miniGL

Bringing C++ to OpenGL

Overview

miniGL is a library built on top of openGL, which itself is an API for low-level, high-performance computer graphics. Defined back in the 90's and maintained by Khronos Group since 2006, openGL is (or was) the library of choice for developers of performance-focused graphical programs: games, game engines, simulators, and anything that needs to interact with GPU's.
It is complicated, verbose, and very expert-friendly. Most libraries that are built on openGL usually target a set of applications and hide as much of the messiness as possible, and miniGL does a bit of that too; it offers some high-level functionality for making shapes and responding to user inputs that can make some simple, fast, and useful programs.
But the main intent behind miniGL is to show that some of the messiness in openGL might not need to exist at all. C++ abstractions are powerful and have become more so in recent years. If it wasn't before, it should now be possible to tame the beast and turn openGL into something simpler but just as powerful.

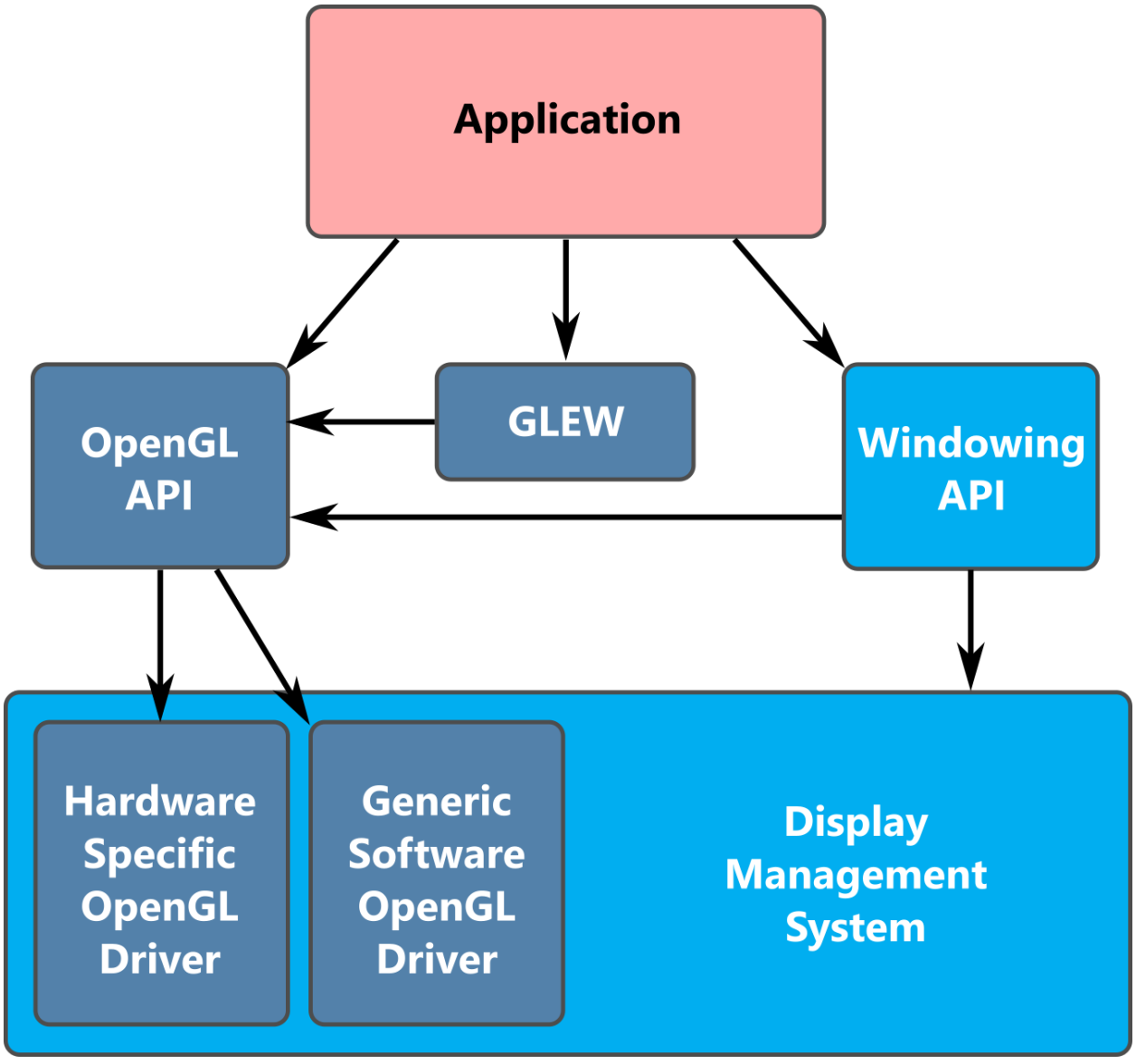
Design Goals

- **Make simple things simple** - Total beginners should be able to make simple programs easily, with very few lines of code and almost no graphics knowledge. Ideally these applications should be things like pong, the original mac_paint, and other simple games and interfaces
- **Make hard things less hard** - People with graphics experience or who are interested in learning more complicated functionality should be able to apply their expertise without having to write raw opengl.
- **Be Fast** - High performance is one of the main reasons people use GPUs in the first place. Wherever possible make as little difference in speed as possible.
- **Be Typesafe** - Type safety is almost nonexistent in parts of opengl, and typing often needs to be tracked manually across multiple files! Fix this.

