Tiny Lib Documentation

Release 0.1

roxlu

March 14, 2016

Contents

Tiny Lib is a one file, header only cross platform creative coding "library" aimed at OpenGL development. Tiny Lib tries to stay tiny but to support as much as features that one needs for common creative coding tasks. Tiny Lib provides features for:

- OpenGL: shaders, vertex types, debug drawing, OBJ loading
- Image loading and saving: for JPG and PNG
- Math: mat4, mat3, vec4, vec3, vec2, Perlin noise, Spline
- Curl: loading http pages
- Utils: string to number (and visa versa), file loading, time utils, file and directory support

Contents:

CHAPTER 1

Programmers Guide

In this guide we will describe how to use Tiny Lib in your project.

1.1 Introduction to Tiny Lib

Tiny Lib is a "one header" library. This means that the declarations (the functions, types, etc..) and the definitions (the actual source code/implementation) are all in the same file. To make sure that the definitions are compiled we need to inject it somewhere in your project.

There are two reasons why we store the definitions and declarations in the same file:

- This allows you to use this library in your project w/o any complicated compile scripts
- This allows you to use your own openGL wrapper code that includes the GL headers

1.2 How to use Tiny Lib in your project

As described above we need to inject the definitions somewhere in your code. To do this you need to define `ROXLU_IMPLEMENTATION` in one file (and only one file!) in your project before including tinylib.h. A common place to do this is in your main.cpp after you've included all the necessary openGL headers (if you want to make use of the tiny lib opengl features). Besides defining the `ROXLU_IMPLEMENTATION` you need to tell tiny lib what features you want to use. You can use the following defines:

- `ROXLU_USE_MATH` To enable mat4, mat3, vec4, vec3, vec2, Perlin, Spline
- `ROXLU_USE_PNG` To enable loading and saving of PNG files
- `ROXLU_USE_JPG` To enable loading of JPG files
- `ROXLU_USE_OPENGL` To enable Shader, Program, OBJ, Painter, VertexP, etc..
- `ROXLU_USE_CURL` To enable loading of remote data over http
- `ROXLU_USE_AUDIO` To enable audio output
- `ROXLU_USE_FONT` To enable PixelFont to draw bitmap strings (with opengl)
- `ROXLU_USE_ALL` To enable everything

Note that some of these defines enable code that are depending on other libraries see below in the dependencies section. In the following code examples we show you an example of how to use the `ROXLU_IMPLEMENTATION` in your main.cpp file and how to enable features in a common header (*MyApplication*).

Example main.cpp:

```
#define GLFW_INCLUDE_GLCOREARB
#include <GLFW/glfw3.h>

#define ROXLU_USE_MATH
#define ROXLU_USE_PNG
#define ROXLU_USE_OPENGL
#define ROXLU_IMPLEMENTATION
#include <tinylib.h>
```

Example MyApplication.h:

```
#define ROXLU_USE_OPENGL
#define ROXLU_USE_MATH
#include <tinylib.h>

class MyApplication() {

}
```

1.3 Dependencies

When you enable certain features there are some dependencies which are listed below:

ROXLU_USE_CURL

- libcurl

ROXLU_USE_PNG

- libpng

ROXLU_USE_JPG

- libjpeg

ROXLU_USE_AUDIO

- libcubeb
- libsndfile

On Mac you need to link with the following libraries and frameworks:

- pthreads
- AudioUnit framework
- CoreAudio framework
- AudioToolbox framework

On Linux you need to link with the following libraries:

- pthread
- dl
- asound
- z
- vorbis

- vorbisenc
- FLAC
- ogg

CHAPTER 2

API Reference

The Tiny Lib api contains a plain functions which are all prefixed with `rx_` and some classes. Below we describe the API per category.

