use CX::CX_Display::testBufferSwapping(), which tests buffer swapping under few of conditions and

provides a summary of results along with raw data from the test. A part of the test is a visual one, in which you should be able to visually discriminate between correct and erroneous behavior. Even though errors occur one a time scale that is not normally perceptable, the structure of the visual displays should allow most people to recognize errors if they know what to look for, which is explained in the documentation for the function.

Another way to check to see if you are having problems with Vsync is to use a feature of CX::CX_SlidePresenter to learn about the timing of your stimuli. CX::CX_SlidePresenter::printLastPresentationInformation() 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. You can also try different buffer swapping modes of CX::CX_SlidePresenter (see CX::CX_SlidePresenter::SwappingMode). One of the swapping modes (MULTI_CORE) swaps the buffers every frame in a secondary thread which avoids issues that arise from not swapping the buffers every frame. 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.

If you have tried CX_Display::useHardwareVSync() and it does not appear to do anything, one 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 in my experience seems to be more reliable than turning Vsync on or off from within CX. If you force Vsync to a setting in the video card driver, CX::CX_Display↩ ::useHardwareVSync() will probably not do anything, but CX::CX_Display::useSoftwareVSync() possibly would still do something (although it is not clear that you would want to have both hardware and software Vsync enabled at the same time). CX_Display::testBufferSwapping() includes a test with both kinds of Vsync enabled simultaneously, so you can check to see if it is working correctly.

If you are experiencing problems with Vsync 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. This is a good reason to always run experiments in full screen mode.

4. Audio Input and Output

Audio input and output in CX is based on a number of classes. The two most important are CX_SoundStream and CX_SoundBuffer. Additionally, CX_SoundBufferPlayer and CX_SoundBufferRecorder combine together a C↩ X_SoundStream and a CX_SoundBuffer to play back or record audio, respectively. Finally, for people wanting to synthesize audio in real time (or ahead of time), the CX::Synth namespace provides a multitude of ways to synthesize audio. We will go through these components in a practical order.

Setting up the CX_SoundStream for Playback

Because the CX_SoundStream is what actually does audio input and output, in order to get any sounds out of a CX program, you must configure a CX_SoundStream for use. This requires that a CX_SoundStream::Configuration struct be filled out with the desired settings and given to CX_SoundStream::setup() as the argument. There are several configuration options for the CX_SoundStream, but, if the gods smile on you today, you will only need one, which is the number of output channels. If you try this and the gods seem to be frowning, check out the Troubleshooting Audio Problems section below. We will use stereo output, so 2 output channels.

```
CX_SoundStream::Configuration ssConfig;
ssConfig.outputChannels = 2; //Stereo output
//ssConfig.api = RtAudio::Api::WINDOWS_DS; //The most likely thing you will need to change is the low-level
       audio API.

//Create the CX_SoundStream and set it up with the configuration.
CX_SoundStream soundStream;
soundStream.setup(ssConfig);

//Check for any error messages.
Log.flush();
```

If there were any errors during setup of the sound stream, they will be logged. Check the console for any messages. You can also check if the return value of `setup()` or call CX_SoundStream::isStreamRunning() to see if setup was successful.

Playback

Now that we have a CX_SoundStream set up, next next thing we need to do in order to play the contents of the sound file is to load the file into CX. This is done by creating a CX_SoundBuffer and then loading a sound file into the sound buffer, as follows.

```
CX_SoundBuffer soundBuffer;
soundBuffer.loadFile("sound_file.wav");
```

If there wasn't an error loading the file, `soundBuffer` now contains the contents of the sound file in a format that can be played by CX. Once you have a sound file loaded into a CX_SoundBuffer, there are a number of things you can do with it. You can remove leading silence with CX_SoundBuffer::stripLeadingSilence() or add silence to the beginning or end with CX_SoundBuffer::addSilence(). You can delete part of the sound, starting from the beginning or end, with CX_SoundBuffer::deleteAmount(). You can reverse the order of the samples, so as to be able to play the sound backwards with CX_SoundBuffer::reverse(). These are just some examples. See the documentation for CX_SoundBuffer and the `soundBuffer` example for more things you can do with it.

Now that you have CX_SoundBuffer with sound data loaded into it, you can play it back using a CX_SoundBuffer↩ Player. Before you can use a CX_SoundBufferPlayer, you have to configure it with CX_SoundBufferPlayer::setup(). `setup()` takes either a structure holding configuration options for the CX_SoundStream that will be used by the CX_SoundBufferPlayer or a pointer to a CX_SoundStream that has already been set up. We will use the CX_↩ SoundStream called `soundStream` that we configured in the previous section.

```
CX_SoundBufferPlayer player;
player.setup(&soundStream);
```

Now that we have a configured CX_SoundBufferPlayer, we just need to give it a CX_SoundBuffer to play by using CX_SoundBufferPlayer::setSoundBuffer() and play the sound.

```
player.setSoundBuffer(&soundBuffer);

player.play();

//Wait for it to finish playing.
while (player.isPlaying())
    ;
```

Because playback does not happen in the main thread, we wait in the main thread until playback is complete before going on.

Playing Multiple Sounds Simultaneously

A CX_SoundBufferPlayer can have a single CX_SoundBuffer assigned to it as the active sound buffer. This means that you cannot play more than one sound at once with a CX_SoundBufferPlayer. This limitation is by design, but also by design there are ways to play multiple sounds at once. The preferred way involves merging together multiple CX_SoundBuffers using CX_SoundBuffer::addSound(). What this does is take a CX_SoundBuffer and add it to an another CX_SoundBuffer at a given offset. This guarantees that the two sounds will be played at the correct time relative to one another, because there is no latency when the second sound starts playing. An example of merging sound buffers:

```
CX_SoundBuffer otherBuffer;
otherBuffer.loadFile("other_sound_file.wav");

CX_SoundBuffer combinedBuffer = soundBuffer;

combinedBuffer.addSound(otherBuffer, 500); //Add the second sound to the first,
    //with the second starting 500 ms after the first.

player.setSoundBuffer(&combinedBuffer);
player.play();
while(player.isPlaying())
    ;
```

Another way to play multiple sounds at once is to create multiple CX_SoundBufferPlayers, all of which use the same CX_SoundStream. Then you can assign different CX_SoundBuffers to each player and call CX_SoundBuffer↩ Player::play() whenever you want to play the specific sound.

```
CX_SoundBufferPlayer player2;
player2.setup(&soundStream);
player2.setSoundBuffer(&otherBuffer);

player.play();
Clock.sleep(500);
player2.play();
while (player.isPlaying() || player2.isPlaying())
    ;
```

You can also put multiple CX_SoundBuffers and CX_SoundBufferPlayers into C++ standard library containers, like `std::vector`. However, I must again stress that using CX_SoundBuffer::addSound() is a better way to do things because it provides 100% predictable relative onset times of sounds (unless there are glitches in audio playback, but that's a different serious problem).

Recording Audio

To record audio, you can use a CX_SoundBufferRecorder. You set it up with a CX_SoundStream, just like CX_SoundBufferPlayer. The only difference is that for recording, we need input channels instead of output channels. We will stop the currenly running CX_SoundStream and reconfigure it to also have 1 input channel. We then set up the CX_SoundBufferRecorder using `soundStream`, create a new CX_SoundBuffer for it to record into, and set that buffer to be recorded to.

```
soundStream.stop();
ssConfig.inputChannels = 1; //Most microphones are mono.
soundStream.setup(ssConfig);

CX_SoundBufferRecorder recorder;
recorder.setup(&soundStream);

CX_SoundBuffer recordedSound;
recorder.setSoundBuffer(&recordedSound);

Log.flush(); //As usual, let's check for errors during setup.
```

Now that we have set up the recorder, we will record for 5 seconds, then play back what we have recorded.

```
cout << "Starting to record." << endl;
recorder.start();
Clock.sleep(CX_Seconds(5));
recorder.stop();
cout << "Done recording." << endl;
```

We sleep the main thread for 5 seconds while the recording takes place in a secondary thread. The implication of the use of secondary threads for recording is that you can start a recording, do whatever you feel like in the main thread – draw visual stimuli, collect responses, etc. – all while the recording keeps happening in a secondary thread.

Once our recording time is complete, we will set a CX_SoundBufferPlayer to play the recorded sound in the normal way.

```
player.setSoundBuffer(&recordedSound);
player.play();
while (player.isPlaying())
    ;
```

Be careful that you are not recording to a sound buffer at the same time you are playing it back, because who knows what might happen (it would probably be fine, actually). To be careful, you can "detach" a CX_SoundBuffer from either a player or a recorder by calling, e.g., CX_SoundBufferPlayer::setSoundBuffer() with `nullptr` as the argument.

```
recorder.setSoundBuffer(nullptr); //Make it so that no buffers are associated with the recorder.
```

All of the pieces of code from above in one place:

```
#include "CX.h"

void runExperiment(void) {
    //Sound stream configuration
    CX_SoundStream::Configuration ssConfig;
    ssConfig.outputChannels = 2; //Stereo output
    //ssConfig.api = RtAudio::Api::WINDOWS_DS; //The most likely thing you will need to change is the
       low-level audio API.

    //Create the CX_SoundStream and set it up with the configuration.
    CX_SoundStream soundStream;
    soundStream.setup(ssConfig);

    //Check for any error messages.
    Log.flush();

    //If things aren't working, try uncommenting this line to learn about the devices available on your
       system for the given api.
    //cout << CX_SoundStream::listDevices(RtAudio::Api::WINDOWS_DS) << endl;

    //Playback
    CX_SoundBuffer soundBuffer;
    soundBuffer.loadFile("sound_file.wav");

    CX_SoundBufferPlayer player;
    player.setup(&soundStream);
    player.setSoundBuffer(&soundBuffer);

    player.play();

    //Wait for it to finish playing.
    while (player.isPlaying())
        ;

    Log.flush();

    soundBuffer.deleteChannel(1);
    player.setSoundBuffer(&soundBuffer);

    player.play();

    //Wait for it to finish playing.
    while (player.isPlaying())
        ;

    Log.flush();


    //Playing multiple sounds at once
    CX_SoundBuffer otherBuffer;
    otherBuffer.loadFile("other_sound_file.wav");

    CX_SoundBuffer combinedBuffer = soundBuffer;

    combinedBuffer.addSound(otherBuffer, 500); //Add the second sound to the first,
    //with the second starting 500 ms after the first.

    player.setSoundBuffer(&combinedBuffer);
    player.play();
    while (player.isPlaying())
        ;

    CX_SoundBufferPlayer player2;
    player2.setup(&soundStream);
    player2.setSoundBuffer(&otherBuffer);

    player.play();
    Clock.sleep(500);
    player2.play();
    while (player.isPlaying() || player2.isPlaying())
        ;


    //Recording
    soundStream.stop();
    ssConfig.inputChannels = 1; //Most microphones are mono.
    soundStream.setup(ssConfig);


    CX_SoundBufferRecorder recorder;
    recorder.setup(&soundStream);

    CX_SoundBuffer recordedSound;
    recorder.setSoundBuffer(&recordedSound);

    Log.flush(); //As usual, let's check for errors during setup.
```

```
    cout << "Starting to record." << endl;
    recorder.start();
    Clock.sleep(CX_Seconds(5));
    recorder.stop();
    cout << "Done recording." << endl;

    player.setSoundBuffer(&recordedSound);
    player.play();
    while (player.isPlaying())
        ;

    recorder.setSoundBuffer(nullptr); //Make it so that no buffers are associated with the recorder.
}
```

Synthesizing Audio

You can synthesize audio in real time, or ahead of time, using the classes in the CX::Synth namespace. See the modularSynth example for some uses of synthesizer modules.

Direct Control of Audio IO

If you want to be really fancy, you can directly read and write the audio data that a CX_SoundStream is sending or receiving. This is a relatively advanced operation and is unlikely to be needed in very many cases, but it's there if need be.

In order to directly access the data that a CX_SoundStream is transmitting, you need to create a class containing a function that will be called every time the CX_SoundStream needs to send more data to the sound card. For example, you could have a class like this that creates a sine wave.

```
class ExampleOutputClass {
public:
    void callbackFunction(CX_SoundStream::OutputEventArgs& args) {
        static float wavePosition = 0;

        float sampleRate = args.instance->getConfiguration().sampleRate;
        const float frequency = 524;
        float positionChangePerSampleFrame = 2 * PI * frequency / sampleRate;

        for (unsigned int sampleFrame = 0; sampleFrame < args.bufferSize; sampleFrame++) {

            //For every channel, put the same data on that channel. This is like playing a mono stream on
       every channel at the same time.
            for (unsigned int channel = 0; channel < args.outputChannels; channel++) {
                args.outputBuffer[(sampleFrame * args.outputChannels) + channel] = sin(wavePosition);
            }

            //Update where in the sine wave we are. A single sine wave happens every 2 * PI.
            wavePosition = fmod(wavePosition + positionChangePerSampleFrame, 2 * PI);
        }
    }
};
```

The only thing going on in this class is `callbackFunction`. This function takes a reference to a CX_Sound↩ Stream::OutputEventArgs struct, which contains important data. Most importantly, `args`, as I have called it in this example, contains a pointer to an array of data that should be filled by the function, called `outputBuffer`. The number of sample frames of data that should be put into `outputBuffer` is given by `bufferSize`. It is important here to be clear about the fact that a sample frame contains 1 sample per channel of sound data, so if `bufferSize` is 256 and the stream is running in stereo (2 channels), the total number of samples that need to be put into `outputBuffer` must be 512. Also note that the sound samples must be interleaved, which means that samples within a sample frame are stored contiguously in the buffer, which means that for the stereo example, even numbered indices would contain data for channel 0 and off numbered indices would contain data for channel 1.

Of course, if you really wanted to create sine waves in real time, you would use CX::Synth::Oscillator and CX::↩ Synth::StreamOutput, but for the sake of example, lets use this class. Once you have defined a class that creates the sound data, create an instance of your class and add it as a listener to the `outputEvent` of a CX_SoundStream.

```
CX_SoundStream soundStream; //Assume this has been or will be set up elsewhere.
ExampleOutputClass sineOut; //Make an instance of the class.

//For event soundStream.outputEvent, targeting class instance sineOut, call callbackFunction of that class
      instance.
ofAddListener(soundStream.outputEvent, &sineOut, &ExampleOutputClass::callbackFunction);
```

From now on, whenever `soundStream` needs more output data, `sineOut.callbackFunction` will be called automatically. The data that you put into the output buffer must be of type `float` and bounded between -1 and 1, inclusive. You can remove a listener to an event with `ofRemoveListener`. More information about the events used by openFrameworks can be found here: http://www.openframeworks.cc/documentation/events/ofEvent.html.

Directly accessing input data works in a very similar way. You need a class with a function that takes a reference to a CX::CX_SoundStream::InputEventArgs struct and returns `void`. Instead of putting data into the output buffer, you would read data out of the input buffer.

Troubleshooting Audio Problems

It is often the case that audio playback problems arise due to the wrong input or output device being used. For this reason, CX_SoundStream has a utility function that lists the available devices on your system so that you can select the correct one. You do this with CX::CX_SoundStream::listDevices() like so:

```
cout << CX_SoundStream::listDevices() << endl;
```

Note that `listDevices()` is a static function, so you use the name of the CX_SoundStream class and `::` to access it.

CX_SoundStream uses RtAudio (http://www.music.mcgill.ca/~gary/rtaudio/) internally. It is possible that some problems could be solved with help from the RtAudio documentation. For example, one of the configuration options for CX_SoundStream is the low level audio API to use (see CX::CX_SoundStream::Configuration::api), about which the RtAudio documentation provides some help (http://www.music.mcgill.ca/~gary/rtaudio/classRtAudio.html#ac9b6f625da88249d08a8409a9db0d849). You can get a pointer to the RtAudio instance being used by the CX_SoundStream by calling CX::CX_SoundStream::getRtAudioInstance(), which should allow you to do just about anything with RtAudio.

Audio Latency

OpenFrameworks and, by extension, CX use the RtAudio library, which provides a cross-platform wrapper for different audio APIs. RtAudio provides a consistent interface across platforms and APIs and that interface is what will be described here.

The core of how audio data is given to the hardware that creates the physical stimuls is very similar to the way in which visual stimuli are presented. For visual stimuli, there are two data buffers. One of the buffers is actively being presented while the other buffer is being filled with data in preparation for presentation. Similarly, for audio data, there are at least two buffers. One buffer is presented while another buffer is being filled with data. Whenever the sound hardware finishes presenting one buffer's worth of data, it starts presenting the next buffer of data and it requests a new buffer of data from the audio data source (e.g. a CX_SoundBufferPlayer). When the hardware is done presenting a buffer of data, if the next buffer is not ready, an audio glitch will occur. Typically what happens is that there is a brief moment of silence before the next buffer of data is ready. Also, "clicks" or "pops" are often heard surrounding the silence. Due to these glitches, it is very important that the next buffer of audio data can consistently be filled before the active buffer is emptied. Glitches can be reduced or eliminated by 1) using multiple buffers (which is possible with some audio APIs, but not many) or 2) increasing the size of the buffers. By increasing the size of the buffers, it increases the probability that a buffer of new data will be filled before the active buffer is emptied.

On the other hand, we would like to minimize latency. Latency occurs in audio data because at the time at which you request a sound to be played, there is already some amount of data already in the buffers ahead of the sound you want to play. The new sound that you requested is put at the back of the queue. Typically, the next time that a new buffer worth of data is requested, your sound data will be put into the start of that buffer. The fewer buffers that are in front of the new data, the sooner the new data will make its way to the front of the line and actually get presented. The smaller the buffers are, the smaller the delays between buffers, so your data can be queued more quickly. The best way to minimize latency is to 1) have fewer buffers (minimum of 2: the active buffer and the buffer being filled) and 2) reduce the size of the buffers. Clearly, there is a tradeoff between latency and glitches. The standard recommendation is to aggressively minimize latency until you start to notice glitches, then back off until you stop detecting glitches.

Notice that the latency as determined by the buffer size and number of buffers, while typically represented as a constant value, is not actually a constant. At the time at which the playback of a sound is requested, it could

be that the buffers have just swapped, so the new sound will need to wait nearly the whole length of time that it takes to present a buffer before it even gets put into the queue. Another possibility is that the sound playback is requested right before a buffer swap, so the new data will be put into the queue right away. Thus, the typical latency measurement is more of an upper bound than a constant. CX_SoundBufferPlayer has functionality to help deal with this fact, in that you can request that sounds start at a specified time in the future using CX_SoundBufferPlayer↩ ::startPlayingAt(). When this function is used, when the sound data is to be put into a buffer, it is not necessarily put at the beginning of the buffer, but is put wherever in the buffer it needs to be in order to be played at the right time relative to the buffer onset. However, note that there may still be some constant latency involved even when this function is used that it does not account for.

CX_SoundStream has some functions like CX_Display for learning about the buffer swapping process. There are CX_SoundStream::hasSwappedSinceLastCheck(), CX_SoundStream::waitForBufferSwap(), CX_SoundStream↩ ::getLastSwapTime(), and CX_SoundStream::estimateNextSwapTime().

CX tries to use audio in as latency-controlled a way as possible. The way it does this is by opening an audio stream and keeping it open throughout an experiment. This helps to avoid the latency associated with creating a new audio stream every time a sound is played. This makes it so that the only latency comes from the number and size of the buffers that are used. By keeping an audio stream open all the time, there is a small CPU cost, but it is fairly small. In spite of the design of CX audio, there will always be latency, so understanding the reasons for the latency can help you to deal with it appropriately.

One downside of the use of only a single stream of audio data is that if multiple sounds are to be played at once, the sound levels need to be adjusted in software in CX to prevent clipping. To explain why this is, a little background is needed. Sound data in CX is treated as a `float` and the amplitudes of the data go from -1 to 1. If an amplitude goes outside of this range, it will be clipped, which produces a type of distortion because the waveform is deformed by flattening the peaks of the waves. When multiple sounds are played at once in a single stream, their amplitudes are added, which means that even if each sound in within an acceptable range, once added together, they will be outside of the interval [-1,1] and clipping will occur. For this reason, when multiple sounds are to played together, regardless of whether more than one CX_SoundBufferPlayer is used or the sounds are merged together in a single CX_SoundBuffer, the ampltudes of the sounds need to be constrained. The easiest way of doing this is to use CX_SoundBuffer::normalize() on the sounds. Normalizing a sound means to make the highest peak in the sound have some set amplitude. Assume you had two sounds with unknown maximum amplitudes. If they were both normalized to have a maximum amplitude of 0.5 each, then when added together, they could never clip.

5. Response Input

CX provides built-in support for collecting input from keyboards, mice, and joysticks. The design of the input subsystem of CX is outlined in this section, with a focus on collecting keyboard input. The structure of input collection from other devices parallels keyboard input collection, so the ideas from this section generalize to other input devices. We begin with a code example of how to collect and process keyboard input.

```
#include "CX.h"

void runExperiment(void) {
    Input.setup(true, false);

    while (true) {
        Input.pollEvents();

        while (Input.Keyboard.availableEvents() > 0) {

            CX_Keyboard::Event ev = Input.Keyboard.getNextEvent();

            if (ev.key == 'P' && ev.type == CX_Keyboard::PRESSED) {
                cout << "P is for Press" << endl;
            }

            if (ev.key == 'R' && ev.type == CX_Keyboard::RELEASED) {
                cout << "R is for Release" << endl;
            }

            if (ev.key == Keycode::PAGE_UP && ev.type == CX_Keyboard::REPEAT) {
                cout << "Page up is for repeat" << endl;
            }

            cout << "Key:          " << ev.key << endl <<
                "Key char:  \"" << (char)ev.key << "\"" << endl <<
```

```
                "Type:        " << ev.type << endl <<
                "Time:        " << ev.time << endl <<
                "Uncertainty: " << ev.uncertainty << endl << endl;

        }
    }
}
```

Keyboard, mouse, and joystick input in CX is handled through an instance of the CX_InputManager class called CX::Instances::Input. By default, no input devices are enabled, so the first step is usually to enable the input devices you plan on using. To do so, call the setup function

```
Input.setup(true, false); //enable keyboard, don't enable mouse
```

which enables the keyboard but not the mouse. You can optionally pass a third argument, which is the index of a joystick that you want to use. We want this example to run indefinitely, so we put most of the code into a while loop, with the conditional expression set to the constant value `true`, so that the loop goes indefinitely. Inside the while loop, we check for new input on all enabled input devices by calling

```
Input.pollEvents();
```

which checks for new input on all input devices. We are mostly interested in the keyboard, which we can access with the Keyboard member of Input. With

```
while (Input.Keyboard.availableEvents() > 0) {

    CX_Keyboard::Event ev = Input.Keyboard.getNextEvent();

    //...
}
```

we say that as long as there are any available events for the keyboard, we want to keep looping. Each time through the loop, the next event from the keyboard is accessed with Input.Keyboard.getNextEvent(). getNextEvent() returns the oldest event stored in an input device, deleting it from the device's event queue. As the events are processed, eventually they will all be removed from the queue of available events and availableEvents() will return 0. At that point, the event processing loop will end and the code will go back to checking for new input. The value that is returned by Input.Keyboard.getNextEvent() is a CX::CX_Keyboard::Event, which is a `struct` that contains information about the event, like what key was used and how the key was used. With

```
if (ev.key == 'P' && ev.type == CX_Keyboard::PRESSED) {
    cout << "P is for Press" << endl;
}
```

we test to see if the key that was used was the P key and that it was pressed. Notice that for many keys, we can compare the value of `ev.key` directly to character literals, which in C++ are enclosed in single quotes (strings are enclosed in double quotes). Also notice that the character 'P' is uppercase, which is the standard for the letter keys. To check that the key was pressed, we compare the type of the event to CX_Keyboard::PRESSED. We do a very similar thing for the R key

```
if (ev.key == 'R' && ev.type == CX_Keyboard::RELEASED) {
    cout << "R is for Release" << endl;
}
```

but instead of checking for a key press, we check for a release. The third and final type of key event is a key repeat, which happens after a key has been held for some time and multiple keypresses are sent repeatedly.

```
if (ev.key == Keycode::PAGE_UP && ev.type == CX_Keyboard::REPEAT) {
    cout << "Page up is for repeat" << endl;
}
```

Here we check for a key repeat for the page up key. For special keys for which it is not possible to represent the key with a character literal, you can compare the value of `ev.key` to special values from the CX::Keycode namespace/enum.

To end the example, we print out information about each key that was used, regardless of how it was used, with

```
cout << "Key:          " << ev.key << endl <<
    "Key char:  \"" << (char)ev.key << "\"" << endl <<
    "Type:        " << ev.type << endl <<
    "Time:        " << ev.time << endl <<
    "Uncertainty: " << ev.uncertainty << endl << endl;
```

The key and type members have been discussed, but the time and uncertainty members have not. The time is the time at which the input event was received by CX. This is not the time at which the response was made, because there is some amount of latency between a response being made and information about that response filtering its way through the computer to CX. The uncertainty member gives the difference in time between the last and second to last times at which input events were polled with Input.pollEvents(). Because events cannot be polled for literally constantly, there is always some amount of time between each polling of events. The uncertainty member captures this uncertainty about when the event actually made it to CX, because the event could have become available at any point between the last and second to last polls. Generally, if the uncertainty is less than 1 ms, there is little cause for concern. If the uncertainty is high, you should use that as an indication that Input.pollEvents() should be called more frequently in the code. Naturally, uncertainty is the lower bound on the true uncertainty. For all CX knows, for any input event, that event could have occurred at any time in the past, so the true uncertainty is effectively unbounded. Because of the uncertantly and latency involved in input, the only correct interpretation of the time member of the event is that it is the time at which the event was given a timestamp by CX. The CX timestamps are probably pretty well correlated with the actual event time.

This example has shown one way of working with keyboard input events. There are a few other important functions for getting input. Instead of using getNextEvent() repeatedly, you can use copyEvents() to get a vector containing a copy of all of the currently stored events. Using copyEvents() does not delete the stored events the way getNext↩ Event() does. To delete all of the stored events, you can use clearEvents().

For working with mouse events, there is very little that is different from working with keyboard events. CX::↩ Instances::Input::Mouse can be worked with in much the same way as the keyboard. The functions to check for available events and to get events have the same names and behaviors (availableEvents() and getNextEvent()). The type of the events for the mouse is CX_Mouse::Event, which has the same time and uncertainty members with identical interpretation as the keyboard events. It also has a type member that you can use to determine if the mouse was moved, clicked, dragged, etc. Instead of a key member, mouse events have a button member, which gives the number of the button that was used. It also gives x and y coordinates of the mouse for the event. CX_Joystick provides a similar interface for joysticks.

For a discussion of some of the issues involved in timestamping responses, see the Timing Issues page.

6. Storing and Outputting Data

Overview

The purpose of having participants do tasks is to get data from them. CX provides the CX_DataFrame as a mechanism for storing data in a structured way and outputting that data for further processing.

The structure of a data frame is essentially the same as a spreadsheet. There are some number of rows, each with a numeric index, and some number of columns, each of which has a unique name. At each combination of row and column, one cell of data is stored.

The type of data stored within each cell is independent of the type of data in other cells of the data frame. However, generally the type of data within a column will be the same for the whole column. Many different types of data can be stored in a CX_DataFrame, the specifics of which will be described later.

In addition to being able to store arbitrary types of data in each cell, a CX_DataFrame can store an arbitrary number of items in each cell. As such, a CX_DataFrame is rectangular in the row and column dimensions, but jagged in the third dimension of cell vectors. This is useful in cases where cells in a column of data are each a logical unit but can vary in length from row to row (perhaps from trial to trial). Without the ability to have a vector of data within a cell, several columns would have to be used instead. To maintain backwards compatibility with spreadsheet software that cannot store vectors of data within a single cell, CX_DataFrame::convertAllVectorColumnsToMultipleColumns() does what its name says.

All of the data that is given to a CX_DataFrame is converted to and stored as a string. If the data is extracted out of the CX_DataFrame, it is converted from a string back to the requested type. This conversion process is slow, so

CX_DataFrames are not suited for computational tasks in which data is repeatedly retrieved and stored in the data frame. In addition, it is possible for floating point data to lose precision when converted to a string, although the loss of precision should be insignificant and, by default, there should be no loss of precision with floating point types.

Once an experiment is complete, the contents of a CX_DataFrame can be printed to a file in a delimited format (tab-delimited by default). Delimited data can be read by pretty much any software that could be used to process it (e.g. R using read.delim, Excel, etc.).

Usage

Using a CX_DataFrame is fairly straightforward. To start, we need to create a CX_DataFrame and put stuff into it.

```
CX_DataFrame df;
df("double", 0) = 3.14;
df("double", 1) = 1.5;
```

On the first line, we created a CX_DataFrame named `df`. On the next line, we start putting in data. Cells of the data frame can be accessed by using operator(), with the user providing a column name (a string) and a row index (an unsigned integer). We access the cell in the column named "double" at row 0 and assign the value 3.14 to it. Then, we do the same for the second row of the "double" column and assign the value 0.5 to it.

Notice that we did not need to tell the data frame that we wanted to make a new column with 2 rows, it did it for us. CX_DataFrames automatically resize themselves to fit new data when you use operator().

We can make other columns with other kinds of data in them. For example, we could make a column with the names of some types of dwellings in it.

```
df("dwellings", 1) = "house";
```

Instead of the `double`s that we used earlier, these values are strings, which works just fine. Note that we skipped over row 0 for the "dwellings" column, which means that there will be an empty cell in row 0. In addition to being able to store a variety of data types, data frames can also store vectors of data.

```
df("vect", 0) = CX::Util::sequence(1, 3, 1);
df(1, "vect") = CX::Util::sequence(9, 5, -2);
```

In the first case, we assigned a vector containing {1, 2, 3} to a cell. In the second case, we did a similar thing, except note that the row index and column name were given in reverse order. Now that we have some data in the data frame, let's look at the data. We can print the contents of a data frame to a string and then print that string to the console.

```
string dataFrameString = df.print("/");
cout << "The initial data in the data frame: " << endl << dataFrameString << endl;
```

First, the contents of `df` are printed with a forward-slash used as the delimiter between cells (the delimiter between the elements of the vectors are left at the default).

You can also extract data from a CX_DataFrame. This is done by reversing which side of the assignment operator the data frame and data appear on.

```
double d = df("double", 0);
int whoops = df("double", 1);
```

Here we read out some values. In the first case, it is a `double` that is being assigned to, so the contents of the cell in the data frame are implicitly converted to `double`. The second line has a programming error on it: We originally put in the value `1.5`, but now we are saying that we want an `int` out, which is not the same type as the value that we put in, which was `double`. In this case, a best-effort attempt is made to give you data of the requested type, but a warning will be logged. The warning will tell you that there is a mismatch between the type of data that was inserted and the type that was extracted. If you notice this in the logs, take care! You could have a serious problem on your hands. Moving on to extracting more types of data:

```
vector<int> intVector = df("vect", 1);
```

You can just put a vector of some type of the left hand side of the assignment operator and the data will be converted to that type. If you only put in a single value, but ask for a vector, you will get a vector of length 1. To extract strings, you need to explicitly request a string:

```
string house = df("dwellings", 1).toString();
```

There is no good reason why you should need to call a function to get a string out, but there is an issue extracting strings that I have not been able to work around. It is possible to explicitly specify which type of data you want to extract by using to<T>(), replacing `T` with the type of your choice. In addition, if you want a vector of data from a cell, you can use toVector<T>(). Using `to()` or `toVector()` might be neccessary if you are not storing the extracted value in a variable, but instead using it in an expression. If you are storing an extracted value in a variable, it's relatively easy for the compiler to figure out what type of data you want without you needing to specify it explicitly (but it is not always possible). It is harder for the compiler to figure it out when a value is used in an expression with other values of various types.

The questioning mind wonders what would happen if we were to read from a cell that has not yet been created, like this:

```
int i = df("ints", 2);
```

First off, a new cell would be created at ("ints", 2) with nothing in it (i.e. a zero-length vector). Then, an attempt will be made to convert that empty cell to an `int`, which is impossible to do in any meaningful way, so an error will be logged and a freshly constructed `int` with some value (probably 0) will be returned. In general, if a type `T` is requested, but there is nothing in the cell, an instance of `T` will be constructed using the default constructor and that value will be returned. If instead of a logged error and an invalid value, you would like to get an exception when attempting an out-of-bounds access, you can use CX::CX_DataFrame::at() instead of operator(). at() does not dynamically resize the data frame, instead throwing a std::out_of_range exception if you are accessing something that does not exist.

Once you have a CX_DataFrame that is full of data, you can easily output it to a file for later processing. You do this with CX::CX_DataFrame::printToFile(), which has several versions. The most simple version only needs a file name and uses default values for the various outputting options. With the following call

```
df.printToFile("myData.txt");
```

the data frame contents will be printed to `PROJECT_DIR/bin/data/myData.txt`. The file will be a tab-delimited text file, which is a file type that can be read by pretty much any program that does data analysis (and if not, you can open it with Excel and change its format). Some of the settings that you can change are the delimiter between cells (default tab: "\t"), whether to output row numbers (default false), what to enclode vector cells with (default double quote: "\""), and what to delimit elements of a vector with (default semicolon: ";"). In addition, you can choose to only print out specific rows and columns of the data frame. To use these more advanced options, there are a number of versions of CX_DataFrame::printToFile() that can be used, the most thorough one taking a CX::CX_DataFrame::OutputOptions struct.

