Nyx

Hunter Fehlan

Nov 30, 2020

CONTENTS

CHAPTER

ONE

ABOUT

Nyx is a fast, easy to use graphics and game development framework created with simplicty in mind–but without sacrificing useability or features.

CHAPTER

TWO

TUTORIAL: START HERE

This tutorial is intended to introduce the concepts used in Nyx–especially to clear up differences from most graphics libraries. I promise, the API for Nyx is simple to use once you learn it, but there is a learning curve as there are features added that make programs run much faster, but also are not typically seen.

To start, `#include "nyx/nyx.hpp"` at the top of your source file. This should bring in all the normal parts of Nyx that you may need to use.

2.1 Basics

Nyx is centered around an `Engine` which controls everything and handles resource management for you. Isn't that nice?

Creating one is simple:

```
auto engine = nyx::Engine();
```

`Engine` does not take any parameters, it simply sets up the base to start building a program off of.

Once you have an `Engine` created, you're going to need a window. After all, a graphical program isn't very useful if you can't see what you're doing.

To make a `Window`, you request one from the `Engine`:

```
auto window = engine.create_window("Tutorial", 800, 600);
```

The `create_window` method takes up to 4 arguments: a caption, width, height, and optionally, a boolean to enable/disable vsync. By default, vsync is enabled, thought you may wish to disable it if you are trying to take a performance benchmark.

Note: For now, only one `Window` object is supported at a time, though there are plans in the future to implement more than one.

The final piece we need to get up and running is at least one `Context`. On their own, `Windows` are not capable of doing any drawing, and the only thing you can do is clear it with a color. While that isn't nothing, it'll be much cooler if we can actually draw something. `Contexts` belong to `Windows`, and they are created like this:

```
auto ctx = window->create_context();
```

The `create_context` method does not take any arguments.

You can have more than once `Context`, and we'll talk about why you might want to do that in a later tutorial.

Each of the objects we just created did a lot of work in the background to set up an environment to start putting content on the screen, but that is all (mostly) hidden from the user. This is by design to keep the library as simple as possible. In the future, there may be more options for customizability, but the defaults for now are sensible and should be completely fine for the majority of cases.

2.2 Main loop

Now that we have our framework, we can talk about the main loop of all Nyx programs. You are free to create a loop however you like, but there are two things that *must* happen every time a frame is rendered. For simplicity, let's assume we have the objects we created before, then this would be a standard loop:

```
do {
  engine.update();

  window->clear();

  engine.show();
} while (engine.is_running);
```

There is a lot going on in only a few lines here, so let's break it down.

`Engine::update()` is the backbone of all event handling in Nyx. It polls all the events like keypresses, mouse clicks, mouse position, window closing, etc, and passes them where they need to go. This *must* be called at the beginning of the loop for it to function correctly. It also updates the internal timers, but we'll get to that later.

`Window::clear()` does exactly what it sounds like it does–it clears the window. It takes an optional argument of the form of a `nyx::Color` which will determine which color to clear it with.

`Engine::show()` *must* go at the bottom of the loop and is what tells Nyx to draw something onto the screen. Before `show` is called, any graphical changes to the window are buffered but not pushed to the GPU. Once `show` is called, those changes are submitted and the window will update.

`Engine::is_running` is true so long as there is at least one active window. For now, since there is only one window allowed, this becomes false as soon as the window is closed.

2.3 Drawing

You may have noticed that there are no actual drawing commands in the main loop, and now it's time to fix that. If the program was run at this point, a window would pop up on the screen, but would be very boring. Let's add a line using that context we created earlier.

The method for adding a line takes 5 parameters: an `(x1, y1)` pair for the first point, an `(x2, y2)` pair for the second point, and a color. Let's draw a line from `(0, 0)` to `(window->w, window->h)` in red:

```
do {
  engine.update();

  window->clear();

  ctx->clear();
  ctx->line(0, 0, window->w, window->h, nyx::Color(0xff0000ff));

  engine.show();
} while (engine.is_running);
```

`window->w` and `window->h` simply get the width and height of the window.

Running the program now, you should see a red line that crosses the window diagonally from the top left to the bottom right. Congrats, you've drawn something!

