Advanced Web Design - Logic Gate Simulator

Haled Odat (201516)

1. Project Structure

The project is divided into several parts:

1.1 Root Structure

```
logic_gate_simulator
├── assets          - Contains textures, shaders, objects, etc
├── vendor          - Contains Third-party libraries
├── src             - Contains the source code
├── CMakeLists.txt  - The CMAKE build system config needed to build.
└── examples        - Contains serialized logic gate circuit examples
```

1.2 Assets Directory

This directory contains all the needed resources needed during runtime it also divided into some parts:

```
assets
├── 2D                  - Contains resources used in 2D rendering
│   ├── shaders          - Shaders used in 2D
│   └── textures         - Contains textures used in 2D
├── 3D                  - Contains resources used in 3D rendering
│   ├── materials.json
│   ├── objects          - Contains 3D object files that are rendered in 3D
│   ├── shaders          - Contains shaders that are used in 3D rendering
│   └── textures         - Contains textures used in 2D
│      └── skybox.hdr    - This is the HDR texture that is used to render the skybox
└── themes               - The directory that contains the current theme
       └── default.json  - The default theme. Serialized json theme
```

On web builds the assets are combined into one binary blob than is lz4 compresses for faster downloads.

1.3 Vendor Directory

This directory contains all the third party libraries

```
vendor
├── glad  - Responsible to load the OpenGL functions/constants
├── glfw  - Responsible to create the window
├── glm   - Responsible for providing math data type (vecN, matN, etc)
├── stb   - Responsible for loading and parsing images
└── CMakeLists.txt - The CMAKE sub-build file that builds the libraries
```

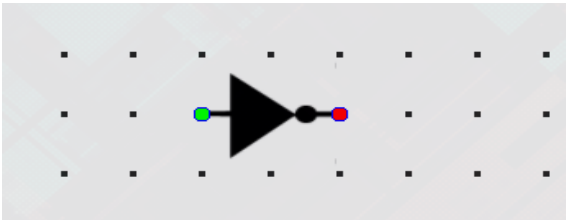
1.4 Src Directory

The source code is divided into models that each handle a specific task

```
src
├── Core                - Things that are used throughout the codebase (logger,
typedefs, etc)
├── Editor              - Contains the logic of the program, how to traverse the
graph, what to render, etc.
├── Events              - Contains the event system.
├── Renderer            - Contains the 2D and 3D renderer as well as resources code
abstractions needed for rendering
├── Serializer          - Contains a Json serializer/deserializer
├── Utils               - Utility functions/types
├── Window.[hpp|cpp]    - GLFW window abstraction that connects with the event
system.
└── Application.[hpp|cpp] - The application start point that contains the main
function.
```

3. Components

Components are the building blocks of logic gate circuits as seen below of an **NOT** gate, the built-in components are basic/simple but they can be combined to create addition, subtraction, multiplication, etc:



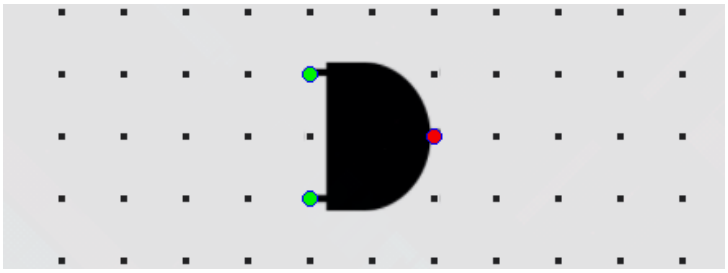
3.1. Component Pins

Component pins denote the input and output of the components which can be further connected with wires.

The input pins are displayed by a green circle while the output is a red circle (as can be seen in the above image).

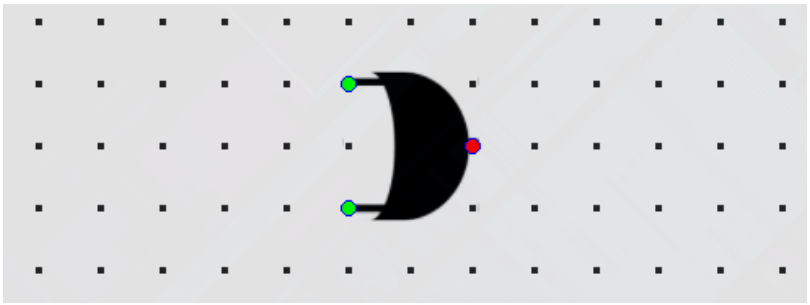
3.2. Component Types

3.2.1. AND Component



The AND gate is a fundamental digital logic gate that takes two binary inputs, often labeled as A and B, and produces a single binary output. Its operation is based on the principle that the output is only true (1) when both of its inputs are true (1). In other words, if both input A and input B are set to 1, the output will be 1; otherwise, the output will be 0. This behavior makes the AND gate useful for various applications in digital logic, such as data validation and control systems, where it's necessary to ensure that multiple conditions are met before a particular action is taken. The AND gate's truth table is straightforward, reflecting its behavior: A 0 input results in a 0 output, and only when both inputs are 1 does it produce a 1 output.