A common issue is that altough it is fine to use vector-containing cells within CX, other software does not support vectors of data within single cells. To deal with this, call

```
df.convertAllVectorColumnsToMultipleColumns(1, true); //Arguments are: start index, delete originals
```

which does what it claims to do. With this function call, we are saying to convert the vectors to multiple columns, naming the new vectors with indices starting from 1 and deleting the old vector column once the conversion is complete. In our example, there is a column named "vect" that has 3 elements per cell. After this function call, there will be 3 columns, named vect1, vect2, and vect3, with each column containing one of the elements of the original cells of vect. If the number of elements per cell is not the same in every row of a column, some rows will have empty cells that follow the last data cell. For the sake of explicitness, the empty cells will be filled with the string "NA".

What types can be stored in a data frame?

All of the types that are built into C++ (e.g. int, float) can be stored. Most of the types that are in the standard library (e.g. std::string) can be stored. Many of the types built into openFrameworks (e.g. ofColor, ofPoint) can be stored. In general, any type `T` can be stored into and retrieved from a CX_DataFrame as long as `T` has overloaded versions of the folowing functions, which are the stream insertion and extraction operators:

```
std::ostream& operator<<(std::ostream&, const T&)
std::istream& operator>>(std::istream&, const T&)
```

It is possible to only define one of these functions and have partial functionality, but honestly, why would you ever do that? Inside of the `src` folder for the dataFrame example, see `myType.h` for an example of how to define these functions for a class. See also the following website: http://geeksquiz.com/overloading-stream-insertion-operators-c/ Alternatively, search the internet for "c++ overload stream insertion and extraction operators".

7. Timing Issues

Input Timing

The cause of problems with input timing can be explained with a rough outline of 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 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 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 CX::CX_InputManager::pollEvents() is not called regularly, input events will be received and everything will appear to be working correctly, but the timestamps will indicate that the event occurred later than it actually did.

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 location (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 many steps in the process that no experimental 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. If you do not use some kind of low-latency button box, my 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 (just don't systematically use different computers for different participant groups). Any random error will simply slightly inflate the estimated response latency variance, but not to any meaningful extent given the base magnitude of human variability.

If you would like to learn more about the internals of how input is handled in CX, you can see how GLFW (the windowing system used by openFrameworks) and openFrameworks manage input by examining the source code

in the respective repositories (also check out the CX source code, of course). There is no documentation on input timing that I am aware of for either GLFW or openFrameworks. For most applications, computers are fast enough that it is not an area that is emphasized by most software developers.

∗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

When it comes to stimulus presentation, CX, like several other pieces of psychology experiment software, uses OpenGL for visual stimuli. With OpenGL, rendering commands are put into buffers and eventually flushed to the video card for actual rendering and presentation. One function that sends the commands is called glFlush(). The OpenGL documentation has this to say about what happens to rendering commands after glFlush() is called: "↩Though this execution may not be completed in any particular time period, it does complete in finite time." (https↩://www.opengl.org/sdk/docs/man2/xhtml/glFlush.xml) That is quite a strong timing guarantee. I mean, it's a lot stronger than allowing for the possibility that the commands might never complete. But seriously, OpenGL makes no timing guarantees. You get what you get and have to empirically verify that it's all going fast enough. Thus, I cannot make any stong claims about stimulus timing in CX because I depend on something that makes no claims about timing, other than that things happen in "finite time". No one who uses OpenGL to render stimuli can make any strong claims about stimulus timing without substantial emperical validation. (As a side note, you don't need to worry about calling glFlush(); it's taken care of for you by CX.)

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 very hard to detect by eye in most cases, so it is important that there is some way to detect errors in stimulus presentation. The best way is with external hardware that provides timestamps giving actual stimulus onset. Barring that, there are some software mechanisms that can be used to attempt to detect errors.

The primary method of presenting time-locked visual stimuli is the CX_SlidePresenter which 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() once a presentation of stimuli is complete. 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_Slide↩Presenter provides a very easy-to-use way to present visual stimuli. 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_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_SoundBuffers 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. More information about audio input and output can be found on the Audio Input and Output page.

Response Latency

Usually, we are not interested in response time (the absolute time at which a response occurred), but rather response latency of a response to a specific stimulus. However, the time at which the stimulus was actually presented

may be misreported by audio or video hardware/software, so even if the response time data had no error whatsoever, when it is compared with the stimulus presentation time, the response latency (response time minus stimulus presentation time) would be wrong due to errors in measured stimulus presentation time. If the timestamps for either stimulus presentation time or response time are off, response latency will be off. No matter how low-latency of a button box you have, if stimulus presentation times are wrong, your button box will not solve the problem of measuring accurate response latencies. Based on this, 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, with the response latency calculated based on these accurate event time measurements. If you do not have a setup like this, then just try to relax and don't worry too much about timing. The vast majority of psychological effects are not that time sensitive and standard computers provide sufficient timing precision for them. However, if what you are studying is dependent on timing, then get proper hardware and software and verify that it all works within the desired error tolerance.

Millisecond Precision

Given the timing issues outlined in this section, it seems natural to end with a brief discussion of the mythical millisecond precision of psychology stimulus software. Psychologists claim to be very concerned with obtaining millisecond precision from their equipment, yet they consistently use off-the-shelf hardware and software that are incapable of providing the desired level of precision. Then they install expensive software for psychology experiments that make claims about the submillisecond precision provided by the software. These claims are usually at best lies of omission. For example, a claim like "CX is accurate to the millisecond" is completely true in a very literal sense. However, the missing footnote on my claim is that it is trivial to write software that is accurate to the millisecond. I can write a little loop that gets a timestamp every time through the loop and show that it takes less than a microsecond to run through the loop on most computers. In that sense, CX is accurate to the microsecond. Awesome! But wait, I said nothing in my claim about controls over stimulus presentation and response collection timing. The reason I didn't is that because CX is software running in a normal operating system, it has very little control over timing. As I have outlined in this section, modern computers are almost literally Rube Goldberg machines, with multiple layers of queues that all data must pass through to get from one point to another. At each queue, there is software that is in charge of managing that queue and moving the data from that queue onward. Each step involves a delay, with both systematic and non-systematic components of the delay. As software running in an operating system, CX is far removed from each of these stages, with no ability to directly affect them. As such, it would be irresponsible to make strong guarantees about timing of stimulus presentation and response collection. Empirically verify that your hardware and software configuration meets your timing needs. I hope that CX is helpful to that end.

8. 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:

```
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. Additionally, code that waits on network resources (e.g. loading a file from an external server) is also typically blocking. Also, loading large files, such as long audio files or video files from the hard drive can be blocking. This is why you should try to load all of your stimuli into RAM at the beginning of the experiment rather than loading them from the hard drive just before they should be presented.

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, 0.033);
```

which requires the allocation of about 300 MB of RAM to store the sequence of numbers from 0 to 1000000 in increments of 0.033. This code doesn't wait for anything to happen, it just takes a long time to execute. It is important to stress that the vast majority of code that does not wait on input or output is not blocking, even if it

is doing something very complex, because modern computers are extremely fast. Allocating a lot of memory can take a long time. Also, applying an algorithm to a large amount of data can take a long time. However, most of the things usually done in psychology experiments take very little time. If you are curious about the amount of time being taken to execute a piece of code, CX provides CX::Util::CX_SegmentProfiler and CX::Util::CX_LapTimer to help with measuring time taken. See the documentation for those classes for information on usage.

If you are trying to collect accurate response data or present stimuli at specific times, one of the worst things you can do is have a section of blocking code that runs while a time-sensitive task is taking place. The main concern with blocking code is that while the code is running some critical timing period passes and a stimulus is not presented or a response is collected but not timestamped correctly. The question, then, is how long can code run before it is considered blocking code. As a rough guideline, if a section of code takes longer than about 1 ms to run, it has the potential to disrupt timing meaningfully.

Using blocking code 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.
```

As long as you aren't also trying to present stimuli, by constantly polling for input, the code will notice input as soon as possible and the timestamp for the input event will be very accurate.

9. Deploying an Experiment

Most of this manual covers how to program an experiment in CX. This section covers how to take an experiment that works in debug mode on a development computer and distribute that experiment to computers on which the experiment will be run.

For all operating systems, one of the most important things to do is to recompile the experiment in a non-debug mode (often called "release" mode). When a program is compiled in debug mode, the compiler adds a large amount of code to the program so that it is easier to debug. One example of something that might be added to a debug version is bounds checking on containers. Assume that you have a vector with 10 elements and you try to access the 12th element using operator[] (the standard way to get data out). This will result in erroneous program behavior regardless of the way in which the program is compiled. However, in debug mode, extra bounds-checking code has been added and when the out-of-bounds access is detected, the debugger triggers a breakpoint and complains that your code is misbehaving. In release mode, no bounds checking is performed, so that what would happen is that whatever is in memory just past the end of the vector would get accessed. This could result in the program crashing right away, or it could result in gradual memory corruption that eventually results in a crash. Thus, release mode programs are much harder to debug than debug mode programs. However, release mode programs do not have all of the extra debugging code added to them and, as a result, run much more quickly than debug mode programs. This speed boost is why you should compile your programs in release mode when preparing to use them to collect data. In addition, there is a class of strange timing-related bugs that only happen in debug mode.

Because the deployment process differs depending on your operating system, each OS will be covered individually. For the big picture, the steps are

1. Compile the experiment in release mode.
2. Collect any additional dependencies (depends on operating system).
3. Copy the contents of `PROJECT_DIRECTORY/bin` to the target computer.

Windows

The result of compilation of a CX experiment is an executable file. This file is `OFDIR/apps/myApps/PROJE↩CT_NAME/bin/PROJECT_NAME.exe`, where `OFDIR` is the root directory of your openFrameworks installation

and PROJECT_NAME is the name of your project. If you have not built a release version of your program yet, you won't have this file. If you have built a debug version of your program, you should have a file called `PROJECT_NAME_debug.exe` in this directory. Also in the `bin` directory, there are a number of files with the same name as the project, but with different file extensions than `.exe`. You do not need these files and may delete them.

Although you have an executable file, it is not without dependencies. In addition to the .exe, there are several .dll files in the `bin` directory. Many of these dlls are needed in order for your program to be able to run and which ones are needed depends on specifics about what your code does. If you would like to test whether you need any of these dlls, you can rename them by, for example, added an "_" to the front of their name and running the executable file. If the experiment opens without error, you do not need the dll that you renamed. A more general way to learn about the dependencies of a Windows executable is to use the Dependency Walker (http://www.dependencywalker.com/).

In addition to the dlls that are in the `bin` directory, you will need some additional dlls that are used by Visual Studio. For Visual Studio 2012, their names are msvcp110.dll and msvcr110.dll (the 110 is because, internally, VS 2012 is version 11.0 of Visual Studio). The first file (distinguished by the "p" in the name) is the C++ standard library implementation and the second file (with "r") is the C runtime library. If the default installation location for Visual Studio was used, these files can be found in `C:\Program Files (x86)\Microsoft Visual Studio 11.0\VC\redist\x86\Microsoft.VC110.CRT\`. Copy them from that location to the `bin` directory in your project to make them available to your executable file regardless of where you copy the `bin` directory to. For more information, you can see this page: https://msdn.microsoft.com/en-us/library/ms235299%28v=vs.110%29.aspx.

If for whatever reason (perhaps testing), you need to deploy a program that was compiled in debug mode, you will need the debug versions of msvcp110.dll and msvcr110.dll. The debug versions of these files are msvcp110d.dll and msvcr110d.dll (note the "d" at the end of the names) and can be found in `C:\Program Files (x86)\Microsoft Visual Studio 11.0\VC\redist\Debug_NonRedist\x86\Microsoft.VC110.DebugCRT\`. Note that part of that path is `Debug_NonRedist`, indicating that these dlls are not supposed to be redistributed under the Visual Studio license terms. I am not a lawyer, but I suspect that if you deploy the debug dlls along with your project to computers within your lab (or even multiple labs within a single institution) that you should be fairly safe from Microsoft coming down on you, especially if you have a good reason (like testing). However, I would strongly recommend against including the debug dlls in anything that is released publicly. In any case, you should only deploy programs that have been compiled in release mode.

Once you have added the two MSCV dlls to the `bin` directory, you should be able to just copy all of the contents of `bin` to another computer and have the experiment run. This is why the `data` directory is within `bin`: So that the `bin` directory is a self-contained copy of the experiment, including all data files.

In Visual Studio, switching from debug mode to release mode is easy. In the default view, in the toolbars at the top of the screen, there should be a dropdown menu that has the word "Debug" in it as the currently selected option. Click on the dropdown and select "Release". The next time you compile the project, it will be compiled in release mode. It may take a long time to compile the first time, as everything needs to be recompiled.

Windows XP

If you are deploying to a computer running Windows XP, there is an additional step. You must compile your project and the openFrameworks project to be compatible with Windows XP, which it is not done by default (Windows XP is obsolete). Assume that you are in Visual Studio and the Solution Explorer pane is visible. At the top level, there will be a Solution that has the same name as your project. In terms of the hierarchy, directly under the Solution there will be your project and a project titled openframeworksLib. You must complete the following steps for both projects.

Right-click on the project and select Properties. A window titled PROJECT_NAME Property Pages should open. On the left, under Configuration Properties, select General. On the right, there should be a setting called "Platform Toolset". If you click on the selection, a dropdown button should appear and if you click on that, a dropdown menu should appear. Select the option called "Visual Studio 2012 - Windows XP (v110)".

Again, do this for both your project and the openframeworksLib project, then recompile the project. If you compile for Windows XP, it will work on some newer versions of Windows as well, but there may be some versions of Windows that it will not work for.

Linux

In general, the situation is a bit easier on Linux than on Windows. For one thing, many of the program dependencies that come as dlls on Windows get compiled into the executable on Linux. You should not need to dig around to get extra dependencies and should be able to just deploy the contents of the `bin` directory to other Linux computers. Of course, you would probably be well-served to use the same distribution and version of Linux on the development and deployment computers to avoid any hassle about different versions of Linux.

10. Frequently Asked Questions

How do I cite CX?

There are a couple of details you will need to modify in this citation, depending on which version of CX you are using.

Hardman, K.O. (2015). CX (Version 0.1.1) [Computer Software]. Available from https://github.com/hardmanko/ofxCX/releases/tag/v0.1.1.

Make sure to update the version number to whatever version you are using and get the "Retrieved from" URL by going to the release page on GitHub for the release that you are using. Note that the trailing period on the URL is there because APA format dictates it, not because it is part of the URL.

If you are using a development version of the software (e.g. you downloaded the latest version of the master branch), give the version as the hash of the commit you downloaded. If you don't remember which commit you downloaded, you have almost no recourse other than trying to figure out which commit you downloaded, which is hard. I strongly recommend only using tagged releases.

IF YOU HAVE MODIFIED THE SOURCE CODE FOR CX IN ANY WAY, YOU MUST MENTION THIS IN TEXT. In this case, you should still cite CX, but say that you have forked it.

Can I have multiple versions of CX installed at the same time?

Yes, you just need to have the different versions in different directories in the openFrameworks addons directory. For example, if you wanted to have CX 0.1.1 and CX 0.1.0 both installed, you would need to have two folders in addons, one called (for example) ofxCX-0.1.1 and the other called ofxCX-0.1.0. To choose which version of CX you want to use, when you create a project with the openFrameworks project generator, you just need to select the version of the CX addon that you want to use.

Where should I put files related to an experiment?

All files related to an experiment should go into `PROJECT_DIRECTORY/bin/data`. This makes it possible to just copy the `bin` directory when deploying an experiment, because all neccessary files are in `bin` (the executable program files) or the `data` subdirectory within `bin`.

Where are the files I created in a CX experiment if I didn't give an absolute path?

Any files that are created (e.g. printing a CX_DataFrame to a file) for which you don't provide an absolute path go into `PROJECT_DIRECTORY/bin/data`.

My experiment does not work on Windows XP!

See the Deploying an Experiment page for a solution.

I've found a bug in CX. How do I tell you about it?

Submit a bug report on the `CX issues page` on GitHub.

11. Program Model

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 in turn 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).

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.

Additionally, CX uses the colorspace package by Pascal Getreuer (http://www.getreuer.info/home/colorspace).

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_SoundBufferPlayer.

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) and 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.

Normally in an oF app, there are update and draw events that fire regularly. These events can be listed to by various pieces of code so that they can know when to update their state and when to draw anything that they need to draw. In CX, this functionality is broken by default. However, if you are using something that needs these events, you can call ofNotifyUpdate() to notify that anything listening should update. Also, after you start drawing, e.g. with CX_Display::beginDrawingToBackBuffer(), you can call ofNotifyDraw() to tell anything listening to draw itself.

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 look at 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 . 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 events. This is explained more in the Response 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 designed to make doing that as easy and painless as possible, while still giving you as much control over your experiment as possible.

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. If you don't like the default rendering environment, use CX::reopenWindow() to make a new window with different settings. 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. The monitor's refresh rate is checked with CX_Display↩ ::estimateFramePeriod().

12. license

The code in this repository (hardmanko/ofxCX), with exception of anything within the libs subfolder, is available under the MIT License (https://secure.wikimedia.org/wikipedia/en/wiki/Mit_license). Anything within the libs subfolder is copyrighted by the respective copyright holder under whatever license they chose.

Copyright (c) 2015 Kyle Hardman

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, IN↩ CLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

13. Module Index

13.1. Modules

Here is a list of all modules:

14. Namespace Index

14.1. Namespace List

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

15. Hierarchical Index

15.1. Class Hierarchy

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

16. Class Index

16.1. Class List

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

17. Module Documentation

17.1. Data

Classes

- class CX::CX_DataFrame::IoOptions
- struct CX::CX_DataFrame::OutputOptions
- class CX::CX_DataFrame
- class CX::CX_DataFrameColumn
- class CX::CX_DataFrameRow
- class CX::CX_DataFrameCell

17.1.1 Detailed Description

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

17.2. Entry Point

Macros

- #define CX_NO_MAIN

Functions

- void runExperiment (void)

Variables

- CX_InputManager CX::Instances::Input = CX::Private::inputManagerFactory()
- CX_Display CX::Instances::Disp
- CX_Logger CX::Instances::Log
- CX_RandomNumberGenerator CX::Instances::RNG

17.2.1 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()).

17.2.2 Macro Definition Documentation

17.2.2.1 #define CX_NO_MAIN

If this preprocessor macro is defined, CX will not produce a `main` function, leaving it up to the user to produce such a function.

A `main` function can be as simple as:

```
void main (void) {
    CX::initializeCX(CX_InitConfiguation());

    runExperiment();
}
```

17.2.3 Function Documentation

17.2.3.1 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() at any point.
}
```

17.2.4 Variable Documentation

17.2.4.1 CX::CX_Display CX::Instances::Disp

An instance of CX::CX_Display that is lightly hooked into the CX backend. The only thing that happens outside of user code is that during CX setup, before reaching user code in runExperiment(), CX_Display::setup() is called.

17.2.4.2 CX::CX_InputManager CX::Instances::Input = CX::Private::inputManagerFactory()

An instance of CX_InputManager that is exceedingly lightly hooked into the CX backend. The only way in which this is used that is not in user code, is that input events are polled for once during setup, which helps operating systems know that the program is still responding.

17.2.4.3 CX::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. After runExperiment() returns, CX::Instances::Log.flush() is called.

17.2.4.4 CX_RandomNumberGenerator CX::Instances::RNG

An instance of CX_RandomNumberGenerator that is very lightly hooked into the CX backend. The only way this is used outside of user code is to generate random numbers internally in, e.g., Algo::BlockSampler.

17.3. Input Devices

Classes

- class CX::CX_InputManager
- struct CX::CX_Joystick::Event
- class CX::CX_Joystick
- struct CX::CX_Keyboard::Keycodes
- struct CX::CX_Keyboard::Event
- class CX::CX_Keyboard
- struct CX::CX_Mouse::Event
- class CX::CX_Mouse

Enumerations

- enum CX::CX_Joystick::EventType { CX::CX_Joystick::BUTTON_PRESS, CX::CX_Joystick::BUTTON_RE↩ LEASE, CX::CX_Joystick::AXIS_POSITION_CHANGE }
- enum CX::CX_Keyboard::EventType { CX::CX_Keyboard::PRESSED, CX::CX_Keyboard::RELEASED, CX↩ ::CX_Keyboard::REPEAT }
- enum CX::CX_Mouse::Buttons { **LEFT** = OF_MOUSE_BUTTON_LEFT, **MIDDLE** = OF_MOUSE_BUTTO↩ N_MIDDLE, **RIGHT** = OF_MOUSE_BUTTON_RIGHT }
- enum CX::CX_Mouse::EventType { CX::CX_Mouse::MOVED, CX::CX_Mouse::PRESSED, CX::CX_Mouse::RELEASED, CX::CX_Mouse::DR↩ AGGED, CX::CX_Mouse::SCROLLED }

17.3.1 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/o Serial.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.

17.3.2 Enumeration Type Documentation

17.3.2.1 enum CX::CX_Mouse::Buttons

Names of the mouse buttons corresponding to some of the integer button identifiers.

17.3.2.2 enum CX::CX_Joystick::EventType

The type of the joystick event.

Enumerator

BUTTON_PRESS A button on the joystick has been pressed. See Event::buttonIndex and Event::buttonState for the event data.

BUTTON_RELEASE A button on the joystick has been released. See Event::buttonIndex and Event::button↩ State for the event data.

AXIS_POSITION_CHANGE The joystick has been moved in one of its axes. See Event::axisIndex and Event::axisPosition for the event data.

17.3.2.3 enum CX::CX_Mouse::EventType

The type event that caused the creation of a CX_Mouse::Event.

Enumerator

MOVED The mouse has been moved without a button being held. Event::button should be -1 (meaningless).

PRESSED A mouse button has been pressed. Check Event::button for the button index and Event::x and Event::y for the location.

RELEASED A mouse button has been released. Check Event::button for the button index and Event::x and Event::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. Event::button may not be meaningful because the held button

SCROLLED Event::x if your mouse has a wheel that can move horizontally. The mouse wheel has been scrolled. Check Event::y to get the change in the standard mouse wheel direction, or

17.3.2.4 enum CX::CX_Keyboard::EventType

The type of the keyboard event.

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 key-presses to be rapidly sent. This event is one of the many repeats.

17.4. Message Logging

Classes

- struct CX::CX_Logger::MessageFlushData
- class CX::CX_Logger

Enumerations

- enum CX::CX_Logger::Level {
  LOG_ALL = 0, **LOG_VERBOSE** = 1, **LOG_NOTICE** = 2, **LOG_WARNING** = 3,
  LOG_ERROR = 4, **LOG_FATAL_ERROR** = 5, **LOG_NONE** = 6 }

17.4.1 Detailed Description

This module is designed for logging error, warnings, and other messages. The primary interface is the CX_Logger class, in particular the preinstantiated CX::Instances::Log.

17.4.2 Enumeration Type Documentation

17.4.2.1 enum CX::CX_Logger::Level [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).

17.5. Randomization

Classes

- class CX::CX_RandomNumberGenerator

17.5.1 Detailed Description

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

An instance of this class is preinstsantiated for you: CX::Instances::RNG.

17.6. Sound

Namespaces

- CX::Synth

Classes

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

17.6.1 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.

17.7. Timing

Classes

- class CX::CX_Clock
- class CX::CX_BaseClockInterface
- struct CX::CX_Time_t< TimeUnit >::PartitionedTime
- class CX::CX_Time_t< TimeUnit >
- class CX::Util::CX_LapTimer
- class CX::Util::CX_SegmentProfiler

Variables

- CX_Clock CX::Instances::Clock

17.7.1 Detailed Description

This module provides methods for timestamping events in experiments.

17.7.2 Variable Documentation

17.7.2.1 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.

17.8. Utility

Namespaces

- CX::Util

Classes

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

Enumerations

- enum CX::Util::CX_RoundingConfiguration { CX::Util::CX_RoundingConfiguration::ROUND_TO_NEARES↩T, CX::Util::CX_RoundingConfiguration::ROUND_UP, CX::Util::CX_RoundingConfiguration::ROUND_DO↩WN, CX::Util::CX_RoundingConfiguration::ROUND_TOWARD_ZERO }

17.8.1 Detailed Description

17.8.2 Enumeration Type Documentation

17.8.2.1 enum CX::Util::CX_RoundingConfiguration [strong]

The way in which numbers should be rounded with CX::Util::round().

Enumerator

ROUND_TO_NEAREST Round to the nearest number.

ROUND_UP Round to the number above the current number.

ROUND_DOWN Round to the number below the current number.

ROUND_TOWARD_ZERO Round toward zero.

17.9. Video

Namespaces

- CX::Draw

Classes

- class CX::CX_Display
- struct CX::CX_SlidePresenter::FinalSlideFunctionArgs
- struct CX::CX_SlidePresenter::PresentationErrorInfo
- struct CX::CX_SlidePresenter::Configuration
- struct CX::CX_SlidePresenter::SlideTimingInfo
- struct CX::CX_SlidePresenter::Slide
- class CX::CX_SlidePresenter

Enumerations

- enum CX::Draw::LineCornerMode { **OUTER_POINT**, **BEZIER_ARC**, **STRAIGHT_LINE** }
- enum CX::CX_SlidePresenter::ErrorMode { **PROPAGATE_DELAYS** }
- enum CX::CX_SlidePresenter::SwappingMode { CX::CX_SlidePresenter::SwappingMode::SINGLE_COR↩E_BLOCKING_SWAPS, CX::CX_SlidePresenter::SwappingMode::MULTI_CORE }
- enum CX::CX_SlidePresenter::Slide::PresStatus : int { CX::CX_SlidePresenter::Slide::PresStatus::NOT_STARTED, CX::CX_SlidePresenter::Slide::PresStatus↩::RENDERING, CX::CX_SlidePresenter::Slide::PresStatus::SWAP_PENDING, CX::CX_SlidePresenter::↩Slide::PresStatus::IN_PROGRESS, CX::CX_SlidePresenter::Slide::PresStatus::FINISHED }

17.9.1 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 with openFrameworks: See the graphics and 3d sections of this page: http://www.openframeworks.cc/documentation/.

17.9.2 Enumeration Type Documentation

17.9.2.1 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.

17.9.2.2 enum CX::Draw::LineCornerMode [strong]

Settings for how the corners are drawn for the lines() function.

17.9.2.3 enum CX::CX_SlidePresenter::Slide::PresStatus : int [strong]

The possible presentation statuses of the slide.

Enumerator

NOT_STARTED The slide is somewhere in the queue awaiting start.

RENDERING The slide is next in line for presentation and its rendering has started.

SWAP_PENDING The slide is next in line for presentation and its rendering has completed, but it has not been swapped in.

IN_PROGRESS The slide has been swapped in and is now on screen, assuming that the rending completed before the swap.

FINISHED The slide has been replaced with a new slide.

17.9.2.4 enum CX::CX_SlidePresenter::SwappingMode [strong]

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 theoretically 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 of 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 CX↩ _SlidePresenter::Configuration::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. However, it avoids thread synchronization issues, which is a huge plus.

Enumerator

SINGLE_CORE_BLOCKING_SWAPS The slide presenter does bufer swapping in the main thread, blocking briefly during the buffer swap.

MULTI_CORE The slide presenter does bufer swapping in a secondary thread, which means that there is no blocking in the main thread when buffers are swapping.

18. Namespace Documentation

18.1. CX::Algo Namespace Reference

Classes

- class BlockSampler
- class LatinSquare

Functions

- template<typename dataT , typename distT >
  std::vector< dataT > generateSeparatedValues (int count, distT minDistance, std::function< distT(dataT, dataT)> distanceFunction, std::function< dataT(void)> randomDeviate, unsigned int maxSequentialFailures, int maxRestarts)
- template<typename T >
  std::vector< std::vector< T > > fullyCross (std::vector< std::vector< T > > factors)
- template<typename T >
  CX_DataFrame fullyCross (std::map< std::string, std::vector< T >> &factors)

18.1.1 Detailed Description

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

18.1.2 Function Documentation

18.1.2.1 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
```

18.1.2.2 template<typename T > CX_DataFrame CX::Algo::fullyCross (std::map< std::string, std::vector< T >> & *factors*)

This function does the same thing as CX::Algo::fullyCross(std::vector< std::vector<T> > factors), except that it returns a CX_DataFrame, which means that you can access factor values by the name of the factor, rather than an index. You can see this in the example.

```
string shapes[3] = { "square", "rectangle", "triangle" };
vector<string> shapesV = Util::arrayToVector(shapes, 3);

string numbers[2] = { "1.5", "3.7" };
vector<string> numbersV = Util::arrayToVector(numbers, 2);

map<string, vector<string>> factors;
factors["shapes"] = shapesV;
factors["numbers"] = numbersV;

CX_DataFrame crossed = Algo::fullyCross(factors);

cout << crossed.print() << endl;

double firstNumber = crossed(0, "numbers").toDouble();
string secondShape = crossed(1, "shapes").toString();
```

Template Parameters

T	The type of data to use. Typically, using `string`s works well, as you can stringify a number (or other type) and then extract that type from the CX_DataFrame, as can be seen in the example with the "numbers" factor.

Parameters

factors	A map that uses the name of a factor as the key and a vector of factor levels as the value.

18.1.2.3 template<typename dataT , typename distT > std::vector< dataT > CX::Algo::generateSeparatedValues (int *count*, distT *minDistance*, std::function< distT(dataT, dataT)> *distanceFunction*, std::function< dataT(void)> *randomDeviate*, unsigned int *maxSequentialFailures*, int *maxRestarts*)

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. Distance does not even need to be unidimensional: note that the type of data used for distance is a template parameter. The distance type must have operator<(distT,distT) defined.

The second function generates random values of the type.

Template Parameters

<*dataT*>	The type of data you are working with.
<*distT*>	The type of distance units used.

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.
max↩ Sequential↩ Failures	The maximum number of times in a row that a newly-generated value can be less than min↩ Distance from at least one other value. If this number of failures is reached, the process will be restarted depending on the setting of maxRestarts. This is to help make sure that if the algorithm is having a hard time finding a value that works given the other values that have been selected, it doesn't just run forever, but tries over with new values.
maxRestarts	If non-negative, the number of times that the algorithm will restart before giving up. If negative, the algorithm will never give up. Note that this may result in an infinite loop if it is impossible to get enough samples.

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 from each other.
std::vector<ofPoint> getObjectLocations(int locCount, float minDistance, ofPoint minimumValues, ofPoint
      maximumValues) {
    //pointDistance is an anonymous function that takes two ofPoints as arguments and returns a float.
    auto pointDistance = [](ofPoint a, ofPoint b) -> float {
        return a.distance(b);
    };

    //randomPoint is an anonymous function that takes no arguments explicitly, but it captures by reference
       everything from
    //the enclosing environment (specifically minimumValues and maximumValues).
    auto randomPoint = [&]() -> ofPoint {
        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, float>(locCount, minDistance, pointDistance,
      randomPoint, 1000, 100);
}

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

18.2. CX::Draw Namespace Reference

Classes

- struct EnvelopeProperties
- class Gabor
- struct GaborProperties
- struct WaveformProperties

Enumerations

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

Functions

- ofPath squircleToPath (double radius, double amount)
- void squircle (ofPoint center, double radius, double amount, double rotationDeg)
- 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, float rotationDeg)

- void centeredString (int x, int y, std::string s, ofTrueTypeFont &font)
- void centeredString (ofPoint center, std::string s, ofTrueTypeFont &font)
- std::string wordWrap (std::string s, float width, ofTrueTypeFont &font)
- void lines (std::vector< ofPoint > points, float lineWidth, bool circleJoins)
- 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)
- std::vector< ofPoint > getBezierVertices (std::vector< ofPoint > controlPoints, std::vector< float > times)
- std::vector< ofPoint > getBezierVertices (std::vector< ofPoint > controlPoints, unsigned int resolution)
- void bezier (std::vector< ofPoint > controlPoints, float width, unsigned int resolution)
- std::vector< double > convertColors (std::string conversionFormula, double S1, double S2, double S3)
- ofFloatColor convertToRGB (std::string inputColorSpace, double S1, double S2, double S3)
- std::vector< ofPoint > getFixationCrossVertices (float armLength, float armWidth)
- ofPath fixationCrossToPath (float armLength, float armWidth)
- void fixationCross (ofPoint location, float armLength, float armWidth)
- void saveFboToFile (ofFbo &fbo, std::string filename)
- ofPath lines (std::vector< ofPoint > points, float width, LineCornerMode cornerMode)
- template<typename ofColorType >
  std::vector< ofColorType > getRGBSpectrum (unsigned int colorCount)
- template<typename T >
  ofVbo colorArcToVbo (ofPoint center, std::vector< ofColor_< T >> colors, float radiusX, float radiusY, float width, float angleBegin, float angleEnd)
- template<typename T >
  void colorArc (ofPoint center, std::vector< ofColor_< T >> colors, float radiusX, float radiusY, float width, float angleBegin, float angleEnd)
- template<typename T >
  ofVbo colorWheelToVbo (ofPoint center, std::vector< ofColor_< T >> colors, float radius, float width, float angle)
- template<typename T >
  void colorWheel (ofPoint center, std::vector< ofColor_< T >> colors, float radius, float width, float angle)
- template<typename T >
  void patternMask (ofPoint center, float width, float height, float squareSize, std::vector< ofColor_< T >> colors=std::vector< ofColor_< T >>(0))
- ofFloatPixels waveformToPixels (const WaveformProperties &properties)
- ofFloatPixels envelopeToPixels (const EnvelopeProperties &properties)
- ofFloatPixels gaborToPixels (const GaborProperties &properties)
- ofFloatPixels gaborToPixels (ofColor color1, ofColor color2, const ofFloatPixels &wave, const ofFloatPixels &envelope)
- ofTexture gaborToTexture (const GaborProperties &properties)
- ofTexture gaborToTexture (ofColor color1, ofColor color2, const ofFloatPixels &wave, const ofFloatPixels &envelope)
- void gabor (ofPoint center, const GaborProperties &properties)
- void gabor (ofPoint center, ofColor color1, ofColor color2, const ofFloatPixels &wave, const ofFloatPixels &envelope)

18.2.1 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.

18.2.2 Function Documentation

18.2.2.1 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.

18.2.2.2 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 in the positive y-direction 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.

18.2.2.3 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(std::vector<ofPoint>, float) 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 (`resolution + 1` points).

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

Equivalent to a call to CX::Draw::centeredString(ofPoint, std::string, ofTrueTypeFont&) with the x and y values in the point.

18.2.2.5 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.

18.2.2.6 template<typename T > void CX::Draw::colorArc (ofPoint *center*, std::vector< ofColor_< T >> *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`.

18.2.2.7 template<typename T > ofVbo CX::Draw::colorArcToVbo (ofPoint *center*, std::vector< ofColor_< T >> *colors*, float *radiusX*, float *radiusY*, float *width*, float *angleBegin*, float *angleEnd*)

See CX::Draw::colorArc(ofPoint, std::vector<ofColor_<T>>, float, float, float, float, float) for documentation of the parameters for this function. The only difference is that this function returns an ofVbo, which a complicated thing you can learn about here: http://www.openframeworks.cc/documentation/gl/ofVbo.html The ofVbo is ready to be drawn without any further processing as in the following code snippet.