There is something important happening here with the `Context`. Notice that we are calling `ctx->clear()` before adding the line. This is very important, as if we didn't, the `Context` would simply fill up drawing the same line one more time every frame. This would gradually make the framerate worse and worse! `Contexts` keep their state between frames unless a clear is called, and this is one of the big features of Nyx. If all you wanted was a static line drawn on the screen, the more idiomatic way in Nyx would look like this:

```
ctx->line(0, 0, window->w, window->h, nyx::Color(0xff0000ff));

do {
  engine.update();

  window->clear();

  engine.show();
} while (engine.is_running);
```

For drawing a single line, this won't make much of a performance difference, but when drawing large numbers of objects, taking advantage of the persistent state of a `Context` can *massively* improve drawing performance–which in a game can be the difference between being playable or not. The less time rendering is taking up, the more time you have for game logic.

2.4 Review

We created an `Engine`, used it to create a `Window`, and then used that to create a drawing `Context`. From there, we created a loop to draw onto the screen, and drew a single line in two different ways. For reference, here is the completed source for the program:

```
#include "nyx/nyx.hpp"

int main(int, char *[]) {
  auto engine = nyx::Engine();
  auto window = engine.create_window("Tutorial", 800, 600);
  auto ctx = window->create_context();

  ctx->line(0, 0, window->w, window->h, nyx::Color(0xff0000ff));

  do {
    engine.update();

    window->clear();

    engine.show();
  } while (engine.is_running);

  return 0;
}
```

CHAPTER

THREE

TUTORIAL: YOU SPIN ME RIGHT ROUND

This tutorial is going to introduce a few more features of Nyx, and to do that, we're going to make a spinning square that changes colors!

3.1 Setup

We're going to start with the same code we had at the end of the previous tutorial with a small modification (not drawing a line). Here it is:

```
#include "nyx/nyx.hpp"

int main(int, char *[]) {
  auto engine = nyx::Engine();
  auto window = engine.create_window("Tutorial", 800, 600);
  auto ctx = window->create_context();

  do {
    engine.update();

    window->clear();

    engine.show();
  } while (engine.is_running);

  return 0;
}
```

3.2 Drawing a Square

Drawing a square is easy with Nyx! We're going add a simple bit of code. Last tutorial, I mentioned that you often want to draw it outside the loop if it's going to be static, but since this is going to eventually rotate, we're going to have to redraw every frame, meaning we're going to be clearing the `Context` each time. Here is that part of the code changed to our new needs of drawing a white, 100x100 square at the center of the screen:

```
int sq_w = 100;
int sq_h = 100;

do {
  engine.update();
```

(continues on next page)

(continued from previous page)

```
  window->clear();

  ctx->clear();
  ctx->rectangle(
      (window->w / 2) - (sq_w / 2),
      (window->h / 2) - (sq_h / 2),
      sq_w,
      sq_h,
      nyx::Color(0xffffffff));

  engine.show();
} while (engine.is_running);
```

Easy! `Context::rectangle()` takes up to 6 parameters: an x, y, width, height, angle, and color. Angle is optional and you can just put a color where it goes instead. We'll use the angle parameter in a second, but the above snippet doesn't use it.

We are subtracting half the size of the square from the center point of the window so the center of the square is what's actually at the center of the window. Like most libraries, squares/rectangles are drawn from the top left corner.

Now we want it to rotate. Let's do that now by creating a new variable named `angle` that we will increment every frame:

```
int sq_w = 100;
int sq_h = 100;

double angle = 0.0;

do {
  engine.update();

  angle += 1.0;

  window->clear();

  ctx->clear();
  ctx->rectangle(
      (window->w / 2) - (sq_w / 2),
      (window->h / 2) - (sq_h / 2),
      sq_w,
      sq_h,
      angle,
      nyx::Color(0xffffffff));

  engine.show();
} while (engine.is_running);
```

This works, but there's a problem. There isn't any good way to decide how much to increment the angle each frame. As of now, it's based on the framerate. The higher the framerate, the faster it will spin. Not to worry, there's an easy fix.

3.3 Delta Time

Unless you've looked at the API already, you won't know that `Engine::update()` actually returns a `double` value. This value is the delta time of the previous frame–how many seconds it between the last frame and this one. You can use this value to decide how much to change values between frames. Let's use that now to make the square rotate 90 degrees per second:

```
int sq_w = 100;
int sq_h = 100;

double angle = 0.0;

do {
  double dt = engine.update();

  angle += 90.0 * dt;

  window->clear();

  ctx->clear();
  ctx->rectangle(
      (window->w / 2) - (sq_w / 2),
      (window->h / 2) - (sq_h / 2),
      sq_w,
      sq_h,
      angle,
      nyx::Color(0xffffffff));

  engine.show();
} while (engine.is_running);
```

If you run this now, you'll see the square smoothly rotates around in a circle at ~1/4 turn per second. This is now completely independent of how fast the program runs because we are basing the rotation off how fast the program is running. Isn't that cool?

