MiniGL: Bringing Modern C++ to OpenGL

Jackson Welles, Nathan Cuevas, and Emily Rhyu

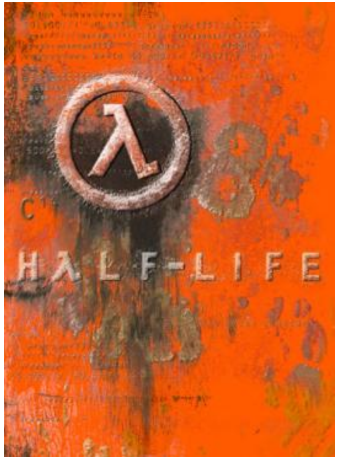
Introduction

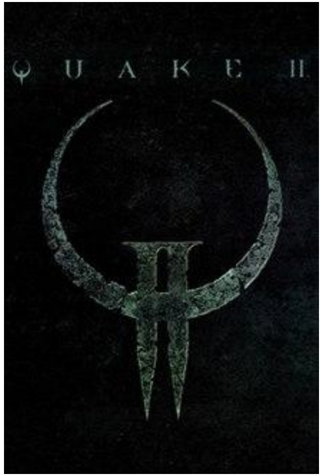
What is OpenGL? : A brief history

- Silicon Graphics (SGI), a leader in 3D graphics in the 90's, created a software library called IRIS GL (Integrated Raster Imaging System Graphical Library)
- Problem: too tied to SGI's platform and competition was closing in
- 1992: SGI released OpenGL, a cross-platform standardized API based off of IRIS GL

What is OpenGL? : Today

- A multi-platform API for building 2D and 3D graphics
- Managed by Khronos Group since 2006
  - Prior : ARB (OpenGL Architecture Review Board)
- Applications: Virtual Reality, Video Games, Information Visualization, Computer-Aided Design (CAD)
  - Ex: Quake 2, Unreal, Half-Life





What is OpenGL? : Clarification

- OpenGL alone is a *specification*, meaning that it provides what the output/performance should look like but not the actual implementation
- The *implementation* comes from hardware and software vendors, often the graphics card manufacturers.
    - As a result, works for many platforms and many devices

OpenGL : Advantages and Disadvantages

Advantages

- Allows users to interact with graphics hardware (i.e. access GPU)
- Works across many devices and platforms
- Support for Extensions
  - Vendors can add additional functionality in drivers
  - Ex: _NV(NVIDIA), AGL(Apple)

Disadvantages

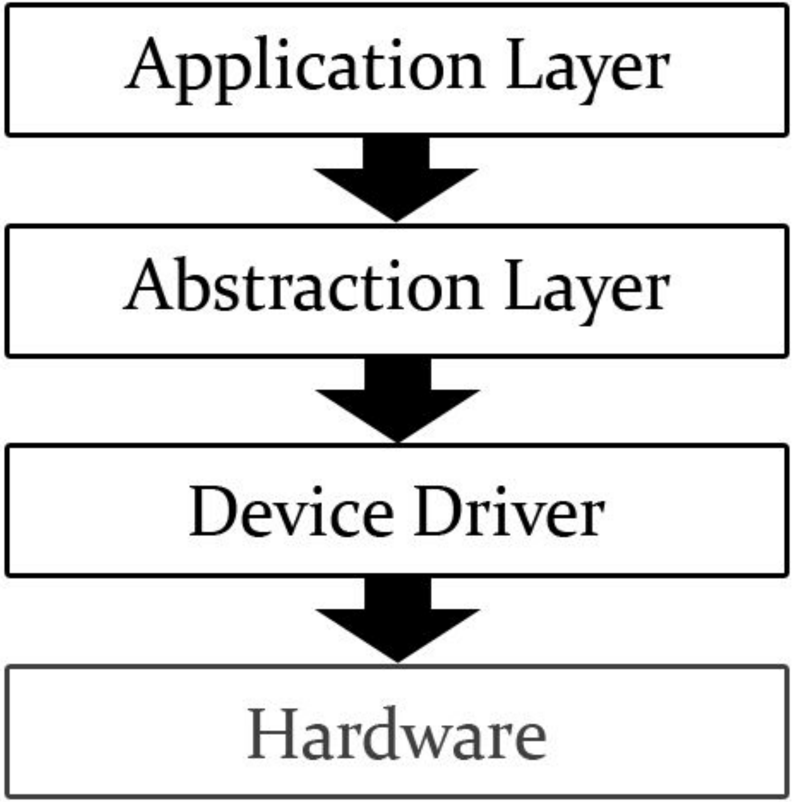
- Bulky
  - Hundreds of lines of code to draw a triangle
- Not intuitive
  - Written like C library
- Starting to be replaced by other APIs?
  - Apple deprecated in Mojave in 2018, saying that was based on past principles