```
ofVbo colorArc = Draw::colorArcToVbo( ***arguments go here*** );
colorArc.draw(GL_TRIANGLE_STRIP, 0, colorArc.getNumVertices());
```

The arguments given to the draw function should be given exactly as in the example, except for the name of the ofVbo instance.

18.2.2.8 template<typename T > void CX::Draw::colorWheel (ofPoint *center*, std::vector< ofColor_< T >> *colors*, float *radius*, float *width*, float *angle*)

Draws a color wheel (really, a ring) with specified colors. It doesn't look quite right if there isn't any empty space in the middle of the ring.

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.

```
//This code snippet draws an isoluminant color wheel to the screen using color conversion from LAB to RGB.
//Move the mouse and turn the scroll wheel to see different slices of the LAB space.
#include "CX.h"

void runExperiment(void) {

    Input.setup(false, true);

    float L = 50;
    float aOff = 40;
    float bOff = 40;

    while (true) {
```

```
        if (Input.pollEvents()) {
            while (Input.Mouse.availableEvents() > 0) {
                CX_Mouse::Event mev = Input.Mouse.getNextEvent();

                if (mev.type == CX_Mouse::SCROLLED) {
                    L += mev.y;
                }

                if (mev.type == CX_Mouse::MOVED) {
                    aOff = mev.x - Disp.getCenter().x;
                    bOff = mev.y - Disp.getCenter().y;
                }
            }

            //Now that input has been received, redraw the color wheel
            vector<ofFloatColor> wheelColors(100);

            for (int i = 0; i < wheelColors.size(); i++) {
                float angle = (float)i / wheelColors.size() * 2 * PI;
                float A = sin(angle) * aOff;
                float B = cos(angle) * bOff;

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

            Disp.beginDrawingToBackBuffer();
            ofBackground(0);
            Draw::colorWheel(Disp.getCenter(), wheelColors, 200, 70, 0);

            stringstream ss;
            ss << "L: " << L << "\nA offset: " << aOff << "\nB offset: " << bOff;
            ofSetColor(255);
            ofDrawBitmapString(ss.str(), Disp.getCenter().x, Disp.
    getCenter().y);

            Disp.endDrawingToBackBuffer();
            Disp.swapBuffers();
        }
    }
}
```

18.2.2.9 template<typename T > ofVbo CX::Draw::colorWheelToVbo (ofPoint *center,* std::vector< ofColor_< T >> *colors,* float *radius,* float *width,* float *angle*)

See CX::Draw::colorWheel(ofPoint, std::vector<ofColor_<T>>, float, float, float) for documentation. The only difference is that this function returns an ofVbo. See CX::Draw::colorArcToVbo() for an example for how to draw the ofVbo.

18.2.2.10 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.

Ranges for the values for some common color spaces:

- HSV/HSB/HSL/HSI: For any of these color spaces, H is in the range [0,360) and the other components are in the range [0,1].
- RGB: All in [0,1].
- LAB: L is in the range [0,100]. A and B have vague ranges, because at certain values, the color that results cannot exist (an "imaginary color"). However, in general, A and B should be in the approximate range [-128,128], although the edges are likely to be imaginary.

Parameters

S1	Source coordinate 1. Corresponds to, e.g., the R in RGB.
S2	Source coordinate 2. Corresponds to, e.g., the G in RGB.
S3	Source coordinate 3. Corresponds to, e.g., the B in RGB.

Returns

An vector of length 3 containing the converted coordinates in the destination color space. The value at index 0 corresponds to the first letter in the resulting color space and the next two indices proceed as expected.

```
vector<double> hslValues = Draw::convertColors("XYZ -> HSL", .7, .4, .6); //Convert
       x=.7, y=.4, z=.6 to HSL values.
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 (something to RGB).

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

This function converts from an arbitrary 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. This uses CX::Draw::convertColors(std↩ ::string, double, double, double), which provides more options.

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; it's the same interface in both the MATLAB and C versions).
S1	Source coordinate 1. Corresponds to, e.g., the R in RGB.
S2	Source coordinate 2. Corresponds to, e.g., the G in RGB.
S3	Source coordinate 3. Corresponds to, e.g., the B in RGB.

Returns

An `ofFloatColor` contaning the RGB coordinates. Instances of `ofFloatColor` can be implicitly converted in assignment to other `ofColor` types.

See also

Example code in the documentation for CX::Draw::colorWheel() uses this function.

18.2.2.12 ofFloatPixels CX::Draw::envelopeToPixels (const EnvelopeProperties & *properties*)

Draws a two-dimensional envelope to an ofFloatPixels. An example of how this can be used is to create the alpha blending falloff effect seen in gabor patches as they fade out toward their edges. There is only a single channel in the pixels, which can be used for alpha blending or other kinds of blending effects. Because the type of the color that is used is ofFloatColor, you can access the value of each pixel like this:

```
ofFloatPixels result = Draw::envelopeToPixels(properties); //Get the pixels
float level = result.getColor(1,2).getBrightness(); //where 1 and 2 are some x and y coordinates
```

Parameters

properties	The properties of the envelope.

Returns

An ofFloatPixels containing the envelope.

18.2.2.13 void CX::Draw::fixationCross (ofPoint *location,* float *armLength,* float *armWidth*)

Draws a standard fixation cross (plus sign).

Parameters

location	Where to draw the fixation cross.
armLength	The length of the arms of the cross (end to end, not from the center).
armWidth	The width of the arms.

18.2.2.14 ofPath CX::Draw::fixationCrossToPath (float *armLength,* float *armWidth*)

Draws a standard fixation cross (plus sign) to an ofPath. The fixation cross will be centered on (0,0) in the ofPath.

Parameters

armLength	The length of the arms of the cross (end to end, not from the center).
armWidth	The width of the arms.

Returns

An ofPath containing the fixation cross.

18.2.2.15 void CX::Draw::gabor (ofPoint *center,* const GaborProperties & *properties*)

Draws a gabor pattern with the specified properties. See the renderingTest example for an example of the use of this function.

Parameters

center	The location of the center of the pattern.
properties	The settings to be used to generate the pattern.

See also

CX::Draw::Gabor for a more computationally efficient way to draw gabors.

18.2.2.16 void CX::Draw::gabor (ofPoint *center,* ofColor *color1,* ofColor *color2,* const ofFloatPixels & *wave,* const ofFloatPixels & *envelope*)

This version of gabor() uses precalculated waves and envelopes. This can save time. However, if speed is the primary concern, the class CX::Draw::Gabor should be used instead.

Parameters

center	The location of the center of the pattern.
color1	The first color of the waves.
color2	The second color of the waves.

wave	A precalculated waveform pattern. Must only have a single channel of color data(i.e.↩ is greyscale).
envelope	A precalculated envelope. Must only have a single channel of color data(i.e.is greyscale).

Returns

An ofFloatPixels containing the gabor pattern.It cannot be drawn directly, but can be put into an ofTexture and drawn from there, for example.

18.2.2.17 ofFloatPixels CX::Draw::gaborToPixels (const GaborProperties & *properties*)

Just like Draw::gabor(ofPoint, const GaborProperties&), except that instead of drawing the pattern, it returns it in an ofFloatPixels object.

Parameters

properties	The settings to be used to generate the pattern.

Returns

An ofFloatPixels containing the gabor pattern. It cannot be drawn directly, but can be put into an ofTexture and drawn from there, for example.

18.2.2.18 ofFloatPixels CX::Draw::gaborToPixels (ofColor *color1,* ofColor *color2,* const ofFloatPixels & *wave,* const ofFloatPixels & *envelope*)

This version of gaborToPixels uses precalculated waves and envelopes. This can save time. However, if speed is the primary concern, the class CX::Draw::Gabor should be used instead.

Parameters

color1	The first color of the waves.
color2	The second color of the waves.
wave	A precalculated waveform pattern. Must only have a single channel of color data (i.e. is greyscale).
envelope	A precalculated envelope. Must only have a single channel of color data (i.e. is greyscale).

Returns

An ofFloatPixels containing the gabor pattern. It cannot be drawn directly, but can be put into an ofTexture and drawn from there, for example.

18.2.2.19 ofTexture CX::Draw::gaborToTexture (const GaborProperties & *properties*)

Just like Draw::gabor(ofPoint, const GaborProperties&), except that instead of drawing the pattern, it returns it in an ofTexture object.

18.2.2.20 ofTexture CX::Draw::gaborToTexture (ofColor *color1,* ofColor *color2,* const ofFloatPixels & *wave,* const ofFloatPixels & *envelope*)

Just like Draw::gabor(ofPoint, ofColor, ofColor, const ofFloatPixels&, const ofFloatPixels&), except that instead of drawing the pattern, it returns it in an ofTexture object.

18.2.2.21 std::vector< ofPoint > CX::Draw::getBezierVertices (std::vector< ofPoint > *controlPoints,* std::vector< float > *times*)

Gets the vertices needed to draw a bezier curve.

Parameters

controlPoints	Control points for the bezier.
times	A vector of "times" in the interval [0,1] giving the times at which to evaluate the bezier curve. Values outside of the interval [0,1] are clamped to be in the interval.

Returns

A vector of points along the bezier curve.

18.2.2.22 std::vector< ofPoint > CX::Draw::getBezierVertices (std::vector< ofPoint > *controlPoints,* unsigned int *resolution*)

Gets the vertices needed to draw a bezier curve.

Parameters

controlPoints	Control points for the bezier.
resolution	Controls the approximation of the bezier curve. There will be `resolution` line segments drawn to complete the curve (`resolution + 1` points).

Returns

A vector of points created based on the `controlPoints`.

18.2.2.23 std::vector< ofPoint > CX::Draw::getFixationCrossVertices (float *armLength,* float *armWidth*)

Gets teh vertices defining the perimeter of a standard fixation cross (plus sign).

Parameters

armLength	The length of the arms of the cross (end to end, not from the center).
armWidth	The width of the arms.

Returns

A vector with the 12 needed vertices.

18.2.2.24 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, or ofColor_<someOtherType>.

Parameters

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

Returns

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

18.2.2.25 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).

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

This function draws a line from p1 to p2 with the given width. Note that this function is purely 2D: The Z coordinate is basically ignored and should be 0 for best performance.

Parameters

p1	One end of the line.
p2	The other end of the line.
width	The width of the line.

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

This function draws a series of line segments to connect the given points. At each point, if `circleJoins` is `true` the line segments are joined with a circle, which results in overdraw. Overdraw means that some areas are drawn twice, which means that if transparency is used, it results in differing colors at the overdrawn areas. A (very inefficient) workaround is to draw with max alpha into an fbo and then draw the fbo with transparency. A more advanced version of this function that attempts to prevent overdraw is Draw::lines(std::vector<ofPoint> points, float width, LineCornerMode cornerMode), but that function can break in various ways.

Parameters

points	The points to connect with lines.
lineWidth	The width of the line.
circleJoins	Whether each junction of two lines should have a circle drawn over it.

18.2.2.28 ofPath CX::Draw::lines (std::vector< ofPoint > *points*, float *width*, LineCornerMode *cornerMode*)

This function is an experimental attempt to draw a collection of lines in an idealized way.

18.2.2.29 template<typename T > void CX::Draw::patternMask (ofPoint *center*, float *width*, float *height*, float *squareSize*, std::vector< ofColor_< T >> *colors* = `std::vector<ofColor_<T>>(0)`)

This function draws a pattern mask created with a large number of small squares.

Parameters

center	The mask will be centered at this point.
width	The width of the area to draw to, in pixels.
height	The height of the are ato draw to, in pixels.
squareSize	The size of each small square making up the shape, in pixels.
colors	Optional. If a vector of colors is provided, colors will be sampled in blocks using an Algo::↩ BlockSampler from the provided colors. If no colors are provided, each color will be chosen randomly by sampling a hue value in the HSB color space, with the S and B held constant at maximum values (i.e. each color will be a bright, fully saturated color).

18.2.2.30 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 with ofCircle.

18.2.2.31 void CX::Draw::saveFboToFile (ofFbo & *fbo,* std::string *filename*)

Saves the contents of an ofFbo to an image 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. If no file extention is given, nothing gets 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).

18.2.2.32 void CX::Draw::squircle (ofPoint *center,* double *radius,* double *amount,* double *rotationDeg*)

This function draws an approximation of a squircle (`http://en.wikipedia.org/wiki/Squircle`) using Bezier curves.

Parameters

center	The squircle will be drawn centered at `center`.
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.
rotationDeg	The amount to rotate the squircle, in degrees.

Note

If more control over the drawing of the squircle is desired, use squircleToPath() and then modify the ofPath.

18.2.2.33 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 to an ofPath. 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.

18.2.2.34 void CX::Draw::star (ofPoint *center,* unsigned int *numberOfPoints,* float *innerRadius,* float *outerRadius,* 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.
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.

18.2.2.35 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.

See also

CX::Draw::star()

18.2.2.36 ofFloatPixels CX::Draw::waveformToPixels (const **WaveformProperties** & *properties*)

This function draws a two-dimensional waveform pattern to an ofFloatPixels objects. The results of this function are not intended to be used directly, but to be applied to an image, for example. The pattern lacks color information, but can be used as an alpha mask, used to control color mixing, or otherwise.

Parameters

properties	The properties that will be used to create the pattern.

Returns

An ofFloatPixels object containing the pattern.

18.2.2.37 std::string CX::Draw::wordWrap (std::string *s*, float *width*, ofTrueTypeFont & *font*)

Performs a word wrapping procedure, splitting `s` into multiple lines so that each line is no more than `width` wide. The algorithm attempts to end lines at whitespace, so as to avoid splitting up words. However, if there is no whitespace on a line, the line will be broken just before it would exceed the width and a hyphen is inserted. If the width is absurdly narrow (less than 2 characters), the algorithm will break.

Parameters

s	The string to wrap.
width	The maxmimum width of each line of `s`, in pixels.
font	A configured ofTrueTypeFont.

Returns

A string with newlines inserted to keep lines to be less than `width` wide.

18.3. CX::Instances Namespace Reference

Variables

- CX_Clock Clock

- CX_Display Disp
- CX_InputManager Input = CX::Private::inputManagerFactory()
- CX_Logger Log
- CX_RandomNumberGenerator RNG

18.3.1 Detailed Description

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

18.4. CX::Keycode Namespace Reference

Enumerations

- enum {
UNKNOWN = GLFW_KEY_UNKNOWN, **SPACE** = GLFW_KEY_SPACE, **APOSTROPHE** = GLFW_KEY↩
_APOSTROPHE, **COMMA** = GLFW_KEY_COMMA,
MINUS = GLFW_KEY_MINUS, **PERIOD** = GLFW_KEY_PERIOD, **SLASH** = GLFW_KEY_SLASH, **NR_0** = GLFW_KEY_0,
NR_1 = GLFW_KEY_1, **NR_2** = GLFW_KEY_2, **NR_3** = GLFW_KEY_3, **NR_4** = GLFW_KEY_4,
NR_5 = GLFW_KEY_5, **NR_6** = GLFW_KEY_6, **NR_7** = GLFW_KEY_7, **NR_8** = GLFW_KEY_8,
NR_9 = GLFW_KEY_9, **SEMICOLON** = GLFW_KEY_SEMICOLON, **EQUAL** = GLFW_KEY_EQUAL, **A** = GLFW_KEY_A,
B = GLFW_KEY_B, **C** = GLFW_KEY_C, **D** = GLFW_KEY_D, **E** = GLFW_KEY_E,
F = GLFW_KEY_F, **G** = GLFW_KEY_G, **H** = GLFW_KEY_H, **I** = GLFW_KEY_I,
J = GLFW_KEY_J, **K** = GLFW_KEY_K, **L** = GLFW_KEY_L, **M** = GLFW_KEY_M,
N = GLFW_KEY_N, **O** = GLFW_KEY_O, **P** = GLFW_KEY_P, **Q** = GLFW_KEY_Q,
R = GLFW_KEY_R, **S** = GLFW_KEY_S, **T** = GLFW_KEY_T, **U** = GLFW_KEY_U,
V = GLFW_KEY_V, **W** = GLFW_KEY_W, **X** = GLFW_KEY_X, **Y** = GLFW_KEY_Y,
Z = GLFW_KEY_Z, **LEFT_BRACKET** = GLFW_KEY_LEFT_BRACKET, **RIGHT_BRACKET** = GLFW_KE↩
Y_RIGHT_BRACKET, **BACKSLASH** = GLFW_KEY_BACKSLASH,
GRAVE_ACCENT = GLFW_KEY_GRAVE_ACCENT, **WORLD_1** = GLFW_KEY_WORLD_1, **WORLD_2** = GLFW_KEY_WORLD_2, **ESCAPE** = GLFW_KEY_ESCAPE,
ENTER = GLFW_KEY_ENTER, **TAB** = GLFW_KEY_TAB, **BACKSPACE** = GLFW_KEY_BACKSPACE, **I**↩
NSERT = GLFW_KEY_INSERT,
DELETE_ = GLFW_KEY_DELETE, **RIGHT_ARROW** = GLFW_KEY_RIGHT, **LEFT_ARROW** = GLFW_K↩
EY_LEFT, **DOWN_ARROW** = GLFW_KEY_DOWN,
UP_ARROW = GLFW_KEY_UP, **PAGE_UP** = GLFW_KEY_PAGE_UP, **PAGE_DOWN** = GLFW_KEY_PA↩
GE_DOWN, **HOME** = GLFW_KEY_HOME,
END = GLFW_KEY_END, **CAPS_LOCK** = GLFW_KEY_CAPS_LOCK, **SCROLL_LOCK** = GLFW_KEY_↩
SCROLL_LOCK, **NUM_LOCK** = GLFW_KEY_NUM_LOCK,
PRINT_SCREEN = GLFW_KEY_PRINT_SCREEN, **PAUSE** = GLFW_KEY_PAUSE, **F1** = GLFW_KEY_F1, **F2** = GLFW_KEY_F2,
F3 = GLFW_KEY_F3, **F4** = GLFW_KEY_F4, **F5** = GLFW_KEY_F5, **F6** = GLFW_KEY_F6,
F7 = GLFW_KEY_F7, **F8** = GLFW_KEY_F8, **F9** = GLFW_KEY_F9, **F10** = GLFW_KEY_F10,
F11 = GLFW_KEY_F11, **F12** = GLFW_KEY_F12, **F13** = GLFW_KEY_F13, **F14** = GLFW_KEY_F14,
F15 = GLFW_KEY_F15, **F16** = GLFW_KEY_F16, **F17** = GLFW_KEY_F17, **F18** = GLFW_KEY_F18,
F19 = GLFW_KEY_F19, **F20** = GLFW_KEY_F20, **F21** = GLFW_KEY_F21, **F22** = GLFW_KEY_F22,
F23 = GLFW_KEY_F23, **F24** = GLFW_KEY_F24, **F25** = GLFW_KEY_F25, **KP_0** = GLFW_KEY_KP_0,
KP_1 = GLFW_KEY_KP_1, **KP_2** = GLFW_KEY_KP_2, **KP_3** = GLFW_KEY_KP_3, **KP_4** = GLFW_KE↩
Y_KP_4,
KP_5 = GLFW_KEY_KP_5, **KP_6** = GLFW_KEY_KP_6, **KP_7** = GLFW_KEY_KP_7, **KP_8** = GLFW_KE↩
Y_KP_8,
KP_9 = GLFW_KEY_KP_9, **KP_PERIOD** = GLFW_KEY_KP_DECIMAL, **KP_DIVIDE** = GLFW_KEY_KP_↩
DIVIDE, **KP_MULTIPLY** = GLFW_KEY_KP_MULTIPLY,
KP_SUBTRACT = GLFW_KEY_KP_SUBTRACT, **KP_ADD** = GLFW_KEY_KP_ADD, **KP_ENTER** = GLF↩
W_KEY_KP_ENTER, **KP_EQUAL** = GLFW_KEY_KP_EQUAL,
LEFT_SHIFT = GLFW_KEY_LEFT_SHIFT, **LEFT_CONTROL** = GLFW_KEY_LEFT_CONTROL, **LEFT_A**↩

LT = GLFW_KEY_LEFT_ALT, **LEFT_SUPER** = GLFW_KEY_LEFT_SUPER,
RIGHT_SHIFT = GLFW_KEY_RIGHT_SHIFT, **RIGHT_CONTROL** = GLFW_KEY_RIGHT_CONTROL, **RIGHT_ALT** = GLFW_KEY_RIGHT_ALT, **RIGHT_SUPER** = GLFW_KEY_RIGHT_SUPER,
MENU = GLFW_KEY_MENU }

18.4.1 Detailed Description

This namespace is a wrapper around an anonymous enum containing a number of keycodes. The values in the enum can be compared with CX::CX_Keyboard::Event::key.

18.5. CX::Synth Namespace Reference

Classes

- class Adder
- class AdditiveSynth
- class Clamper
- class Envelope
- class Filter
- class FIRFilter
- class FunctionModule
- class GenericOutput
- class Mixer
- class ModuleBase
- class ModuleParameter
- class Multiplier
- class Oscillator
- class RingModulator
- class SoundBufferInput
- class SoundBufferOutput
- class Splitter
- class StereoSoundBufferOutput
- class StereoStreamOutput
- class StreamInput
- class StreamOutput
- class TrivialGenerator

Functions

- double sinc (double x)
- double relativeFrequency (double f, double semitoneDifference)
- ModuleBase & operator>> (ModuleBase &l, ModuleBase &r)
- void operator>> (ModuleBase &l, ModuleParameter &r)

18.5.1 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.

18.5.2 Function Documentation

18.5.2.1 ModuleBase & CX::Synth::operator>> (ModuleBase & *l*, ModuleBase & *r*)

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.
```

18.5.2.2 void CX::Synth::operator>> (ModuleBase & *l*, ModuleParameter & *r*)

This operator connects a module to the module parameter. It is not possible to connect a module parameter as an input for anything: They are dead ends.

```
using namespace CX::Synth;
Oscillator osc;
Envelope fenv;
Adder add;
add.amount = 500;
fenv >> add >> osc.frequency; //Connect the envelope as the input for the frequency of the
       oscillator with an offset of 500 Hz.
```

18.5.2.3 double CX::Synth::relativeFrequency (double *f*, double *semitoneDifference*)

This function returns the frequency that is `semitoneDifference` semitones from `f`.

Parameters

f	The starting frequency.
semitone↩ Difference	The difference (positive or negative) from `f` to the desired output frequency.

Returns

The final frequency.

18.5.2.4 double CX::Synth::sinc (double *x*)

The sinc function, defined as `sin(x)/x`.

18.6. CX::Util Namespace Reference

Classes

- class CX_BaseUnitConverter
- class CX_CoordinateConverter
- class CX_DegreeToPixelConverter
- class CX_LapTimer
- class CX_LengthToPixelConverter
- class CX_SegmentProfiler

Enumerations

- enum CX_RoundingConfiguration { CX_RoundingConfiguration::ROUND_TO_NEAREST, CX_Rounding↩ Configuration::ROUND_UP, CX_RoundingConfiguration::ROUND_DOWN, CX_RoundingConfiguration::R↩ OUND_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 log)
- bool setProcessToHighPriority (void)
- bool writeToFile (std::string filename, std::string data, bool append)
- double round (double d, int roundingPower, CX::Util::CX_RoundingConfiguration c)
- std::map< std::string, std::string > readKeyValueFile (std::string filename, std::string delimiter, bool trim↩ Whitespace, std::string commentString)
- bool writeKeyValueFile (const std::map< std::string, std::string > &kv, std::string filename, std::string delimiter)
- float getAngleBetweenPoints (ofPoint p1, ofPoint p2)
- ofPoint getRelativePointFromDistanceAndAngle (ofPoint start, float distance, float angle)
- 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 >
  std::vector< T > stringToVector (std::string s, std::string delimiter)
- 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 > exclude (const std::vector< T > &vals, const std::vector< T > &exclude)
- template<typename T >
  std::vector< T > concatenate (const std::vector< T > &A, const std::vector< T > &B)
- template<typename T >
  std::vector< T > concatenate (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)

18.6.1 Detailed Description

This namespace contains a variety of utility functions.

18.6.2 Function Documentation

18.6.2.1 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.

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

Checks that the version of oF that is used during compilation matches the requested version. If the desired version was 0.8.1, simply input (0, 8, 1) for `versionMajor`, `versionMinor`, and `versionPatch`, respectively.

Parameters

versionMajor	The major version (the X in X.0.0).
versionMinor	The minor version (0.X.0).
versionPatch	The patch version (0.0.X).
log	If `true`, a version mismatch will result in a warning being logged.

Returns

`true` if the versions match, `false` otherwise.

18.6.2.3 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.

18.6.2.4 template<typename T > std::vector< T > CX::Util::clamp (std::vector< T > *vals,* T *minimum,* T *maximum*)

Clamps a vector of values. See CX::Util::clamp().

Parameters

vals	The values 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 values.

18.6.2.5 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}.

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

Concatenates together the value A and the vector B. This is essentially push_front for vectors.

Parameters

A	The first value.
B	The vector of values.

Returns

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

18.6.2.7 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 `pixelsPer↩Unit`.

Returns

The number of pixels needed.

18.6.2.8 template<typename T > std::vector< T > CX::Util::exclude (const std::vector< T > & *values,* const std::vector< T > & *exclude*)

Gets the values from `values` that do not match the values in `exclude`.

Parameters

values	The starting set of values.
exclude	The set of values to exclude from `values`. This may contain duplicates.

Returns

A vector containing the values that were not excluded. This vector may be empty.

18.6.2.9 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 create a vector T = [1, 0]. Now if you "rotate" T in the positive direction, like the hand of a clock, 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., the angle between the mouse cursor and the center of the screen.

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 in degrees between p1 and p2. The values are in the range [0, 360).

18.6.2.10 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().

18.6.2.11 ofPoint CX::Util::getRelativePointFromDistanceAndAngle (ofPoint *start,* float *distance,* float *angle*)

This function begins at point `start` and travels `distance` from that point along `angle`, returning the resulting point.

This is useful for, e.g., drawing an object at a position relative to the center of the screen.

Parameters

start	The starting point.
distance	The distance to travel.
angle	The angle to travel on, in degrees.

18.6.2.12 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.

Parameters

start	The starting value.
end	The ending value. If `end == start`, this will return `start`.

Returns

A vector of the values int the sequence.

18.6.2.13 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.

18.6.2.14 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.

18.6.2.15 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.

18.6.2.16 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.

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

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

18.6.2.18 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:

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

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. If this is `false`, in the example, one of the key-value pairs would be ("blue ", " 0000FF"). Generally, you would want to trim.
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 each key string accesses a value string.

18.6.2.19 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.

18.6.2.20 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.

18.6.2.21 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.

18.6.2.22 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 the value 34.56, the results with different rounding powers (and `c = ROUND_TO_NEAREST`) are as follows: `RP = 0 -> 35; RP = 1 -> 30; RP = -1 -> 34.6`.
c	The type of rounding to do, from the CX::Util::CX_RoundingConfiguration enum. You can round up, down, to nearest (default), and toward zero.

Returns

The rounded value.

18.6.2.23 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`. `end` may be less than `start`, but only if `stepSize` is less than 0. If `end` is greater than `start`, `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.
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. It may be empty.

18.6.2.24 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.

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

Make a sequence starting from that value given by `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.

18.6.2.26 bool CX::Util::setProcessToHighPriority (void)

Attempts to set the process that CX is running in to high priority.

Note: This function only works on Windows.

Returns

`false` if a known error is encountered.

18.6.2.27 template<typename T > std::vector< T > CX::Util::stringToVector (std::string *s*, std::string *delimiter*)

This function takes a string, splits it on the delimiter, and converts each delimited part of the string to T, returning a vector<T>.

Template Parameters

T	The type of the data encoded in the string.

Parameters

s	The string containing the encoded data.
delimiter	The string that delimits the elements of the data.

Returns

A vector of the encoded data converted to T.

18.6.2.28 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	The vector of values to find unique values in.

Returns

A vector containing the unique values in `vals`.

18.6.2.29 template<typename T > T CX::Util::var (std::vector< T > *vals*)

Calculates the sample variance of a vector of values.

Template Parameters

T	The type of data.

Parameters

vals	The data.

Returns

The sample variance.

18.6.2.30 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.

18.6.2.31 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.

18.6.2.32 bool CX::Util::writeKeyValueFile (const std::map< std::string, std::string > & *kv,* std::string *filename,* std::string *delimiter*)

Write key-value pairs stored in a map<string, string> to a file. A delimiter is inserted between each key-value pair. Each key-value pair in on its own line.

Parameters

kv	A map<string, string> with the keys and values.
filename	The file to write to.
delimiter	A string to separate keys from values. Defaults to "=".

If the contents of `kv` are `{ "pigs" : "happy", "cowsMilked" : "no" }` and `delimiter` is "=", the output could be

```
pigs=happy
cowsMilked=no
```

18.6.2.33 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.

19. Class Documentation

19.1. CX::Synth::Adder Class Reference

`#include <CX_Synth.h>`

Inherits CX::Synth::ModuleBase.

Public Member Functions

- double getNextSample (void) override

Public Attributes

- ModuleParameter amount

  The amount that will be added to the input signal.

Additional Inherited Members

19.1.1 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.

19.1.2 Member Function Documentation

19.1.2.1 double CX::Synth::Adder::getNextSample (void) [override], [virtual]

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

Returns

The value of the next sample from the module.

Reimplemented from CX::Synth::ModuleBase.

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

- CX_Synth.h
- CX_Synth.cpp

19.2. CX::Synth::AdditiveSynth Class Reference

`#include <CX_Synth.h>`

Inherits CX::Synth::ModuleBase.

Public Types

- enum HarmonicSeriesType { **MULTIPLE**, **SEMITONE** }
- enum AmplitudePresets { **SINE**, **SQUARE**, **SAW**, **TRIANGLE** }
- typedef double amplitude_t

  A floating-point type used for the waveform amplitudes.
- typedef double frequency_t

  A floating-point type used for the frequencies of the waves.

Public Member Functions

- void setStandardHarmonicSeries (unsigned int harmonicCount)
- void setHarmonicSeries (unsigned int harmonicCount, HarmonicSeriesType type, double controlParameter)
- void setHarmonicSeries (std::vector< frequency_t > harmonicSeries)
- void setAmplitudes (AmplitudePresets a)
- void setAmplitudes (AmplitudePresets a1, AmplitudePresets a2, double mixture)
- void setAmplitudes (std::vector< amplitude_t > amps)
- std::vector< amplitude_t > calculateAmplitudes (AmplitudePresets a, unsigned int count)
- void pruneLowAmplitudeHarmonics (double tol)
- double getNextSample (void) override

Public Attributes

- ModuleParameter fundamental

  The fundamental frequency (the first harmonic) of the synth.

Additional Inherited Members

19.2.1 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.

19.2.2 Member Enumeration Documentation

19.2.2.1 enum CX::Synth::AdditiveSynth::AmplitudePresets [strong]

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.

19.2.2.2 enum CX::Synth::AdditiveSynth::HarmonicSeriesType [strong]

The type of function that will be used to create the harmonic series for the additive synth.

19.2.3 Member Function Documentation

19.2.3.1 std::vector< AdditiveSynth::amplitude_t > CX::Synth::AdditiveSynth::calculateAmplitudes (AmplitudePresets *a*, unsigned int *count*)

This is a specialty function that only works when the standard harmonic series is being used. If so, it calculates the amplitudes needed for the hamonics so as to produce the specified waveform type.

Parameters

a	The type of waveform that should be output from the additive synth.
count	The number of harmonics.

Returns

A vector of amplitudes.

19.2.3.2 double CX::Synth::AdditiveSynth::getNextSample (void) [override], [virtual]

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

Returns

The value of the next sample from the module.

Reimplemented from CX::Synth::ModuleBase.

19.2.3.3 void CX::Synth::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 harmonic with the greatest absolute amplitude. The result of this pruning is that the synthesizer will be more computationally efficient but provide a less precise 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 cutoff point.

Note

Because only harmonics with an amplitude less than or equal to the tolerance times an amplitude are pruned, setting `tol` to 0 will remove harmonics with 0 amplitude, but no others.

19.2.3.4 void CX::Synth::AdditiveSynth::setAmplitudes (AmplitudePresets *a*)

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

a	The type of wave calculate amplitudes for.

19.2.3.5 void CX::Synth::AdditiveSynth::setAmplitudes (AmplitudePresets *a1*, AmplitudePresets *a2*, 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.

Parameters

a1	The first preset.
a2	The second present.
mixture	Should be in the interval [0,1]. The proportion of `a1` that will be used, with the remainder (`1 - mixture`) used from `a2`.

19.2.3.6 void CX::Synth::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.

19.2.3.7 void CX::Synth::AdditiveSynth::setHarmonicSeries (unsigned int *harmonicCount*, HarmonicSeriesType *type*, double *controlParameter*)

Set the harmonic series for the AdditiveSynth.

Parameters

harmonicCount	The number of harmonics to use.
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.

19.2.3.8 void CX::Synth::AdditiveSynth::setHarmonicSeries (std::vector< frequency_t > *harmonicSeries*)

This function applies the harmonic series from a vector of harmonics supplied by the user.

Parameters

harmonicSeries	A vector frequencies that create a harmonic series. These values will be multiplied by the fundamental frequency in order to obtain the final frequency of each harmonic. The multiplier for the first harmonic is at index 0, so by convention you might want to set harmonicSeries[0] equal to 1, so that when the fundamental frequency is set with setFundamentalFrequency(), the first harmonic is actually the fundamental frequency, but this is not enforced.

Note

If `harmonicSeries.size()` is greater than the current number of harmonics, the new harmonics will have an amplitude of 0. If `harmonicSeries.size()` is less than the current number of harmonics, the number of harmonics will be reduced to the size of `harmonicSeries`.

19.2.3.9 void CX::Synth::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_Synth.h
- CX_Synth.cpp

19.3. CX::Algo::BlockSampler< T > Class Template Reference

`#include <CX_Algorithm.h>`

Public Member Functions

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

19.3.1 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 could use this class.