This is a technique used widely throughout game development, and can be used any time you need a frame indepenedent way to move things around. Don't forget that `update` gives you that `dt`, you don't need to calculate it yourself!

3.4 Changing Color With Timers

Timers are built into Nyx, and can be used to call a snippet of code periodically. This snippet is taken in the form of a C++11 lambda, and there are many different timer types. Take a look at the API to see all of them, but in this tutorial, we are going to use the `every` timer.

`every` timers execute the code snippet given every N seconds, where N is a `double` that you provide. We want to change the color to something random each frame, which is going to be very easy with this technique. Here is the full code snippet, comments included where changes were made:

```
#include "nyx/nyx.hpp"

#include <stdlib.h>  // rand

// Moved square size up here and made it constant
const int SQ_W = 100;
```

(continues on next page)

(continued from previous page)

```
const int SQ_H = 100;

int main(int, char *[]) {
  srand(time(NULL));  // Seed the random generator

  auto engine = nyx::Engine();
  auto window = engine.create_window("Tutorial", 800, 600);
  auto ctx = window->create_context();

  auto color = nyx::Color(0xffffffff);  // Start with the color white

  // Create a timer that executes the given function every 1.0 seconds
  engine.timers->every(1.0, [&]{
    // Set the color to a random color given 3 random values for r, g, and b
    color = nyx::Color(rand()%255, rand()%255, rand()%255);
  });

  double angle = 0.0;

  do {
    double dt = engine.update();

    angle += 90.0 * dt;

    window->clear();

    ctx->clear();
    ctx->rectangle(
        (window->w / 2) - (SQ_W / 2),
        (window->h / 2) - (SQ_H / 2),
        SQ_W,
        SQ_H,
        angle,
        color);  // Use the new color variable to
                 // determine the color

    engine.show();
  } while (engine.is_running);

  return 0;
}
```

If all is well, you should see the square rotating, and also changing colors every second. This concludes this tutorial.

CHAPTER

FOUR

API

4.1 Class Hierarchy

4.2 File Hierarchy

4.3 Full API

4.3.1 Namespaces

4.3.1.1 Namespace nyx

Contents

4.3.1.1.1 Classes

4.3.1.1.2 Functions

- *Function nyx::ms*
- *Function nyx::ns*
- *Function nyx::read_file*

4.3.2 Classes and Structs

4.3.2.1 Class Color

- Defined in file_include_nyx_color.hpp

4.3.2.1.1 Class Documentation

class `nyx::Color`

Color is used for any function that requires a color. There are two ways to construct a color, with integer values given for the r, g, b, and a channels, or a hex number which must include an alpha channel as the final part of the number.

Public Functions

Color(unsigned int *hex*)

Instantiate a Color using a hex value

Parameters

- `hex`: integer of the form 0xRRGGBBAA

Color(int *r*, int *g*, int *b*, int *a* = 255)

Instantiate a Color using 3-4 integer values

Parameters

- `r`: the red channel, [0, 255]
- `g`: the green channel, [0, 255]
- `b`: the blue channel, [0, 255]
- `a`: the alpha channel, [0, 255]

Public Members

int **r**

the red channel, [0, 255]

int **g**

the green channel, [0, 255]

int **b**

the blue channel, [0, 255]

int **a**

the alpha channel, [0, 255]

float **rf**
the red channel, [0.0f, 1.0f]

float **gf**
the green channel, [0.0f, 1.0f]

float **bf**
the blue channel, [0.0f, 1.0f]

float **af**
the alpha channel, [0.0f, 1.0f]

4.3.2.2 Class Context

- Defined in file_include_nyx_context.hpp

4.3.2.2.1 Class Documentation

class `nyx::`**Context**
Context is how everything in Nyx is drawn. Contexts have multiple methods that are used to draw things, including primitives and textures. Each context has persistent state, meaning that when objects are added to them, they are kept buffered in OpenGL until the `clear()` method is called. This allows incredibly fast rendering as scenes don't need to be redrawn every frame.

Public Functions

Context`()`
Create a new Context, should not be used directly, instead ask a Window to create a context for you.

~Context`()`

void **clear**`()`
Reset the state of the context, clearing the OpenGL buffers and all objects this Context currently draws to the screen.

void **pixel**(float *x*, float *y*, *Color* *color*)
Draw a pixel at (x, y) using the specified Color.