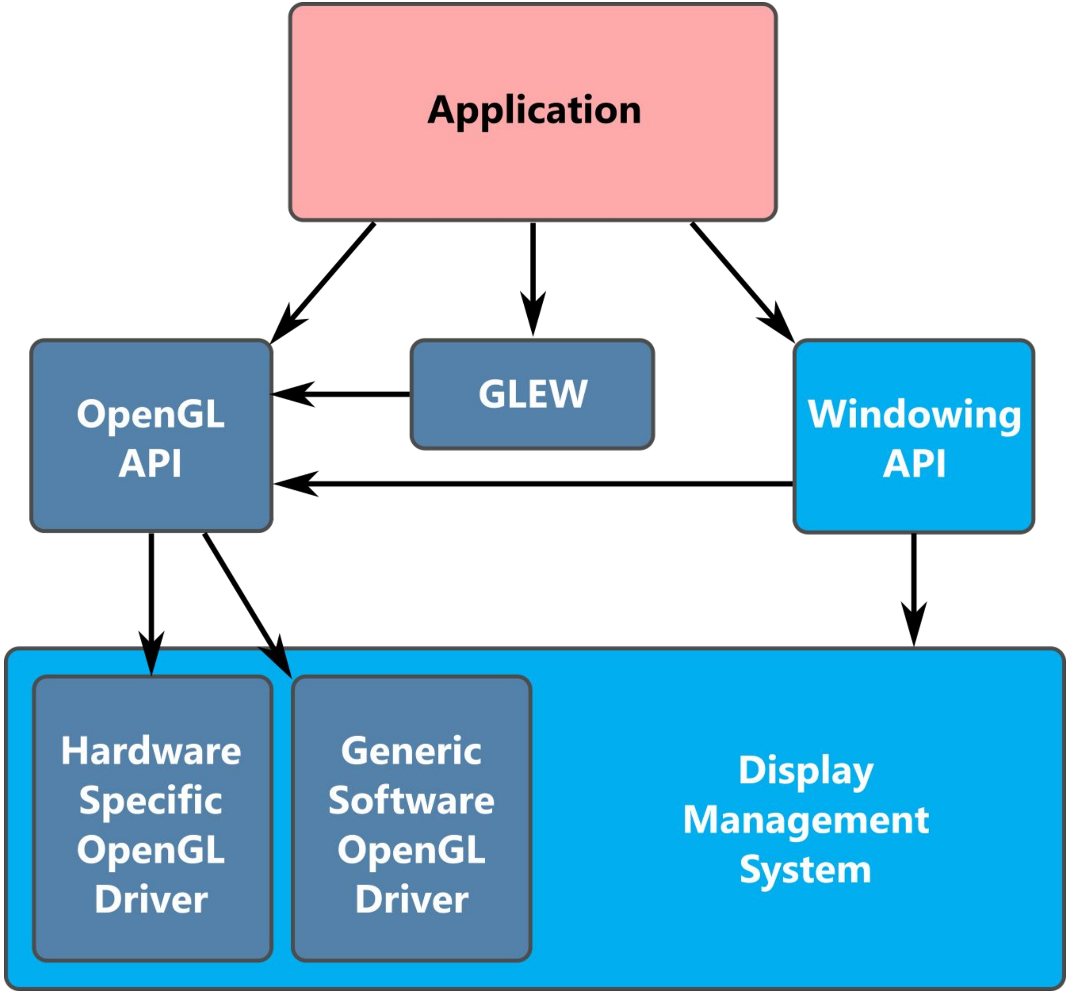
Our Project : MiniGL

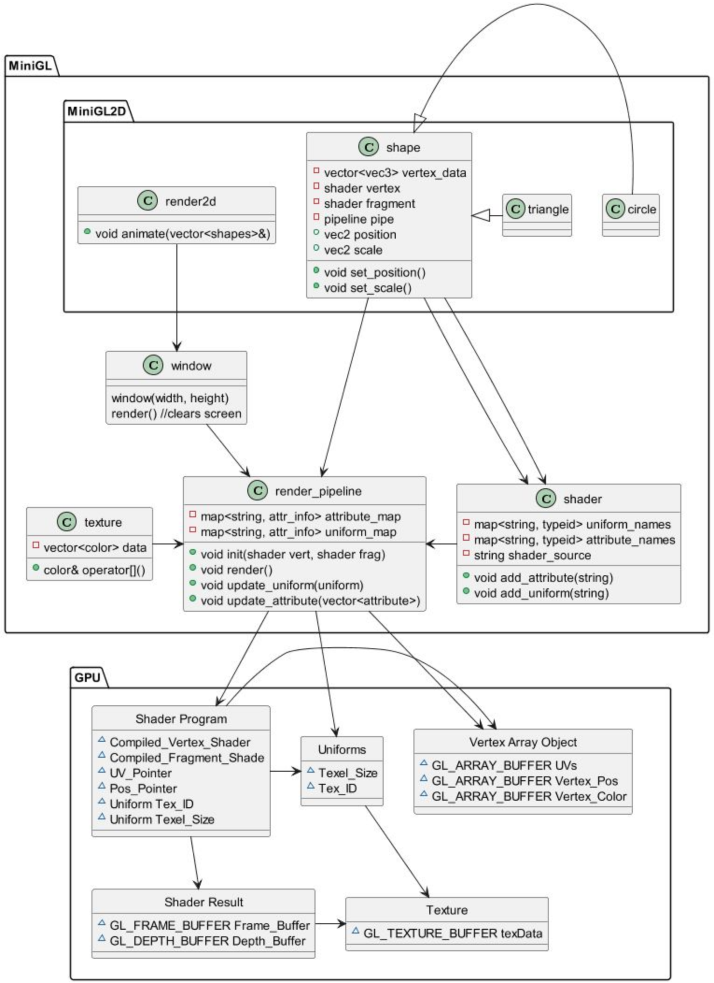
- Goal : Apply modern C++ tools to make OpenGL safer, simpler, and more intuitive without incurring significant overhead
- What is MiniGL : a C++ graphics library built on top of the existing OpenGL API that provides an expressive interface for graphics

How does it all work?

- **Application Layer :** program calling drawing commands. Oversees user level operations and calls OpenGl
- **Abstraction Layer :** OpenGL implementations
- **Device Driver :** software communicate with hardware
- **Hardware :** the GPU




Application
OpenGL API
GLEW
Windowing API
Hardware Specific OpenGL Driver
Generic Software OpenGL Driver
Display Management System


MiniGL
MiniGL2D
shape
vector<vec3> vertex_data
shader vertex
shader fragment
pipeline pipe
vec2 position
vec2 scale
void set_position()
void set_scale()
triangle
circle
render2d
void animate(vector<shapes>&)
window
window(width, height)
render() //clears screen
texture
vector<color> data
color& operator[]()
render_pipeline
map<string, attr_info> attribute_map
map<string, attr_info> uniform_map
void init(shader vert, shader frag)
void render()
void update_uniform(uniform)
void update_attribute(vector<attribute>)
shader
map<string, typeid> uniform_names
map<string, typeid> attribute_names