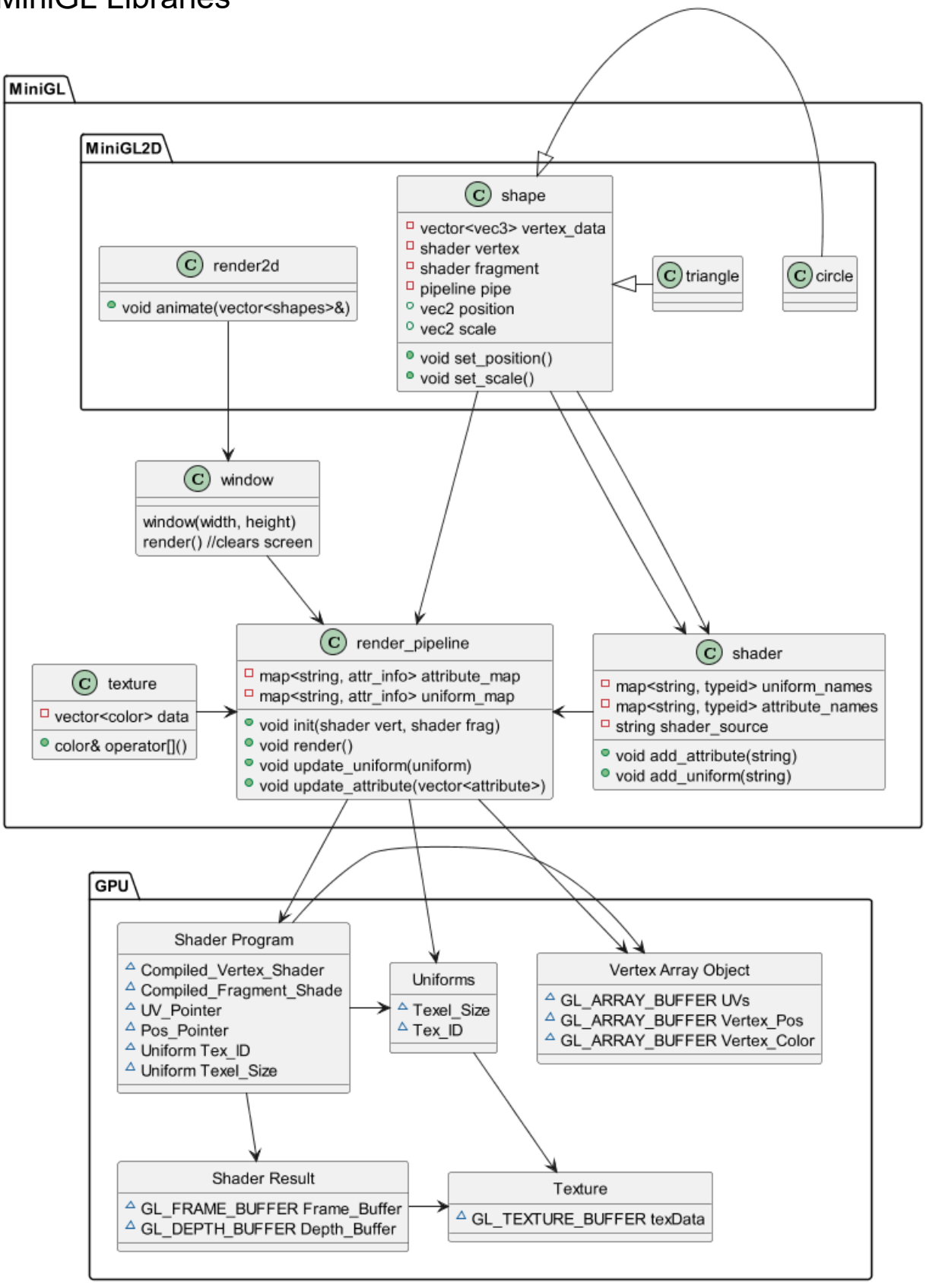
External Libraries

Before we can get started we need some tools to help us. We'd like our library to have zero dependencies but for this kind of turn-around time that's simply impossible, so we'll be using some libraries to help us.

- **GLM** - openGL Mathematics is a lightweight library with convenient flexible types that can be fed into openGL with minimal overhead.
- **GLFW** - A GL framework library that does the hard work of setting up windows and initializing a rendering context on multiple platforms.
- **GLEW** - The GL extension wrangler that hunts down and collects various function pointers in a given system's graphics hardware. This lets us use more up-to-date versions of openGL.



MiniGL Libraries



Design Process

Fundamentals - Abstracting the Shader

The main reason that openGL code is so unfriendly is that most graphical applications have a significant component that runs on specially designed graphics processing units (GPUs). This is an entirely separate chip on most modern systems, and it runs its own programs in its own language which is substantially different from C++ or anything like it. A program that runs on a GPU is called a shader, and making a good abstraction for one has many complications:

Problems with Shaders

- Shaders are compiled at runtime, and the result of compilation is loaded into the GPU at run time. This is not something we can fix directly, although `constexpr` compilation is certainly possible, it's beyond the scope of this project.