Parameters

- `x`: x coordinate of the pixel
- `y`: y coordinate of the pixel
- `color`: the color to draw the pixel with

void **line**(float *x1*, float *y1*, float *x2*, float *y2*, *Color* *color*)
Draw a line from the point (x1, y1) to (x2, y2) using the specified Color.

Parameters

- `x1`: x coordinate of first point
- `y1`: y coordinate of first point
- `x2`: x coordinate of second point
- `y2`: y coordinate of second point

- `color`: the color to draw the line with

void **`triangle`**(float *x1*, float *y1*, float *x2*, float *y2*, float *x3*, float *y3*, float *angle*, *Color color*)
Draw a triangle with the points (x1, y1), (x2, y2), and (x3, y3), rotated at angle, using the specified color.

Parameters

- `x1`: x coordinate of first point
- `y1`: y coordinate of first point
- `x2`: x coordinate of second point
- `y2`: y coordinate of second point
- `x3`: x coordinate of third point
- `y3`: y coordinate of third point
- `angle`: angle in degrees
- `color`: the color to draw the triangle with

void **`triangle`**(float *x1*, float *y1*, float *x2*, float *y2*, float *x3*, float *y3*, *Color color*)
Draw a triangle with the points (x1, y1), (x2, y2), and (x3, y3) using the specified color.

Parameters

- `x1`: x coordinate of first point
- `y1`: y coordinate of first point
- `x2`: x coordinate of second point
- `y2`: y coordinate of second point
- `x3`: x coordinate of third point
- `y3`: y coordinate of third point
- `color`: the color to draw the triangle with

void **`rectangle`**(float *x*, float *y*, float *w*, float *h*, float *angle*, *Color color*)
Draw a rectangle at (x, y) with a size of (w, h), rotated at angle, using the specified color.

Parameters

- `x`: the x coordinate
- `y`: the y coordinate
- `w`: the width
- `h`: the height
- `angle`: angle in degrees
- `color`: the color to draw the rectangle with

void **`rectangle`**(float *x*, float *y*, float *w*, float *h*, *Color color*)
Draw a rectangle at (x, y) with a size of (w, h), using the specified color.

Parameters

- `x`: the x coordinate

- `y`: the y coordinate
- `w`: the width
- `h`: the height
- `color`: the color to draw the rectangle with

void **`asset`**(std::string **`const`** &*name*, std::string **`const`** &*path*, *Color* *mask*, bool *retro* = false)
Load an asset (texture)

Parameters

- `name`: the name to give the asset
- `path`: the path to the asset
- `mask`: the color mask to use with the asset
- `retro`: if true, use nearest-neighbor filtering

void **`asset`**(std::string **`const`** &*name*, std::string **`const`** &*path*, bool *retro* = false)
Load an asset (texture), using color mask 0xff00ffff (magenta)

Parameters

- `name`: the name to give the asset
- `path`: the path to the asset
- `retro`: if true, use nearest-neighbor filtering

void **`texture`**(std::string **`const`** &*path*, float *x*, float *y*, *Color* *color* = *Color*(0xffffffff))
Draw a texture at full size at (x, y)

Parameters

- `path`: the name of the texture
- `x`: the x coordinate
- `y`: the y coordinate
- `color`: the color to multiply into the texture

void **`texture`**(std::string **`const`** &*path*, float *x*, float *y*, float *angle*, *Color* *color* = *Color*(0xffffffff))
Draw a texture at full size at (x, y) at an angle.

Parameters

- `path`: the name of the texture
- `x`: the x coordinate
- `y`: the y coordinate
- `angle`: angle in degrees
- `color`: the color to multiply into the texture

void **`texture`**(std::string **`const`** &*path*, float *x*, float *y*, float *w*, float *h*, *Color* *color* = *Color*(0xffffffff))
Draw a texture at (x, y) with size (w, h)

Parameters

- `path`: the name of the texture
- `x`: the x coordinate
- `y`: the y coordinate
- `w`: the width to scale to
- `h`: the height to scale to
- `color`: the color to multiply into the texture

void **texture**(std::string **const** &*path*, float *x*, float *y*, float *w*, float *h*, float *angle*, *Color* *color* = *Color*(0xffffffff))

Draw a texture at (x, y) with size (w, h) at an angle.