string shader_source
void add_attribute(string)
void add_uniform(string)
GPU
Shader Program
Compiled_Vertex_Shader
Compiled_Fragment_Shade
UV_Pointer
Pos_Pointer
Uniform Tex_ID
Uniform Texel_Size
Uniforms
Texel_Size
Tex_ID
Vertex Array Object
GL_ARRAY_BUFFER UVs
GL_ARRAY_BUFFER Vertex_Pos
GL_ARRAY_BUFFER Vertex_Color
Shader Result
GL_FRAME_BUFFER Frame_Buffer
GL_DEPTH_BUFFER Depth_Buffer
Texture
GL_TEXTURE_BUFFER texData

External Libraries

- **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.

Graphics Processing Unit (GPU)

- Accelerate computer graphics workloads
- Does floating-point operations quickly, freeing up the Central Processing Unit (CPU) to do integer and more general operations
- Discrete and Integrated
- Applications : Video Editing, Gaming, Machine Learning

CPUs with integrated graphics

Discrete graphics card

Shaders

- Programs that run directly on the GPU, rather than CPU
- Written in OpenGL Shading Language (GLSL), a C-like language
- Specifically tailored towards vector and matrix manipulation

```
#version version_number
in type in_variable_name;
in type in_variable_name;

out type out_variable_name;

uniform type uniform_name;

void main()
{
  // process input(s) and do some weird graphics stuff
  ...
  // output processed stuff to output variable
  out_variable_name = weird_stuff_we_processed;
}
```