- Shaders are written in GLSL, which is not C++ and can not be easily converted to or from C++. Although writing a metalanguage is certainly possible it would again be beyond the scope of this project, and is ill-advised regardless.
- Shaders store data in graphics memory, which can only be accessed through several calls of setter and getter functions in openGL. Setting is generally easier than getting.
- Shaders and their parameters are accessed by arbitrary integers that must be matched across GLSL and C++. This is an absolute paradise for tricky, annoying, time consuming bugs where IDs and types are mismatched.
- All memory and resources used by shaders must be initialized and destroyed, and if they are not the negative effects can persist after a program closes!

First Steps: RAII

The general flow to create a shader resource looks something like this:

```
GLuint arr_id;
glGenVertexArrays(1, &arr_id);
glBindVertexArray(arr_id);
/* Do some stuff */
glDeleteVertexArrays(1, &arr_id);
```

A constructor and destructor pair wraps this nicely, but what's more important here is that not calling glDeleteVertexArrays() can lead to problems that persist even after the program crashes. So we must make sure that when errors occur destructors are still called. So crashing on unmanageable errors is not an option. Given how sensitive the GLFW execution environment can be, errors must be handled elegantly, and probably the easiest way to guarantee object destruction is to use exceptions for most error types and cases that can't be handled elegantly.