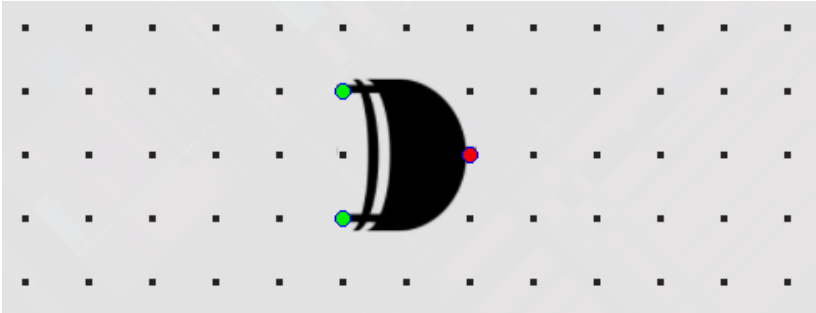
3.3.2. OR Component



(image of an and component)

The OR gate is another fundamental digital logic gate that, like the AND gate, takes two binary inputs, often labeled as A and B. However, its operation is different from the AND gate. The OR gate produces a binary output that is true (1) if at least one of its inputs is true (1). In other words, if either input A or input B is set to 1 (or both are set to 1), the output will be 1; otherwise, the output will be 0. This logical behavior is valuable in scenarios where you want an action to be triggered if any of multiple conditions are met. The OR gate's truth table reflects this behavior, showing that it produces a 1 output whenever at least one input is 1.

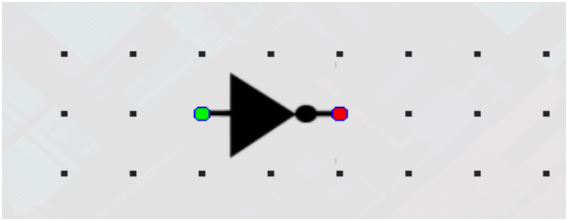
3.3.3. XOR Component



(image of a xor component)

The XOR (exclusive OR) gate, like the AND and OR gates, takes two binary inputs, often labeled as A and B. However, its operation is distinct from both of these gates. The XOR gate produces a binary output that is true (1) if the number of true inputs is odd. In other words, the output is 1 when the inputs are different. If both inputs are the same (both 0 or both 1), the XOR gate produces a 0 output. XOR gates are commonly used in digital systems for various purposes, including error detection, data encryption, and creating flip-flop circuits for memory storage. The XOR gate's truth table highlights its unique behavior, with a 1 output only when the inputs differ.

3.3.4. NOT Component

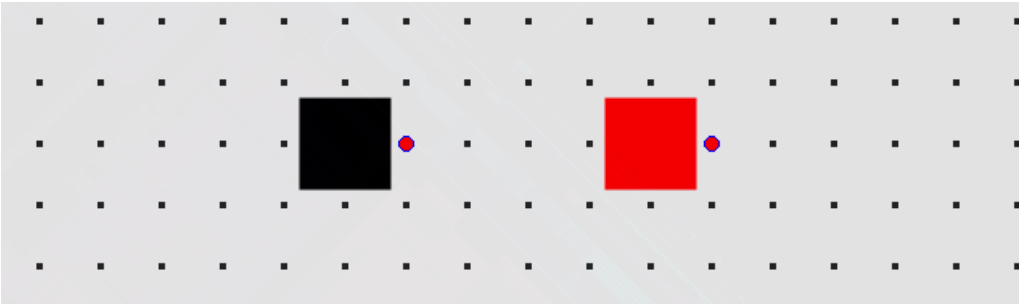


(image of a not component)

The NOT gate, often referred to as an inverter, is a simple digital logic gate that has only one input, typically labeled as A, and produces a single binary output. Its function is to

complement or negate the input. In other words, if the input A is 0, the NOT gate outputs 1, and if the input A is 1, the NOT gate outputs 0. This fundamental gate is essential in digital logic circuits for tasks like signal inversion, enabling various logical operations, and controlling the state of flip-flops and memory elements. The NOT gate's truth table is straightforward, showing that it flips the input value to produce the output value.