Building a Shader Abstraction (HARD)

- Shaders aren’t written in C++!
  - And moreover they can’t be translated easily
- Shader data is stored in GPU memory
  - Ideally this data is set and never checked
- Shaders use ints as handles for everything
  - Lots of typeless pointers create errors, we need type safety!
- Shaders resources must be initialized and destroyed
  - Failing to call the proper cleanup can crash a whole system

The First Step: RAII

In OpenGL resources are generated and deleted:

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

Almost every shader resource behaves like this
→ Constructors and Destructors are an obvious choice

Data Ownership

```
class attr_bundle
{
    std::map<string, variant<vec2, vec3, vec4, mat2, mat3, mat4>> attributes;
    std::map<string, variant<vec2, vec3, vec4, mat2, mat3, mat4>> uniforms;
    void set_attribute(string name, vector<vec2> values);
    void render()
    {
        for (auto pair : attributes)
        {
            // ~10 lines of opengl code to set each attribute
        }
        glDrawArrays();
    }
};
```

Clunky, user updates object, object updates shader

Render Pipeline

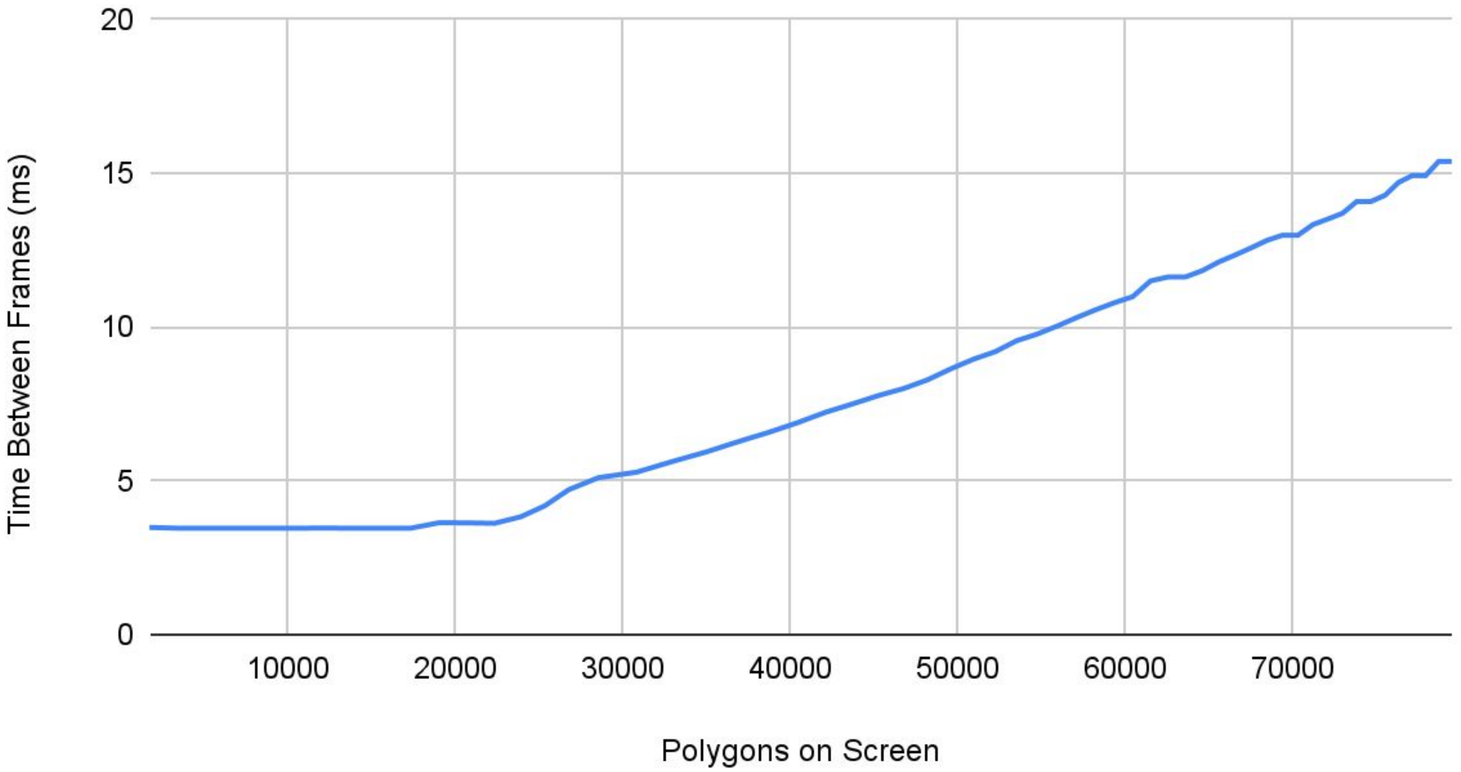
- Doesn't store data, manages handles to pass data through to shader
  - When users set data it is applied directly to GPU programs
- Type checks incoming data at run time.
  - Not as good as compile time, but much better than nothing
  - Uses `std::type_index`
- Shaders map to data 1 to 1
  - May not be optimal!

```
pipeline["shape_color"] = color(colors::red); VS
```

```
glBindVertexArray(vao_id);
glUseProgram(shader_program_id);
glEnableVertexAttribArray(node.array_num);
glBindBuffer(GL_ARRAY_BUFFER, node.buffer_id);
glBufferData(GL_ARRAY_BUFFER, min_verticies * sizeof(T), new_value.data(), GL_DYNAMIC_DRAW);
glVertexAttribPointer( node.array_num, vertex_size, GL_FLOAT, GL_FALSE,0, (void *)0);
```