2.1 Overview

String and conversion utils

- *rx_to_float()*
- *rx_to_int()*
- *rx_int_to_string()*
- *rx_float_to_string()*
- *rx_string_replace()*
- *rx_split()*

File Utils

- *rx_get_exe_path()*
- *rx_to_data_path()*
- *rx_get_file_mtime()*
- *rx_is_dir()*
- *rx_file_exists()*
- *rx_strip_filename()*
- *rx_strip_dir()*
- *rx_create_dir()*
- *rx_create_path()*
- *rx_get_file_ext()*
- *rx_get_files()*
- *rx_norm_path()*
- *rx_read_file()*

Time Utils

- *rx_hrtime()*
- *rx_millis()*
- *rx_strftime()*
- *rx_get_year()*
- *rx_get_month()*
- *rx_get_day()*
- *rx_get_hour()*
- *rx_get_minute()*
- *rx_get_time_string()*
- rx_get_date_string()

Image Utils

- rx_save_png()
- rx_load_png()
- rx_save_jpg()
- rx_load_jpg()

OpenGL

- *rx_create_shader()*
- *rx_create_shader_from_file()*
- *rx_create_program_with_attribs()*
- *rx_get_uniform_location()*
- *rx_uniform_1i()*
- *rx_uniform_1f()*
- *rx_uniform_mat4fv()*
- *rx_create_png_screenshot()*
- *rx_create_jpg_screenshot()*

2.2 String and conversion utils

float **rx_to_float**(**const** std::string&)
Convert a string to a float value.

int **rx_to_int**(**const** std::string&)
Convert a string to a integer value.

std::string **rx_string_replace**(std::string, char *from*, char *to*)
Convert a character in a string into another value

std::string **rx_string_replace**(std::string, std::string *from*, std::string *to*)
Convert a string from one value into another.

std::string **rx_int_to_string**(**const** int &*v*)
Convert an integer value into a string.

std::string **rx_float_to_string** (**const** float &*v*)
Convert a float to a string.

std::vector<std::string> **rx_split** (std::string *str*, char *delim*)
Split a string on the given delimiter and return a `std::vector<std::string>` with the separate parts.

2.3 File utils

std::string **rx_get_exe_path** ()
Returns the the full path to the directory where the executable is located.

std::string **rx_to_data_path** (**const** std::string *filename*)
Returns a path to what we call a data path. A data path is a directory where you can store things like textures, shaders, fonts, etc. By default this will return a path to your executable with `data/`.

uint64_t **rx_get_file_mtime** (std::string *filepath*)
Returns the file modification time since unix epoch in nanoseconds.

bool **rx_is_dir** (std::string *filepath*)
Checks if the given path is an directory

bool **rx_file_exists** (std::string *filepath*)
Checks if the given filepath exists and returns true if it does, otherwise it will return false.

std::string **rx_strip_filename** (std::string *filepath*)
Removes the filename from the given path, returning only the full path.

std::string **rx_strip_dir** (std::string *filepath*)
Removes the path from the given filepath keeping only the filename.

bool **rx_create_dir** (std::string *path*)
Create the given (sub) directory.

bool **rx_create_path** (std::string *filepath*)
Create a complete path. This will create all the given paths that don't exist.

```
rx_create_path(rx_to_data_path("2014/01/16/"));
```

std::string **rx_get_file_ext** (std::string *filename*)
This will return the filename extension, like "jpg", "gif", etc..

```
std::string ext = rx_get_file_ext("/file/image.jpg");
printf("%s\n", ext.c_str()); // will print 'jpg'
```

std::vector<std::string> **rx_get_files** (std::string *path*, std::string *ext* = "")
Get all the files in the given path. You can specify a file extension filter like "jpg", "gif" etc..

std::string **rx_norm_path** (std::string *path*)
Creates a normalized, cross platform path. Always pass in forward slashes; on windows we will convert these to backslahes:

```
std::string normpath = rx_norm_path("/path/to/my/dir");
```

std::string **rx_read_file** (std::string *filepath*)
Read a file into a string.