```
#include "CX.h"

void runExperiment(void) {
    //Construct a BlockSampler using RNG as the random number generator
    //and integer values 1 to 4 as the data to sample from.
    Algo::BlockSampler<int> bs(&RNG, Util::intVector(1, 4));

    //Generate 4 blocks of values and print those values along with information about the block and
       position
    cout << "Block, Position: Value" << endl;
    while (bs.getBlockNumber() < 4) {
        cout << bs.getBlockNumber() << ", " << bs.getBlockPosition() << ": ";
        cout << bs.getNextValue() << endl;
    }
}
```

Note

Another way of getting blocked random samples is to use CX::CX_RandomNumberGenerator::sample↩ Blocks().

19.3.2 Constructor & Destructor Documentation

19.3.2.1 template<typename T> CX::Algo::BlockSampler< T >::BlockSampler (CX_RandomNumberGenerator * *rng*, const std::vector< T > & *values*) [inline]

Constructs a BlockSampler with the given settings. See setup() for the meaning of the parameters.

19.3.3 Member Function Documentation

19.3.3.1 template<typename T> unsigned int CX::Algo::BlockSampler< T >::getBlockNumber (void) const [inline]

Returns the index of the block that is currently being sampled. Because it is zero-indexed, you can alternately think of the value as the number of completed blocks.

19.3.3.2 template<typename T> unsigned int CX::Algo::BlockSampler< T >::getBlockPosition (void) const [inline]

Returns the index of the sample that will be taken the next time getNextValue() is called. If 0, it means that a block of samples was just finished. If within the current block 4 samples had already been taken, this will return 4

19.3.3.3 template<typename T> T CX::Algo::BlockSampler< T >::getNextValue (void) [inline]

Get the next value sampled from the provided data.

Returns

An element sampled from the provided values, or, if there were no values provided, a warning will be logged and a default-constructed instance of T will be returned.

19.3.3.4 template<typename T> void CX::Algo::BlockSampler< T >::restartSampling (void) [inline]

Restarts sampling to be at the beginning of a block of samples. Also resets the block number (

19.3.3.5 template<typename T> void CX::Algo::BlockSampler< T >::setup (CX_RandomNumberGenerator * *rng*, const std::vector< T > & *values*) [inline]

Set up the BlockSampler.

Parameters

rng	A pointer to a CX_RandomNumberGenerator that will be used to randomize the sampled data.
values	A vector of values from which to sample.

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

- CX_Algorithm.h

19.4. CX::Synth::Clamper Class Reference

`#include <CX_Synth.h>`

Inherits CX::Synth::ModuleBase.

Public Member Functions

- double getNextSample (void) override

Public Attributes

- ModuleParameter low

  The lowest possible output value.
- ModuleParameter high

  The highest possible output value.

Additional Inherited Members

19.4.1 Detailed Description

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

19.4.2 Member Function Documentation

19.4.2.1 double CX::Synth::Clamper::getNextSample (void) [override], [virtual]

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

Returns

The value of the next sample from the module.

Reimplemented from CX::Synth::ModuleBase.

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

- CX_Synth.h
- CX_Synth.cpp

19.5. CX::CX_SlidePresenter::Configuration Struct Reference

`#include <CX_SlidePresenter.h>`

Public Attributes

- CX_Display * display

  A pointer to the display on which to present the slides.
- 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. In this function, you can add additional slides to the slide presenter and do other tasks, like process input.
- CX_SlidePresenter::ErrorMode errorMode

  This sets how errors in slide presentation should be handled. Currently, the only available mode is the default, so this should not be changed.
- bool deallocateCompletedSlides

  If `true`, once a slide has been presented, its framebuffer will be deallocated to conserve video memory. This only matters if you are using a large number of slides at once and add slides during slide presentation.
- SwappingMode swappingMode

  The mode used for swapping slides. See the SwappingMode enum for the possible settings. Defaults to `SINGLE_CORE_BLOCKING_SWAPS`.
- 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.
- bool useFenceSync

Hint that fence sync should be used to check that slides are fully rendered to the back buffer before they are swapped in. This will allow the slide presenter to notify you if slides are swapped into the front buffer before it is confirmed that they were fully rendered. Defaults to `true`*. See also waitUntilFenceSyncComplete.*

- 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 rendered into the back buffer. This prevents vertical tearing, but may cause slides to be swapped in late if the confirmation that rendering has completed is delayed but the rendering has actually occurred on time. Does nothing if `swappingMode` *is* `MULTI_CORE`*.*

19.5.1 Detailed Description

This struct is used for configuring a CX_SlidePresenter. See CX_SlidePresenter::setup(const CX_SlidePresenter↩ ::Configuration&).

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

- CX_SlidePresenter.h

19.6. 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 less than 0 will cause the system default input device to be used.
- int outputDeviceId

  The ID of the desired output device. A value less than 0 will cause the system default output device to be used.

19.6.1 Detailed Description

This struct controls the configuration of the CX_SoundStream.

19.6.2 Member Data Documentation

19.6.2.1 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#ac9b6f625da88249d08a8409a9 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::UN↩ SPECIFIED is used.

19.6.2.2 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).

19.6.2.3 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.

19.6.2.4 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

19.7. CX::CX_BaseClockInterface Class Reference

`#include <CX_Clock.h>`

Inherited by CX::CX_StdClockWrapper< stdClock >.

Public Member Functions

- virtual cxTick_t nanos (void)=0

  Returns the current time in nanoseconds.
- virtual void resetStartTime (void)=0

  Resets the start time, so that an immediate call to nanos() would return 0.
- virtual std::string getName (void)

  Returns a helpful name describing the clock implementation.

19.7.1 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

19.8. 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)
- virtual std::vector< float > operator() (const std::vector< float > &vx)
- virtual std::vector< float > inverse (const std::vector< float > &vy)

19.8.1 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()`.

19.8.2 Member Function Documentation

19.8.2.1 virtual float CX::Util::CX_BaseUnitConverter::inverse (float *y*) [inline], [virtual]

`inverse()` should perform the inverse operation as `operator()`.

Reimplemented in CX::Util::CX_LengthToPixelConverter, and CX::Util::CX_DegreeToPixelConverter.

19.8.2.2 std::vector< float > CX::Util::CX_BaseUnitConverter::inverse (const std::vector< float > & *vy*) [virtual]

Applies the inverse unit conversion to a whole vector.

19.8.2.3 virtual float CX::Util::CX_BaseUnitConverter::operator() (float *x*) [inline], [virtual]

`operator()` should perform the unit conversion.

Reimplemented in CX::Util::CX_LengthToPixelConverter, and CX::Util::CX_DegreeToPixelConverter.

19.8.2.4 std::vector< float > CX::Util::CX_BaseUnitConverter::operator() (const std::vector< float > & *vx*) [virtual]

Applies the unit conversion to a whole vector.

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

- CX_UnitConversion.h
- CX_UnitConversion.cpp

19.9. CX::CX_Clock Class Reference

`#include <CX_Clock.h>`

Public Member Functions

- void setImplementation (CX_BaseClockInterface *impl)
- PrecisionTestResults 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")

19.9.1 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.

19.9.2 Member Function Documentation

19.9.2.1 void CX::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 effectively a static function of the CX_Clock class.

19.9.2.2 std::string CX::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".

19.9.2.3 std::string CX::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.

19.9.2.4 CX_Millis CX::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.

19.9.2.5 CX_Clock::PrecisionTestResults CX::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 blocking. See Blocking Code.

Parameters

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

Returns

A CX_Clock::PrecisionTestResults struct containing some information about the precision of the clock.

19.9.2.6 void CX::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()).

19.9.2.7 void CX::CX_Clock::setImplementation (CX::CX_BaseClockInterface * *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_StdClockWrapper to wrap any of the clocks from the std::chrono namespace or any clock that conforms to the standard of those clocks. You can also write your own low level clock that implements CX_BaseClockInterface.

Parameters

impl	A pointer to an instance of a class implementing CX::CX_BaseClockInterface.

Note

This function resets the experiment start time of `impl`, but does not reset the experiment start time date/time string.

19.9.2.8 void CX::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.

Parameters

t	The requested sleep duration. If 0, the thread yields rather than sleeping.

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

- CX_Clock.h
- CX_Clock.cpp

19.10. CX::Util::CX_CoordinateConverter Class Reference

`#include <CX_UnitConversion.h>`

Public Member Functions

- CX_CoordinateConverter (void)
- CX_CoordinateConverter (ofPoint origin, bool invertX, bool invertY, bool invertZ=false)
- void setAxisInversion (bool invertX, bool invertY, bool invertZ=false)
- void setOrigin (ofPoint newOrigin)
- void setMultiplier (float multiplier)
- void setUnitConverter (CX_BaseUnitConverter *converter)
- 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)
- std::vector< ofPoint > operator() (const std::vector< ofPoint > &p)
- std::vector< ofPoint > inverse (const std::vector< ofPoint > &p)

19.10.1 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 coordinates in your own system and convert those coordinates into the standard coordinates that are used by the drawing functions. Note that this does not always play nicely when dealing with angles.

See CX_CoordinateConverter::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(Disp.getCenter(), 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 of the use of this class can be found in the advancedChangeDetection example experiment.

If you want to invert the y-axis, you may be better off using CX::CX_Display::setYIncreasesUp().

19.10.2 Constructor & Destructor Documentation

19.10.2.1 CX::Util::CX_CoordinateConverter::CX_CoordinateConverter (void)

Constructs a CX_CoordinateConverter with the default settings. The settings can be changed later with setAxis↩ Inversion(), setOrigin(), setMultiplier(), and/or setUnitConverter().

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

Constructs a CX_CoordinateConverter 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::getCenter() 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). The other way of saying this is that smaller (increasingly negative) values are farther away.

19.10.3 Member Function Documentation

19.10.3.1 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.

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

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

19.10.3.3 std::vector< ofPoint > CX::Util::CX_CoordinateConverter::inverse (const std::vector< ofPoint > & *p*)

Applies the inverse conversion on a whole vector of points at once.

Parameters

p	The vector of points to inverse convert.

Returns

The inverse converted points.

19.10.3.4 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.

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

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

19.10.3.6 std::vector< ofPoint > CX::Util::CX_CoordinateConverter::operator() (const std::vector< ofPoint > & *p*)

Applies the conversion on a whole vector of points at once.

Parameters

p	The vector of points to convert.

Returns

The converted points.

19.10.3.7 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).

19.10.3.8 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, assuming that the CX_CoordinateConverter is used throughout. If it has not been set, the multiplier is 1 by default.

Parameters

multiplier	The amount to multiply user coordinates by.

19.10.3.9 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::getCenter() as the value for this argument.

19.10.3.10 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(Disp.getCenter());
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

19.11. CX::CX_DataFrame Class Reference

`#include <CX_DataFrame.h>`

Classes

- struct InputOptions
- class IoOptions
- struct OutputOptions

Public Types

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

  An unsigned integer type used for indexing the rows of a CX_DataFrame.

Public Member Functions

- CX_DataFrame & operator= (const CX_DataFrame &df)
- void append (CX_DataFrame df)
- CX_DataFrame copyRows (std::vector< CX_DataFrame::rowIndex_t > rowOrder) const
- CX_DataFrame copyColumns (std::vector< std::string > columns)
- void clear (void)
- CX_DataFrameCell operator() (std::string column, rowIndex_t row)
- CX_DataFrameCell operator() (rowIndex_t row, std::string column)

  Behaves just like CX_DataFrame::operator()(std::string, rowIndex_t).
- CX_DataFrameCell at (rowIndex_t row, std::string column)
- CX_DataFrameCell at (std::string column, rowIndex_t row)
- void appendRow (CX_DataFrameRow row)
- void insertRow (CX_DataFrameRow row, rowIndex_t beforeIndex)
- bool deleteRow (rowIndex_t row)
- CX_DataFrameRow operator[ ] (rowIndex_t row)
- void setRowCount (rowIndex_t rowCount)
- rowIndex_t getRowCount (void) const

  Returns the number of rows in the data frame.
- bool reorderRows (const vector< CX_DataFrame::rowIndex_t > &newOrder)
- void shuffleRows (void)
- void shuffleRows (CX_RandomNumberGenerator &rng)
- void addColumn (std::string columnName)
- bool deleteColumn (std::string columnName)
- std::vector< std::string > getColumnNames (void) const
- bool columnExists (std::string columnName) const

  Returns `true` if the named column exists in the `CX_DataFrame`.
- CX_DataFrameColumn operator[ ] (std::string column)
- template<typename T >
  std::vector< T > copyColumn (std::string column) const
- template<typename T >
  std::vector< std::vector< T > > copyVectorColumn (std::string column) const
- bool columnContainsVectors (std::string columnName) const

  Returns `true` if the named column contains any cells which contain vectors (i.e. have a length > 1).
- std::vector< std::string > convertVectorColumnToColumns (std::string columnName, int startIndex, bool deleteOriginal, std::string newBaseName="")
- void convertAllVectorColumnsToMultipleColumns (int startIndex, bool deleteOriginals)
- std::string print (std::string delimiter="\t", bool printRowNumbers=false) const

- std::string print (const std::set< std::string > &columns, std::string delimiter="\t", bool printRow↩ Numbers=false) const
- std::string print (const std::vector< rowIndex_t > &rows, std::string delimiter="\t", bool printRow↩ Numbers=false) const
- std::string print (const std::set< std::string > &columns, const std::vector< rowIndex_t > &rows, std::string delimiter="\t", bool printRowNumbers=false) const
- std::string print (OutputOptions oOpt) const
- bool printToFile (std::string filename, std::string delimiter="\t", bool printRowNumbers=false) const
- bool printToFile (std::string filename, const std::set< std::string > &columns, std::string delimiter="\t", bool printRowNumbers=false) const
- bool printToFile (std::string filename, const std::vector< rowIndex_t > &rows, std::string delimiter="\t", bool printRowNumbers=false) const
- bool printToFile (std::string filename, const std::set< std::string > &columns, const std::vector< rowIndex_t > &rows, std::string delimiter="\t", bool printRowNumbers=false) const
- bool printToFile (std::string filename, OutputOptions oOpt) const
- bool readFromFile (std::string filename, InputOptions iOpt)
- bool readFromFile (std::string filename, std::string cellDelimiter="\t", std::string vectorEncloser="\"", std::string vectorElementDelimiter=";")

Friends

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

19.11.1 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 CX_Data↩ Frame::operator(), which dynamically resizes the data frame. When an experimental session is complete, the data can be written to a file using CX_DataFrame::printToFile().

See example-dataFrame for 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.

19.11.2 Member Function Documentation

19.11.2.1 void CX::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.

19.11.2.2 void CX::CX_DataFrame::append (CX_DataFrame *df*)

Appends a data frame to this data frame. Internally, CX_DataFrame::appendRow() is used to copy over the rows of `df` one at a time.

Parameters

df	The CX_DataFrame to append.

19.11.2.3 void CX::CX_DataFrame::appendRow (CX_DataFrameRow *row*)

Appends the row to the end of the data frame.

Parameters

row	The row of data to add.

Note

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

19.11.2.4 CX_DataFrameCell CX::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.

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

Equivalent to `CX::CX_DataFrame::at(rowIndex_t, std::string)`.

19.11.2.6 void CX::CX_DataFrame::clear (void)

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

19.11.2.7 void CX::CX_DataFrame::convertAllVectorColumnsToMultipleColumns (int *startIndex*, bool *deleteOriginals*)

For all columns with at least one cell that contains a vector, that column is converted into multiple columns with CX_DataFrame::convertVectorColumnToColumns(). The name of the new columns will be the same as the name of the original column, plus an index suffix.

Parameters

startIndex	The number at which to being suffixing the multiple columns derived from a vector column. This value is used for each vector column (it's not cumuluative for all columns created with this function call, because that would be bizarre).
deleteOriginals	If `true`, the original vector columns will be deleted once they have been converted into multiple columns.

19.11.2.8 std::vector< std::string > CX::CX_DataFrame::convertVectorColumnToColumns (std::string *columnName*, int *startIndex*, bool *deleteOriginal*, std::string *newBaseName* = " ")

Converts a column which contains vectors of data into multiple columns which are given names with an ascending integer suffix. Each new column will contain the data from one location in the previous vectors of data. For example, if you have length 3 vectors in a column and use this function on that column, you will end up with three columns, each of which contains one of the elements of those vectors, with order maintained, of course.

If you have vectors with different lengths within the same column, this function still works, it just fills empty cells of new columns with the string "NA".

Parameters

columnName	The name of the column to convert to multiple columns. If the named column does not exist or it does not contain any vectors, this function has no effect.
startIndex	The value at which to start giving suffix indices. For example, if it is 1, the first new column will be named "newBaseName1", the second "newBaseName2", etc..
deleteOriginal	If `true`, the original column, `columnName`, will be deleted once the data has been copied into the new columns.
newBaseName	If this is the empty string, `columnName` will be used as the base for the new column names. Otherwise, `newBaseName` will be used.

Returns

A vector of strings containing the new names. If an error occurred or nothing needed to be done, this vector will be of length 0.

Note

If any of the names of the new columns conflicts with an existing column name, the new column will be created, but its name will be changed by appending "_NEW". If this new name conflicts with an existing name, the process will be repeated until the new name does not conflict.

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

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

Note that if it is a vector column, you will only get the first element of each cell. Use CX::CX_DataFrame::copyVectorColumn() to get all of the elements of each cell.

Template Parameters

T	The type of data to extract.

Parameters

column	The name of the column to copy data from.

Returns

A vector containing the copied data.

19.11.2.10 CX_DataFrame CX::CX_DataFrame::copyColumns (std::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.

19.11.2.11 CX_DataFrame CX::CX_DataFrame::copyRows (std::vector< CX_DataFrame::rowIndex_t > *rowOrder*) const

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.

19.11.2.12 template<typename T > std::vector< std::vector< T > > CX::CX_DataFrame::copyVectorColumn (std::string *column*) const

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

Template Parameters

T	The type of data to extract.

Parameters

column	The name of the column to copy data from.

Returns

A vector of vectors containing the copied data.

19.11.2.13 bool CX::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.

19.11.2.14 bool CX::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.

19.11.2.15 std::vector< std::string > CX::CX_DataFrame::getColumnNames (void) const

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

Returns

Vector of strings with the column names.

19.11.2.16 void CX::CX_DataFrame::insertRow (CX_DataFrameRow *row,* rowIndex_t *beforeIndex*)

Inserts a row into the data frame.

Parameters

row	The row of data to insert.
beforeIndex	The index of the row before which `row` should be inserted. If >= the number of rows currently stored, `row` will be appended to the end of the data frame.

Note

If `row` has columns that do not exist in the data frame, those columns will be added to the data frame. This may be a blocking operation, depending on the size of the data frame.

19.11.2.17 CX_DataFrameCell CX::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 resized in order to fit the new row(s) and/or column.

Parameters

row	The row number.
column	The column name.

Returns

A CX_DataFrameCell that can be read from or written to.

19.11.2.18 CX_DataFrame & CX::CX_DataFrame::operator= (const 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.

19.11.2.19 CX_DataFrameRow CX::CX_DataFrame::operator[ ] (rowIndex_t *row*)

Extract a row from the data frame. Note that the returned value is not a copy of the original row. Rather, it represents the original row so that if the returned row is modified, it will also modify the original data in the parent data frame.

Parameters

row	The index of the row to extract.

Returns

A CX_DataFrameRow.

19.11.2.20 CX_DataFrameColumn CX::CX_DataFrame::operator[ ] (std::string *column*)

Extract a column from the data frame. Note that the returned value is not a copy of the original column. Rather, it represents the original column so that if the returned column is modified, it will also modify the original data in the parent data frame.

Parameters

column	The name of the column to extract.

Returns

A CX_DataFrameColumn.

See also

See also copyColumn() for a way to copy out a column of data.

19.11.2.21 std::string CX::CX_DataFrame::print (std::string *delimiter* = "\t", bool *printRowNumbers* = false) const

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

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

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

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

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

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

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 "\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.

19.11.2.25 std::string CX::CX_DataFrame::print (OutputOptions *oOpt*) const

Prints the contents of the CX_DataFrame to a string with formatting options specified in oOpt.

Parameters

oOpt	Output formatting options.

Returns

A string containing a formatted representation of the data frame contents.

19.11.2.26 bool CX::CX_DataFrame::printToFile (std::string *filename*, std::string *delimiter* = "\t", bool *printRowNumbers* = false) const

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

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

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

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

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

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

This function is equivalent in behavior to CX::CX_DataFrame::print() except that instead of returning a string containing the printed contents of the data frame, the string is printed directly to a file. If the file exists, it will be overwritten. All paramters shared with print() are simply passed along to print(), so they have the same behavior.

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.
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

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

19.11.2.30 bool CX::CX_DataFrame::printToFile (std::string *filename*, OutputOptions *oOpt*) const

This function is equivalent in behavior to CX::CX_DataFrame::print() except that instead of returning a string containing the printed contents of the data frame, the string is printed directly to a file. If the file exists, it will be overwritten. All paramters shared with print() are simply passed along to print(), so they have the same behavior.

Parameters

filename	The name of the output file.

oOpt	Output formatting options.
oOpt	The output options.

Returns

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

19.11.2.31 bool CX::CX_DataFrame::readFromFile (std::string *filename*, InputOptions *iOpt*)

Equivalent to a call to readFromFile(string, string, string, string), except that the last three arguments are taken from iOpt.

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.
iOpt	Input options, such as the delimiter between cells in the input file.

19.11.2.32 bool CX::CX_DataFrame::readFromFile (std::string *filename*, std::string *cellDelimiter* = "\t", std::string *vectorEncloser* = "\"", std::string *vectorElementDelimiter* = ";")

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.

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 data in the input file. Consecutive delimiters are not treated as a single delimiter.
vectorEncloser	A string containing the character(s) that surround cells that contain a vector of data in the input file. 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. If vectorEncloser is the empty string, this function will not attempt to read in vectors: everything that looks like a vector will just be treated as a string.
vectorElement↩ Delimiter	The delimiter between the elements of the vector.

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.

19.11.2.33 bool CX::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.

19.11.2.34 void CX::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 the extra rows.

19.11.2.35 void CX::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.

19.11.2.36 void CX::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

19.12. 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 (bool log=true) const
- std::string toString (void) const

  Equivalent to a call to to<string>().
- 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 (bool log=true) const
- template<typename T >
  void storeVector (std::vector< T > values)
- bool isVector (void) const

  Returns `true` if more than one element is stored in the CX_DataFrameCell.
- unsigned int size (void) const

  Returns the number of elements stored in the cell.
- void copyCellTo (CX_DataFrameCell *targetCell) const
- std::string getStoredType (void) const
- void deleteStoredType (void)
- void clear (void)

  Delete the contents of the cell.
- template<>
  std::string to (bool log) const
- template<>
  std::vector< std::string > toVector (bool log) const

Static Public Member Functions

- static void setFloatingPointPrecision (unsigned int prec)
- static unsigned int getFloatingPointPrecision (void)

19.12.1 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.

19.12.2 Constructor & Destructor Documentation

19.12.2.1 CX::CX_DataFrameCell::CX_DataFrameCell (const char * *c*)

Constructs the cell with a string literal, treating it's type as the same as a `std::string`.

19.12.3 Member Function Documentation

19.12.3.1 void CX::CX_DataFrameCell::copyCellTo (CX_DataFrameCell * *targetCell*) const

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

Parameters

targetCell	A pointer to the cell to copy data to.

19.12.3.2 void CX::CX_DataFrameCell::deleteStoredType (void)

If for whatever reason the type of the data stored in the CX_DataFrameCell should be ignored, you can delete it with this function.

19.12.3.3 unsigned int CX::CX_DataFrameCell::getFloatingPointPrecision (void) [static]

Get the current floating point precision, set by CX_DataFrameCell::setFloatingPointPrecision().

19.12.3.4 std::string CX::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.

You can test if the type of data stored in a CX_DataFrameCell is some type T with the following code snippet.

```
CX_DataFrameCell cell; //Assume this actually has some data in it.
bool typesMatch = (cell.getStoredType() == typeid(T).name())
```

If the stored data is a vector (i.e. has length > 1), the returned string is "vector<TID>", where "TID" is replaced with the type string. In other words, you can test if the stored type is a vector<T> by looking for "vector<" at the beginning of the return value of this function, getting the type name between the surrounding angle brackets, and then using `typeid(T).name()` as in the code snippet above.

Returns

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

19.12.3.5 CX_DataFrameCell & CX::CX_DataFrameCell::operator= (const char * *c*)

Assigns a string literal to the cell, treating it's type as the same as a `std::string`.

19.12.3.6 void CX::CX_DataFrameCell::setFloatingPointPrecision (unsigned int *prec*) [static]

Set the precision with which floating point numbers (`float`s and `double`s) are stored, in number of significant digits. This value will be used for all `CX_DataFrameCell`s. Changing this value after storing data will not change the precision of the stored data.

Defaults to `std::numeric_limits<double>::max_digits10` significant digits. To quote cppreference.com, "The value of std::numeric_limits<T>::max_digits10 is the number of base-10 digits that are necessary to uniquely represent all distinct values of the type T, such as necessary for serialization/deserialization to text." That is to say, the default value is sufficient for lossless conversion between a double precision float and the string representation of that value stored by the CX_DataFrameCell.

Parameters

prec	The number of significant digits.

19.12.3.7 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.

19.12.3.8 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.

19.12.3.9 template<typename T > T CX::CX_DataFrameCell::to (bool *log* = `true`) 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.

19.12.3.10 std::string CX::CX_DataFrameCell::to (bool *log*) 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.

19.12.3.11 template<typename T > std::vector< T > CX::CX_DataFrameCell::toVector (bool *log* = `true`) 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.

19.12.3.12 std::vector< std::string > CX::CX_DataFrameCell::toVector (bool *log*) const

Converts the contents of the CX_DataFrame cell to a vector of strings.

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

- CX_DataFrameCell.h
- CX_DataFrameCell.cpp

19.13. CX::CX_DataFrameColumn Class Reference

`#include <CX_DataFrame.h>`

Public Member Functions

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

  Returns the number of rows in the column.

Friends

- class **CX_DataFrame**

19.13.1 Detailed Description

This class represents a column from a CX_DataFrame. It has special behavior that may not be obvious. If it is extracted from a CX_DataFrame with the use of CX::CX_DataFrame::operator[ ](std::string), then the extracted column is linked to the original column of data such that if either are modified, both will see the effects.

19.13.2 Constructor & Destructor Documentation

19.13.2.1 CX::CX_DataFrameColumn::CX_DataFrameColumn (void)

Constructs a CX_DataFrameColumn without linking it to a CX_DataFrame.

19.13.3 Member Function Documentation

19.13.3.1 CX_DataFrameCell CX::CX_DataFrameColumn::operator[ ] (CX_DataFrame::rowIndex_t *row*)

Accesses the element in the specified row of the column.

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

- CX_DataFrame.h
- CX_DataFrame.cpp

19.14. CX::CX_DataFrameRow Class Reference

`#include <CX_DataFrame.h>`

Public Member Functions

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

  Returns a vector containing the names of the columns in this row.
- void clear (void)

  Clears the contents of the row.

Friends

- class **CX_DataFrame**

19.14.1 Detailed Description

This class represents a row from a CX_DataFrame. It has special behavior that may not be obvious. If it is extracted from a CX_DataFrame with the use of CX::CX_DataFrame::operator[ ](CX_DataFrame::rowIndex_t), then the extracted row is linked to the original row of data such that if either are modified, both will see the effects. See the code example. If a CX_DataFrameRow is constructed normally (not extracted from a CX_DataFrame) it is not linked to any data frame.

```
//Create a CX_DataFrame and put some stuff in it.
CX_DataFrame df;
df(0, "a") = 2;
df(0, "b") = 5;

CX_DataFrameRow row0 = df[0]; //Extract row 0 from the data frame.
row0["a"] = 10; //Modify it.

cout << df.print() << endl; //See that the data frame has been modified.

df.appendRow(row0); //Append the row to the end of the data frame.

cout << df.print() << endl;

row0["a"] = 3; //Although row0 has been appended, it still only refers to row 0, not both rows,
//so this will only affect row 0 and not row 1.

cout << df.print() << endl;
```

19.14.2 Constructor & Destructor Documentation

19.14.2.1 CX::CX_DataFrameRow::CX_DataFrameRow (void)

Construct a CX_DataFrameRow without linking it to a CX_DataFrame.

19.14.3 Member Function Documentation

19.14.3.1 CX_DataFrameCell CX::CX_DataFrameRow::operator[ ] (std::string *column*)

Accesses the element in the specified column of the row.

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

- CX_DataFrame.h
- CX_DataFrame.cpp

19.15. 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)
- void setup (float pixelsPerUnit, float viewingDistance, bool roundResult=false)
- float operator() (float degrees) override
- float inverse (float pixels) override
- bool configureFromFile (std::string filename, std::string delimiter="=", bool trimWhitespace=true, std::string commentString="//")

19.15.1 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.
```

19.15.2 Constructor & Destructor Documentation

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

Constructs an instance of a CX_DegreeToPixelConverter with the given configuration. See setup() for the meaning of the parameters.

19.15.3 Member Function Documentation

19.15.3.1 bool CX::Util::CX_DegreeToPixelConverter::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_DegreeToPixelConverter. The format of the file is provided in the example code below.

Sample configuration file:

```
D2PC.pixelsPerUnit = 35
D2PC.viewingDistance = 50
D2PC.roundResult = true
```

All of the configuration keys are used in this example. Note that the "D2PC" prefix allows this configuration to be embedded in a file that also performs other configuration functions.

See CX_DegreeToPixelConverter::setup() for details about the meanings of the configuration options.

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 (""), everything on a line following the first instance of `commentString` will be ignored.