Still Fast?

miniGL Polygons vs Frame Interval



About 80,000 polygons before frame rate drops below 60 (standard for most monitors)

For Reference

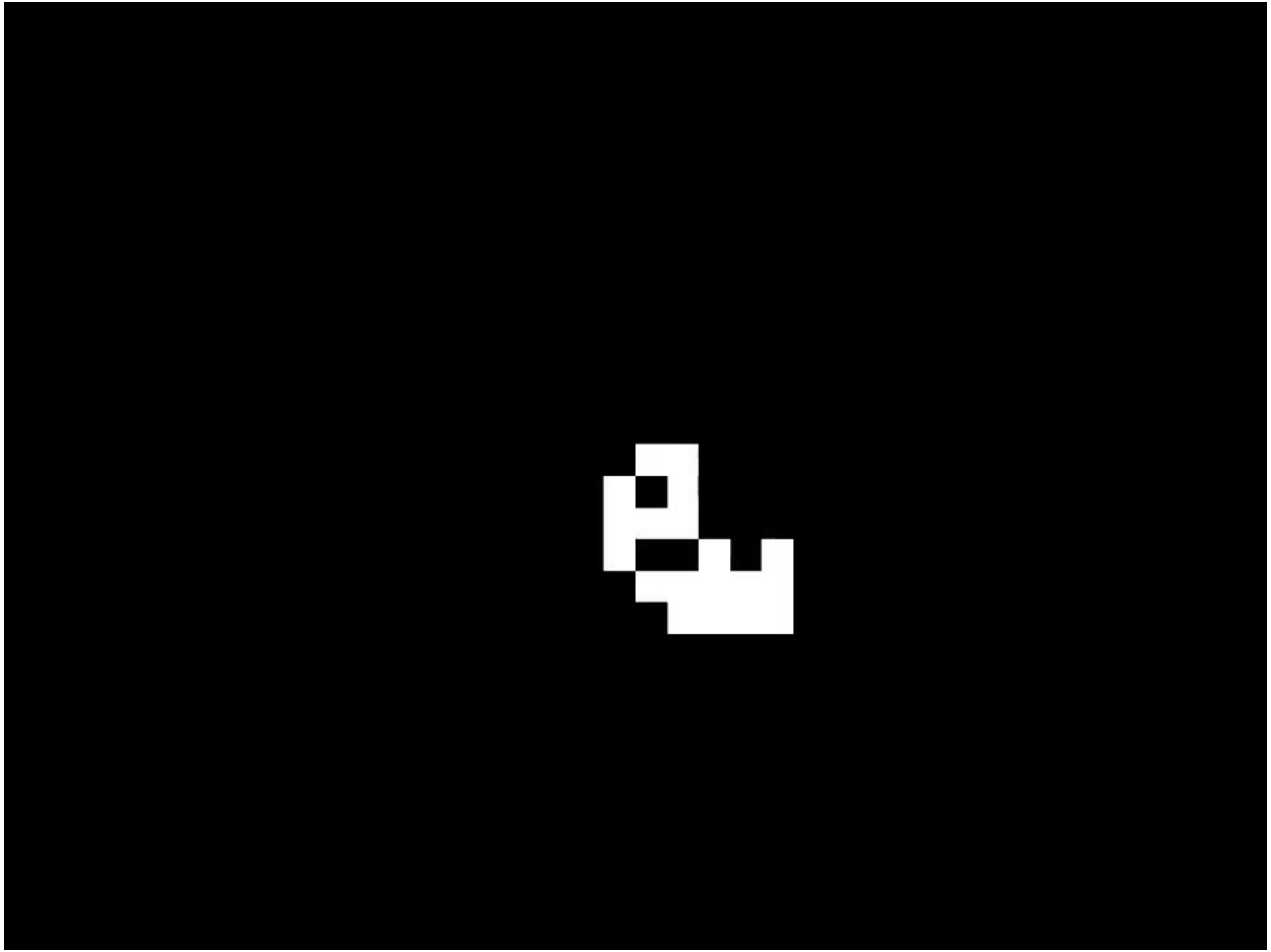
- This dragon is 5756 polygons
  - Can be in one render pipeline
- We can render 80,000 polygons
  - Across 6,000 pipelines
- 15 Dragons is good enough

