miniGL Manual

Getting Started:

Dependencies:

miniGL needs a few libraries to help it work with openGL. These are:
GLFW : https://www.glfw.org/
GLEW : https://sourceforge.net/projects/glew/postdownload
GLM : https://github.com/g-truc/glm

Linux Setup:

Although it's possible to set up miniGL on windows, mac, and linux, linux packages are by far the simplest to install. To install dependencies on linux, simply run:
`sudo apt install libglfw3-dev libglm-dev libglew-dev`

Building the Demos:

If all is set up correctly, running `make` should generate a bunch of demo executables that show various aspects of the program. `fluids` and `pong` are some of the most exciting.

Writing Your Own Code:

miniGL is meant to be used in programs you create. After checking out the demos, you can try messing around with your own miniGL applications in `sandbox.cpp` or modifying any of the demos. The bulk of this manual is dedicated to descriptions of how to use the miniGL library functions to create programs.

miniGL Base (minigl.hpp)

`class minigl::window`

The window class manages the openGL rendering context and display window associated with a miniGL program. Any programs using render_pipelines directly should create a window before creating any pipelines.

window::window() = delete;
window::window(const window&) = delete;
window::window(pixels w, pixels h, std::string name)

Creates an openGL rendering context.

w - window width in pixels
h - window height in pixels
name - name to be displayed on window

bool window::ok(void)

Returns true if the window was successfully initialized, false otherwise.

Void window::set_background_color(color c)

Sets the background color of the window.

c - color for the background

glm::vec2 window::cursor_pos(void)

Returns a 2D vector (not to be confused with std::vector) with the pixel position of the cursor in the window. Coordinates start at (0,0) in the top left corner, x increases left to right, y increases top to bottom. All values within the window will be positive and less than window and height. Moving the cursor out of the window will give negative values or values greater than width and height.

int width(void)

Returns the width of the window in pixels.

int height(void)

Returns the height of the window in pixels

template <typename C, typename... R>
requires std::invocable<C, R...>
void render(C oper, R &&...args)

Begins rendering a window, this will block program execution until the window has been closed or an exception has been thrown.

oper - An invocable object that will be invoked once per rendered frame. This is a good place to call render functions or update parameters.

args - A variable length list of arguments to be passed to oper when it is executed.

`class minigl::shader`

The shader class is used to define shader programs to run on a graphics unit. These programs must be partially composed in GLSL, the openGL shader language. If you are unfamiliar with this shading language it is recommended that you first take a look at some of the demos provided with miniGL to understand the basics. They're not as scary as they look.

template <typename T>
concept valid_uniform

The valid_uniform concept specifies types that are valid uniform inputs to a shader program. A uniform input is an input that is consistent across an entire rendered image, so for example the color of a monochrome plane, or a matrix to transform an object's position.

template <typename T>
concept valid_vertex_attribute

The valid_vertex_attribute concept specifies types that are valid attribute inputs to a shader program. An attribute input is different for every vertex in an object, for example the position of points in a cube.

shader::shader() = default
shader::shader(const shader&) = default
shader::shader(shader_types t)

Constructs a shader object. Shaders can be either vertex or fragment shaders. Generally, vertex shaders run on sets of vertex_attributes. A simple triangle might for example have 3 vertices each with its own position. A vertex shader would run on each of the 3 vertices in the triangle and compute where that vertex will end up on the screen. A fragment shader is run once for every pixel in the final image that gets shown on the screen, it takes the transformed position from the vertex_shader and uses that to compute which pixels are within the defined shape.

t - either shader_types::vertex or shader_types::fragment

template <typename T>
requires valid_uniform<T>
void add_uniform(std::string name)

Creates a uniform input of the specified name and type. This named parameter can be referenced later in the body of the shader and can also be assigned once the shader is compiled into a render pipeline.

name - name of the uniform input

template <typename T>
requires valid_vertex_attribute<T>
void add_attribute(std::string name)

Creates an attribute input of the specified name and type. This named parameter can be referenced later in the body of the shader and can also be assigned once the shader is compiled into a render pipeline. Attribute parameters can only be added to vertex shaders, fragment shaders do not operate on vertex attributes but rather on texel or pixel fragments.

name - name of the attribute input

void define_shader(std::string body)

Defines the GLSL body of a shader to be compiled, specified attribute and uniform inputs can be referenced and parameters to be passed between shaders can be defined as well. As an important note, vertex shaders output position into gl_Position variable, additionally fragment shaders must set at least one output in order to draw their result to the screen or a texture.

body - a string containing the shader body.