Returns

`true` if there were no problems reading in the file, `false` otherwise.

19.15.3.2 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.

19.15.3.3 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.

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

Sets up a CX_DegreeToPixelConverter with the given configuration.

Parameters

pixelsPerUnit	The number of pixels within one length unit (e.g. inches, centimeters). This can be measured by drawing an object with a known size 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.

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

- CX_UnitConversion.h
- CX_UnitConversion.cpp

19.16. 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, false otherwise.
- 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) const
- bool hasSwappedSinceLastCheck (void)
- void waitForBufferSwap (void)
- CX_Millis getLastSwapTime (void) const
- CX_Millis estimateNextSwapTime (void) const
- uint64_t getFrameNumber (void) const
- void estimateFramePeriod (CX_Millis estimationInterval, float minRefreshRate=40, float maxRefresh↩Rate=160)
- CX_Millis getFramePeriod (void) const
- CX_Millis getFramePeriodStandardDeviation (void) const
- void setFramePeriod (CX_Millis knownPeriod)
- void setWindowResolution (int width, int height)
- ofRectangle getResolution (void) const
- ofPoint getCenter (void) const
- void waitForOpenGL (void)
- std::map< std::string, CX_DataFrame > testBufferSwapping (CX_Millis desiredTestDuration, bool test↩SecondaryThread)
- ofFbo makeFbo (void)
- void copyFboToBackBuffer (ofFbo &fbo)
- void copyFboToBackBuffer (ofFbo &fbo, ofPoint destination)
- void copyFboToBackBuffer (ofFbo &fbo, ofRectangle source, ofPoint destination)
- void setYIncreasesUpwards (bool upwards)
- bool getYIncreasesUpwards (void)

  Do y-axis values increase upwards?

19.16.1 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 a context in which things can be drawn.

An instance of this class is created for the user. It is called CX::Instances::Disp. The user should not need another instance of this class.

19.16.2 Member Function Documentation

19.16.2.1 void CX::CX_Display::beginDrawingToBackBuffer (void)

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

```
Disp.beginDrawingToBackBuffer();
//Draw stuff...
Disp.endDrawingToBackBuffer();
```

19.16.2.2 void CX::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 below:

```
display.windowWidth = 600
display.windowHeight = 300
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 omitted, in which case there is no change in the configuration of the CX_Display for that option. 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 (""), everything on a line following the first instance of `commentString` will be ignored.

19.16.2.3 void CX::CX_Display::copyFboToBackBuffer (ofFbo & *fbo*)

Copies an `ofFbo` to the back buffer using a potentially very slow but pixel-perfect blitting operation. The slowness of the operation is hardware-dependent, with older hardware often being faster at this operation. Generally, you should just draw the `ofFbo` directly using its `draw()` function.

Note

This function 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).

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

Copies an `ofFbo` to the back buffer using a potentially very slow but pixel-perfect blitting operation. The slowness of the operation is hardware-dependent, with older hardware often being faster at this operation. Generally, you should just draw the `ofFbo` directly using its `draw()` function.

Note

This function 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.
destination	The point on the back buffer where the fbo will be placed.

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

Copies an `ofFbo` to the back buffer using a potentially very slow but pixel-perfect blitting operation. The slowness of the operation is hardware-dependent, with older hardware often being faster at this operation. Generally, you should just draw the `ofFbo` directly using its `draw()` function.

Note

This function 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.
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:

```
Disp.beginDrawingToBackBuffer();
ofSetColor( 255 ); //If the color is not set to white, the fbo will be drawn mixed with whatever the
       current color is.
fbo.draw( x, y, width, height ); //Replace these variables with the destination location (x,y) and
       dimensions of the FBO.
Disp.endDrawingToBackBuffer();
```

19.16.2.6 void CX::CX_Display::endDrawingToBackBuffer (void)

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

19.16.2.7 void CX::CX_Display::estimateFramePeriod (CX_Millis *estimationInterval*, float *minRefreshRate* = `40`, float *maxRefreshRate* = `160`)

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 during setup 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.
minRefreshRate	The minimum allowed refresh rate, in Hz. If an observed duration is less than 1/minRefreshRate seconds, it will be ignored for purposes of estimating the frame period.
maxRefreshRate	The maximum allowed refresh rate, in Hz. If an observed duration is greater than 1/minRefreshRate seconds, it will be ignored for purposes of estimating the frame period.

19.16.2.8 CX_Millis CX::CX_Display::estimateNextSwapTime (void) const

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().

19.16.2.9 ofPoint CX::CX_Display::getCenter (void) const

Returns an ofPoint representing the center of the display. Works in both windowed and full screen mode.

19.16.2.10 uint64_t CX::CX_Display::getFrameNumber (void) const

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(true)) 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.

19.16.2.11 CX_Millis CX::CX_Display::getFramePeriod (void) const

Gets the estimate of the frame period estimated with CX_Display::estimateFramePeriod().

19.16.2.12 CX_Millis CX::CX_Display::getFramePeriodStandardDeviation (void) const

Gets the estimate of the standard deviation of the frame period estimated with CX_Display::estimateFramePeriod().

19.16.2.13 CX_Millis CX::CX_Display::getLastSwapTime (void) const

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().

19.16.2.14 ofRectangle CX::CX_Display::getResolution (void) const

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.

19.16.2.15 bool CX::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.

19.16.2.16 bool CX::CX_Display::isAutomaticallySwapping (void) const

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

19.16.2.17 ofFbo CX::CX_Display::makeFbo (void)

Makes an ofFbo with dimensions equal to the size of the current display with standard settings and allocates memory for it. The FBO is configured to use RGB plus alpha for the color settings. The MSAA setting for the FBO is set to the current value returned by CX::Util::getMsaaSampleCount(). This is to help the FBO have the same settings as the front and back buffers so that rendering into the FBO will produce the same output as rendering into the back buffer.

Returns

The configured FBO.

19.16.2.18 void CX::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 for up to 1 frame to due the requirement that it synchronize with the thread.

19.16.2.19 void CX::CX_Display::setFramePeriod (CX_Millis *knownPeriod*)

During setup, CX tries to estimate the frame period of the display using CX::CX_Display::estimateFramePeriod(). However, this does not always work, and the estimated value is wrong. If you know that this is happening, you can use this function to set the correct frame period. A typical call might be

```
Disp.setFramePeriod(CX_Seconds(1.0/60.0));
```

to set the frame period for a 60 Hz refresh cycle. However, note that this will not fix the underlying problem that prevented the frame period from being estimated correctly, which usually has to do with problems with the video card doing vertical synchronization incorrectly. Thus, this may not fix anything.

Parameters

knownPeriod	The known refresh period of the monitor.

Note

This function sets the standard deviation of the frame period to 0.

19.16.2.20 void CX::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.

19.16.2.21 void CX::CX_Display::setup (void)

Set up the display. Must be called for the display to function correctly. This is called during CX setup; the user should not need to call it.

19.16.2.22 void CX::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.

19.16.2.23 void CX::CX_Display::setYIncreasesUpwards (bool *upwards*)

Set whether the y-axis vales should increase upwards.

Parameters

upwards	If `true`, y-values will increase upwards. If `false`, y-values will increase downwards (the default).

19.16.2.24 void CX::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

19.16.2.25 void CX::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.

19.16.2.26 std::map< std::string, CX_DataFrame > CX::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 work better than others for the tested computer.

In the resulting data, there are three 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 CX_Display::useHardwareVSync() and CX_Display::useSoftwareVSync()). Other columns, giving data from the tests, are explained below. Whatever combination of Vsync works best can be set up for use in experiments using CX_Display::useHardwareVSync() and CX_Display::useSoftwareVSync() to set the Vsync mode in code or with CX_Display::configureFromFile() to set the values based on a configuration file.

The threading mode that is used in stimulus presentation is primarily determined by CX_SlidePresenter with the CX::CX_SlidePresenter::Configuration::SwappingMode setting, although some experiments might want to use threaded swaps directly. If you are not using a multi-threaded swapping mode with a CX_SlidePresenter, you probably don't need to do these tests with a secondary thread, which you can do by setting the argument `testSecondaryThread` to false when you call this function.

Continuous swapping test

This test examines the case of constantly swapping the front and back buffers. It measures the amount of time between swaps, which should always approximately equal the frame period. The raw data from this test can be found in the "continuousSwapping" CX_DataFrame in the returned map. The raw data are in flat field format, with the duration data in the "duration" column and the test conditions in the "hardVSync", "softVSync", and "thread"

columns. A summary of this test can be found in the "summary" data frame in the returned map. In the summary, columns related to this test are prefixed with "cs" and give the mean, standard deviation, minimum, and maximum swap duration in each of the conditions that were tested.

If the swapping durations are not very consistent, which can be determined by visual examination and by looking at the standard deviation, min, and max, 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, so that it might nearly appear to be a shade of grey. If you see slow flickering or solid black or white, that is an error. If there are horizontal lines that alternate black and white, that is a signature of vertical tearing, which is an error (except for when both kinds of Vsync are turned off, in which case it is allowable and a good demonstration of the value of Vsync).

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). The raw swap durations for this test can be found in the "waitSwap" data frame in the returned map, with the test conditions given in the "hardVSync", "softVSync", and "thread" columns. The "type" column indicates whether a given swap duration was long, short, or normal and the "duration" column gives the durations of the swaps. Summary data from this test can be found in the "summary" data frame in the returned map. The columns in the summary data that correspond to this test are prefixed "ws".

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 (i.e. the length of one frame), 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 on and off (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.

If there are horizontal lines that alternate between black and white, that is a sign of vertical tearing, which is an error.

Note: 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 in the secondary thread so there will be no wait swaps. You can enable constant swapping in the secondary thread with CX_Display::setAutomatic↩ Swapping().

Remedial measures

If all of the tests fail, there are a number of possible reasons.

One of the primary reasons for failure is that the video card driver is not honoring the requested vertical synchronization settings that CX tries during the test. A workaround for this issue is to force vertical synchronization on in the video driver settings, which can be done through the GUIs for the drivers. In my experience, this is a good first thing to try and often improves things substantially.

It should not be assumed that using both hardware and software Vsync is better than using only one of the two. The failure case I typically observe if both are enabled is that each buffer swap will take twice the nominal frame period. If this error occurrs, try using just one type of Vsync.

If none of the wait swap test configurations result in acceptable behavior, the implication is that there is an error in the implementation of Vsync for your computer. If this is the case, you should be careful about using stimulus presentation code that requests two or more swaps in a row (i.e. to swap in two different stimuli on two consecutive frames) following a multi-frame interval in which there were no buffer swaps. What may happen is that the first

stimulus may never be presented (especially if the "short" duration on the test is ∼0). If the short duration is not 0, then that stimulus should be presented, but if the long duration is less than 3P, the preceding stimulus may be cut short. In cases like this, you may want to configure the CX_Display to swap buffers automatically in a secondary thread all the time (see CX_Display::setAutomaticSwapping()), so that there are never swaps after several frames without swaps. The "animation" example shows how to use CX_Display::hasSwappedSinceLast↩ Check() to synchronize rendering in the main thread with buffer swaps in the secondary thread. Note that if your computer does not have at least a 2 core CPU, using a secondary thread to constantly swap buffers is not a good solution, because the secondary thread will peg 1 CPU at 100% usage.

If none of these remedial measures corrects you problems, you may want to try another psychology experiment package. However, many of them use OpenGL and so if the problem is your OpenGL configuration (hardware and software), switching to another package that uses OpenGL is unlikely to fix your problem (if it does, let me know because that could point to an issue in CX or openFrameworks).

Parameters

desiredTest↩ Duration	An approximate amount of time to spend performing all of the tests, so the time if divided among all of 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 map containing CX_DataFrames. One data frame, named "summary" in the map, contains summary statistics. Another data frame, named "constantSwapping", contains raw data from the constant swapping test. Another data frame, named "waitSwap", contains raw data from the wait swap test.

Note

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

19.16.2.27 void CX::CX_Display::useHardwareVSync (bool *b*)

Sets whether the display is using hardware VSync to control frame presentation. Without some form of Vsync, vertical tearing may 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 Visual Stimuli for information on what VSync is.

19.16.2.28 void CX::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 use↩ HardwareVSync() 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 Visual Stimuli for information on what Vsync is.

19.16.2.29 void CX::CX_Display::waitForBufferSwap (void)

If the display is automatically swapping, this function blocks until a buffer swap has ocurred. If the display is not automatically swapping, it returns immediately.

19.16.2.30 void CX::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 with other tasks. To demystify things, this function simply calls glFinish().

See also

Blocking Code

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

- CX_Display.h
- CX_Display.cpp

19.17. CX::CX_InputManager Class Reference

`#include <CX_InputManager.h>`

Public Member Functions

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

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().

Friends

- CX_InputManager **Private::inputManagerFactory** (void)

19.17.1 Detailed Description

This class is responsible for managing three basic input devices: the keyboard, mouse, and, if available, joystick. You access each of these devices with the corresponding class member: 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. Alternately, you can call CX_Mouse::enable() or CX_Keyboard::enable(), if that makes more sense to you.

The overall structure of input in CX revolves around polling for new input, with CX_InputManager::pollEvents(). This is the only way to get new input events for the keyboard and mouse. When pollEvents() is called, CX checks to see if any keyboard or mouse input has been given since the last time pollEvents() was called. If there are new events, they are put into input device specific queues. You can find out how many input events are available in, for example, the keyboard queue by calling `CX_Keyboard::availableEvents()`. If there are any available events, you can the first one with CX_Keyboard::getNextEvent(). CX_Keyboard::getNextEvent() returns a CX_Keyboard::Event struct that contains information about the event. This all works the same way for the mouse.

```
//Basic sequence of events:
if (Input.pollEvents()) { //Returns true if there are any events available on any input
      devices.
    while (Input.Keyboard.availableEvents()) { //For each new event
        CX_Keyboard::Event keyEvent = Input.Keyboard.getNextEvent(); //Pop that
       event out of the queue
        if (keyEvent.type == CX_Keyboard::PRESSED) { //If a key has been pressed
            //Process the keypress...
            if (keyEvent.key == 'a') {
                //do something...
            }
        }
    }
}
```

If the timing of input is critical for you application, you should poll for input regularly, because the quality of input timestamps is based on the regularity of polling.

This class has a private constructor because you should never need more than one of them. If you really, really need more than one, you can use CX::Private::inputManagerFactory() to make one.

19.17.2 Member Function Documentation

19.17.2.1 void CX::CX_InputManager::clearAllEvents (bool *poll* = `false`)

This function clears all events on all input devices.

Parameters

poll	If `true`, events are polled before they are cleared, so that events that hadn't yet made it into the device specific queues (e.g. the Keyboard queue) are cleared as well.

19.17.2.2 bool CX::CX_InputManager::pollEvents (void)

This function polls for new events on all of the configured input devices (see CX_InputManager::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. Note that the events do not neccessarily need to be new events in order for this to return `true`. If there were events that were already stored in Mouse, Keyboard, or Joystick that had not been processed by user code at the time this function was called, this function will return `true`.

19.17.2.3 bool CX::CX_InputManager::setup (bool *useKeyboard*, bool *useMouse*, int *joystickIndex* = `-1`)

Set up 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

19.18. CX::CX_Joystick Class Reference

`#include <CX_Joystick.h>`

Classes

- struct Event

Public Types

- enum EventType { BUTTON_PRESS, BUTTON_RELEASE, AXIS_POSITION_CHANGE }

Public Member Functions

- bool setup (int joystickIndex)
- std::string getJoystickName (void)
- int getJoystickIndex (void)
- bool pollEvents (void)
- int availableEvents (void)
- CX_Joystick::Event getNextEvent (void)
- void clearEvents (void)
- std::vector< CX_Joystick::Event > copyEvents (void)

  Return a vector containing a copy of the currently stored events. The events stored by the input device are unchanged. The first element of the vector is the oldest event.
- std::vector< float > getAxisPositions (void)
- std::vector< unsigned char > getButtonStates (void)
- void appendEvent (CX_Joystick::Event ev)

19.18.1 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. Unlike CX_Keyboard and CX_Mouse, CX_Joystick does not need to be in a CX_InputManager to work.

19.18.2 Member Function Documentation

19.18.2.1 void CX::CX_Joystick::appendEvent (CX_Joystick::Event *ev*)

Appends a joystick event to the event queue without any modification (e.g. the timestamp is not set to the current time, it is left as-is). This can be useful if you want to have a simulated participant perform the task for debugging purposes.

Parameters

ev	The event to append.

19.18.2.2 int CX::CX_Joystick::availableEvents (void)

Get the number of available events for this input device. Events can be accessed with CX_Joystick::getNextEvent() or CX_Joystick::copyEvents().

19.18.2.3 void CX::CX_Joystick::clearEvents (void)

Clear (delete) all events from this input device.

Note

Unpolled events are not cleared by this function, which means that responses made after a call to CX_Input↩Manager::pollEvents() but before a call to clearEvents() will not be removed by calling clearEvents().

19.18.2.4 vector< float > CX::CX_Joystick::getAxisPositions (void)

This function returns in the current positions of the joystick axes.

Returns

A vector of the current axis positions.

19.18.2.5 vector< unsigned char > CX::CX_Joystick::getButtonStates (void)

This function returns in the current states of the joystick buttons.

Returns

A vector of the current button states.

19.18.2.6 int CX::CX_Joystick::getJoystickIndex (void)

Get the integer index of the currently selected joystick.

19.18.2.7 std::string CX::CX_Joystick::getJoystickName (void)

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

19.18.2.8 CX_Joystick::Event CX::CX_Joystick::getNextEvent (void)

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

19.18.2.9 bool CX::CX_Joystick::pollEvents (void)

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

Returns

True if there are new events.

19.18.2.10 bool CX::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

19.19. CX::CX_Keyboard Class Reference

`#include <CX_Keyboard.h>`

Classes

- struct Event
- struct Keycodes

Public Types

- enum EventType { PRESSED, RELEASED, REPEAT }

Public Member Functions

- void enable (bool enable)
- bool enabled (void)

  Returns `true` if the keyboard is enabled.
- int availableEvents (void) const
- CX_Keyboard::Event getNextEvent (void)
- void clearEvents (void)
- std::vector< CX_Keyboard::Event > copyEvents (void)

  Return a vector containing a copy of the currently stored events. The events stored by the input device are unchanged. The first element of the vector is the oldest event.
- bool isKeyHeld (int key) const
- CX_Keyboard::Event waitForKeypress (int key, bool clear=true, bool eraseEvent=false)

  Identical to CX_Keyboard::waitForKeypress(std::vector<int>, bool, bool), except that this only takes a length 1 vector.
- CX_Keyboard::Event waitForKeypress (std::vector< int > keys, bool clear=true, bool eraseEvent=false)
- bool isChordHeld (const std::vector< int > &chord) const
- void appendEvent (CX_Keyboard::Event ev)

Friends

- class **CX_InputManager**

19.19.1 Detailed Description

This class is responsible for managing the keyboard. You should not need to create an instance of this class: use the instance of CX_Keyboard within CX::Instances::Input instead.

19.19.2 Member Function Documentation

19.19.2.1 void CX::CX_Keyboard::appendEvent (CX_Keyboard::Event *ev*)

Appends a keyboard event to the event queue without any modification (e.g. the timestamp is not set to the current time, it is left as-is). This can be useful if you want to have a simulated participant perform the task for debugging purposes. If the event type is CX_Keyboard::PRESSED or CX_Keyboard::RELEASED, the key of the event will be added to or removed from the list of held keys, depending on event type.

Parameters

ev	The event to append.

19.19.2.2 int CX::CX_Keyboard::availableEvents (void) const

Get the number of available events for this input device. Events can be accessed with CX_Keyboard::getNextEvent() or CX_Keyboard::copyEvents().

19.19.2.3 void CX::CX_Keyboard::clearEvents (void)

Clear (delete) all events from this input device.

Note

Unpolled events are not cleared by this function, which means that responses made after a call to CX_Input↩Manager::pollEvents() but before a call to clearEvents() will not be removed by calling clearEvents().

19.19.2.4 void CX::CX_Keyboard::enable (bool *enable*)

Enable or disable the keyboard.

Parameters

enable	If `true`, the keyboard will be enabled; if `false` it will be disabled.

19.19.2.5 CX_Keyboard::Event CX::CX_Keyboard::getNextEvent (void)

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

19.19.2.6 bool CX::CX_Keyboard::isChordHeld (const std::vector< int > & *chord*) const

Checks whether the given key chord is held, i.e. all of the keys in `chord` are held simultaneously.

Returns

`false` if `chord` is empty or if not all of the keys in `chord` are held. `true` if all of the keys in `chord` are held.

19.19.2.7 bool CX::CX_Keyboard::isKeyHeld (int *key*) const

This function checks to see if the given key is held, which means a keypress has been received, but not a key release.

Parameters

key	The character literal for the key you are interested in or special key code from CX::Keycode.

Returns

`true` if the given key is held, `false` otherwise.

19.19.2.8 CX_Keyboard::Event CX::CX_Keyboard::waitForKeypress (std::vector< int > *keys*, bool *clear* = `true`, bool *eraseEvent* = `false`)

Wait until the first of the given `keys` is pressed. This specifically checks that a key has been pressed: If it was already held at the time this function was called and then released, it will have to be pressed again before this function will return. Returns a CX_Keyboard::Event for the key that was waited on, optionally removing that event that caused this function to return from the queue of stored events if `eraseEvent` is `true`.

Parameters

keys	A vector of key codes for the keys that will be waited on. If any of the codes are -1, any keypress will cause this function to return. Should be character literals or from CX::Keycode.
clear	If `true`, all waiting events will be flushed with CX_InputManager::pollEvents() and then all keyboard events will be cleared both before and after waiting for the keypress. If `false` and `this->availableEvents() > 0`, it is possible that one of the available events will include a keypress for one of the keys to be waited on, in which case this function will return immediately.
eraseEvent	If `true`, the event that caused this function to return will be erased from the queue of stored events. The implication of this removal is that the return value of this function is the only opportunity to gain access to the event that caused this function to return. The advantage of this approach is that if, after some given key is pressed, all events in the queue are processed, you are guaranteed to not hit the same event twice (once from the return value of this function, once from processing the queue).

Returns

A CX_Keyboard::Event with information about the keypress that caused this function to return.

Note

If the keyboard is not enabled at the time this function is called, it will be enabled for the duration of the function and then disabled at the end of the function.

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

- CX_Keyboard.h
- CX_Keyboard.cpp

19.20. CX::Util::CX_LapTimer Class Reference

`#include <CX_TimeUtilities.h>`

Public Member Functions

- CX_LapTimer (CX_Clock *clock, unsigned int logSamples=0)
- void setup (CX_Clock *clock, unsigned int logSamples=0)
- void restart (void)
- void takeSample (void)
- unsigned int collectedSamples (void)
- CX_Millis mean (void)

  Get the mean value of the stored lap times.
- CX_Millis min (void)

  Get the shortest stored lap time.
- CX_Millis max (void)

  Get the longest stored lap time.
- CX_Millis stdDev (void)

  Get the standard deviation of the stored lap times.
- std::string getStatString (void)

Public Attributes

- std::string name

  If this is set, it will be printed at the start of the string returned by getStatString() and in automatically logged messages.

19.20.1 Detailed Description

This class can be used for profiling loops. It measures the amount of time that elapses between subsequent calls to takeSample(). One possible use is to determine how long it takes between calls to an important function, like CX_InputManager::pollEvents() or CX_Display::swapBuffers()

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

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

19.20.2 Constructor & Destructor Documentation

19.20.2.1 CX::Util::CX_LapTimer::CX_LapTimer (CX_Clock * *clock*, unsigned int *logSamples* = 0)

Construct and set up a CX_LapTimer. See CX_LapTimer::setup() for a description of the parameters.

19.20.3 Member Function Documentation

19.20.3.1 unsigned int CX::Util::CX_LapTimer::collectedSamples (void)

Returns the number of lap durations that have been collected.

19.20.3.2 std::string CX::Util::CX_LapTimer::getStatString (void)

Get a string summarizing some basic descriptive statistics for the currently stored lap durations.

Returns

A string containing the minimum, mean, maximum, and standard deviation, in ms, of the collected samples.

19.20.3.3 void CX::Util::CX_LapTimer::restart (void)

Restart data collection. All collected samples are cleared.

19.20.3.4 void CX::Util::CX_LapTimer::setup (CX_Clock * *clock*, unsigned int *logSamples* = 0)

Set up the CX_LapTimer with the selected clock source and the number of samples to log between each automatic logging of results.

Parameters

clock	The instance of CX_Clock to use.
logSamples	If this is not 0, then every `logSamples` samples, a string containing information about the last `logSamples` samples will be logged and then those samples will be cleared.

19.20.3.5 void CX::Util::CX_LapTimer::takeSample (void)

Take a single sample of time. If at least one previous sample has been taken, the difference between the current time and the previous time is stored as the duration of that "lap" through the code.

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

- CX_TimeUtilities.h
- CX_TimeUtilities.cpp

19.21. 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)
- void setup (float pixelsPerUnit, bool roundResult=false)
- float operator() (float length) override
- float inverse (float pixels) override
- bool configureFromFile (std::string filename, std::string delimiter="=", bool trimWhitespace=true, std::string commentString="//")

19.21.1 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 distance unit right and 2 units down from that point.
```

19.21.2 Constructor & Destructor Documentation

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

Constructs a CX_LengthToPixelConverter with the given configuration. See setup() for the meaning of the parameters.

19.21.3 Member Function Documentation

19.21.3.1 bool CX::Util::CX_LengthToPixelConverter::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_LengthToPixelConverter. The format of the file is provided in the example code below.

Sample configuration file:

```
L2PC.pixelsPerUnit = 35
L2PC.roundResult = true
```

All of the configuration keys are used in this example. Note that the "L2PC" prefix allows this configuration to be embedded in a file that also performs other configuration functions.

See CX_LengthToPixelConverter::setup() for details about the meanings of the configuration options.

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 (""), everything on a line following the first instance of `commentString` will be ignored.

Returns

`true` if there were no problems reading in the file, `false` otherwise.

19.21.3.2 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.

19.21.3.3 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.

19.21.3.4 void CX::Util::CX_LengthToPixelConverter::setup (float *pixelsPerUnit*, bool *roundResult* = `false`)

Sets up 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.

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

- CX_UnitConversion.h
- CX_UnitConversion.cpp

19.22. CX::CX_Logger Class Reference

`#include <CX_Logger.h>`

Classes

- struct MessageFlushData

Public Types

- enum Level {
  LOG_ALL = 0, **LOG_VERBOSE** = 1, **LOG_NOTICE** = 2, **LOG_WARNING** = 3,
  LOG_ERROR = 4, **LOG_FATAL_ERROR** = 5, **LOG_NONE** = 6 }

Public Member Functions

- CX::Private::CX_LogMessageSink log (Level level, std::string module="")
- CX::Private::CX_LogMessageSink verbose (std::string module="")

  Equivalent to `log(CX_Logger::Level::LOG_VERBOSE, module).`
- CX::Private::CX_LogMessageSink notice (std::string module="")

  Equivalent to `log(CX_Logger::Level::LOG_NOTICE, module).`
- CX::Private::CX_LogMessageSink warning (std::string module="")

  Equivalent to `log(CX_Logger::Level::LOG_WARNING, module).`
- CX::Private::CX_LogMessageSink error (std::string module="")

  Equivalent to `log(CX_Logger::Level::LOG_ERROR, module).`
- CX::Private::CX_LogMessageSink fatalError (std::string module="")

  Equivalent to `log(CX_Logger::Level::LOG_FATAL_ERROR, module).`
- void level (Level level, std::string module)
- void levelForAllModules (Level level)
- void levelForConsole (Level level)

  Set the log level for messages to be printed to the console.
- void levelForFile (Level level, std::string filename="CX_LOGGER_DEFAULT")
- void levelForAllExceptions (Level level)
- void levelForExceptions (Level level, std::string module)
- Level getModuleLevel (std::string module)
- void flush (void)
- void clear (void)

  Clear all stored log messages.
- void timestamps (bool logTimestamps, std::string format="%H:%M:%S.%i")
- void setMessageFlushCallback (std::function< void(MessageFlushData &)> f)
- void captureOFLogMessages (bool capture)

19.22.1 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 much better to use the preinstantiated CX::Instances::Log.

example-logging shows a number of the features of CX_Logger.

CX_Logger is designed to help prevent timing errors. Messages can be logged at any time during program execution. If these messages were immediately outputted to the console or to files, it could disrupt a timing-critical section of code. For this reason, logged messages are stored by the CX_Logger until the user requests that all stored messages be outputted to the logging targets with CX_Logger::flush(). The user can choose an appropriate, non-timing-critical time at which to call flush().

By default, messages are logged to the console window that opens with CX programs. Optionally, messages can also be logged to any number of files using CX_Logger::levelForFile(). For each logging target (i.e. the console and the logfiles), you can filter out less severe messages. For example, you could have two log files, one of which

contains all messages and the other of which only contains errors and fatal errors. By default, no logfiles are created and all messages (with only a few exceptions) are logged to the console. There are a few openFrameworks classes that are extremely verbose, like ofFbo, and less severe messages from those classes are suppressed by default. You can undo this behavior by simply calling CX_Logger::levelForAllModules() with CX_Logger::Level::LOG_ALL as the argument.

This class is designed to be partially thread safe. It is safe to use any of the message logging functions (log(), verbose(), notice(), warning(), error(), and fatalError()) in multiple threads at once. Other than those functions, the other functions should be called only from one thread (presumably the main thread).

19.22.2 Member Function Documentation

19.22.2.1 void CX::CX_Logger::captureOFLogMessages (bool *capture*)

Set this instance of CX_Logger to be the target of any messages created by openFrameworks logging functions. This function is called during CX setup for CX::Instances::Log. You do not need to call it yourself.

Parameters

capture	If `true`, capture oF log messages. If `false`, oF log messages go to the console.

19.22.2.2 void CX::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).

Note

This function is not 100% thread-safe: Only call it from the main thread.

19.22.2.3 CX_Logger::Level CX::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 level for `module`.

19.22.2.4 void CX::CX_Logger::level (Level *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_Logger::Level enum for valid values.
module	A string representing one of the modules from which log messages are generated.

19.22.2.5 void CX::CX_Logger::levelForAllExceptions (Level *level*)

See CX::CX_Logger::levelForExceptions() for more information.

Parameters

level	The default exception level.

19.22.2.6 void CX::CX_Logger::levelForAllModules (Level *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.

19.22.2.7 void CX::CX_Logger::levelForExceptions (Level *level*, std::string *module*)

When a logged message is stored, if its log level is greater than or equal to the exception level for the given module, an exception will be thrown. The exception will be a std::runtime_error. By default, the exception level is LOG_NONE, i.e. that no logged messages will cause an exception to be thrown.

You might want to use this feature for two reasons: 1) There are certain really serious errors that sometimes happen while the experiment is running that are not themselves exceptions but that you want to be exceptions so that they will not allow the program to continue in an erroneous state. 2) For debugging purposes. When an exception is thrown it triggers a breakpoint in some IDEs. When that happens, you have a full stack trace and interactive debugger environment to work with to help determine why the logged message was logged.

Note that, for technical reasons, this exception throwing feature may not work properly on some systems. If you get unexpected program termination when trying to use this feature, you may just want to not use it.

Parameters

level	The desired exception level. The naming of the values in CX_LogLevel is slightly confusing in the context of this function. Instead of, e.g., LOG_WARNING, think of the value as EXCEPTION_ON_WARNING (or greater).
module	The module to set the exception level for.

19.22.2.8 void CX::CX_Logger::levelForFile (Level *level*, std::string *filename* = "CX_LOGGER_DEFAULT")

Sets the log level for the file with the 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 (typically the console).

Parameters

level	Log messages with level greater than or equal to this level will be outputted to the file. See the CX::CX_Logger::Level enum for valid values.
filename	The name of the file to output to. 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.

19.22.2.9 CX::Private::CX_LogMessageSink CX::CX_Logger::log (Level *level*, std::string *module* = " ")

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

```
Log.log(CX_Logger::Level::LOG_WARNING, "moduleName") << "Special message number: " << 20;
```

Possible output: "[warning] <moduleName> Special message number: 20"

A newline is inserted automatically at the end of each message.

Parameters

level	Log level for this message. This has implications for message filtering. See CX::CX_Logger::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 CX::CX_Logger::level().

Returns

An object that can have log messages given to it as though it were a std::ostream.

Note

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

19.22.2.10 void CX::CX_Logger::setMessageFlushCallback (std::function< void(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.

19.22.2.11 void CX::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).