2.4 Time utils

uint64_t **rx_hrtime**()
A high resolution timer in nano seconds.

```
// somewhere we have a defined a delay and timeout
uint64_t delay = 1000ull * 1000ull * 1000ull; // 1 second, 1000 millis
uint64_t timeout = rx_hrtime() + delay;

// then somewhere else you can check if this delay has been reached
uint64_t now = rx_hrtime();
if(now > timeout) {
   // Do something every second.
   timeout = rx_hrtime() + delay; // set new delay
}
```

float **rx_millis**()
Returns the time since the first call to this function in milliseconds.

std::string **rx_strftime**(**const** std::string *fmt*)
Wrapper around `strftime` which returns a a time/date.

```
std::string datetime = rx_strftime("%Y/%m/%d");
printf("%s\n", datetime.c_str()); // prints e.g. 2014/01/16
```

std::string **rx_get_year**()
Get the current year with 4 digits, eg. 2014

std::string **rx_get_month**()
Get the current month with 2 digits, [00-11]

std::string **rx_get_day**()
Get the current day of the month with 2 digits, [00-031]

std::string **rx_get_hour**()
Get the current hour with 2 digits, [00-23]

std::string **rx_get_minute**()
Get the current minutes with 2 digits, [00-60]

std::string **rx_get_time_string**()
Returns a string for the current date-time with milli second accuracy. This function is handy if you want to create unique filenames for example (as long as there is some time between each time you call this function to prevent duplicates).

```
std::string time_string = rx_get_time_string();
printf("%s\n", time_string.c_str()); // prints something like: 2014.01.16_19.01.00_328
```

2.5 Image utils

bool rx_save_png(std::string file, unsigned char* pix, int w, int h, int nchannels, bool fl
Save the given pixels to the given file path.

```
int width = 320;
int height = 240;
unsigned char* pix = new unsigned char[width * height * 3];
```

```
    // some pixel data
    for(int i = 0; i < width; ++i) {
       for(int j = 0; j < height; ++j) {
           int dx = j * width * 3 + i * 3;
           if(i < (width/2)) {
               pix[dx + 0] = 255;
               pix[dx + 1] = 255;
               pix[dx + 2] = 255;
           }
           else {
               pix[dx + 0] = 0;
               pix[dx + 1] = 0;
               pix[dx + 2] = 0;
           }
       }
    }

    std::string outfile = rx_to_data_path("test.png");

    if(rx_save_png(outfile, pix, width, height, 3) == false) {
        printf("Error: cannot save PNG: %s\n", outfile.c_str());
        ::exit(EXIT_FAILURE);
    }
```

Parameters

- **string** – `file` Full file path where to save the image
- **unsigned char*** – `pix` Pointer to the raw pixels you want to save
- **int** – `w` The width of the pixel buffer
- **int** – `h` The height of the pixel buffer
- **int** – `nchannels` The number of color components (e.g. 1 for grayscale, 3 for RGB)
- **bool** – `flip` Set to true if you want to flip the image horizontally (handy when using `glReadPixels()`)

Returns boolean true on success, else false.

bool rx_load_png(std::string file, unsigned char pix, int& w, int& h, int& nchannels, int**

Load a PNG file from the given filepath and create a pixel buffer, set width, height and nchannels. See https://gist.github.com/roxlu/9b9d555cf784385d67ba for examples on how to load using the same memory buffer.

```
    int w = 0;
    int h = 0;
    int channels = 0;
    unsigned char* pix = NULL;
    int allocated = 0; /* when given to rx_load_png, it should contain the number of bytes in `pix`,
    int bytes_in_image = 0;

    bytes_in_image = rx_load_png("test.png", &pix, w, h, channels, &allocated);
    if (bytes_in_image < 0) {
       printf("Error: cannot load the png.\n");
       ::exit(EXIT_FAILURE)
    }

    printf("Width: %d\n", w);
```

```
    printf("Height: %d\n", h);
    printf("Color Channels: %d\n", channels);
    printf("Bytes in buffer: %d\n", bytes_in_image);
```

Parameters

- **string** – file Load the png from this filepath.
- **unsigned char*** – pix (out) We will allocate a unsigned char buffer for you; you need to delete this buffer yourself!
- **int&** – w (out) Reference to the width result. We will set the width value of the loaded image to w.
- **int&** – h (out) Reference to the height result. We will set the height value of the loaded image to h.
- **int&** – nchannels (out) The number of color channels in the loaded png.
- **int*** – allocated (out,in) The number of bytes in the pix parameter. When given this must hold the correct value of the buffer size. We will try to reuse or reallocate the buffer. The allocated param may be 0 (it will just allocate a new buffer then.)