Parameters

- `path`: the name of the texture
- `x`: the x coordinate
- `y`: the y coordinate
- `w`: the width to scale to
- `h`: the height to scale to
- `angle`: angle in degrees
- `color`: the color to multiply into the texture

void **texture**(std::string **const** &*path*, float *x*, float *y*, float *sx*, float *sy*, float *sw*, float *sh*, *Color* *color* = *Color*(0xffffffff))

Draw a portion of a texture at (x, y), using the portion at (sx, sy) with a size of (sw, sh)

Parameters

- `path`: the name of the texture
- `x`: the x coordinate
- `y`: the y coordinate
- `sx`: the source x
- `sy`: the source y
- `sw`: the source width
- `sh`: the source height
- `color`: the color to multiply into the texture

void **texture**(std::string **const** &*path*, float *x*, float *y*, float *sx*, float *sy*, float *sw*, float *sh*, float *angle*, *Color* *color* = *Color*(0xffffffff))

Draw a portion of a texture at (x, y), using the portion at (sx, sy) with a size of (sw, sh), at an angle.

Parameters

- `path`: the name of the texture
- `x`: the x coordinate

- `y`: the y coordinate
- `sx`: the source x
- `sy`: the source y
- `sw`: the source width
- `sh`: the source height
- `angle`: angle in degrees
- `color`: the color to multiply into the texture

void **`texture`**(std::string **`const`** &*path*, float *x*, float *y*, float *w*, float *h*, float *sx*, float *sy*, float *sw*, float *sh*, *Color* *color* = *Color*(0xffffffff))

Draw a portion of a texture at (x, y) with a size of (w, h), using the portion at (sx, sy) with a size of (sw, sh)

Parameters

- `path`: the name of the texture
- `x`: the x coordinate
- `y`: the y coordinate
- `w`: the width to scale to
- `h`: the height to scale to
- `sx`: the source x
- `sy`: the source y
- `sw`: the source width
- `sh`: the source height
- `color`: the color to multiply into the texture

void **`texture`**(std::string **`const`** &*path*, float *x*, float *y*, float *w*, float *h*, float *sx*, float *sy*, float *sw*, float *sh*, float *angle*, *Color* *color* = *Color*(0xffffffff))

Draw a portion of a texture at (x, y) with a size of (w, h), using the portion at (sx, sy) with a size of (sw, sh), at an angle.

Parameters

- `path`: the name of the texture
- `x`: the x coordinate
- `y`: the y coordinate
- `w`: the width to scale to
- `h`: the height to scale to
- `sx`: the source x
- `sy`: the source y
- `sw`: the source width
- `sh`: the source height
- `angle`: angle in degrees
- `color`: the color to multiply into the texture

4.3.2.3 Class Engine

- Defined in file_include_nyx_engine.hpp

4.3.2.3.1 Class Documentation

class nyx::**Engine**

Engines are the main hub for everything in Nyx. Engines are responsible for handling all

Public Functions

Engine()

Create a new engine.

~Engine()

Window ***create_window**(std::string **const** &*title*, unsigned int *width*, unsigned int *height*, bool *vsync* = true)

Create a new window, adding it to the list of windows for this engine

Return a pointer to the new Window

Parameters

- title: the caption/title for the window
- width: the width of the new window
- height: the height of the new window
- vsync: whether the window has vsync enabled

double **update**()

Update all the timers and events for this window, and the fps for all windows.

void **show**()

TODO: Document.

Public Members

bool **is_running**

If true, there is at least one active window.

TimerMgr ***timers**

Used to manage timers automatically for this Engine instance.

4.3.2.4 Class EventMgr

- Defined in file_include_nyx_event_mgr.hpp

4.3.2.4.1 Class Documentation

class nyx::**EventMgr**
TODO: Document

Public Functions

EventMgr()
TODO: Document.

~EventMgr()
TODO: Document.

bool **pressed**(std::string **const** &*key*)
TODO: Document.

bool **released**(std::string **const** &*key*)
TODO: Document.

bool **down**(std::string **const** &*key*, double *interval*, double *delay*)
TODO: Document.

bool **down**(std::string **const** &*key*, double *interval*)
TODO: Document.

bool **down**(std::string **const** &*key*)
TODO: Document.

Public Members

int **mouse_x**
TODO: Document.

int **mouse_y**
TODO: Document.

int **mouse_dx**
TODO: Document.

int **mouse_dy**
TODO: Document.

4.3.2.5 Class Font437

- Defined in file_include_nyx_font_font_437.hpp

4.3.2.5.1 Class Documentation

class nyx::**Font437**
TODO: Document

Public Functions

Font437(nyx::*Context* *ctx, std::string **const** &path, int char_w, int char_h)
TODO: Document.