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

- CX_Logger.h
- CX_Logger.cpp

19.23. CX::CX_Mouse Class Reference

`#include <CX_Mouse.h>`

Classes

- struct Event

Public Types

- enum Buttons { **LEFT** = OF_MOUSE_BUTTON_LEFT, **MIDDLE** = OF_MOUSE_BUTTON_MIDDLE, **RIGHT** = OF_MOUSE_BUTTON_RIGHT }
- enum EventType { MOVED, PRESSED, RELEASED, DRAGGED, SCROLLED }

Public Member Functions

- void enable (bool enable)
- bool enabled (void)

  Returns `true` if the mouse is enabled.
- int availableEvents (void)
- CX_Mouse::Event getNextEvent (void)
- std::vector< CX_Mouse::Event > copyEvents (void)

  Return a vector containing a copy of the currently stored events. The events stored by the input device are unchanged. The first element of the vector is the oldest event.
- void clearEvents (void)
- void showCursor (bool show)
- void setCursorPosition (ofPoint pos)
- ofPoint getCursorPosition (void)
- bool isButtonHeld (int button) const
- CX_Mouse::Event waitForButtonPress (int button, bool clear=true, bool eraseEvent=false)

  Identical to CX_Mouse::waitForButtonPress(std::vector<int>, bool, bool), except that this only takes a length 1 vector.
- CX_Mouse::Event waitForButtonPress (std::vector< int > buttons, bool clear=true, bool eraseEvent=false)
- void appendEvent (CX_Mouse::Event ev)

Friends

- class **CX_InputManager**

19.23.1 Detailed Description

This class is responsible for managing the mouse. You should not need to create an instance of this class: use the instance of CX_Mouse within CX::Instances::Input instead.

19.23.2 Member Function Documentation

19.23.2.1 void CX::CX_Mouse::appendEvent (CX_Mouse::Event *ev*)

Appends a mouse event to the event queue without any modification (e.g. the timestamp is not set to the current time, it is left as-is). This can be useful if you want to have a simulated participant perform the task for debugging purposes. If the event type is CX_Mouse::PRESSED or CX_Mouse::RELEASED, the button of the event will be added to or removed from the list of held buttons, depending on event type.

Parameters

ev	The event to append.

19.23.2.2 int CX::CX_Mouse::availableEvents (void)

Get the number of available events for this input device. Events can be accessed with CX_Mouse::getNextEvent() or CX_Mouse::copyEvents().

19.23.2.3 void CX::CX_Mouse::clearEvents (void)

Clear (delete) all events from this input device.

Note

Unpolled events are not cleared by this function, which means that responses made after a call to CX_Input↩Manager::pollEvents() but before a call to clearEvents() will not be removed by calling clearEvents().

19.23.2.4 void CX::CX_Mouse::enable (bool *enable*)

Enable or disable the mouse.

Parameters

enable	If `true`, the mouse will be enabled; if `false` it will be disabled.

19.23.2.5 ofPoint CX::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.

19.23.2.6 CX_Mouse::Event CX::CX_Mouse::getNextEvent (void)

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

19.23.2.7 bool CX::CX_Mouse::isButtonHeld (int *button*) const

This function checks to see if the button key is held, which means a button press has been received, but not a button release.

Parameters

button	The index of a button to check for. For the most common named buttons, see the CX_Mouse::Buttons enum.

Returns

`true` if the given key is held, `false` otherwise.

19.23.2.8 void CX::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.

19.23.2.9 void CX::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.

19.23.2.10 CX_Mouse::Event CX::CX_Mouse::waitForButtonPress (std::vector< int > *buttons*, bool *clear* = `true`, bool *eraseEvent* = `false`)

Wait until the first of the given `buttons` is pressed. This specifically checks that a button has been pressed: If it was already held at the time this function was called and then released, it will have to be pressed again before this function will return. Returns a CX_Mouse::Event for the buttons that were waited on, optionally removing that event that caused this function to return from the queue of stored events if `eraseEvent` is `true`.

Parameters

buttons	A vector of button indices for the buttons that will be waited on. If any of the values are -1, any button press will cause this function to return. The button indices may be from CX_Mouse::Buttons or just raw integers.
clear	If `true`, all waiting events will be flushed with CX_InputManager::pollEvents() and then all mouse events will be cleared both before and after waiting for the keypress. If `false` and `this->availableEvents() > 0`, it is possible that one of the available events will include a press for one of the buttons to be waited on, in which case this function will return immediately.
eraseEvent	If `true`, the event that caused this function to return will be erased from the queue of stored events. The implication of this removal is that the return value of this function is the only opportunity to gain access to the event that caused this function to return. The advantage of this approach is that if, after some given key is pressed, all events in the queue are processed, you are guaranteed to not hit the same event twice (once from the return value of this function, once from processing the queue).

Returns

A CX_Mouse::Event with information about the button press that caused this function to return.

Note

If the mouse is not enabled at the time this function is called, it will be enabled for the duration of the function and then disabled at the end of the function.

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

- CX_Mouse.h
- CX_Mouse.cpp

19.24. CX::CX_RandomNumberGenerator Class Reference

`#include <CX_RandomNumberGenerator.h>`

Public Member Functions

- CX_RandomNumberGenerator (void)
- void setSeed (unsigned long seed)
- void setSeed (const std::string &s)
- 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 > sampleExclusive (unsigned int count, const std::vector< T > &values, const T &exclude, bool withReplacement)
- template<typename T >
  std::vector< T > sampleExclusive (unsigned int count, const std::vector< T > &values, const std::vector< T > &exclude, bool withReplacement)
- 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)
- template<typename T >
  std::vector< T > sampleUniformRealizations (unsigned int count, T lowerBound_closed, T upperBound_↩ open)
- template<typename T >
  std::vector< T > sampleNormalRealizations (unsigned int count, T mean, T standardDeviation)
- std::vector< unsigned int > sampleBinomialRealizations (unsigned int count, unsigned int trials, double probSuccess)
- std::mt19937_64 & getGenerator (void)

19.24.1 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).

When an instance of this class is constructed, it is automatically seeded from a high-entropy source. In particular, a `std::random_device`. See the documentation for CX_RandomNumberGenerator::CX_RandomNumber↩ Generator() for more information.

The monolithic structure of CX_RandomNumberGenerator provides a certain important feature that a collection of loose functions does not have, which is the ability to easily 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 an instance of this class. CX_RandomNumberGenerator::getSeed() allows you to recover 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.

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++ standard library random number generators are not thread safe, CX_Random↩ NumberGenerator is not thread safe. If you want to use a CX_RandomNumberGenerator in a thread, that thread should have its own CX_RandomNumberGenerator. You should create a new CX_RandomNumberGenerator for the thread. You may seed the thread's new CX_RandomNumberGenerator with CX::Instances::RNG, for example.

19.24.2 Constructor & Destructor Documentation

19.24.2.1 CX::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 always going to be 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 (this is not cryptography, just fooling humans).

19.24.3 Member Function Documentation

19.24.3.1 std::mt19937_64 & CX::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());
```

19.24.3.2 CX_RandomInt_t CX::CX_RandomNumberGenerator::getMaximumRandomInt (void)

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

Returns

The maximum value.

19.24.3.3 CX_RandomInt_t CX::CX_RandomNumberGenerator::getMinimumRandomInt (void)

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

Returns

The minimum value.

19.24.3.4 unsigned long CX::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_RandomNumber↩ Generator.

19.24.3.5 double CX::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.

19.24.3.6 CX_RandomInt_t CX::CX_RandomNumberGenerator::randomInt (void)

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

Returns

The int.

19.24.3.7 CX_RandomInt_t CX::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.

19.24.3.8 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.

19.24.3.9 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.
with↩ Replacement	Sample with or without replacement.

Returns

A vector of the sampled values.

Note

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

19.24.3.10 std::vector< int > CX::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.

19.24.3.11 std::vector< unsigned int > CX::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.

19.24.3.12 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 `values.size() * blocksToSample` elements.

19.24.3.13 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.

19.24.3.14 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.

19.24.3.15 template<typename T > std::vector< T > CX::CX_RandomNumberGenerator::sampleExclusive (unsigned int *count,* const std::vector< T > & *values,* const T & *exclude,* bool *withReplacement*)

Sample some number of random values, with or without replacement, from a vector without the possibility of getting the excluded value.

Parameters

count	The number of values to sample.
values	The vector of values to sample from.
exclude	The vector of values to exclude from sampling.
withReplacement	If true, values will be sampled with replacement (i.e. the same value can be sampled more than once).

Returns

The sampled values, of equal number to count, unless an error has occurred.

Note

If all of the values are excluded, an error will be logged and an empty vector will be returned.

19.24.3.16 template<typename T > std::vector< T > CX::CX_RandomNumberGenerator::sampleExclusive (unsigned int *count,* const std::vector< T > & *values,* const std::vector< T > & *exclude,* bool *withReplacement*)

Sample some number of random values, with or without replacement, from a vector without the possibility of getting any of the excluded values.

Parameters

count	The number of values to sample.
values	The vector of values to sample from.
exclude	The vector of values to exclude from sampling.
withReplacement	If true, values will be sampled with replacement (i.e. the same value can be sampled more than once).

Returns

The sampled values, of equal number to count, unless an error has occurred.

Note

If all of the values are excluded, an error will be logged and an empty vector will be returned.

19.24.3.17 template<typename T > std::vector< T > CX::CX_RandomNumberGenerator::sampleNormalRealizations (unsigned int *count,* T *mean,* T *standardDeviation*)

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

Template Parameters

T	The precision with which to sample (should be `float` or `double` most of the time).

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.

19.24.3.18 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));
```

19.24.3.19 template<typename T > std::vector< T > CX::CX_RandomNumberGenerator::sampleUniformRealizations (unsigned int *count*, T *lowerBound_closed*, T *upperBound_open*)

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

Template Parameters

T	The precision with which to sample (should be `float` or `double` most of the time).

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.

19.24.3.20 void CX::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.

19.24.3.21 void CX::CX_RandomNumberGenerator::setSeed (const std::string & *seedString*)

This function provides a method of setting the seed using an arbitrary string (e.g. date-time and participant number) as the seed. A CRC32 checksum is used to convert the string into an unsinged long, which is then used as the seed for the CX_RandomNumberGenerator. You can retrieve the seed with getSeed().

Parameters

seedString	The string from which the new seed will be calculated.

19.24.3.22 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.

19.24.3.23 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

19.25. CX::Util::CX_SegmentProfiler Class Reference

`#include <CX_TimeUtilities.h>`

Public Member Functions

- CX_SegmentProfiler (CX_Clock *clock, unsigned int logSamples=0)
- void setup (CX_Clock *clock, unsigned int logSamples=0)
- void t1 (void)
- void t2 (void)
- unsigned int collectedSamples (void)
- void restart (void)
- std::string getStatString (void)
- CX_Millis mean (void)

  Get the mean of the stored segment durations.
- CX_Millis min (void)

  Get the shortest of the stored segment durations.
- CX_Millis max (void)

  Get the longest of the stored segment durations.
- CX_Millis stdDev (void)

  Get the standard deviation of the stored segment durations.

Public Attributes

- std::string name

  If this is set, it will be printed at the start of the string returned by getStatString() and in automatically logged messages.

19.25.1 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;
```

19.25.2 Constructor & Destructor Documentation

19.25.2.1 CX::Util::CX_SegmentProfiler::CX_SegmentProfiler (CX_Clock * *clock*, unsigned int *logSamples* = 0)

Set up the CX_SegmentProfiler with the selected clock source and the number of samples to log between each automatic logging of results.

Parameters

clock	The instance of CX_Clock to use.
logSamples	If this is not 0, then every `logSamples` samples, a string containing information about the last `logSamples` samples will be logged and then those samples will be cleared.

19.25.3 Member Function Documentation

19.25.3.1 unsigned int CX::Util::CX_SegmentProfiler::collectedSamples (void)

Returns

The number of collected samples.

19.25.3.2 std::string CX::Util::CX_SegmentProfiler::getStatString (void)

Get a string summarizing some basic descriptive statistics for the currently stored data.

Returns

A string containing the minimum, mean, maximum, and standard deviation, in ms, of the stored data.

19.25.3.3 void CX::Util::CX_SegmentProfiler::restart (void)

Restart data collection. All collected samples are cleared.

19.25.3.4 void CX::Util::CX_SegmentProfiler::setup (CX_Clock * *clock*, unsigned int *logSamples* = 0)

Set up the CX_SegmentProfiler with the selected clock source and the number of samples to log between each automatic logging of results.

Parameters

clock	The instance of CX_Clock to use.
logSamples	If this is not 0, then every `logSamples` samples, a string containing information about the last `logSamples` samples will be logged.

19.25.3.5 void CX::Util::CX_SegmentProfiler::t1 (void)

This function takes a timestamp at the current time and will be compared with the timestamp taken with t2().

19.25.3.6 void CX::Util::CX_SegmentProfiler::t2 (void)

This function stores the difference between the current time and the time captured with t1(). If enough samples have been collected, equal to the value of `logSamples` during setup(), a summary statistics string will be automatically logged.

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

- CX_TimeUtilities.h
- CX_TimeUtilities.cpp

19.26. 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** }
- enum SwappingMode { SwappingMode::SINGLE_CORE_BLOCKING_SWAPS, SwappingMode::MULTI_↩ CORE }

Public Member Functions

- bool setup (CX_Display *display=&CX::Instances::Disp)
- bool setup (const CX_SlidePresenter::Configuration &config)
- 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 a slide presentation, if any is in progress.
- bool isPresentingSlides (void) const

  Returns `true` if slide presentation is in progress, even if the first slide has not yet been presented.
- bool presentSlides (void)

- void clearSlides (void)
- std::vector< CX_SlidePresenter::Slide > & getSlides (void)
- CX_SlidePresenter::Slide & getSlideByName (std::string name)
- std::string getLastPresentedSlideName (void) const
- std::vector< CX_Millis > getActualPresentationDurations (void)
- std::vector< unsigned int > getActualFrameCounts (void)
- CX_SlidePresenter::PresentationErrorInfo checkForPresentationErrors (void) const
- std::string printLastPresentationInformation (void) const
- CX_DataFrame getLastPresentationInformation (void) const

19.26.1 Detailed Description

This class is a useful abstraction that presents slides (i.e. 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(&Disp); //Set up the slide presenter to use Disp as the display.

//Everything drawn after beginDrawingNextSlide and before the next call to it will be drawn to that slide.
slidePresenter.beginDrawingNextSlide(2000, "circle"); //We need to give a duration for the slide, plus an
       optional name.
ofBackground(50);
ofSetColor(ofColor::red);
ofCircle(Disp.getCenter(), 40);

//Begin drawing another slide.
slidePresenter.beginDrawingNextSlide(1000, "rectangle");
ofBackground(50);
ofSetColor(ofColor::green);
ofRect(Disp.getCenter() - ofPoint(100, 100), 200, 200);

//The duration of the last slide, as long as it is greater than 0, is ignored.
slidePresenter.beginDrawingNextSlide(1, "off");
ofBackground(50);
slidePresenter.endDrawingCurrentSlide(); //it is not necessary to call this, but the slide presenter will
       warn if you don't.

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!
    Input.pollEvents(); //It's also a good idea to poll for input events constantly.
}

//Or you could just call this function, which does the updating and input polling operations for you.
//slidePresenter.presentSlides();
```

19.26.2 Member Function Documentation

19.26.2.1 void CX::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 a few 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). If equal to the empty string ("" ; the default), the name will be set to "Slide N", where N is the slide number, indexed from 0.

Parameters

slide	The slide to append.

19.26.2.2 void CX::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 back 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. If equal to the empty string ("" ; the default), the name will be set to "Slide N", where N is the slide number, indexed from 0.

Note

See Visual Stimuli for more information about framebuffers.

One of the most tedious 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.h"

CX_SlidePresenter SlidePresenter;

//This is the function we want to use to draw a stimulus, but it takes two
//arguments. It needs to take 0 arguments in order to be used by the CX_SlidePresenter.
void drawRectangle(ofRectangle r, ofColor col) {
    ofBackground(0);
    ofSetColor(col);
    ofRect(r);
}

//One option is to use a functor to shift around where the arguments to the function come from. With a
//functor, like rectFunctor, below, you can define an operator() that takes no arguments directly, but gets
      its
//data from the position and color members of the structure. 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.
struct rectFunctor {
    ofRectangle position;
    ofColor color;
    void operator() (void) {
        drawRectangle(position, color);
    }
};

void runExperiment(void) {

    SlidePresenter.setup(&Disp);


    //Here we use the functor. We set up the values for position and color and then give the functor to
       `appendSlideFunction()`.
    rectFunctor rf;
    rf.position = ofRectangle(100, 100, 50, 80);
    rf.color = ofColor(0, 255, 0);
    SlidePresenter.appendSlideFunction(rf, 2000.0, "functor rect");


    //The other method is to use std::bind to "bake in" values for the arguments of drawRectangle. We will
```

```
    //set up the rectPos and rectColor values to bind to the arguments of drawRectangle.
    ofRectangle rectPos(100, 50, 100, 30);
    ofColor rectColor(255, 255, 0);

    //With the call to std::bind, we bake in the values rectPos and rectColor to their respective
       arguments,
    //resulting in a function that takes 0 arguments, which we pass into appendSlideFunction().
    SlidePresenter.appendSlideFunction(std::bind(drawRectangle, rectPos, rectColor), 2000.0, "bind rect");


    SlidePresenter.startSlidePresentation();
    while (SlidePresenter.isPresentingSlides()) {
        SlidePresenter.update();
    }
}
```

19.26.2.3 void CX::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. If equal to the empty string (" "; the default), the name will be set to "Slide N", where N is the slide number, indexed from 0.

```
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();
```

19.26.2.4 CX_SlidePresenter::PresentationErrorInfo CX::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 presentationErrors↩ SuccessfullyChecked member set to false and an error will be logged.

19.26.2.5 void CX::CX_SlidePresenter::clearSlides (void)

Clears (deletes) all of the slides contained in the slide presenter and stops presentation, if it was in progress.

19.26.2.6 void CX::CX_SlidePresenter::endDrawingCurrentSlide (void)

Ends drawing to the framebuffer of the slide that is currently being drawn to. See beginDrawingNextSlide().

19.26.2.7 std::vector< unsigned int > CX::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.

19.26.2.8 std::vector< CX_Millis > CX::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.

19.26.2.9 CX_DataFrame CX::CX_SlidePresenter::getLastPresentationInformation (void) const

This function produces a CX_DataFrame with the following information related to slide presentation for each slide (drawn directly from the CX_SlidePresenter struct used by each slide): name, intended and actual timing information, and copyToBackBufferCompleteTime. In addition, the slide index is given.

The column names are "index", "name", "copyToBackBufferCompleteTime", "actual.startTime", "actual.duration", "actual.startFrame", and "actual.frameCount". Plus, for the intended timings, replace "actual" with "intended" for the 4 intended timings columns.

Returns

The data frame.

19.26.2.10 std::string CX::CX_SlidePresenter::getLastPresentedSlideName (void) const

Returns

The name of the last slide to be presented. If no slides have been presented yet during the current slide presentation, returns "NO_SLIDE_PRESENTED".

19.26.2.11 CX_SlidePresenter::Slide & CX::CX_SlidePresenter::getSlideByName (std::string *name*)

Gets a reference to the slide with the given name, if found. If the named slide is not found, a `std::out_of_range` exception is thrown and an error is logged (although you will never see the log message unless the exception is caught).

Parameters

name	The name of the slide to get.

Returns

A reference to the named slide.

Note

Because the user supplies slide names, there is no guarantee that any given slide name will be unique. Because of this, this function simply returns a reference to the first slide for which the name matches.

19.26.2.12 std::vector< CX_SlidePresenter::Slide > & CX::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.

19.26.2.13 bool CX::CX_SlidePresenter::presentSlides (void)

Performs a "standard" slide presentation in a single function call as a convenience. This function calls startSlide↩ Presentation() to begin the presentation and then calls update() and CX::Instances::Input.pollEvents() continuously as long as isPresentingSlides() returns true.

Returns

`true` if the slide presentation completed successfully or `false` if the slide presentation could not be started.

19.26.2.14 std::string CX::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. Errors are marked with two asterisks (**).

19.26.2.15 bool CX::CX_SlidePresenter::setup (CX_Display * *display* = &CX::Instances::Disp)

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.

19.26.2.16 bool CX::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, `true` otherwise.

19.26.2.17 bool CX::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.

19.26.2.18 void CX::CX_SlidePresenter::update (void)

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

19.27. 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)
- bool deleteChannel (unsigned int channel)
- void setChannelData (unsigned int channel, const std::vector< float > &data)
- void reverse (void)
- void multiplySpeed (float speedMultiplier)

- void resample (float newSampleRate)
- float getSampleRate (void) const

  Returns the sample rate of the sound data stored in this CX_SoundBuffer.
- bool setChannelCount (unsigned int channels, bool average=true)
- int getChannelCount (void) const

  Returns the number of channels in the sound data stored in this CX_SoundBuffer.
- uint64_t getTotalSampleCount (void) const
- uint64_t getSampleFrameCount (void) const
- 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.

19.27.1 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 example for an introduction on how to use this class along with a CX_SoundBufferPlayer.

To record from a microphone into 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.

19.27.2 Member Function Documentation

19.27.2.1 void CX::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.

19.27.2.2 bool CX::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.

19.27.2.3 bool CX::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.

19.27.2.4 bool CX::CX_SoundBuffer::applyGain (float *decibels*, int *channel* = `-1`)

Apply gain 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 mute.
channel	The channel that the gain should be applied to. If channel is less than 0, the gain is applied to all channels.

19.27.2.5 void CX::CX_SoundBuffer::clear (void)

Clears all data stored in the sound buffer and returns it to an uninitialized state.

19.27.2.6 void CX::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.

19.27.2.7 bool CX::CX_SoundBuffer::deleteChannel (unsigned int *channel*)

Delete the specified channel from the data.

Parameters

channel	A 0-indexed index of the channel to delete.

Returns

`true` if there were no errors.

19.27.2.8 CX_Millis CX::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.

19.27.2.9 float CX::CX_SoundBuffer::getNegativePeak (void)

Finds the minimum amplitude in the sound buffer.

Returns

The minimum amplitude.

Note

Amplitudes are between -1 and 1, inclusive.

19.27.2.10 float CX::CX_SoundBuffer::getPositivePeak (void)

Finds the maximum amplitude in the sound buffer.

Returns

The maximum amplitude.

Note

Amplitudes are between -1 and 1, inclusive.

19.27.2.11 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!

19.27.2.12 uint64_t CX::CX_SoundBuffer::getSampleFrameCount (void) const

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.

19.27.2.13 uint64_t CX::CX_SoundBuffer::getTotalSampleCount (void) const [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.

19.27.2.14 bool CX::CX_SoundBuffer::isLoadedSuccessfully (void) [inline]

Checks to see if sound data has been successfully loaded into this CX_SoundBuffer from a file.

19.27.2.15 bool CX::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.

19.27.2.16 bool CX::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.

19.27.2.17 bool CX::CX_SoundBuffer::multiplyAmplitudeBy (float *amount*, int *channel* = `-1`)

Apply gain in terms of amplitude. The original value is simply multiplied by `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.

19.27.2.18 void CX::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().

19.27.2.19 void CX::CX_SoundBuffer::normalize (float *amount* = `1.0`)

Normalizes the contents of the sound buffer.

Parameters

amount	The peak with the greatest absolute amplitude will be set to `amount` and all other samples will be scaled proportionally so as to retain their relationship with the greatest absolute peak. Should be in the interval [0,1], unless clipping is desired. Should be positive unless you want to invert the waveform.

19.27.2.20 void CX::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.

19.27.2.21 void CX::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.

19.27.2.22 bool CX::CX_SoundBuffer::setChannelCount (unsigned int *newChannelCount*, bool *average* = `true`)

Sets the number of channels of the sound. Depending on the old number of channels (O) and the new number of channels (N), the conversion is performed in different ways. The cases in this list are evaluated in order an only 1 is executed, so a later case cannot be reached if an earlier case has already evaluated to true. When a case says anything about the average of existing data, it of course means the average on a sample-by-sample basis, not the average of all the samples.

- If `O == N`, nothing happens.
- If `O == 0`, the number of channels is just set to N. However, `O == 0`, that usually means that there is no sound data available, so changing the number of channels is kind of meaningless.
- If `N == 0`, the CX_SoundBuffer is cleared: all data is deleted. If you have no channels, you cannot have data in those channels.
- If `O == 1`, each of the `N` new channels is set equal to the value of the single old channel.
- If `N == 1`, and `average == true` the new channel is set equal to the average of the `O` old channels. If `average == false`, all but the first channel are removed.
- If `N > O`, the first `O` channels are preserved unchanged. If `average == true`, the `N - O` new channels are set to the average of the `O` old channels. If `average == false`, the `N - O` new channels are set to 0.
- If `N < O`, and `average == false`, the data from the `O - N` to-be-removed channels is discarded. If `average == true` the data from the `O - N` to-be-removed channels are averaged and added on to the `N` remaining channels. The averaging is done in an unusual way, so that the average intensitity of the kept channels is equal to the average intensity of the removed channels. An example to show why this is done: Assume that you have 3 channels – a, b, and c – and are switching to 2 channels, removing c. The average of c is just c, so when c is added to a and b, you now have c in 2 channels, whereas it was just in 1 channel originally: (a + c) + (b + c) = a + b + 2c. Thus, the final intensity of c is too high. What we want to do is scale c down by the number of channels it is being added to so that the total amount of c is equal both before and after changing the number of channels, so you divide c by the number of channels it is being added to (2). Now, (a + c/2) + (b + c/2) = a + b + c. However, there is another problem, which is that abs(a + c/2) can be greater than 1 even if the absolute value of both is no greater than 1. Now we need to scale each sample so that it is constrained to the proper range. We do that by taking the ratio of the new and old channels and multiplying the samples by that ratio. `N/O = 2/3` in the example. Now we have 2/3 $*$ (a + c/2) = 2a/3 + c/3, which is bounded between -1 and 1, as long as a and c are both bounded. Also, 2/3 $*$ [(a + c/2) + (b + c/2)] = 2a/3 + 2b/3 + 2c/3, so the ratios of the components of the original sound are constant in the final sound.

Parameters

newChannelCount	The number of channels the CX_SoundBuffer will have after the conversion.
average	If `true` and case `N < O` is reached, then the `O - N` old channels that are being removed will be averaged and this average will be added back into the `N` remaining channels. If `false` (the default), the channels that are being removed will actually be removed.

Returns

`true` if the conversion was successful, `false` if the attempted conversion is unsupported.

19.27.2.23 void CX::CX_SoundBuffer::setChannelData (unsigned int *channel*, const std::vector< float > & *data*)

Set the contents of a single channel from a vector of float data.

Parameters

channel	The channel to set the data for. If greater than any existing channel, new channels will be created so that the number of stored channels is equal to `channel + 1`. If you don't want a bunch of new empty channels, make sure you don't use a large channel number.

data	A vector of sound samples. These values must be in the interval [-1, 1], which is not checked for. See CX::Util::clamp() for one method of making sure your data are in the correct range. If the other channels in the CX_SoundBuffer are longer than `data`, `data` will be extended with zeroes. If the other channels in the CX_SoundBuffer are shorter than `data`, those channels will be extended with zeroes.

19.27.2.24 bool CX::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, which is checked for!
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.

19.27.2.25 void CX::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).

19.27.2.26 void CX::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.

19.27.2.27 bool CX::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

`true` for successfully saving the file. `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

19.28. 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 setup (CX_SoundStream *ss)
- bool play (void)
- bool startPlayingAt (CX_Millis experimentTime, CX_Millis offset)
- bool stop (void)
- bool isPlaying (void) const

  Check if the sound is currently playing.
- bool isQueuedToStart (void) const

  Check if the sound is queued to play.
- Configuration getConfiguration (void)
- bool setSoundBuffer (CX_SoundBuffer *sound)
- CX_SoundBuffer * getSoundBuffer (void)
- CX_SoundStream * getSoundStream (void)
- void seek (CX_Millis time)

19.28.1 Detailed Description

This class is used for playing CX_SoundBuffers. See the soundBuffer tutorial for an example of how to use this class.

19.28.2 Member Function Documentation

19.28.2.1 CX_SoundBufferPlayer::Configuration CX::CX_SoundBufferPlayer::getConfiguration (void)

Returns the configuration used for this CX_SoundBufferPlayer.

19.28.2.2 CX_SoundBuffer * CX::CX_SoundBufferPlayer::getSoundBuffer (void)

This function provides access to the CX_SoundBuffer that is in use by the CX_SoundBufferPlayer. If this function is called during playback of the sound buffer, a warning will be logged, but the buffer pointer will still be returned.

Returns

A pointer to the CX_SoundBuffer that is currently assigned to the CX_SoundBufferPlayer.

19.28.2.3 CX_SoundStream* CX::CX_SoundBufferPlayer::getSoundStream (void) [inline]

This function provides direct access to the CX_SoundStream used by the CX_SoundBufferPlayer.

19.28.2.4 bool CX::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.

19.28.2.5 void CX::CX_SoundBufferPlayer::seek (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.

Note

If used while the sound is playing, a warning will be logged but the function will still have its normal effect.

19.28.2.6 bool CX::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.

19.28.2.7 bool CX::CX_SoundBufferPlayer::setup (Configuration *config*)

Configures the CX_SoundBufferPlayer with the given configuration. A CX_SoundStream will be set up within the CX_SoundBufferPlayer and the sound stream will be started.

Parameters

config	The configuration to use for the CX_SoundBufferPlayer, which is really all about configuring the CX_SoundStream used internally by the CX_SoundBufferPlayer.

19.28.2.8 bool CX::CX_SoundBufferPlayer::setup (CX_SoundStream * *ss*)

Set up the sound buffer recorder from an existing CX_SoundStream. The CX_SoundStream is not started automatically. The CX_SoundStream must exist for the lifetime of the CX_SoundBufferPlayer.

Parameters

ss	A pointer to a fully configured CX_SoundStream.

Returns

`true` in all cases.

19.28.2.9 bool CX::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.

The start time is adjusted by an estimate of the latency of the sound stream. This is calculated as (N_b - 1) * S_b / SR, where N_b is the number of buffers, S_b is the size of the buffers (in sample frames), and SR is the sample rate, in sample frames per second.

In order for this function to have any meaningful effect, the request start time, plus any latency adjustments, must be in the future. If `experimentTime` plus `latencyOffset` minus the estimated stream latency is not in the future, the sound will start playing immediately and a warning will be logged.

Parameters

experimentTime	The desired experiment time at which the sound should start playing.
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 latency offset is in the past. `true` otherwise.

Note

See CX_SoundBufferPlayer::seek() for a way to choose the current time point within the sound.

19.28.2.10 bool CX::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`.

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

- CX_SoundBufferPlayer.h
- CX_SoundBufferPlayer.cpp

19.29. 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)
- bool setup (CX_SoundStream *ss)
- Configuration getConfiguration (void)
- CX_SoundStream * getSoundStream (void)

  This function provides direct access to the CX_SoundStream used by the CX_SoundBufferRecorder.
- void setSoundBuffer (CX_SoundBuffer *soundBuffer)
- CX_SoundBuffer * getSoundBuffer (void)
- void start (bool clearExistingData=false)
- void stop (void)

  Stop recording sound data.
- bool isRecording (void) const

  Returns `true` is currently recording.

19.29.1 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.start();
Clock.sleep(CX_Seconds(5));
recorder.stop();

//Write the recording to a file
recording.writeToFile("recording.wav");
```

19.29.2 Member Function Documentation

19.29.2.1 CX_SoundBufferRecorder::Configuration CX::CX_SoundBufferRecorder::getConfiguration (void)

Returns the configuration used for this CX_SoundBufferRecorder.

19.29.2.2 CX_SoundBuffer * CX::CX_SoundBufferRecorder::getSoundBuffer (void)

This function returns a pointer to the CX_SoundBuffer that is currently in use by the CX_SoundBufferRecorder.

19.29.2.3 void CX::CX_SoundBufferRecorder::setSoundBuffer (CX_SoundBuffer * *soundBuffer*)

This function associates a CX_SoundBuffer with the CX_SoundBufferRecorder. The CX_SoundBuffer will be recorded to when start() is called.

Parameters

soundBuffer	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.

19.29.2.4 bool CX::CX_SoundBufferRecorder::setup (CX_SoundBufferRecorder::Configuration & *config*)

This function sets up the CX_SoundStream that CX_SoundBufferRecorder uses to record audio data.

Parameters

config	A reference to a CX_SoundBufferRecorder::Configuration struct that will be used to configure an internally-stored CX_SoundStream.

Returns

`true` if configuration of the CX_SoundStream was successful, `false` otherwise.

19.29.2.5 bool CX::CX_SoundBufferRecorder::setup (CX_SoundStream * *ss*)

Set up the sound buffer recorder from an existing CX_SoundStream. The CX_SoundStream is not started automatically. The CX_SoundStream must remain in scope for the lifetime of the CX_SoundBufferRecorder.

Parameters

ss	A pointer to a fully configured CX_SoundStream.

Returns

`true` in all cases.

19.29.2.6 void CX::CX_SoundBufferRecorder::start (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.

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

- CX_SoundBufferRecorder.h
- CX_SoundBufferRecorder.cpp

19.30. 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 estimateTotalLatency (void) const
- CX_Millis estimateLatencyPerBuffer (void) const
- bool hasSwappedSinceLastCheck (void)
- void waitForBufferSwap (void)
- CX_Millis getLastSwapTime (void) const
- CX_Millis estimateNextSwapTime (void) const
- RtAudio * getRtAudioInstance (void) const

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 (std::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.