Returns true on success, else false

bool rx_load_jpg(std::string file, unsigned char pix, int& w, int& height, int& nchannels**

Loads a JPG file, see rx_load_png for an example as the function works the same, but only loads a JPG. See https://gist.github.com/roxlu/9b9d555cf784385d67ba for examples on how to load using the same memory buffer.

Parameters

- **string** – file Load the jpg from this filepath.
- **unsigned char*** – pix (out) We will allocate a unsigned char buffer for you; you need to delete this buffer yourself!
- **int&** – w (out) Reference to the width result. We will set the width value of the loaded image to w.
- **int&** – h (out) Reference to the height result. We will set the height value of the loaded image to h.
- **int&** – nchannels (out) The number of color channels in the loaded jpg.
- **int*** – allocated (out,in) The number of bytes in the pix parameter. When given this must hold the correct value of the buffer size. We will try to reuse or reallocate the buffer. The allocated param may be 0 (it will just allocate a new buffer then.)

Returns true on success, else false

bool rx_save_jpg(std::string file, unsigned char* pix, int width, int height, int nchannels

Parameters

- **string** – file Save a jpg to this filepath.
- **unsigned char*** – pix The pixels you want to save.
- **int** – width The width of the pix buffer.
- **int** – height The height of the pix buffer.
- **int** – nchannels The number of color channels. (e.g. 3).

- **int** – `quality` The quality (reasonable values are 65-100, 80 is ok)
- **bool** – `flip` Flip the given input pixels horizontally (e.g. nice when using `glReadPixels()`)
- **J_COLOR_SPACE** – `colorSpace` The JPEG color space that you pass as *pix*, by default `JCS_RGB`. Other options `JCS_GRAYSCALE`, `JCS_YCbCr`, `JCS_CMYK`, `JCS_YCCK`
- **J_DCT_METHOD** – `dctMethod` DCT/IDCT algorithms, by default `JDCT_FASTEST`. Other options `JDCT_ISLOW`, `JDCT_IFAST`, `JDCT_FLOAT`, `JDCT_SLOWEST`

Returns true on success else false

2.6 OpenGL

GLuint **rx_create_shader**(GLenum, **const** char *)

Creates a shader for the given type and source.

```
static const char* MY_VERTEX_SHADER = ""
    "#version 330"
    "uniform mat4 u_pm;"
    "uniform mat4 u_vm;"
    "layout( location = 0 ) in vec4 a_pos;"
    ""
    "void main() {"
    "  gl_Position = u_pm * u_vm * a_pos;"
    "}"
    "";

GLuint vert = rx_create_shader(GL_VERTEX_SHADER, MY_VERTEX_SHADER);
```

Parameters

- **GLenum** – What kind of shader to create `GL_VERTEX_SHADER`, `GL_FRAGMENT_SHADER`
- **const char*** – Pointer to the shader source

Returns GLuint, the created shader.

GLuint **rx_create_shader_from_file**(GLenum, std::string)

Creates a shader for the given type and filepath.

```
GLuint vert = rx_create_shader_from_file(GL_VERTEX_SHADER, "my_shader.vert");
```

Parameters

- **GLenum** – What kind of shader to create `GL_VERTEX_SHADER`, `GL_FRAGMENT_SHADER`
- **string** – The filepath of the shader to load

GLuint **rx_create_program**(GLuint *vert*, GLuint *frag*, bool *link* = false)

Create a shader program from the given vertex and fragment shaders. Set `link` to true if you want to link the shader program as well. Sometimes, especially when using GLSL < 330, you want to bind the attribute locations in your shader. In this case you'll pass `link = false`. Otherwise, when using version 330 you can use the `layout( location = 0 )` directives.