~Font437()
TODO: Document.

void **render**(std::string **const** &text, float x, float y, float scale, *Color* color = *Color*(0xffffffff))
TODO: Document.

void **render**(std::string **const** &text, float x, float y, *Color* color = *Color*(0xffffffff))
TODO: Document.

Public Members

int **char_w**
TODO: Document.

int **char_h**
TODO: Document.

bool **tile_coords**

4.3.2.6 Class FPSClock

- Defined in file_include_nyx_fpsclock.hpp

4.3.2.6.1 Class Documentation

class nyx::**FPSClock**
TODO: Document

Public Functions

FPSClock(int sample_count_)
TODO: Document.

~FPSClock() = default
TODO: Document.

double **tick**()
TODO: Document.

Public Members

double **dt**
TODO: Document.

4.3.2.7 Class Spritesheet

- Defined in file_include_nyx_spritesheet.hpp

4.3.2.7.1 Class Documentation

class nyx::**Spritesheet**
TODO: Document

Public Functions

Spritesheet(*Context* *ctx, std::string **const** &*name*, unsigned int *grid_w*, unsigned int *grid_h*, std::initializer_list<std::tuple<std::string, SpriteDef>> *defs*)
TODO: Document.

~Spritesheet()
TODO: Document.

void **blit**(std::string **const** &*name*, int *x*, int *y*, int *scale* = 1)
TODO: Document.

Public Members

unsigned int **grid_w**
TODO: Document.

unsigned int **grid_h**
TODO: Document.

int **offset_x**

int **offset_y**

bool **tile_coords**

4.3.2.8 Class TimerMgr

- Defined in file_include_nyx_timer_mgr.hpp

4.3.2.8.1 Class Documentation

class nyx::**TimerMgr**
TODO: Document

Public Functions

TimerMgr()
TODO: Document.

~TimerMgr()
TODO: Document.

void **pause**(std::string **const** &*tag*)
TODO: Document.

void **resume**(std::string **const** &*tag*)
TODO: Document.

bool **is_paused**(std::string **const** &*tag*)
TODO: Document.

void **cancel**(std::string **const** &*tag*)
TODO: Document.

void **cancel_all**()
TODO: Document.

std::string **after**(double *delay*, std::function<void)
>*f*TODO: Document.

std::string **during**(double *delay*, std::function<void)
>*f*TODO: Document.

std::string **every**(double *delay*, std::function<void)
>*f*TODO: Document.

std::string **every**(double *delay*, int *count*, std::function<void)
>*f*TODO: Document.

4.3.2.9 Class Window

- Defined in file_include_nyx_window.hpp

4.3.2.9.1 Class Documentation

class nyx::**Window**
TODO: Document

Public Functions

Window(*Engine* *parent, **const** std::string &title, int width, int height, bool vsync = true)
TODO: Document.

~Window()
TODO: Document.

Context ***create_context**()
TODO: Document.

void **set_current**()
TODO: Document.

void **clear**(*Color* color = *Color*(0x000000ff))
TODO: Document.

void **imgui_newframe**()
TODO: Document.

void **imgui_window**(std::string **const** &title, std::function<void()> f)
TODO: Document.

void **imgui_window**(std::string **const** &title, ImGuiWindowFlags flags, std::function<void()> f)
TODO: Document.

void **imgui_window**(std::string **const** &title, ImGuiWindowFlags flags, ImVec2 pos, std::function<void()> f)
TODO: Document.

void **save**(std::string **const** &path)

Public Members

int **w**
The width of this window, will change on resize.

int **h**
The height of this window, will change on resize.

bool **show_fps**
If true, draw an fps counter in the top left.

EventMgr ***events**
Used to keep track of events for this Window.

4.3.3 Functions

4.3.3.1 Function nyx::ms

- Defined in file_include_nyx_utils.hpp

4.3.3.1.1 Function Documentation

long long nyx::**ms**()
TODO: Document.

4.3.3.2 Function nyx::ns

- Defined in file_include_nyx_utils.hpp

4.3.3.2.1 Function Documentation

long long nyx::**ns**()
TODO: Document.

4.3.3.3 Function nyx::read_file

- Defined in file_include_nyx_utils.hpp

4.3.3.3.1 Function Documentation

std::string nyx::**read_file**(**const** std::string &*path*)
TODO: Document.

INDEX

N