19.30.1 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

19.30.2 Member Function Documentation

19.30.2.1 bool CX::CX_SoundStream::closeStream (void)

Closes the sound stream. After the sound stream is closed, CX::CX_SoundStream::setup() must be called to reset the stream.

Returns

`false` if an error was encountered while closing the stream, `true` otherwise.

19.30.2.2 std::string CX::CX_SoundStream::convertApisToString (std::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.

19.30.2.3 std::vector< std::string > CX::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.

19.30.2.4 std::string CX::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.

19.30.2.5 RtAudio::Api CX::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, M↩ ACOSX_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.

19.30.2.6 CX_Millis CX::CX_SoundStream::estimateLatencyPerBuffer (void) const

This function calculates an estimate of the amount of latency per buffer full of data. It is calculated by S_b / SR, where S_b is the size of each buffer in sample frames and SR is the sample rate in samples per second.

Returns

Latency per buffer.

19.30.2.7 CX_Millis CX::CX_SoundStream::estimateNextSwapTime (void) const

Estimate the time at which the next buffer swap will occur. The estimate is based on the buffer size and sample rate, not empirical measurement.

Returns

The estimated time of next swap. This value can be compared with the result of CX::Instances::Clock.now().

19.30.2.8 CX_Millis CX::CX_SoundStream::estimateTotalLatency (void) const

This function gets an estimate of the total stream latency, calculated based on the buffer size, number of buffers, and sample rate. The calculation is N_b $*$ S_b / SR, where N_b is the number of buffers, S_b is the size of the buffers (in sample frames), and SR is the sample rate, in sample frames per second. This is a conservative upper bound on latency. Note that latency is not constant, but it depends on where in the buffer swapping process you start playing a sound. See Audio Input and Output for more information.

Returns

An estimate of the stream latency.

19.30.2.9 std::string CX::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`.

19.30.2.10 std::vector< std::string > CX::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.

19.30.2.11 std::vector< RtAudio::Api > CX::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.

19.30.2.12 const CX_SoundStream::Configuration& CX::CX_SoundStream::getConfiguration (void) const [inline]

Gets the configuration that was used on the last call to setup(). Because some of the configuration options are only suggestions, this function allows you to check what the actual used configuration was.

Returns

A const reference to the configuration struct.

19.30.2.13 std::vector< RtAudio::DeviceInfo > CX::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.

19.30.2.14 CX_Millis CX::CX_SoundStream::getLastSwapTime (void) const

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().

19.30.2.15 RtAudio * CX::CX_SoundStream::getRtAudioInstance (void) const

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/

19.30.2.16 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.

19.30.2.17 bool CX::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.

19.30.2.18 bool CX::CX_SoundStream::isStreamRunning (void) const

Check whether the sound stream is running.

Returns

false if the stream is not setup or not running or if RtAudio has not been initialized. Returns `true` if the stream is running.

19.30.2.19 std::string CX::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.

19.30.2.20 CX_SoundStream::Configuration CX::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. Note that there is a direct correspondence between the names of the keys in the file and the names of the members of a CX_SoundStream::Configuration struct.

Sample configuration file:

```
ss.api = WINDOWS_DS //See convertStringToApi() for valid API names.
ss.sampleRate = 44100
ss.bufferSize = 512
ss.inputChannels = 0
//ss.inputDeviceId //This is not used in this example because no input channels are used. It would take an
      integer.
ss.outputChannels = 2
ss.outputDeviceId = 0 //selects device 0. Can be a negative value, in which case the default output is
      selected.
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. All flags are supported.
//ss.streamOptions.priority is not used in this example. It would take a positive integer.
```

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.

19.30.2.21 bool CX::CX_SoundStream::setup (CX_SoundStream::Configuration & *config*)

Opens the sound stream with the specified configuration. See CX::CX_SoundStream::Configuration for the configuration options. If there were errors during configuration, error messages will be logged. If the configuration was successful, the sound stream will be started automatically.

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`, which is passed by reference, is updated based on the actually used settings. You can alternately check the configuration later using CX::CX_SoundStream::getConfiguration().

Returns

`true` if configuration appeared to be successful, `false` otherwise.

19.30.2.22 bool CX::CX_SoundStream::start (void)

Starts the sound stream. The stream must already be have been set up (see setup()).

Returns

`false` if the stream was not started, `true` if the stream was started or if it was already running.

19.30.2.23 bool CX::CX_SoundStream::stop (void)

Stops the stream. In order to restart the stream, CX::CX_SoundStream::start() must be called. If there is an error, a message will be logged.

Returns

`false` if there was an error, `true` otherwise.

19.30.2.24 void CX::CX_SoundStream::waitForBufferSwap (void)

Blocks until the next swap of the audio buffers. If the stream is not running, it returns immediately.

See also

See Blocking Code

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

- CX_SoundStream.h
- CX_SoundStream.cpp

19.31. CX::CX_Time_t< TimeUnit > Class Template Reference

`#include <CX_Time_t.h>`

Classes

- struct PartitionedTime

Public Member Functions

- PartitionedTime getPartitionedTime (void) const
- CX_Time_t (void)
- CX_Time_t (double t)
- CX_Time_t (int t)
- CX_Time_t (cxTick_t 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 fractions of an hour.
- double minutes (void) const

  Get the time stored by this CX_Time_t in minutes, including fractions of a minute.
- double seconds (void) const

  Get the time stored by this CX_Time_t in seconds, including fractions of a second.
- double millis (void) const

  Get the time stored by this CX_Time_t in milliseconds, including fractions of a millisecond.
- double micros (void) const

  Get the time stored by this CX_Time_t in microseconds, including fractions of a microsecond.
- cxTick_t 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

  Adds together two times.
- template<typename RT >
  CX_Time_t< TimeUnit > operator- (const CX_Time_t< RT > &rhs) const

  Subtracts two times.
- 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)

  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)

  Adds a CX_Time_t to an existing CX_Time_t.
- template<typename RT >
  CX_Time_t< TimeUnit > & operator-= (const CX_Time_t< RT > &rhs)

  Subtracts a CX_Time_t from an existing CX_Time_t.
- template<typename RT >
  bool operator< (const CX_Time_t< RT > &rhs) const

  Compares two times in the expected way.
- template<typename RT >
  bool operator<= (const CX_Time_t< RT > &rhs) const

  Compares two times in the expected way.

- template<typename RT >
  bool operator> (const CX_Time_t< RT > &rhs) const

  Compares two times in the expected way.
- template<typename RT >
  bool operator>= (const CX_Time_t< RT > &rhs) const

  Compares two times in the expected way.
- template<typename RT >
  bool operator== (const CX_Time_t< RT > &rhs) const

  Compares two times in the expected way.
- template<typename RT >
  bool operator!= (const CX_Time_t< RT > &rhs) const

  Compares two times in the expected way.

Static Public Member Functions

- static CX_Time_t< TimeUnit > min (void)

  Get the minimum time value that can be represented with this class.
- static CX_Time_t< TimeUnit > max (void)

  Get the maximum time value that can be represented with this class.
- static CX_Time_t< TimeUnit > standardDeviation (std::vector< CX_Time_t< TimeUnit >> vals)

19.31.1 Detailed Description

template<typename TimeUnit>class CX::CX_Time_t< TimeUnit >

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();
```

19.31.2 Constructor & Destructor Documentation

19.31.2.1 template<typename TimeUnit> CX::CX_Time_t< TimeUnit >::CX_Time_t (void) [inline]

Default constructor for CX_Time_t.

19.31.2.2 template<typename TimeUnit> CX::CX_Time_t< TimeUnit >::CX_Time_t (double *t*) [inline]

Constructs a CX_Time_t with the specified time value.

Parameters

t	A time value with units interpreted depending on the TimeUnit template argument for the instance of the class being constructed.

Example:

```
CX_Minutes quarterHour(15); //Interpreted as 15 minutes
CX_Seconds oneMinute(60); //Interpreted as 60 seconds
```

19.31.2.3 template<typename TimeUnit> CX::CX_Time_t< TimeUnit >::CX_Time_t (int *t*) [inline]

Constructs a CX_Time_t with the specified time value.

Parameters

t	A time value with units interpreted depending on the TimeUnit template argument for the instance of the class being constructed.

Example:

```
CX_Minutes quarterHour(15); //Interpreted as 15 minutes
CX_Seconds oneMinute(60); //Interpreted as 60 seconds
```

19.31.2.4 template<typename TimeUnit> CX::CX_Time_t< TimeUnit >::CX_Time_t (cxTick_t *t*) [inline]

Constructs a CX_Time_t with the specified time value.

Parameters

t	A time value with units interpreted depending on the TimeUnit template argument for the instance of the class being constructed.

Example:

```
CX_Minutes quarterHour(15); //Interpreted as 15 minutes
CX_Seconds oneMinute(60); //Interpreted as 60 seconds
```

19.31.2.5 template<typename TimeUnit> template<typename tArg > CX::CX_Time_t< TimeUnit >::CX_Time_t (const CX_Time_t< tArg > & *t*) [inline]

Constructs a CX_Time_t based on another instance of a CX_Time_t. If the TimeUnit template parameter has a different value for `t` than for the CX_Time_t being constructed, it does not change the amount of time stored. For example, if `t` is a CX_Time_t<std::ratio<60, 1> > (i.e. CX_Minutes) containing 1 minute, and the CX_Time_t that is constructed will contain 1 minute regardless of if that minute is thought of as 1/60 of an hour or 60,000,000 microseconds.

19.31.3 Member Function Documentation

19.31.3.1 template<typename TimeUnit> PartitionedTime CX::CX_Time_t< TimeUnit >::getPartitionedTime (void) const [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).

Returns

A PartitionedTime struct containing whole number amounts of the components of the time.

19.31.3.2 template<typename TimeUnit> template<typename RT > CX_Time_t<TimeUnit>& CX::CX_Time_t< TimeUnit >::operator+= (const CX_Time_t< RT > & *rhs*) [inline]

Adds a CX_Time_t to an existing CX_Time_t.

Parameters

rhs	The value to add.

19.31.3.3 template<typename TimeUnit> template<typename RT > CX_Time_t<TimeUnit>& CX::CX_Time_t< TimeUnit >::operator-= (const CX_Time_t< RT > & *rhs*) [inline]

Subtracts a CX_Time_t from an existing CX_Time_t.

Parameters

rhs	The value to subtract.

19.31.3.4 template<typename TimeUnit> static CX_Time_t<TimeUnit> CX::CX_Time_t< TimeUnit >::standardDeviation (std::vector< CX_Time_t< TimeUnit >> *vals*) [inline], [static]

This function calculates the sample standard deviation for a vector of time values.

19.31.3.5 template<typename TimeUnit> double CX::CX_Time_t< TimeUnit >::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

19.32. CX::CX_WindowConfiguration Struct Reference

`#include <CX_EntryPoint.h>`

Public Attributes

- ofWindowMode mode

  The mode of the window. One of ofWindowMode::OF_WINDOW, ofWindowMode::OF_FULLSCREEN, or ofWindowMode::OF_GAME_MODE.
- int width

  The width of the window, in pixels.
- int height

  The height of the window, in pixels.
- bool resizeable

  Whether or not the window can be resized by the user (i.e. by clicking and dragging the edges). Only works for oF 0.8.4 and newer.
- unsigned int msaaSampleCount

  See CX::Util::getMsaaSampleCount(). If this value is too high, some types of drawing take a really long time.
- ofPtr< ofBaseGLRenderer > desiredRenderer

  If you want to request a specific renderer, you can provide one here. If nothing is provided, a reasonable default is assumed.
- Private::CX_GLVersion desiredOpenGLVersion

  If you want to request a specific OpenGL version, you can provide this value. If nothing is provided, the newest OpenGL version available is used.
- std::string windowTitle
- std::function< void(void)> preOpeningUserFunction

19.32.1 Detailed Description

This structure is used to configure windows opened with CX::reopenWindow().

19.32.2 Member Data Documentation

19.32.2.1 std::function<void(void)> CX::CX_WindowConfiguration::preOpeningUserFunction

A user-supplied function that will be called just before the GLFW window is opened. This allows you to set window hints just before the window is opened. This only works if you are using oF version 0.8.4.

19.32.2.2 std::string CX::CX_WindowConfiguration::windowTitle

A title for the window that is opened.

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

- CX_EntryPoint.h

19.33. CX::Synth::Envelope Class Reference

`#include <CX_Synth.h>`

Inherits CX::Synth::ModuleBase.

Public Member Functions

- double getNextSample (void) override
- void attack (void)

  Trigger the attack of the Envelope.
- void release (void)

  Trigger the release of the Envelope.

Public Attributes

- ModuleParameter gateInput
- ModuleParameter a

  The number of seconds it takes, following the attack, for the level to rise from 0 to 1. Should be non-negative.
- ModuleParameter d

  The number of seconds it takes, once reaching the attack peak, to fall to `s`. Should be non-negative.
- ModuleParameter s

  The level at which the envelope sustains while waiting for the release. Should be between 0 and 1.
- ModuleParameter r

  The number of seconds it takes, following the release, for the level to fall to 0 from `s`. Should be non-negative.

Additional Inherited Members

19.33.1 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.

19.33.2 Member Function Documentation

19.33.2.1 double CX::Synth::Envelope::getNextSample (void) [override], [virtual]

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

Returns

The value of the next sample from the module.

Reimplemented from CX::Synth::ModuleBase.

19.33.3 Member Data Documentation

19.33.3.1 ModuleParameter CX::Synth::Envelope::gateInput

This parameter can be used by another module as a way to signal the Envelope. When the output of the module inputting to `gateInput` changes to 1.0, the attack of the envolpe is triggered. When it changes to 0, the release is triggered.

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

- CX_Synth.h
- CX_Synth.cpp

19.34. CX::Draw::Gabor::Envelope Struct Reference

`#include <CX_Gabor.h>`

Static Public Attributes

- static std::string none = "return 1;"

  Does nothing to affect the wave pattern.
- static std::string circle = "if (d <= cp) return 1; \n return 0;"

  Creates a circle, clipped at a radius set by the control parameter.
- static std::string linear = "if (d > cp) return 0; \n return 1 - (d / cp);"

  Creates linearly decreasing values up to a radius set by the control parameter.
- static std::string cosine = "if (d >= cp) return 0;\n return (cos(d / cp * PI) + 1) / 2;"

  Creates values that decrease with a cosine shape as distance increases, depending on the control parameter for a radius.
- static std::string gaussian = "return exp(-(d * d) / (2 * (cp * cp)));"

  Creates values that decrease with a gaussian shape as distance increases, where the control parameter sets the standard deviation.

19.34.1 Detailed Description

This struct contains several functions that used for calculating the envelope containing the gabor patch (e.g. a fall-off away from the center).

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

- CX_Gabor.h
- CX_Gabor.cpp

19.35. CX::Draw::EnvelopeProperties Struct Reference

`#include <CX_Gabor.h>`

Static Public Member Functions

- static float none (float d, float cp)
- static float circle (float d, float cp)
- static float linear (float d, float cp)
- static float cosine (float d, float cp)
- static float gaussian (float d, float cp)

Public Attributes

- float width

  The width of the envelope, in pixels.
- float height

  The height of the envelope, in pixels.
- std::function< float(float, float)> envelopeFunction
- float controlParameter

19.35.1 Detailed Description

This struct controls the properties of an envelope created with CX::Draw::envelopeToPixels(). The type of the envelope is specified with the envelopeFunction member and depending on the function that is used, controlParameter can quantitatively affect the envelope.

19.35.2 Member Function Documentation

19.35.2.1 float CX::Draw::EnvelopeProperties::circle (float *d*, float *cp*) [static]

Creates a hard clipped circle.

Parameters

d	The distance.
cp	The control parameter, interpreted as a radius.

Returns

1 if `d <= cp`, 0 otherwise.

19.35.2.2 float CX::Draw::EnvelopeProperties::cosine (float *d,* float *cp*) [static]

Creates values that decrease with a cosine shape as `d` increases.

Parameters

d	The distance.
cp	The control parameter, interpreted as a radius.

Returns

A value that drops off with a cosine shape as d increases up to cp, beyond which this returns 0.

19.35.2.3 float CX::Draw::EnvelopeProperties::gaussian (float *d,* float *cp*) [static]

Creates values that decrease with a gaussian shape as `d` increases.

Parameters

d	The distance.
cp	The control parameter, interpreted as the standard deviation of a gaussian distribution.

Returns

A value from a gaussian kernel for deviate `d` with mean 0 and standard deviation `cp`.

19.35.2.4 float CX::Draw::EnvelopeProperties::linear (float *d,* float *cp*) [static]

Creates linearly decreasing values up to a radius set by `cp`.

Parameters

d	The distance.
cp	The control parameter, interpreted as a radius.

Returns

`1 - (d / cp)` if `d <= cp`, 0 otherwise.

19.35.2.5 float CX::Draw::EnvelopeProperties::none (float *d,* float *cp*) [static]

Does nothing to affect the wave pattern.

Parameters

d	The distance.
cp	The control parameter.

Returns

1, regardless of the inputs.

19.35.3 Member Data Documentation

19.35.3.1 float CX::Draw::EnvelopeProperties::controlParameter

A parameter that controls the envelope in different ways, depending on the envelope function. This is passed as the second argument to envelopeFunction() each time it is called.

19.35.3.2 std::function<float(float, float)> CX::Draw::EnvelopeProperties::envelopeFunction

A function used to generate the envelope. Can be one of the static functions of this struct or some user defined function. The first argument it takes is the distance in pixels from the center of the envelope (depend on the width and height). The second argument is the controlParameter, which is set by the user. The function should return a value in the interval [0,1].

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

- CX_Gabor.h
- CX_Gabor.cpp

19.36. CX::CX_Joystick::Event Struct Reference

`#include <CX_Joystick.h>`

Public Attributes

- int buttonIndex

  If `type` is BUTTON_PRESS or BUTTON_RELEASE, this contains the index of the button that was changed.
- unsigned char buttonState

  If `type` is BUTTON_PRESS or BUTTON_RELEASE, this contains the current state of the button.
- int axisIndex

  If `type` is AXIS_POSITION_CHANGE, this contains the index of the axis which changed.
- float axisPosition

  If `type` is AXIS_POSITION_CHANGE, this contains the amount by which the axis changed.
- CX_Millis time

  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 `time`, which represents the difference between the time at which this event was timestamped by CX and the last time that events were checked for.
- EventType type

  The type of the event, from the CX_Joystick::EventType enum.

19.36.1 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.

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

- CX_Joystick.h

19.37. CX::CX_Keyboard::Event Struct Reference

`#include <CX_Keyboard.h>`

Public Attributes

- int key
- CX_Millis time

  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 `time`, which represents the difference between the time at which this event was timestamped by CX and the last time that events were checked for.
- EventType type

  The type of the event: press, release, or key repeat.
- Keycodes codes

  Alternative representations of the pressed key.

19.37.1 Detailed Description

This struct contains the results of a keyboard event, whether it be a key press or release, or key repeat.

The primary representation of the key that was pressed is given by key. Four alternative representations are given in the codes struct.

19.37.2 Member Data Documentation

19.37.2.1 int CX::CX_Keyboard::Event::key

The key that was pressed. This can be compared with character literals for most standard keys. For example, you could use (myKeyEvent.key == 'E') to test if the key was the E key. This does not depend on modifier keys: You always check for uppercase letters. For the number row keys, you check for the number, not the special character that is produced when shift is held, etc.

For special keys, this value can be compared to the values in the CX::Keycode enum.

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

- CX_Keyboard.h

19.38. CX::CX_Mouse::Event Struct Reference

`#include <CX_Mouse.h>`

Public Attributes

- int button

  The relevant mouse button if the event `type` is PRESSED, RELEASED, or DRAGGED. Can be compared with elements of enum CX_Mouse::Buttons to find out about the named buttons.
- float x

  The x position of the cursor at the time of the event, or the change in the x-axis scroll if the `type` is EventType::S↩ CROLLED.
- float y

  The y position of the cursor at the time of the event, or the change in the y-axis scroll if the `type` is EventType::S↩ CROLLED.
- CX_Millis time

  The time at which the event was registered. Can be compared to the result of CX::Clock::now().
- CX_Millis uncertainty

  The uncertainty in `time`, which represents the difference between the time at which this event was timestamped by CX and the last time that events were checked for.
- EventType type

  The type of the event.

19.38.1 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.

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

- CX_Mouse.h

19.39. CX::Synth::Filter Class Reference

`#include <CX_Synth.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

19.39.1 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.

19.39.2 Member Enumeration Documentation

19.39.2.1 enum CX::Synth::Filter::FilterType

The type of filter to use.

19.39.3 Member Function Documentation

19.39.3.1 double CX::Synth::Filter::getNextSample (void) [override], [virtual]

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

Returns

The value of the next sample from the module.

Reimplemented from CX::Synth::ModuleBase.

19.39.4 Member Data Documentation

19.39.4.1 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 `bandwidth` to 100. Of course, past those frequencies the attenuation continues. Larger values result in a less pointy band.

19.39.4.2 ModuleParameter CX::Synth::Filter::cutoff

The cutoff frequency of the filter.

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

- CX_Synth.h
- CX_Synth.cpp

19.40. 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.
- std::string currentSlideName

  The name of the slide that is currently being presented.

19.40.1 Detailed Description

The final slide user function takes a reference to a struct of this type. See CX_SlidePresenter::Configuration::finalSlideCallback for more information.

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

- CX_SlidePresenter.h

19.41. CX::Synth::FIRFilter Class Reference

`#include <CX_Synth.h>`

Inherits CX::Synth::ModuleBase.

Public Types

- enum FilterType {
  LOW_PASS, **HIGH_PASS**, **BAND_PASS**, **BAND_STOP**,
  FilterType::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)
- void setBandCutoffs (double lower, double upper)
- double getNextSample (void)

Additional Inherited Members

19.41.1 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.

19.41.2 Member Enumeration Documentation

19.41.2.1 enum CX::Synth::FIRFilter::FilterType [strong]

The type of filter to use.

Enumerator

USER_DEFINED Should not be used directly.

19.41.2.2 enum CX::Synth::FIRFilter::WindowType

The type of windowing function to apply after convolution.

19.41.3 Member Function Documentation

19.41.3.1 double CX::Synth::FIRFilter::getNextSample (void) [virtual]

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

Returns

The value of the next sample from the module.

Reimplemented from CX::Synth::ModuleBase.

19.41.3.2 void CX::Synth::FIRFilter::setBandCutoffs (double *lower*, double *upper*)

Sets the upper and lower cutoffs for a band filter mode (i.e. BAND_PASS or BAND_STOP).

Parameters

lower	The lower end of the band (Hz).
upper	The upper end of the band (Hz).

19.41.3.3 void CX::Synth::FIRFilter::setCutoff (double *cutoff*)

If using either FilterType::LOW_PASS or FilterType::HIGH_PASS, this function allows you to change the cutoff frequency for the filter. This causes the filter coefficients to be recalculated.

Parameters

cutoff	The cutoff frequency, in Hz.

19.41.3.4 void CX::Synth::FIRFilter::setup (FilterType *filterType,* unsigned int *coefficientCount*)

Set up the FIRFilter with the given filter type and number of coefficients to use.

Parameters

filterType	Should be a type of filter other than FIRFilter::FilterType::FIR_USER_DEFINED. If you want to define your own filter type, use FIRFilter::setup(std::vector<double>) instead.
coefficientCount	The number of coefficients sets the length of time, in samples, that the filter will produce a non-zero output following an impulse. In other words, the filter operates on `coefficient↩ Count` samples at a time to produce each output sample.

19.41.3.5 void CX::Synth::FIRFilter::setup (std::vector< double > *coefficients*)

You can use this function to supply your own filter coefficients, which allows a great deal of flexibility in the use of the FIRFilter. See the fir1 and fir2 functions from the "signal" package for R for a way to design your own filter.

Parameters

coefficients	The filter coefficients to use.

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

- CX_Synth.h
- CX_Synth.cpp

19.42. CX::Synth::FunctionModule Class Reference

`#include <CX_Synth.h>`

Inherits CX::Synth::ModuleBase.

Public Member Functions

- double getNextSample (void) override

Public Attributes

- std::function< double(double)> f

  The user function, which will be called each time getNextSample() is called.

Additional Inherited Members

19.42.1 Detailed Description

This class is an easy way to apply an arbitrary function to modular synth data. The user function, `f`, takes a `double` and returns a `double`. Each time getNextSample() is called, the next sample from the input to this module will be taken and passed to `f`, and the the result of `f` will be returned.

19.42.2 Member Function Documentation

19.42.2.1 double CX::Synth::FunctionModule::getNextSample (void) `[inline], [override], [virtual]`

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

Returns

The value of the next sample from the module.

Reimplemented from CX::Synth::ModuleBase.

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

- CX_Synth.h

19.43. CX::Draw::Gabor Class Reference

`#include <CX_Gabor.h>`

Classes

- struct Envelope
- struct Wave

Public Member Functions

- Gabor (std::string waveFunction, std::string envelopeFunction)
- void setup (std::string waveFunction, std::string envelopeFunction)
- void draw (void)
- void draw (float newX, float newY)
- void draw (ofPoint newCenter)
- void draw (ofPoint newCenter, float fboHeight)
- ofShader & getShader (void)

  Get a reference to the ofShader used by this class. Use this only if you want to do advanced things directly with the shader.

Public Attributes

- ofPoint center

  The center of the gabor.
- float radius
- float fboHeight

  If drawing the gabor into a framebuffer that has a different height than the main window, use this to set the height of that framebuffer. If this is less than 0, the height of the current framebuffer will be inferred to be the height of the main window.
- ofFloatColor color1

  The first color used in the waveforms. There is no meaning to order to the colors.
- ofFloatColor color2

  The second color used in the waveforms. There is no meaning to order to the colors.
- struct {
  float angle
  The angle at which the waves are oriented, in degrees.
  float wavelength
  The distance, in pixels, between the center of each wave within the pattern.
  float phase
  The phase shift of the waves, in degrees.
  } wave

  Settings for the waveforms.

- struct {
  float controlParameter
  Control parameter for the envelope generating function.
  } envelope

  Settings for the envelope.

19.43.1 Detailed Description

This class draws gabor patches using hardware acceleration to speed up the process. Compared to the loose functions, like CX::Draw::gabor(), this class is preferable from a speed perspective, but it is slightly harder to use and not as flexible. You use it by calling the setup function to specify some basic information about the gabor, setting a number of data members of the class to certain values, and calling the draw function. For example:

```
#include "CX.h"

void runExperiment(void) {
    Draw::Gabor gabor; //Make an instance of the Gabor class.

    //Do basic setup for the gabor by setting the wave and envelope functions.
    gabor.setup(Draw::Gabor::Wave::sine,
      Draw::Gabor::Envelope::gaussian);

    gabor.envelope.controlParameter = 50; //Set the control parameter for the envelope (in this case,
       standard deviation).

    gabor.wave.wavelength = 30; //Set the wavelength of the waves, in pixels.
    gabor.wave.angle = 30; //Set the angle of the waves.

    gabor.color1 = ofColor::green; //Choose the two colors to alternate between.
    gabor.color2 = ofColor::red;

    Disp.beginDrawingToBackBuffer();

    ofBackground(127);

    gabor.draw(Disp.getCenter());

    Disp.endDrawingToBackBuffer();
    Disp.swapBuffers();

    Input.Keyboard.waitForKeypress(-1);
}
```

Advanced users: The Gabor class is meant to be somewhat extensible, so that you can add your own wave and envelope functions. To do so, you will need to write a function body that calculates wave amplitudes and envelope amounts using the OpenGL Shading Language (GLSL). These functions will be called for every pixel that is drawn and will be given various pieces of data that will help them calculate the resulting value.
The waveform function has the following type signature:

```
float waveformFunction(in float wp)
```

where `wp` is the current position, in the interval [0,1), for the waveform that you are calculating the amplitide for. The return value is the amplitude of the wave at `wp` and should be in the interval [0,1]. An example of a function body that you could use to generate sine waves is

```
return (sin(wp * 6.283185307179586232) + 1) / 2;
```

where the returned value is scaled to be in the interval [0,1] instead of [-1,1].
The envelope function has the following type signature:

```
float envelopeFunction(in float d, in float cp)
```

where `d` is the distance from the center of the gabor patch and `cp` is the control parameter, which the user can set by modifying Gabor::envelope::controlParameter. The function returns a value in the interval [0,1] that is interpreted as the alpha for the color that is set for the current pixel. For example, for a circular envelope, the alpha is fully opaque for pixels within the radius and fully transparent for pixels outside of the radius, so a function body might be:

```
if (d <= cp) return 1;
return 0;
```

Due to how GLSL works, these function bodies can be written as strings in C++ source code and passed to the GLSL compiler as strings. In this case, you just need to pass the function bodies to Gabor::setup().

19.43.2 Constructor & Destructor Documentation

19.43.2.1 CX::Draw::Gabor::Gabor (std::string *waveFunction,* std::string *envelopeFunction*)

Convenience constructor which sets up the class while constructing it.

19.43.3 Member Function Documentation

19.43.3.1 void CX::Draw::Gabor::draw (void)

Draw the gabor given the current settings

19.43.3.2 void CX::Draw::Gabor::draw (float *newX,* float *newY*)

Draw the gabor, setting a new location for it.

Parameters

newX	The new x coordinate of the center of the gabor.
newY	The new y coordinate of the center of the gabor.

19.43.3.3 void CX::Draw::Gabor::draw (ofPoint *newCenter*)

Draw the gabor, setting a new location for it.

Parameters

newCenter	The new center of the gabor.

19.43.3.4 void CX::Draw::Gabor::draw (ofPoint *newCenter,* float *newFboHeight*)

Draw the gabor, setting a new location for it and new fboHeight.

Parameters

newCenter	The new center of the gabor.
newFboHeight	The new CX::Draw::Gabor::fboHeight.

19.43.3.5 void CX::Draw::Gabor::setup (std::string *waveFunction,* std::string *envelopeFunction*)

Set up the gabor to use certain wave and envelope functions. This is a special setup step because changing the functions changes the source code of the fragment shader used to draw the gabor, so it has to be recompiled. This is a potentially blocking function.

Parameters

waveFunction	A function to use to calculate the mixing between color1 and color2. Most users should use a value from Gabor::Wave. Advanced users can write their own function using GLSL.
envelope↩ Function	A function to use to calculate the envelope giving the falloff of the gabor from the center of the pattern. Most users should use a value from Gabor::Envelope. Advanced users can write their own function using GLSL.

19.43.4 Member Data Documentation

19.43.4.1 float CX::Draw::Gabor::radius

The maximum radius of the gabor. The should generally be larger than the (visible) edge of the envelope that is used. If you have an envelope that should have a smooth (or blended) edge but are seeing a hard-clipped edge, you should try increasing the radius.

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

- CX_Gabor.h
- CX_Gabor.cpp

19.44. CX::Draw::GaborProperties Struct Reference

`#include <CX_Gabor.h>`

Public Attributes

- float width

  The width of the gabor patch.
- float height

  The height of the gabor patch.
- ofColor color1

  The first color.
- ofColor color2

  The second color.
- WaveformProperties wave

  Parameters controlling the waveform used to create the gabor patch.
- EnvelopeProperties envelope

  Parameters controlling the envelope used to contain the waves.

19.44.1 Detailed Description

Draws a gabor patch with two colors that are used for the peaks and troughs of the waves plus an envelope that smooths the edges of the patch. The waves are specified with the wave argument and the envelope with the envelope argument.

The width and height of the wave and envelope do not need to be directly specified as their values are taken from the width and height members of this struct.

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

- CX_Gabor.h

19.45. CX::Synth::GenericOutput Class Reference

`#include <CX_Synth.h>`

Inherits CX::Synth::ModuleBase.

Public Member Functions

- double getNextSample (void) override

Additional Inherited Members

19.45.1 Detailed Description

This class is used within output modules that actually output data. This class serves as an endpoint for data that is then retrieved by the class containing the GenericOutput. See, for example, the StereoStreamOutput class. This class does nothing useful on its own (getNextSample() is just a passthrough).

19.45.2 Member Function Documentation

19.45.2.1 double CX::Synth::GenericOutput::getNextSample (void) [inline], [override], [virtual]

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

Returns

The value of the next sample from the module.

Reimplemented from CX::Synth::ModuleBase.

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

- CX_Synth.h

19.46. 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.

19.46.1 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

19.47. CX::CX_DataFrame::InputOptions Struct Reference

`#include <CX_DataFrame.h>`

Inherits CX::CX_DataFrame::IoOptions.

Additional Inherited Members

19.47.1 Detailed Description

Options for the format of data that are input to a CX_DataFrame.

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

- CX_DataFrame.h

19.48. CX::CX_DataFrame::IoOptions Class Reference

`#include <CX_DataFrame.h>`

Inherited by CX::CX_DataFrame::InputOptions, and CX::CX_DataFrame::OutputOptions.

Public Attributes

- std::string cellDelimiter

  The delimiter between cells of the data frame. Defaults to tab ("\t").
- std::string vectorEncloser

  The string which surrounds a vector of data (i.e. one cell of data, which happens to be a vector). Defaults to double quote ("\"").
- std::string vectorElementDelimiter

  The string which delimits elements of a vector. Defaults to semicolon (";").

19.48.1 Detailed Description

Options for the format of files that are output to or input from a CX_DataFrame.

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

- CX_DataFrame.h

19.49. CX::CX_Keyboard::Keycodes Struct Reference

`#include <CX_Keyboard.h>`

Public Member Functions

- Keycodes (int oF_, int glfw_, int scancode_, unsigned int codepoint_)

Public Attributes

- int oF
- int glfw
- int scancode
- unsigned int codepoint

19.49.1 Detailed Description

This struct contains four alternative representations of the pressed key.

- oF The openFrameworks key representation. This depends on modifier keys.
- glfw The GLFW keycode. This does not depend on modifier keys.
- scancode System-specific scancode. This is not very easy to use, but does not depend on modifier keys.
- codepoint The locale-specific unicode code point for the key. This depends on modifier keys.

19.49.2 Constructor & Destructor Documentation

19.49.2.1 CX::CX_Keyboard::Keycodes::Keycodes (int *oF_*, int *glfw_*, int *scancode_*, unsigned int *codepoint_*) [inline]

Fancy constructor for the struct.

19.49.3 Member Data Documentation

19.49.3.1 unsigned int CX::CX_Keyboard::Keycodes::codepoint

The locale-specific unicode codepoint for the key. This is the most like the natural language value of the key, so it naturally depends on modifier keys.

19.49.3.2 int CX::CX_Keyboard::Keycodes::glfw

The GLFW keycode. These can be compared to the constants defined here: http://www.glfw.org/docs/latest/group__keys.html. This value does not depend on modifier keys. Like oF, the value of this can be compared with character literals for a lot of the standard keys (letters are uppercase).

19.49.3.3 int CX::CX_Keyboard::Keycodes::oF

The openFrameworks keycode. The value of this can be compared with character literals for many of the standard keyboard keys. The value depends on the modifier keys.

For special keys, this can be compared with the key constant values defined in ofConstants.h (e.g. `OF_KEY_ESC`).

For modifier keys, you can check for a specific key using, for example, the constants `OF_KEY_RIGHT_CONTROL` or `OF_KEY_LEFT_CONTROL`. You can alternately check to see if this 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:

```
bool ctrlHeld = (myKeyEvent.key & OF_KEY_CONTROL) == OF_KEY_CONTROL;
```

This works the same way for all of the modifier keys.

19.49.3.4 int CX::CX_Keyboard::Keycodes::scancode

System-specific scancode. These are not very easy to use, but do not depend on modifier keys.

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

- CX_Keyboard.h

19.50. CX::Algo::LatinSquare Class Reference

`#include <CX_Algorithm.h>`

Public Member Functions

- LatinSquare (void)

  Construct a LatinSquare with no contents.
- LatinSquare (unsigned int dimensions)
- void generate (unsigned int dimensions)

  Construct a LatinSquare with no contents.
- 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

  The Latin square.

19.50.1 Detailed Description

This class provides a way to work with Latin squares in a relatively easy way.