Parameters

- **GLuint** – vert The vertex shader.
- **GLuint** – frag The fragment shader.

Returns GLuint We return the newly created shader program id (not linked).

GLuint **rx_create_program_with_attribs** (GLuint *vert*, GLuint *frag*, int *nattribs*, **const** char ***attribs*)

This function is similar to rx_create_program except that it will bind the attribute locations for you. The indices of the given attribs array are used as bind locations. The example below will bind a_pos at index 0, a_tex at 1 and a_col at 2. This function will also link the shader.

```
const char* attribs[] = { "a_pos", "a_tex", "a_col" } ;
GLuint prog = rx_create_program_with_attribs(vert, frag, 3, attribs);
```

Parameters

- **GLuint** – vert The vertex shader.
- **GLuint** – frag The fragment shader.
- **int** – nattribs The number of attributes in the attribs array.
- **const char**** – attribs The attributes that you want to set.

Returns GLuint A linked program.

GLint **rx_get_uniform_location** (GLuint *prog*, std::string *name*)

Safe way to retrieve uniform locations. When compiled in debug mode, this function will make sure that the uniform is found and will cause an assertion if we cannot find the uniform (which often means it's optimized away and thus not used in the shader). Make sure that your shader is active (glUseProgram(prog)).

Parameters

- **GLuint** – prog The shader program.
- **string** – name The name of the uniform for which you want the location.

Returns GLint The location of the uniform (-1 on failure).

void **rx_uniform_1i** (GLuint *prog*, std::string *name*, GLint *v*)

This function will set the uniform with the given name to v. This function is to be used in your setup routines. It's an easy wrapper for e.g. settings uniform locations for your texture samplers.

Parameters

- **GLuint** – prog The program that contains the name uniform.
- **string** – name The name of the uniform you want to set.
- **GLint** – v The value you want to set.

Returns void

void **rx_uniform_1f** (GLuint *prog*, std::string *name*, GLfloat *v*)

Similar to rx_uniform_1i ecxept you use this to set float values.

Parameters

- **GLuint** – prog The program that contains the name uniform.
- **string** – name The name of the uniform you want to set.
- **GLfloat** – v The value you want to set.

Returns void

void **rx_uniform_mat4fv**(GLuint *prog*, std::string *name*, GLint *count*, GLboolean *transpose*, GLfloat **value*)

Wrapper around *glUniformMatrix4fv*. This function will set the given matrix.

Parameters

- **GLuint** – `prog` The program that contains the `name` uniform.
- **string** – `name` The name of the uniform you want to set.
- **GLint** – `count` Number of matrices to set.
- **GLboolean** – `transpose` Transpose the matrix.
- **const float*** – `value` Pointer to the matrix.

Returns void

bool **rx_create_png_screenshot**(std::string *filepath*)

Creates a PNG screenshot of the current read buffer with the size of the current viewport. The image is saved to the given `filepath`. Note that this function allocates some static memory so we don't have to allocate every time you create a screenshot. This does mean that we "leak" a couple of bytes and that you need to be carefull calling this function from different threads at the same time.

NOTE: This means that both `ROXLU_USE_OPENGL` and `ROXLU_USE_PNG` must be enabled.

Parameters **string** – `filepath` The file path where you want to save the screenshot.

Returns boolean, true on success else false.

bool **rx_create_jpg_screenshot**(std::string *filepath*, int *quality*)

Same as `rx_create_png_screenshot()` but this creates a JPG image. Writing JPGs to disk is often faster then writing PNG images.

NOTE: This means that both `ROXLU_USE_OPENGL` and `ROXLU_USE_PNG` must be enabled.

Parameters

- **string** – `filepath` The file path where you want to save the screenshot.
- **integer** – `quality` The quality level for the JPG.

Returns boolean, true on success else false.

2.7 Math

2.8 Audio

- genindex
- search

R