Shader Example Use

```
shader vertex_shader(shader_types::vertex),
fragment_shader(shader_types::fragment);
vertex_shader.add_attribute<glm::vec2>("vertex_position");
vertex_shader.define_shader(R"(
    void main() {
        gl_Position = vec4(vertex_position, 1);
        frag_color = vertex_color.rgb;
    }
)");

fragment_shader.add_uniform<color>("shape_color");
fragment_shader.define_shader(R"(
    out vec3 color;
    void main() {
        color = shape_color;
```

```
    }
)");
```

This snippet defines a pair of shaders for rendering colored shapes, note the output of color in the fragment shader and the setting of vertex position in the vertex shader.

class render_pipeline

The render_pipeline class is a wrapper for shader programs running on the GPU. It can create shader programs, set their arguments, and run them on certain targets. Initializing a render pipeline must be done after a window is created, otherwise the program will swiftly crash as the internal openGL calls have nothing to interact with.

render_pipeline::render_pipeline()

Creates an empty render pipeline object, it must be initialized later using render_pipeline::init() for the object to perform any useful functions.

render_pipeline::render_pipeline(const render_pipeline&) = delete

render_pipeline has no copy constructor. This is because the object doesn't store data that is passed into it, it merely passes the data along to the GPU and tracks the handles for the various resources created. If you need two copies of a render pipeline, initialize each with the same parameters and pass both the same arguments. It's worth noting in most cases this will simply draw two shapes directly on top of each other.

render_pipeline::render_pipeline(render_pipeline&& old);

render_pipeline implements a move constructor. Moved from pipelines should not be used, doing so will result in undefined behavior.

render_pipeline::render_pipeline(const shader &vert_shader, const shader &frag_shader)

Creates a render pipeline based on the shader definitions provided. This constructor compiles the shaders, sends them to the GPU, and creates a host of GPU-side resources needed for rendering. This constructor can fail, and will throw and std::runtime_error specifying what failed during construction (usually a shader compilation issue).

vert_shader - shader object defining the vertex shader
frag_shader - shader object defining the fragment shader

void render_pipeline::init(const shader &vert_shader, const shader &frag_shader)

Initializes an empty render_pipeline (default constructor). Does nothing if a pipeline is already initialized.

vert_shader - shader object defining the vertex shader
frag_shader - shader object defining the fragment shader

bool render_pipeline::ok(void)

Returns true if the pipeline is successfully initialized, false otherwise.

pipeline_helper render_pipeline::operator[](std::string name)

Returns a helper object that can be assigned to in order to modify the attribute or uniform associated with the name string. These associations come from shader definitions.

name - name of the shader program input to update

template <typename T>
requires valid_vertex_attribute<T>
pipeline_helper &operator=(const std::vector<T> &new_value)

Sets a vertex attribute value in the shader program of the render_pipeline that generated this object.

new_value - a vector of vertex attributes to be passed to a shader

template <typename T>
requires valid_uniform<T>
pipeline_helper &operator=(const T &new_value)

Sets a uniform value in the shader program of the render_pipeline that generated this object.

new_value - a uniform value to be passed to a shader.

void render_pipeline::render(pixels w, pixels h, float x_center = 0, float y_center = 0)

Runs the shader associated with a render_pipeline to draw an image to the active display window. Users can specify which region of the window they would like to display to.

w - the width of the active display region
h - the height of the active display region
x_center - (optional) x offset of the active region within the larger window
y_center - (optional) y offset of the active region within the larger window

render_to_texture_helper render_pipeline::render_to_texture(pixels width, pixels height)

Generates a helper object that can be assigned to a pipeline_helper in order to render shader output data into a texture that can be reused by another pipeline.

width - width of the texture to render to in pixels
height - height of the texture to render to in pixels

pipeline_bracket_helper &operator=(render_to_texture_helper helper)

Renders the render_pipeline that generated render_to_texture_helper into a texture in the render_pipeline that created pipeline_bracket_helper. These complicated helpers enable a very simple syntax, so see the render_to_texture example below.

helper - a render to texture helper generated by render_pipeline::render_to_texture

render_pipeline Example Use (continued from Shader Example)

```
render_pipeline pipeline(vertex_shader, fragment_shader);
pipeline["vertex_postion"] = vertexPositions;
pipeline["shape_color"] = color(colors::red);
my_window.render([&](){
    pipeline.render(my_window.width(), my_window.height());
});
```

This snippet defines a pipeline with the basic colored-shape shaders defined earlier. This snippet is all that is required to draw the shape to the screen. vertexPositions simply contains the locations of however many vertices the given shape has.

render_to_texture Example Use

```
tv_object["input"] = camera_pipeline.render_to_texture(width, height);
tv_object.render(width, height);
```