```
Algo::LatinSquare ls(4); //Construct a standard 4x4 LatinSquare.
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;
}
```

19.50.2 Constructor & Destructor Documentation

19.50.2.1 CX::Algo::LatinSquare::LatinSquare (unsigned int *dimensions*)

Construct a LatinSquare with the given dimensions. The generated square is the 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.

19.50.3 Member Function Documentation

19.50.3.1 bool CX::Algo::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.

19.50.3.2 bool CX::Algo::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.

19.50.3.3 unsigned int CX::Algo::LatinSquare::columns (void) const

Returns the number of columns.

19.50.3.4 void CX::Algo::LatinSquare::generate (unsigned int *dimensions*)

Construct a LatinSquare with no contents.

Note

This deletes any previous contents of the latin square.

19.50.3.5 std::vector< unsigned int > CX::Algo::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.

19.50.3.6 std::vector< unsigned int > CX::Algo::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.

19.50.3.7 LatinSquare & CX::Algo::LatinSquare::operator+= (unsigned int *value*)

Adds the given value to all of the values in the latin square.

19.50.3.8 std::string CX::Algo::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.

19.50.3.9 void CX::Algo::LatinSquare::reorderDown (void)

This function moves all of the rows down one place, then moves the bottommost row to the top.

19.50.3.10 void CX::Algo::LatinSquare::reorderLeft (void)

This function shifts the columns to the left and the first column is moved to be the last column.

19.50.3.11 void CX::Algo::LatinSquare::reorderRight (void)

This function shifts the columns to the right and the last column is moved to be the first column.

19.50.3.12 void CX::Algo::LatinSquare::reorderUp (void)

This function moves all of the rows up one place, then moves the topmost row to the bottom.

19.50.3.13 void CX::Algo::LatinSquare::reverseColumns (void)

Reverses the order of the columns in the latin square.

19.50.3.14 void CX::Algo::LatinSquare::reverseRows (void)

Reverses the order of the rows in the latin square.

19.50.3.15 unsigned int CX::Algo::LatinSquare::rows (void) const

Returns the number of rows.

19.50.3.16 void CX::Algo::LatinSquare::swapColumns (unsigned int *c1*, unsigned int *c2*)

Swap the given columns. If either column is out of range, this function has no effect.

19.50.3.17 void CX::Algo::LatinSquare::swapRows (unsigned int *r1*, unsigned int *r2*)

Swap the given rows. If either row is out of range, this function has no effect.

19.50.3.18 bool CX::Algo::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

19.51. CX::CX_Logger::MessageFlushData Struct Reference

`#include <CX_Logger.h>`

Public Member Functions

- MessageFlushData (std::string message_, CX_Logger::Level level_, std::string module_)

Public Attributes

- std::string message

  A string containing the logged message.
- Level level

  The log level of the message.
- std::string module

  The module associated with the message, usually which created the message.

19.51.1 Detailed Description

If a user function is listening for flush callbacks by using setMessageFlushCallback(), each time the user function is called, it gets a reference to an instance of this struct with all the information filled in.

19.51.2 Constructor & Destructor Documentation

19.51.2.1 CX::CX_Logger::MessageFlushData::MessageFlushData (std::string *message_*, CX_Logger::Level *level_*, std::string *module_*) [inline]

Convenience constructor which constructs an instance of the struct with the provided values.

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

- CX_Logger.h

19.52. CX::Synth::Mixer Class Reference

`#include <CX_Synth.h>`

Inherits CX::Synth::ModuleBase.

Public Member Functions

- double getNextSample (void) override

Additional Inherited Members

19.52.1 Detailed Description

This class mixes together a number of inputs. It does no mixing in the usual sense of setting levels of the inputs, which is done with Multipliers. 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.

19.52.2 Member Function Documentation

19.52.2.1 double CX::Synth::Mixer::getNextSample (void) [override], [virtual]

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

Returns

The value of the next sample from the module.

Reimplemented from CX::Synth::ModuleBase.

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

- CX_Synth.h
- CX_Synth.cpp

19.53. CX::Synth::ModuleBase Class Reference

`#include <CX_Synth.h>`

Inherited by CX::Synth::Adder, CX::Synth::AdditiveSynth, CX::Synth::Clamper, CX::Synth::Envelope, CX::Synth::Filter, CX::Synth::FIRFilter, CX::Synth::FunctionModule, CX::Synth::GenericOutput, CX::Synth::Mixer, CX::Synth::Multiplier, CX::Synth::Oscillator, CX::Synth::RingModulator, CX::Synth::SoundBufferInput, CX::Synth::SoundBufferOutput, CX::Synth::Splitter, CX::Synth::StreamInput, 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)
- void disconnect (void)

  Fully disconnect a module from all inputs and outputs.

Protected Member Functions

- void _dataSet (ModuleBase *caller)
- void _setDataIfNotSet (ModuleBase *target)
- void _registerParameter (ModuleParameter *p)
- void _assignInput (ModuleBase *in)
- void _assignOutput (ModuleBase *out)
- virtual void _dataSetEvent (void)
- virtual unsigned int _maxInputs (void)
- virtual unsigned int _maxOutputs (void)
- virtual void _inputAssignedEvent (ModuleBase *in)
- virtual void _outputAssignedEvent (ModuleBase *out)

Protected Attributes

- std::vector< ModuleBase * > _inputs

  The inputs to this module.
- std::vector< ModuleBase * > _outputs

  The outputs from this module.
- std::vector< ModuleParameter * > _parameters

  The ModuleParameters of this module.
- ModuleControlData_t * _data

  The data for this module.

Friends

- ModuleBase & operator>> (ModuleBase &l, ModuleBase &r)
- void operator>> (ModuleBase &l, ModuleParameter &r)

19.53.1 Detailed Description

All modules of the modular synth inherit from this class.

19.53.2 Member Function Documentation

19.53.2.1 void CX::Synth::ModuleBase::_assignInput (ModuleBase * *in*) [protected]

Assigns a module as an input to this module. This is not a reciprocal operation.

Parameters

in	The module to assign as an input.

19.53.2.2 void CX::Synth::ModuleBase::_assignOutput (ModuleBase * *out*) [protected]

Assigns a module as an output from this module. This is not a reciprocal operation.

Parameters

out	The module to asssign as an output.

19.53.2.3 void CX::Synth::ModuleBase::_dataSet (ModuleBase * *caller*) [protected]

This function is called on a module after the data for that module has been set.

Parameters

caller	The module that set the data for this module.

19.53.2.4 void CX::Synth::ModuleBase::_dataSetEvent (void) [protected], [virtual]

This function is a sort of callback that is called whenever _dataSet is called. Within this function, you should do things for your module that depend on the new data values. You should not attempt to propagate the data values to inputs, outputs, or parameters: that is all done for you.

19.53.2.5 void CX::Synth::ModuleBase::_inputAssignedEvent (ModuleBase * *in*) [protected], [virtual]

Does nothing by default, but can be overridden by inheriting classes.

19.53.2.6 unsigned int CX::Synth::ModuleBase::_maxInputs (void) [protected], [virtual]

Returns the maximum number of inputs to this module.

19.53.2.7 unsigned int CX::Synth::ModuleBase::_maxOutputs (void) [protected], [virtual]

Returns the maximum numer of outputs from this module.

19.53.2.8 void CX::Synth::ModuleBase::_outputAssignedEvent (ModuleBase * *out*) [protected], [virtual]

Does nothing by default, but can be overridden by inheriting classes.

19.53.2.9 void CX::Synth::ModuleBase::_registerParameter (ModuleParameter * *p*) [protected]

If you are using a CX::Synth::ModuleParameter in your module, you must register that ModuleParameter during construction (or setup) of the module using this function.

```
class MyModule : public ModuleBase {
public:

    MyModule(void) {
        this->_registerParameter(&myParam);
        //...
    }

    ModuleParameter myParam;
    //...
};
```

19.53.2.10 void CX::Synth::ModuleBase::_setDataIfNotSet (ModuleBase * *target*) [protected]

This function sets the data for a target module if the data for that module has not been set.

Parameters

target	The target module to set the data for.

19.53.2.11 void CX::Synth::ModuleBase::disconnectInput (ModuleBase * *in*)

Disconnect a module that is an input to this module. This is a reciprocal operation: This module's input is disconnected and `in`'s output to this module is disconnected.

19.53.2.12 void CX::Synth::ModuleBase::disconnectOutput (ModuleBase * *out*)

Disconnect a module that this module outputs to. This is a reciprocal operation: This module's output is disconnected and `out`'s input from this module is disconnected.

19.53.2.13 ModuleControlData_t CX::Synth::ModuleBase::getData (void)

Gets the data used by the module.

19.53.2.14 double CX::Synth::ModuleBase::getNextSample (void) [virtual]

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

Returns

The value of the next sample from the module.

Reimplemented in CX::Synth::FIRFilter, CX::Synth::TrivialGenerator, CX::Synth::StreamInput, CX::Synth::SoundBufferInput, CX::Synth::Splitter, CX::Synth::RingModulator, CX::Synth::Oscillator, CX::Synth::Multiplier, CX::Synth::Mixer, CX::Synth::GenericOutput, CX::Synth::FunctionModule, CX::Synth::Filter, CX::Synth::Envelope, CX::Synth::Clamper, CX::Synth::Adder, and CX::Synth::AdditiveSynth.

19.53.2.15 void CX::Synth::ModuleBase::setData (ModuleControlData_t *d*)

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.

19.53.3 Friends And Related Function Documentation

19.53.3.1 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.
```

19.53.3.2 void operator>> (ModuleBase & *l*, ModuleParameter & *r*) [friend]

This operator connects a module to the module parameter. It is not possible to connect a module parameter as an input for anything: They are dead ends.

```
using namespace CX::Synth;
Oscillator osc;
Envelope fenv;
Adder add;
add.amount = 500;
fenv >> add >> osc.frequency; //Connect the envelope as the input for the frequency of the
       oscillator with an offset of 500 Hz.
```

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

- CX_Synth.h
- CX_Synth.cpp

19.54. CX::Synth::ModuleParameter Class Reference

`#include <CX_Synth.h>`

Public Member Functions

- ModuleParameter (void)

  Construct a ModuleParameter with no value.
- ModuleParameter (double d)

  Construct a ModuleParameter with the given start value.
- void updateValue (void)
- bool valueUpdated (bool checkForUpdates=true)
- double & getValue (void)
- operator double (void)

  Implicitly converts the parameter to `double`.
- ModuleParameter & operator= (double d)

  Assign a value to the parameter.

Friends

- class **ModuleBase**
- void operator>> (ModuleBase &l, ModuleParameter &r)

19.54.1 Detailed Description

This class is used to provide modules with the ability to have their control parameters change as a function of incoming data from other modules. For example, if you want to change the frequency of an oscillator, you can feed an LFO into the frequency parameter of the oscillator.

If you create a module that uses a ModuleParameter, you must perform one setup step in the constructor of the module. You must call ModuleBase::_registerParameter() with the ModuleParameter as the argument.

19.54.2 Member Function Documentation

19.54.2.1 double & CX::Synth::ModuleParameter::getValue (void)

Gets the current value of the parameter.

19.54.2.2 void CX::Synth::ModuleParameter::updateValue (void)

Update the value of the module parameter. This gets the next sample from the module that is the input for the ModuleParameter, if any.

19.54.2.3 bool CX::Synth::ModuleParameter::valueUpdated (bool *checkForUpdates* = `true`)

Returns `true` if the value of the ModuleParameter has been updated since the last time this function was called. This should be called right after updateValue() or with `checkForUpdates = true`. Updates to the value resulting from assignment of a new value with `operator=()` count as updates to the value.

If you don't care whether the value has been updated before using it, don't call this function. Instead, just use updateValue() and getValue().

Parameters

checkFor↩ Updates	Check for updates before determining whether the value has been updated.

Returns

`true` if the value has been updated since the last check.

19.54.3 Friends And Related Function Documentation

19.54.3.1 void operator>> (ModuleBase & *l*, ModuleParameter & *r*) [friend]

This operator connects a module to the module parameter. It is not possible to connect a module parameter as an input for anything: They are dead ends.

```
using namespace CX::Synth;
Oscillator osc;
Envelope fenv;
Adder add;
add.amount = 500;
fenv >> add >> osc.frequency; //Connect the envelope as the input for the frequency of the
       oscillator with an offset of 500 Hz.
```

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

- CX_Synth.h
- CX_Synth.cpp

19.55. CX::Synth::Multiplier Class Reference

`#include <CX_Synth.h>`

Inherits CX::Synth::ModuleBase.

Public Member Functions

- Multiplier (double amount)
- double getNextSample (void) override
- void setGain (double decibels)

Public Attributes

- ModuleParameter amount

  The amount that the input signal will be multiplied by.

Additional Inherited Members

19.55.1 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.

19.55.2 Constructor & Destructor Documentation

19.55.2.1 CX::Synth::Multiplier::Multiplier (double *amount_*)

Convenience constructor.

Parameters

*amount_*	The amount to multiply the input by.

19.55.3 Member Function Documentation

19.55.3.1 double CX::Synth::Multiplier::getNextSample (void) [override], [virtual]

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

Returns

The value of the next sample from the module.

Reimplemented from CX::Synth::ModuleBase.

19.55.3.2 void CX::Synth::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_Synth.h
- CX_Synth.cpp

19.56. CX::Synth::Oscillator Class Reference

`#include <CX_Synth.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

  The fundamental frequency of the oscillator.

Additional Inherited Members

19.56.1 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
```

19.56.2 Member Function Documentation

19.56.2.1 double CX::Synth::Oscillator::getNextSample (void) [override], [virtual]

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

Returns

The value of the next sample from the module.

Reimplemented from CX::Synth::ModuleBase.

19.56.2.2 double CX::Synth::Oscillator::saw (double *wp*) [static]

Produces a sawtooth wave.

Parameters

wp	The waveform position to sample, in the interval [0, 1), where 0 is the start of the waveform and 1 is the end of the waveform.

Returns

A value normalized to the interval [-1, 1] containing the value of the waveform function at the given waveform position.

19.56.2.3 void CX::Synth::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.
}
```

19.56.2.4 double CX::Synth::Oscillator::sine (double *wp*) [static]

Produces a sine wave.

Parameters

wp	The waveform position to sample, in the interval [0, 1), where 0 is the start of the waveform and 1 is the end of the waveform.

Returns

A value normalized to the interval [-1, 1] containing the value of the waveform function at the given waveform position.

19.56.2.5 double CX::Synth::Oscillator::square (double *wp*) [static]

Produces a square wave.

Parameters

wp	The waveform position to sample, in the interval [0, 1), where 0 is the start of the waveform and 1 is the end of the waveform.

Returns

A value normalized to the interval [-1, 1] containing the value of the waveform function at the given waveform position.

19.56.2.6 double CX::Synth::Oscillator::triangle (double *wp*) [static]

Produces a triangle wave.

Parameters

wp	The waveform position to sample, in the interval [0, 1), where 0 is the start of the waveform and 1 is the end of the waveform.

Returns

A value normalized to the interval [-1, 1] containing the value of the waveform function at the given waveform position.

19.56.2.7 double CX::Synth::Oscillator::whiteNoise (double *wp*) [static]

Produces white noise.

Parameters

wp	This argument is ignored.

Returns

A random value in the interval [-1, 1].

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

- CX_Synth.h
- CX_Synth.cpp

19.57. 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.

19.57.1 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

19.58. CX::CX_DataFrame::OutputOptions Struct Reference

`#include <CX_DataFrame.h>`

Inherits CX::CX_DataFrame::IoOptions.

Public Attributes

- bool printRowNumbers

  If `true`, a column of row numbers will be printed. The column will be named "rowNumber". Defaults to `true`.
- std::vector< rowIndex_t > rowsToPrint

  The indices of the rows that should be printed. If the vector has size 0, all rows will be printed.
- std::set< std::string > columnsToPrint

  The names of the columns that should be printed. If the set has size 0, all columns will be printed.

19.58.1 Detailed Description

Options for the format of data that are output from a CX_DataFrame.

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

- CX_DataFrame.h

19.59. CX::CX_Time_t< TimeUnit >::PartitionedTime Struct Reference

`#include <CX_Time_t.h>`

Public Attributes

- int hours

  The hours component of the time.
- int minutes

  The minutes component of the time.
- int seconds

  The seconds component of the time.
- int milliseconds

  The milliseconds component of the time.
- int microseconds

  The microseconds component of the time.
- int nanoseconds

  The nanoseconds component of the time.

19.59.1 Detailed Description

template<typename TimeUnit>struct CX::CX_Time_t< TimeUnit >::PartitionedTime

This struct contains the result of CX_Time_t::getPartitionedTime().

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

- CX_Time_t.h

19.60. 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

- std::vector< std::string > namesOfSlidesWithErrors

  The names of all of the slides that had any errors.
- 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

  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.
- 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.
- unsigned int lateStarts

  The number of slides for which the start time was later than the intended start time.

19.60.1 Detailed Description

This struct contains information about errors that were detected during slide presentation. See CX_SlidePresenter::checkForPresentationErrors() for how to get this information.

Note that false positives are possible. For example, when considering late starts, it is possible that a slide was actually presented on time, but CX did not learn that the presentation was started until after the intended start time.

It is possible for errors to be counted multiple times. For example, one slide might be copied to the back buffer late (1 error) and, as a result, presented late (2 errors), which also means that it has an incorrect frame count (3 errors).

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

- CX_SlidePresenter.h

19.61. CX::Synth::RingModulator Class Reference

`#include <CX_Synth.h>`

Inherits CX::Synth::ModuleBase.

Public Member Functions

- double getNextSample (void) override

Additional Inherited Members

19.61.1 Detailed Description

This class is an implementation of a very basic ring modulator. Ringmods need two inputs: the source and the carrier. The order doesn't matter, for this class. If only one input is given, it will just pass that input through.

This is not an analog emulation and it does nothing to deal with aliasing, so it may not work well with non-sinusoidal carriers.

```
StreamInput input;
input.setup(&ss); //Assume that ss is a CX_SoundStream that is configured for input.

StreamOutput output;
output.setup(&ss); //Assume that ss is also configured for output.

Oscillator carrier;
carrier.setGeneratorFunction(Oscillator::sine);
carrier.frequency = 250;

RingModulator rm;

carrier >> rm; //Connect the carrier
input >> rm; //And the source

Multiplier m(0.1);

rm >> m >> output;
```

19.61.2 Member Function Documentation

19.61.2.1 double CX::Synth::RingModulator::getNextSample (void) [override], [virtual]

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

Returns

The value of the next sample from the module.

Reimplemented from CX::Synth::ModuleBase.

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

- CX_Synth.h
- CX_Synth.cpp

19.62. CX::CX_SlidePresenter::Slide Struct Reference

`#include <CX_SlidePresenter.h>`

Public Types

- enum PresStatus : int {
  PresStatus::NOT_STARTED, PresStatus::RENDERING, PresStatus::SWAP_PENDING, PresStatus::IN_↩
  PROGRESS,
  PresStatus::FINISHED }

Public Attributes

- std::string name

  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 drawing↩ Function points to a function, `framebuffer` will not be drawn and drawingFunction will be called instead.
- std::function< void(void)> drawingFunction

  Pointer to a user function that will be called to draw the slide, rather than using the `framebuffer`.
- std::function< void(void)> slidePresentedCallback

  Pointer to a user function that will be called right after slide is presented, i.e. right after the back buffer containing the slide contents is swapped into the front buffer.
- PresStatus presentationStatus

  Presentation status of the slide. 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.

19.62.1 Detailed Description

This struct contains information related to slide presentation using CX_SlidePresenter.

19.62.2 Member Data Documentation

19.62.2.1 std::function<void(void)> CX::CX_SlidePresenter::Slide::drawingFunction

Pointer to a user function that will be called to draw the slide, rather than using the `framebuffer`.

Pointer to a user function that will be called to draw the slide. If this points to a function, any data in `framebuffer` will be ignored.

Note

It is important to note that if you want to do something other than drawing in this function (e.g. examining responses to other stimuli), that the time at which this function is called is not the same time at which the slide's contents appear on screen. If you want a function to be called right after the contents of this slide appear on screen, use CX::CX_SlidePresenter::Slide::slidePresentedCallback instead.

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

- CX_SlidePresenter.h

19.63. 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 Disp.getFrame↩ Number().
- uint32_t frameCount

  The number of frames the slide was/should have been 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.

19.63.1 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

19.64. CX::Synth::SoundBufferInput Class Reference

`#include <CX_Synth.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

19.64.1 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.

19.64.2 Member Function Documentation

19.64.2.1 bool CX::Synth::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.

19.64.2.2 double CX::Synth::SoundBufferInput::getNextSample (void) [override], [virtual]

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

Returns

The value of the next sample from the module.

Reimplemented from CX::Synth::ModuleBase.

19.64.2.3 void CX::Synth::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.

19.64.2.4 void CX::Synth::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_Synth.h
- CX_Synth.cpp

19.65. CX::Synth::SoundBufferOutput Class Reference

`#include <CX_Synth.h>`

Inherits CX::Synth::ModuleBase.

Public Member Functions

- void setup (float sampleRate)
- void sampleData (CX_Millis t)

Public Attributes

- CX::CX_SoundBuffer sb

  The sound buffer that will be filled with samples with sampleData() is called.

Additional Inherited Members

19.65.1 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. See the documentation for CX::Synth::StereoSoundBufferOutput to get an idea of how to use this class.

19.65.2 Member Function Documentation

19.65.2.1 void CX::Synth::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`.

19.65.2.2 void CX::Synth::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_Synth.h
- CX_Synth.cpp

19.66. CX::Synth::Splitter Class Reference

`#include <CX_Synth.h>`

Inherits CX::Synth::ModuleBase.

Public Member Functions

- double getNextSample (void) override

Additional Inherited Members

19.66.1 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;

//In runExperiment:
osc >> sp;
sp >> m1 >> out.left;
sp >> m2 >> out.right;
```

19.66.2 Member Function Documentation

19.66.2.1 double CX::Synth::Splitter::getNextSample (void) [override], [virtual]

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

Returns

The value of the next sample from the module.

Reimplemented from CX::Synth::ModuleBase.

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

- CX_Synth.h
- CX_Synth.cpp

19.67. CX::Synth::StereoSoundBufferOutput Class Reference

`#include <CX_Synth.h>`

Public Member Functions

- void setup (float sampleRate)
- void sampleData (CX_Millis t)

Public Attributes

- GenericOutput left

  The left channel of the buffer.
- GenericOutput right

  The right channel of the buffer.
- CX::CX_SoundBuffer sb

  The sound buffer that will be filled with samples with sampleData() is called.

19.67.1 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.

```
#include "CX.h"

using namespace CX::Synth;

void runExperiment(void) {
    StereoSoundBufferOutput sout;
    sout.setup(44100);

    Splitter sp;
    Oscillator osc;
    Multiplier leftM;
    Multiplier rightM;

    osc.frequency = 400;
    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.
    sout.sb.writeToFile("Stereo.wav");
}
```

19.67.2 Member Function Documentation

19.67.2.1 void CX::Synth::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`.

19.67.2.2 void CX::Synth::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_Synth.h
- CX_Synth.cpp

19.68. CX::Synth::StereoStreamOutput Class Reference

`#include <CX_Synth.h>`

Public Member Functions

- void setup (CX::CX_SoundStream *stream)

Public Attributes

- GenericOutput left

  The left channel of the stream.
- GenericOutput right

  The right channel of the stream.

19.68.1 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.

19.68.2 Member Function Documentation

19.68.2.1 void CX::Synth::StereoStreamOutput::setup (CX::CX_SoundStream * *stream*)

Set up the StereoStreamOutput with the given CX_SoundStream.

Parameters

stream	A CX_SoundStream that is configured for stereo output.

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

- CX_Synth.h
- CX_Synth.cpp

19.69. CX::Synth::StreamInput Class Reference

`#include <CX_Synth.h>`

Inherits CX::Synth::ModuleBase.

Public Member Functions

- void setup (CX::CX_SoundStream *stream)
- double getNextSample (void) override
- void clear (void)

  Clear the contents of the input buffer.
- void setMaximumBufferSize (unsigned int size)

Additional Inherited Members

19.69.1 Detailed Description

This class is a module that takes input from a CX_SoundStream configured for input, so it is good for getting sounds from a microphone or line in. This class is strictly monophonic.

In order to be compatible with the other modules, this module takes in sound data and stores it in an internal buffer. Requests for samples from this class will takes samples from the buffer. If the buffer is empty, this will output 0. If there are no requests for samples from this class for a long time, its buffer can get very large. Then, when samples are requested, the samples it gives out will be very old. For this reason, user code can configure a maximum buffer size using setMaximumBufferSize(). The maximum buffer size defaults to 4096 samples. User code can clear the buffer with clear().

19.69.2 Member Function Documentation

19.69.2.1 double CX::Synth::StreamInput::getNextSample (void) [override], [virtual]

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

Returns

The value of the next sample from the module.

Reimplemented from CX::Synth::ModuleBase.

19.69.2.2 void CX::Synth::StreamInput::setMaximumBufferSize (unsigned int *size*)

Set the maximum number of samples that the input buffer can contain.

Parameters

size	The size of the input buffer, in samples.

19.69.2.3 void CX::Synth::StreamInput::setup (CX::CX_SoundStream * *stream*)

Set up the StreamInput with a CX_SoundStream configured for input.

Parameters

stream	A pointer to the sound stream.

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

- CX_Synth.h
- CX_Synth.cpp

19.70. CX::Synth::StreamOutput Class Reference

`#include <CX_Synth.h>`

Inherits CX::Synth::ModuleBase.

Public Member Functions

- void setup (CX::CX_SoundStream *stream)

Additional Inherited Members

19.70.1 Detailed Description

This class provides a method of playing the output of a modular synth using a CX_SoundStream. This class can only take data from one input, so it is monophonic. However, the sound stream does not need to be configured to only use 1 output channel because this class will put the same data on all available output channels. 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 that ss has been started.
CX_SoundStream ss;
Oscillator osc;

Synth::StreamOutput output;
output.setup(&ss);

osc >> output; //Sound should be playing past this point.
```

19.70.2 Member Function Documentation

19.70.2.1 void CX::Synth::StreamOutput::setup (CX::CX_SoundStream * *stream*)

Set up the StereoStreamOutput with the given CX_SoundStream.

Parameters

stream	A CX_SoundStream that is configured for output to any number of channels.

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

- CX_Synth.h
- CX_Synth.cpp

19.71. CX::Synth::TrivialGenerator Class Reference

`#include <CX_Synth.h>`

Inherits CX::Synth::ModuleBase.

Public Member Functions

- double getNextSample (void) override

Public Attributes

- ModuleParameter value

  The start value.
- ModuleParameter step

  The amount to change on each step.

Additional Inherited Members

19.71.1 Detailed Description

This class is used for numerically, rather than auditorily, testing other modules. It produces samples starting at `value` and increasing by `step`.

19.71.2 Member Function Documentation

19.71.2.1 double CX::Synth::TrivialGenerator::getNextSample (void) [override], [virtual]

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

Returns

The value of the next sample from the module.

Reimplemented from CX::Synth::ModuleBase.

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

- CX_Synth.h
- CX_Synth.cpp

19.72. CX::Draw::Gabor::Wave Struct Reference

`#include <CX_Gabor.h>`

Static Public Attributes

- static std::string saw = "return wp;"

  Produces a saw wave.
- static std::string sine = "return (sin(wp $*$ 6.283185307179586232) + 1) / 2;"

  Produces a sine wave.
- static std::string square = "if (wp $<$ 0.5) return 1; \n return 0;"

  Produces a square wave.
- static std::string triangle = "if (wp $<$ .5) return (2 $*$ wp); \n return 2 - (2 $*$ wp);"

  Produces a triangle wave.

19.72.1 Detailed Description

This struct contains several functions that are used for calculating the mixing between color1 and color2.

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

- CX_Gabor.h
- CX_Gabor.cpp

19.73. CX::Draw::WaveformProperties Struct Reference

`#include <CX_Gabor.h>`

Static Public Member Functions

- static float sine (float wp)
- static float square (float wp)
- static float triangle (float wp)
- static float saw (float wp)

Public Attributes

- float width
- float height
- float angle
- float wavelength

  The distance, in pixels, between the center of each wave within the pattern.
- float phase

  The phase shift of the waves, in degrees.
- std::function< float(float)> waveFunction

19.73.1 Detailed Description

Controls the properties of a waveform drawn with CX::Draw::waveformToPixels().

19.73.2 Member Function Documentation

19.73.2.1 float CX::Draw::WaveformProperties::saw (float *wp*) [static]

Produces a saw wave.

Parameters

wp	The waveform position in the interval [0,1).

Returns

A value in the range [0,1], depending on the waveform position.

19.73.2.2 float CX::Draw::WaveformProperties::sine (float *wp*) [static]

Produces a sine wave.

Parameters

wp	The waveform position in the interval [0,1).

Returns

A sinusoidal value in the range [0,1], depending on the waveform position.

19.73.2.3 float CX::Draw::WaveformProperties::square (float *wp*) [static]

Produces a square wave.

Parameters

wp	The waveform position in the interval [0,1).

Returns

A 0 or 1, depending on the waveform position.

19.73.2.4 float CX::Draw::WaveformProperties::triangle (float *wp*) [static]

Produces a triangle wave.

Parameters

wp	The waveform position in the interval [0,1).

Returns

A value in the range [0,1], depending on the waveform position.

19.73.3 Member Data Documentation

19.73.3.1 float CX::Draw::WaveformProperties::angle

The angle at which the waves are oriented, in degrees.

19.73.3.2 float CX::Draw::WaveformProperties::height

The height of the pattern, in pixels.

19.73.3.3 std::function<float(float)> CX::Draw::WaveformProperties::waveFunction

A function that calculates the height of the wave given a waveform position. It should take the current waveform position as a value in the interval [0,1) and return the relative height of the wave as a value in the interval [0,1]. See the static functions in this struct, like sine(), square(), etc. for some options.

19.73.3.4 float CX::Draw::WaveformProperties::width

The width of the pattern, in pixels.

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

- CX_Gabor.h
- CX_Gabor.cpp

Index