Game of Life

Runs 20 - 100x faster on GPU

Requires Rendering to Textures

Looks Great



miniGL2d

- A minimal, pain free interface for 2d graphics
- An even higher level of abstraction for users to quickly create 2d graphical programs
- A VERY simple API that is surprisingly expressive
- Useful for things like simulations, games, and visualizations

The Shape Abstraction

- The core object of miniGL2d
- We provide with some builtins (circle, rectangle, square, triangle) but the user can define their own shapes
- Abstracts aways all of the ugly details such as glsl shader scripts, resizing, etc.

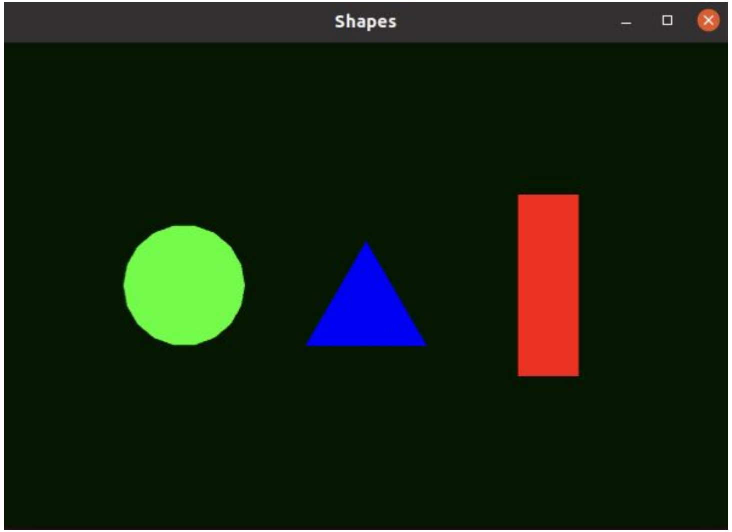
miniGL2d API

```
void shape::translate(position pos);
void shape::set_pos(position pos);
void shape::scale(vec2 u);
void shape::rotate(deg d);
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);
```

```
window2d my_win(600_px, 400_px, color(colors::forest_green), "Shapes");
std::vector<shape> shapes = {
    circle(50_px, colors::green, position(-150, 0)),
    triangle(100_px, colors::blue),
    rectangle(50_px, 150_px, colors::red, position(150, 0)),
};
render2d::draw(my_win, shapes);
```

Would likely take over 100 lines in raw openGL!



We need more!

- Ideally would have animation and interaction of these shapes

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

Called for every frame

```
int width = 700, height = 1000, radius = 100, initial_pos = 350, g = 1, vel = 0, fps = 60;
window2d my_win(pixels(width), pixels(height), color(colors::forest_green), "Bouncing Ball Demo");
std::vector<shape> initial_world = { circle(pixels(radius), position(0, initial_pos)), };
render2d::animate(my_win, fps, initial_world,
[&](std::vector<shape> &world, events e) {
  int dt = 1;
  shape& ball = world[0];
  int relative_pos = ball.get_pos().y - radius + height/2;
  if (relative_pos <= 0) {
    vel = -vel;
    ball.translate(position(0, -relative_pos));
  }
  vel = vel - g * dt;
  ball.set_pos(position(0, ball.get_pos().y + (vel - g * dt) * dt));
});
```