This syntax is possible with the render_to_texture_helper and pipeline_helper objects. In this case an object styled as a virtual TV (possibly for some sort of 3D application) with a uniform texture input called “input” is receiving a rendered image from a camera object that is presumably looking at another object or the same objects from a different angle. Although the details of this tv_object are left open-ended in this example, if a user were going for a TV like effect the texture would displayed on the front of the tv, and then the TV would be drawn into the final scene, which could be a living room or a surveillance office or anything of the sort.

class shape

This is the core abstraction of the miniGL2d API. `shape` provides basic operations to manipulate its attributes. This class hides away details such as glsl scripts and matrix manipulations to create a simple abstraction for users who may not be familiar with graphics programming. The shape class works in conjunction with the `render2d` class to create interactive animations and draw to the screen.

shape::shape (std::vector<glm::vec3> base_vertices, std::vector<color> base_fragments, pixels unit_len, position pos, std::vector<glm::vec2> uv, bool enable_tex, texture tex);

This is the primary constructor used when defining a shape that supports textures.

> **`base_vertices`** - defines the triangles of a shape. Size must be a multiple of 3.
> **`base_fragments`** - defines the fragment data that corresponds to `base_vertices`.
> **`pixels`** - size in pixels of 2 units of length in `base_vertices`.
> **`pos`** - initial position of the shape.
> **`enable_tex`** - set to true if texture should be attached by default.
> **`uv`** - the uv mapping for textures relative to `base_vertices`.
> **`tex`** - the default texture.

shape::shape (std::vector<glm::vec3> base_vertices, std::vector<color> base_fragments, pixels unit_len, position pos);

This is the primary constructor used when defining a shape that doesn't support textures.

> **`base_vertices`** - defines the triangles of a shape. Size must be a multiple of 3.
> **`base_fragments`** - defines the fragment data that corresponds to `base_vertices`.
> **`pixels`** - size in pixels of 2 units of length in `base_vertices`.
> **`pos`** - initial position of the shape.

void shape::translate(position pos);

Moves the position of the shape in pixels relative to its current position. For example, if the current position of the shape is (400px, 505px), then calling `translate(position(50,-25));` Will move the shape to the position (450px, 480px). For the default shapes `circle`, `rectangle` and `triangle`, the position of the shape corresponds to the center.

void shape::set_pos(position pos);

Sets the position of the shape on the screen in pixels. For the default shapes `circle`, `rectangle` and `triangle`, the position of the shape corresponds to the center.

position shape::get_pos();

Returns the position of the shape. For the default shapes `circle`, `rectangle` and `triangle`, the position of the shape corresponds to the center.

void shape::scale(glm::vec2 s);

Scales the shape by a factor given by `s`.

void shape::attach_tex(texture tex);

Attaches texture tex to shape. Note that the texture must be supported for the shape. After attaching a texture, the texture will be overlaid on the shape in a way defined by the `uv` field of `shape`. The behavior of this function is undefined if the shape does not support textures. Currently, `rectangle` is the only shape that supports textures. Custom shapes can also be defined to support textures (see `shape` constructor for more details).

class render2d

This class provides static functions to be able to display `shapes` on the screen. This uses our `render_pipeline` abstraction as the underlying mechanism in order to achieve this. The goal with this class is to provide functions that make it easy to display and interact with shapes without presuming understanding of graphics programming.

static void render2d::draw(const window2d& win, std::vector<shape> shapes)

Displays a vector of shapes onto a defined window.

> **win** - the window class that defines background color, size, and title of the window.
> **`shapes`** - vector of the shapes that will be displayed.

static void shape::animate(const window2d& win, int fps, std::vector<shape> shapes, std::function<void(std::vector<shape>&, events)> func);

This function should be used when creating programs that animate and (optionally) listen to events.

win - the window class that defines background color, size, and title of the window.
fps - the number of times a second that `func` will be called.
shapes - the initial vector of all the shapes that will be displayed.
func - a function that updates the shapes vector per frame

It is recommended that func is defined as a lambda so that external parameters used can be used as state. `func` also takes in an `events` field that allows for listening of keyboard inputs and mouse position. Currently, the following are the supported fields events that can be checked:

```
struct events
{
    bool pressed_keys[NUM_KEYS];
    position cursor_pos;
    bool left_click;
};
```

For `pressed_keys` the following are what is currently supported:

```
E_KEY
A_KEY
D_KEY
W_KEY
S_KEY
SPACE_KEY
Q_KEY
LEFT_KEY
RIGHT_KEY
UP_KEY
DOWN_KEY
```