Shader Data Ownership

We would like to create abstractions that let users own data and programs on the GPU. But the specifics of what classes are needed and what owners actually store is a good deal more complicated.

Development of Render Pipeline

The first naive attempts at getting some abstractions off the ground made objects out of every component that shaders could use. This included frame buffers, depth buffers, textures, vertex buffers, vertex array buffers, and so on. Issues arose with this approach, first was that these abstractions offered very little improvement over plain openGL, saving one or two lines was not transformative, it was more cosmetic.

So the next attempt features some larger objects that could bundle together all of the necessary information to do stuff with shaders, but that looked something like this:

```
class attr_bundle
{
    std::map<string, variant<vec2, vec3, vec4, mat2, mat3, mat4, texture>>
attributes;
    std::map<string, variant<vec2, vec3, vec4, mat2, mat3, mat4, texture>>
uniforms;
    void set_attribute(string name, vector<vec2> values);
};
```

This isn't that bad. Users can now put native types into the shader space with one function. They can also name these values to be referenced or updated later. But there's still some problems. Storing bundles of values to be eventually written into a shader once it's compiled is not very efficient. The user will likely have relevant values in memory anyway, so storing them again and copying them can be a huge waste. Any user writing in C++ will likely already have their data contained in formats that are comfortable and useful in C++.

So the final iteration is something called a render_pipeline, an object that contains only the information necessary to send and receive information from a shader. The functions of this pipeline are essentially just setting a shader's inputs, running it, and collecting the outputs. The member functions of this object, although they have a few different overloads, boil down to:

- `init()`
- `update_parameter()`
- `render()`
- `render_texture()`

These may seem too simple to communicate with a complex thing like a shader, but types, templates, and concepts let us create overloads that optimize for a variety of situations automatically based on the types of inputs users provide.

Type Safety and Shader Definitions

In order for types to behave nicely with shaders we have to define a mapping from C++ types to shader types, and more importantly we need to enforce it absolutely. Fortunately, it's actually pretty easy to determine variable types from a GLSL program. They all start like this:

```
#version 330 core
out vec3 color;
in vec3 fragmentColor;
void main() {
    color = fragmentColor;
}
```

Since shader programs get compiled at run time miniGL can simply read the definitions and figure out the appropriate type. We can also do this in reverse, and let users define typed inputs for the shader programs they write. This makes using shaders simple enough that in many cases we can simply write the body of a shader in a C++ file directly:

```
shader vertex_shader(shader_types::vertex);
vertex_shader.add_uniform<glm::mat4>("MVP");
vertex_shader.add_attribute<glm::vec3>("vertex_position");
vertex_shader.add_attribute<color>("vertex_color");
vertex_shader.define_shader(R"(
    out vec3 frag_color;
    void main() {
        gl_Position = MVP * vec4(vertex_position, 1);
        frag_color = vertex_color.rgb;
    }
)");
```

There's still a bit of mystery here. For example the out vec3 frag_color line is an output to another shader which will have in vec3 frag_color in its header. These params between shaders never leave the confines of the GPU and are simple enough and central enough to GLSL that they're worth leaving in. On the bright side, we've managed to capture type information, and there's no need to look between files to make sure names and numbers are matched.