called each frame

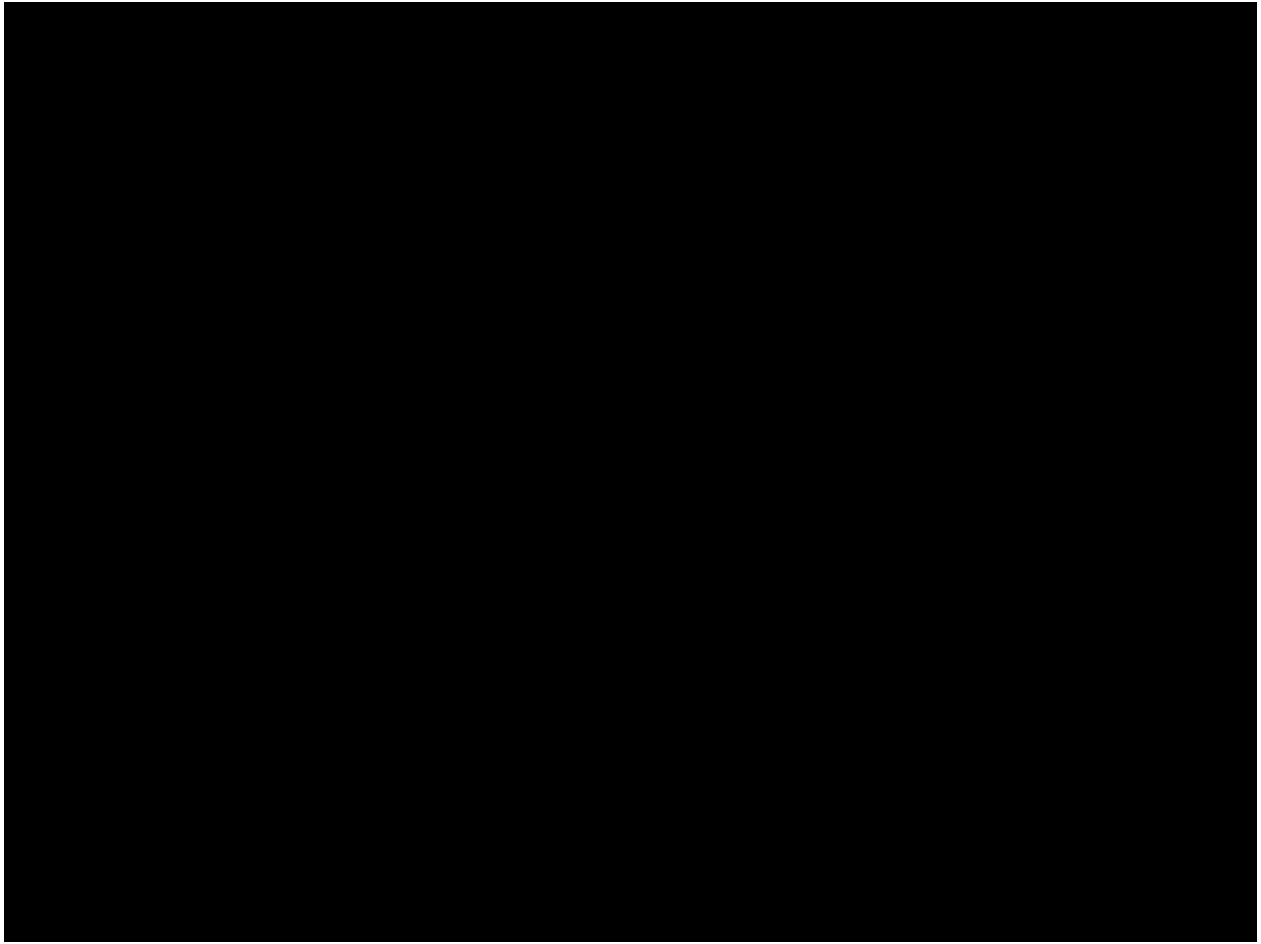
How about events?

- Keyboard inputs, mouse position, etc.