Shaders are in turn used to define render pipelines, so after we define the types they need to be transferred into the pipeline object. It may be possible to get compile-time type information through to a render pipeline from a shader like this, but it would require breaking pipelines into constexpr and non-constexpr implementations with different behaviors. We'll have to settle for run time checking, and we do this by associating an std::type_index with parameter names. So after putting this shader into a pipeline we get this:

```
pipeline.update_uniform("MVP", mat4(1.0f));
```

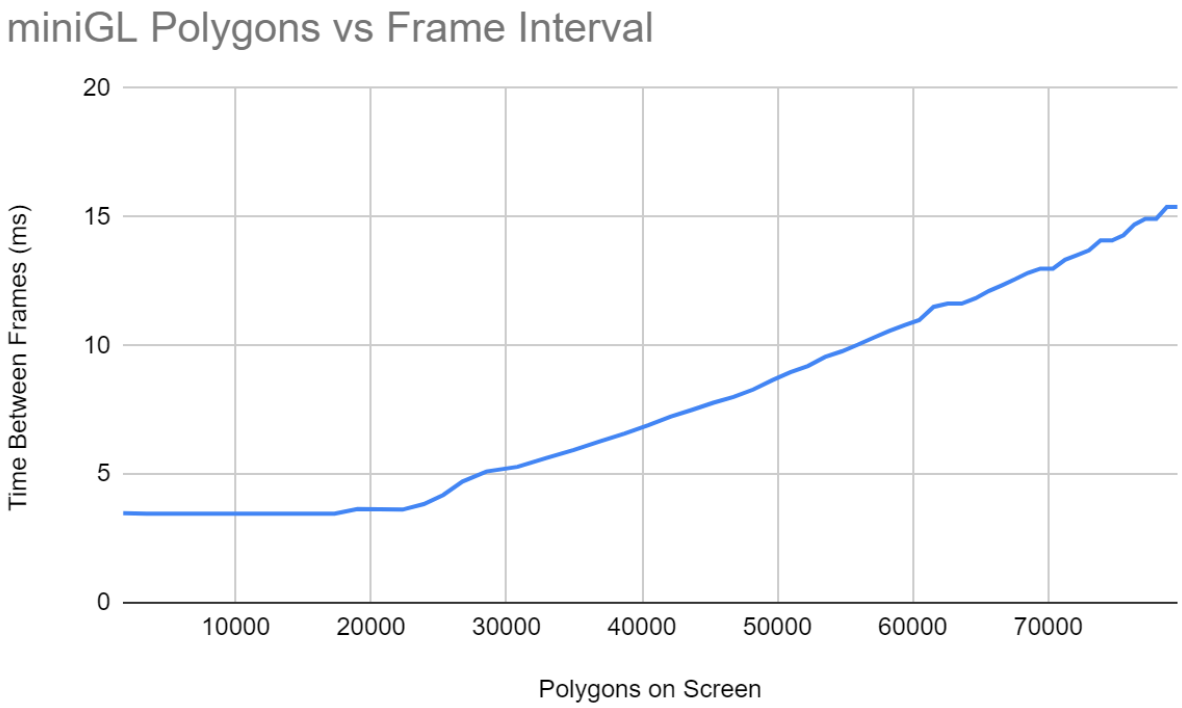
Limits of Render Pipeline

Early in development, render pipelines were conceptualized as connecting one set of GPU resources to one shader program with one output. This is not necessarily how shaders have to work. Any number of objects can be fed to any number of shaders and produce any number of outputs. There's no reason render_pipelines can't accommodate this, but it would have left little development time for anything else. A few tests were run to make sure that this 1 to 1 approach for the render pipeline could still perform reasonably well.

Test Parameters

We created a program that would generate a new render pipeline every frame and configure that pipeline to draw a simple cube made up of 12 triangles. Time between frames was measured as the total number of cubes on screen increased. 3 trials were run.

Results



We were quite happy with the results we saw. Time seems to grow linearly with the number of polygons on screen after a brief period of no movement likely caused by a hardware frame limit. The important thing here is that despite duplicating shaders for each of the cubes being drawn, the program still performed reasonably well. Although again, it would be nice to be able to support multiple objects per shader or multiple shaders per object, our abstraction is good enough to keep running forward for now.

To clarify what we mean by good enough, this cool dragon on the unity asset store made by 3dFoin consists of 5756 polygons.

This dragon could be contained by a single pipeline, and our demo was able to render roughly 80,000 polygons across nearly 6,000 pipelines. 15 Dragons is probably enough for our purposes.

Final Steps: Render to Texture and Helpers

The last shader feature added to miniGL is something called render to texture. Rendering to a texture is not very different from rendering to the screen. The shader takes in object parameters, does some math, and spits out an array of colors. Except instead of sending those colors to a screen, they are saved in memory as a texture that can then be reused. It can be used to do all sorts of things like light-maps and post-processing, and it's necessary to make a lot of fun demos we wanted to do.

The technical details of rendering to texture involve linking the output of one program to an input texture of another. This binding is relatively simple, but it is most efficient when it is done once to link a pair of pipelines together permanently rather than being called every frame. Up until this point, pipeline objects were rendered once per frame, so we introduced a way to bind pipelines for render to texture:

```
void pipeline_connect(render_pipeline &receiver, string rname,
render_pipeline &sender, string sname, width , height)
```

This worked but was absolutely terrible to use, every action with render pipelines up until this point could be done per frame and without much forethought from a programmer. But now programmers had to thoughtfully link a chain of pipelines before they started using them, and overriding a render to texture with a standard texture would break the link.

Eventually a better approach arose:

```
life_pipeline["tex"] = plane_pipeline.render_to_texture(width, height);
```

This is a one time connection that simply outputs the result of a render to a texture, so this can be done on a per frame basis, which is good. We use a helper class returned by render_pipeline::operator[] to get a very clean and intuitive signature which is also quite nice.

The downside of this way of doing things is that on each frame we must link the pipelines anew. And indeed it shows in performance measurements:

Experiment

One exciting application of miniGL that showcases the above issue well is Conway's game of life. It's a simple algorithm that operates on a large grid to simulate a cell-like system that grows and evolves in interesting ways.

It's also perfect to test render to texture, shaders are particularly good at doing many things in parallel and a 2D texture is an excellent choice for storing the game grid. We can compare the time to compute one step in the game from the old pipeline_connect to the new render_to_texture. Running 3 trials of 100,000 iterations on a 50 X 50 grid we get:
pipeline_connect: 0.91 us per iteration
render_to_texture: 73.01 us per iteration

These results were super disappointing, we don't want to use the pipeline_connect syntax but we need to spare ourselves the cost of binding pipelines every frame.
The solution, as the astute reader may have already guessed, is simply to cache the ID of the last texture rendered to (openGL supplies unique texture IDs). This way, if on a subsequent frame we are asked to render to the same texture again, we can skip the binding step and simply perform the render. Running the experiment again with the same optimization we get the following results
pipeline_connect: 0.90 us per iteration
render_to_texture: 0.91 us per iteration

We're able to get the best of both after all. For fun, we can compare the miniGL game of life to a basic C++ version that does the processing on the CPU rather than the GPU. Both were run on the same device with the same compiler settings for 100,000 iterations on a 50x50 grid. The results speak for themselves.

Metric	Raw C++ Version	MiniGL Version
Lines of Code	167	163
Time for 1 Life Iteration	16.96 us	0.91 us

Nathan Cuevas, Emily Rhyu, Jackson Welles

miniGL2d

miniGL also provides a library focused specifically on rendering of 2d objects. This is useful for those who prefer to completely abstract away low level details such as shader scripts, buffers and projection matrices that were prevalent in our previous abstractions. The emphasis with this abstraction is not necessarily performance but ease of creating 2d renderings and programs. Although this API is high level, we still offer the user control over some low level details like textures and vertices if they wish. We named this high level api miniGL2d and can be included through minigl2d.hpp. We anticipate that miniGL2d will be used for creating applications like simulations, games, visualization tools and animations.