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

Contains event data

Called for every frame



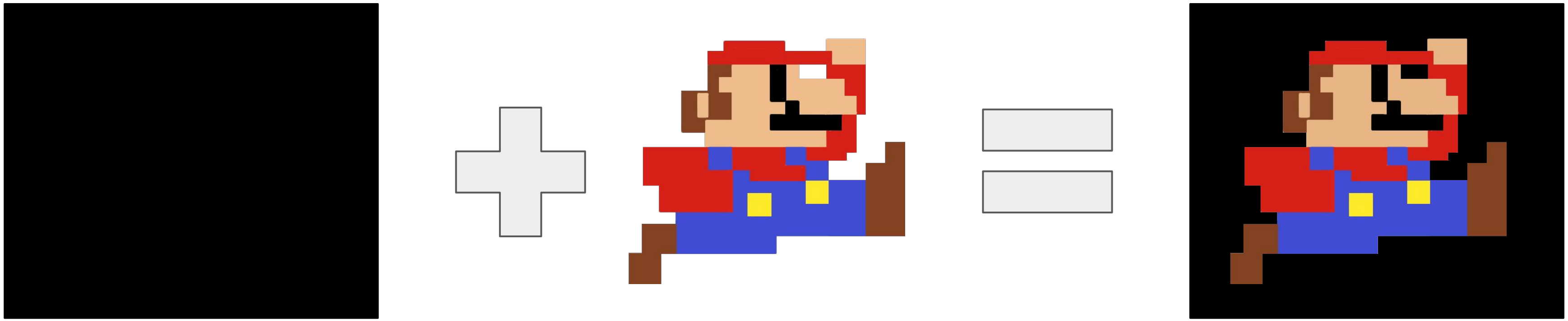
```cpp
window2d win(1200_px, 800_px, color(colors::dark_grey), "Pong");
const int border_width = 1100, border_height = 700, radius = 15, paddle_width = 150,
          paddle_height = 20, paddle_speed = 6, paddle_ypos = -300;
glm::vec2 ball_vel(7, 7);
std::vector<shape> initial_world {
        circle(pixels(radius), colors::dark_grey),
      rectangle(pixels(paddle_width),pixels(paddle_height), colors::dark_grey,position(0, paddle_ypos)),
      rectangle(pixels(border_width), pixels(border_height), colors::cyan),
};
render2d::animate(win, 60, initial_world,
[&](std::vector<shape> &world, events e) {
      shape& ball = world[0];
      shape& paddle = world[1];
      if (e.pressed_keys[A_KEY])
        paddle.translate(position(-paddle_speed, 0));
      if (e.pressed_keys[D_KEY])
        paddle.translate(position(paddle_speed, 0));
      // move ball
      ball.translate(ball_vel);
      // about to hit left wall or right wall
      int horiz_edge = border_width/2 - radius;
      int vert_edge = border_height/2 - radius;
      if (ball.get_pos()[0] + ball_vel[0] >= horiz_edge || ball.get_pos()[0] + ball_vel[0] <= -horiz_edge)
        ball_vel[0] = -ball_vel[0];
      // about to hit the top wall or the bottom wall
      if (ball.get_pos()[1] + ball_vel[1] >= vert_edge || ball.get_pos()[1] + ball_vel[1] <= -vert_edge)
        ball_vel[1] = -ball_vel[1];
      // about to hit top of paddle
      if (ball.get_pos()[1] + ball_vel[1] <= paddle_ypos + paddle_height/2
        && ball.get_pos()[0] + ball_vel[0] >= paddle.get_pos()[0] - paddle_width/2
        && ball.get_pos()[0] + ball_vel[0] <= paddle.get_pos()[0] + paddle_width/2)
        ball_vel[1] = -ball_vel[1];
    });
```

easy to listen to inputs

Shapes are fun but limited

- Can attach textures to our basic shapes

```
rectangle r(20_px, 20_px, colors::black);
r.attach_tex(mario_tex);
```



- Can be very creative with textures
- Textures + cursor events = drawing application

```cpp
window2d my_win(1200_px, 800_px, color(colors::dark_grey), "Basic Paint");
shape canvas = rectangle(1200_px, 800_px);
texture tex = texture(300, 200, colors::white);
canvas.attach_tex(tex);
std::vector<shape> initial_world { canvas, };
render2d::animate(my_win, 144, initial_world,
[&](std::vector<shape> &world, events e) {
    if (e.left_click) {
        int x_pixel = (e.cursor_pos[0] + 600) / 4;
        int y_pixel = (e.cursor_pos[1] + 400) / 4;
        if (x_pixel > 0 && y_pixel > 0 && x_pixel < 300 && y_pixel < 200) {
            tex[y_pixel][x_pixel] = colors::black;
            world[0].attach_tex(tex);
        }
    }
});
```

Fluids Demo

References

https://openglbook.com/chapter-0-preface-what-is-opengl.html
https://learnopengl.com/Getting-started/OpenGL
https://venturebeat.com/2018/06/06/apple-defends-end-of-opengl-as-mac-game-developers-threaten-to-leave/
https://learnopengl.com/Getting-started/Shaders
https://www.intel.com/content/www/us/en/products/docs/processors/what-is-a-gpu.html