The Shape Construct

The miniGL library revolves around this construct known as the `shape`. All of the objects rendered on the screen using miniGL2d will be based on some underlying `shape` object. There are default shapes that are provided by the library such as `circle`, `triangle`, `rectangle`; however, the user can also define their own by specifying the vertices themselves. The idea behind the shape object was so that the user can quickly define one without having to worry about window contexts and scaling to the correct size. For example, to declare a circle, all one has to do is `circle my_circle(100_px, colors::blue, position(-20,50))`; This particular line creates a circle that is 100 pixels in radius and is blue centered at position (-20,50) in the screen. Note that the task of scaling the vertices to the appropriate size based on the window is completely abstracted away from the user, something that is a tedious task when working with raw openGL.

To define a shape, the vertices of the triangles have to be provided. Recall that all shapes rendered in openGL are created by defining triplets of 3d coordinates. To define a shape, these vertices (call them position vertices) must be provided, as they specify the exact relative dimensions of the shape. For instance, the triangle shape is defined by the following vertices.

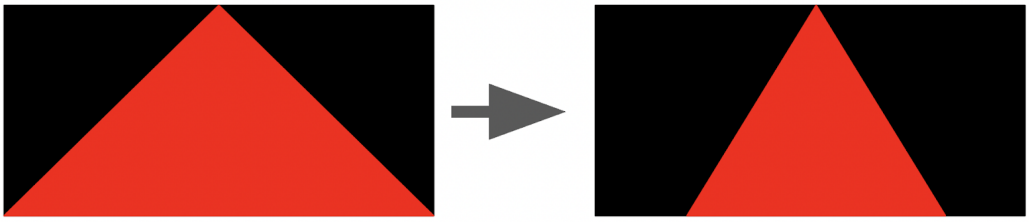
```
std::vector<vec3> {
  {-1.0f, -1.0f,  0.0f},
  {1.0f,  -1.0f,  0.0f},
  {0.0f,  0.732, 0.0f},
},
```

Where each of the vec3 elements define a point in space. Notice that this triplet of coordinates when connected together in space creates an equilateral triangle. The circle in miniGL2d is actually defined as an 18 sided polygon, resulting in a vector of vec3's of size 54. These vertices are stored in the shape object in a field called `base_vertices`. Like mentioned before there are built-ins for `circles`, `rectangles` and `triangles` and the user only has to define their own vertices if they don't want to use one of those shapes.

Now that we have the vertices of a shape defined, there is the challenge of scaling it to the correct size that the user expects in the window. This is done in 3 steps. (1) scale to window, (2) scale to size, (3) translate. See the visualization of these steps below:

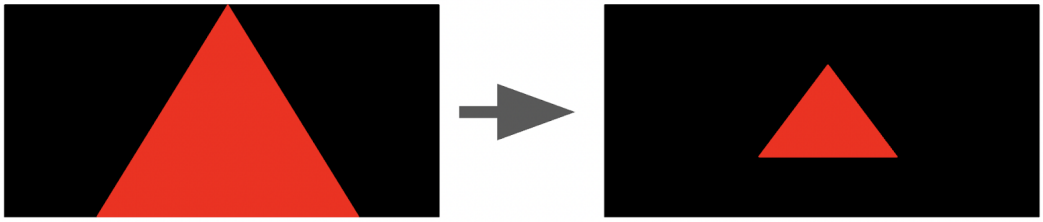
Scale to proportion

If we just feed in base_vertices to openGL, it will simply stretch it to fit the window, with no regard of whether the shape is still in proportion or not. This first step ensures that we rescale the vertices so that it maintains its desired shape. With the case of the equilateral triangle, we want all sides to be of equal length.



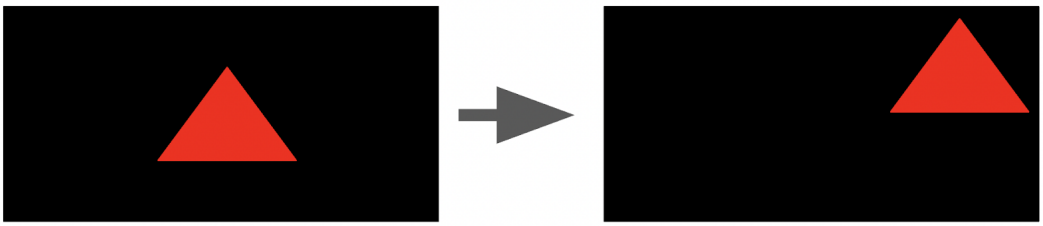
Scale to size

Now that we have the correct proportions of our shape, we need to scale it to the desired size. For example, maybe the user wants this equilateral triangle to have a side length of 50 pixels.



Translate

Finally we have to translate the vertices to the correct position based on the shape's internal position, for example maybe the user wants the center triangle at a window position of (200pixels, 105pixels).



All of this vertex manipulation is done with the help of a predefined vertex shader glsl script in the `shape` object. This glsl script is shown below.

```
out vec2 frag_uv;
      void main()
            {
                  gl_Position = MVP * vec4(vPosition, 1.0);
                  frag_uv = uv;
            }
```

Where all of the 3 transformations we defined above are accumulated in the MVP or “Model View Projection” matrix. These calculations are done for every shape for each frame before they get rendered.

`rectangle`, `circle` and `triangle` are predefined, but the user is also free to define their own shapes. To define your own shape you must inherit from the `shape` base class and call the `shape` base class from the initializer list of the new shape. As an example, below is the `triangle` constructor.

```
triangle::triangle(pixels side_len, color col, position pos) :
    shape(
        std::vector<vec3> {
            {-1.0f, -1.0f,  0.0f},
            {1.0f,  -1.0f,  0.0f},
            {0.0f,  0.732, 0.0f},
        },
        get_color_vec(col, 3),
        pixels(side_len.px),
        pos
    )
{
}
```

Notice that `shape()` is called from the initializer list. The first parameter is a vector of vec3’s that define the shape of the triangle, we call these the vertices. In this case, the vertices specify the shape of an equilateral triangle. Note that the size of this vector has to be a multiple of 3. This is because this vector defines the collection of triangles that define the shape. The second param to the `shape` constructor is the fragment data. Simply call `get_color_vec()` with the color of the shape desired along with the size of the vector in the first parameter. If the user wants to define their own color gradients instead of having a solid color, they can manually define their own fragment data here. The third parameter is the size (in pixels) of 2 units of the vertex vector. This value is used to scale the shape appropriately depending on the pixel size that is desired. For example, consider the above code for the triangle, if we want a triangle whose base is 5px wide We want to pass in 5px into the third param because a single side in the vertex vector is 2

units long. The last param is simply the position of the shape. The origin can be specified by calling the default `position` constructor.

The miniGL2d API is simple, but very expressive. The following essentially defines the entire API:

miniGL2d API

The miniGL2d API is simple, but very expressive. The following essentially defines the entire API:

```
void shape::translate(position pos);
void shape::set_pos(position pos);
void shape::scale(vec2 u);
void shape::attach_tex(texture tex);

void render2d::draw(const window2d& win, vector<shape> shapes);
void render2d::animate(
      const window2d& win,
      int fps,
      vector<shape> shapes,
      function<void(vector<shape>&, events> func);
```

Please see the manual for more details on each of these functions, but the idea is that each shape can be manipulated using `translate()`, `set_pos()`, `attach_tex()` and `scale()`. And these manipulations can be done between each frame to create drawings and interactable animations. Under the hood `render2d::draw()` and `render2d::animate()` call `render_pipeline::render()` to display these shapes on the screen.

References

www.opengl-tutorial.org
cppreference.com
www.khronos.org/registry/OpenGL-Refpages/gl4/

A Tour of C++, 2nd Edition, Bjarne Stroustrup
GPU Gems, 1st Edition, Nvidia, Edited by Randima Fernando
OpenGL Programming Guide, 9th Edition, Kessenich, Sellers, and Shreiner