3.3.5. Switch Component

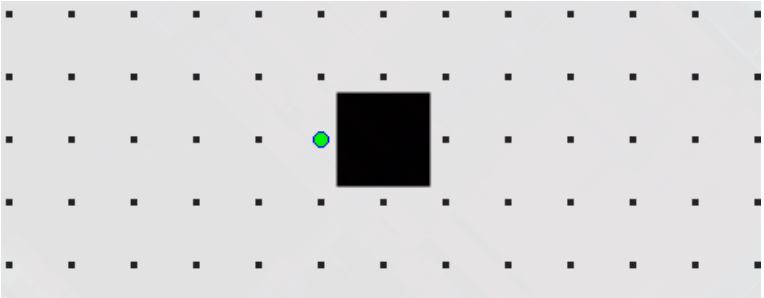


(image of an inactive and active switch component)

A Switch is a component used to control the flow of electrical current in a circuit. It is designed to either open or close a circuit, effectively allowing or blocking the passage of electricity.

This is used for the input of Aggregate/Chip Components.

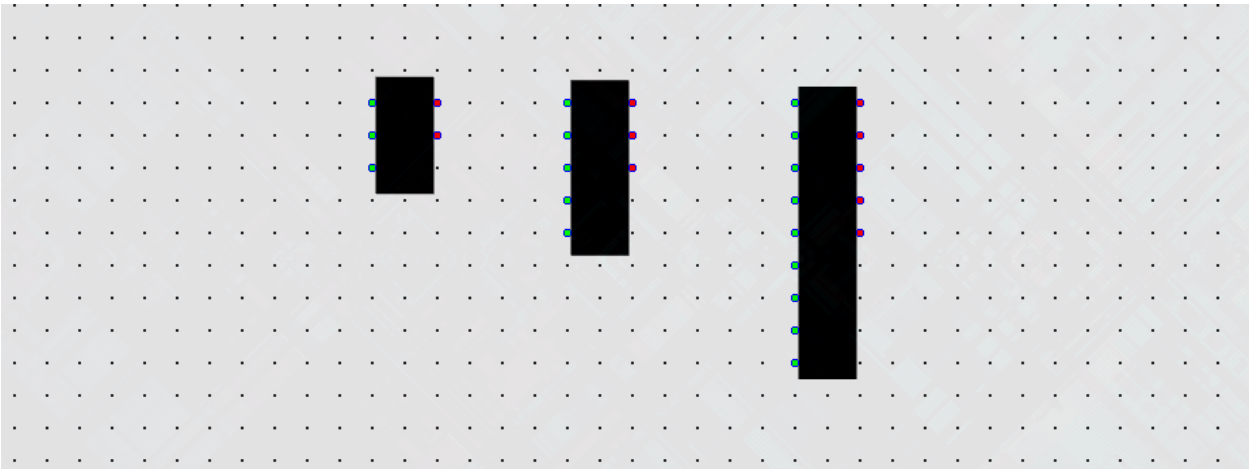
3.3.6. Output Component



(image of an output component)

This is exclusively used for the output of Aggregate/Chip Components.

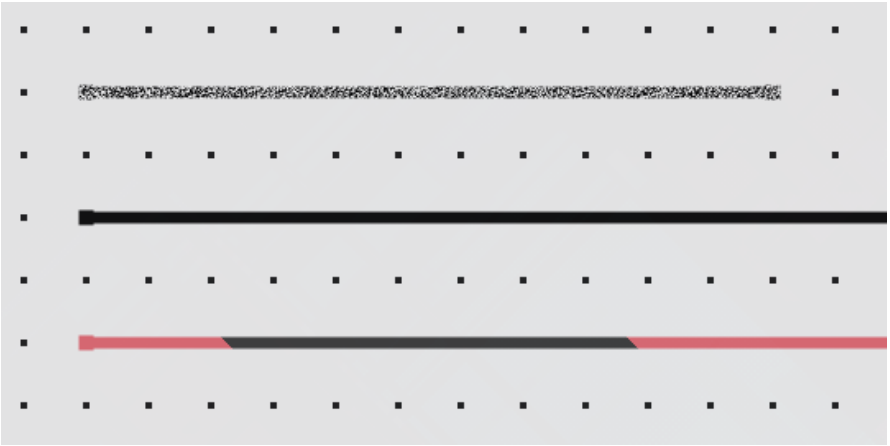
3.4 Aggregate/Chip Components



(image of a 1-bit, 2-bit, and 4-bit full adders)

Aggregate/Chip Components have varying in length in the input (Switch Components) and output (Output Components) pins depending on which chip it is created from. Once created from a chip, its input (Switch Components) and output (Output Components) become immutable and cannot be changed, but other internals can.

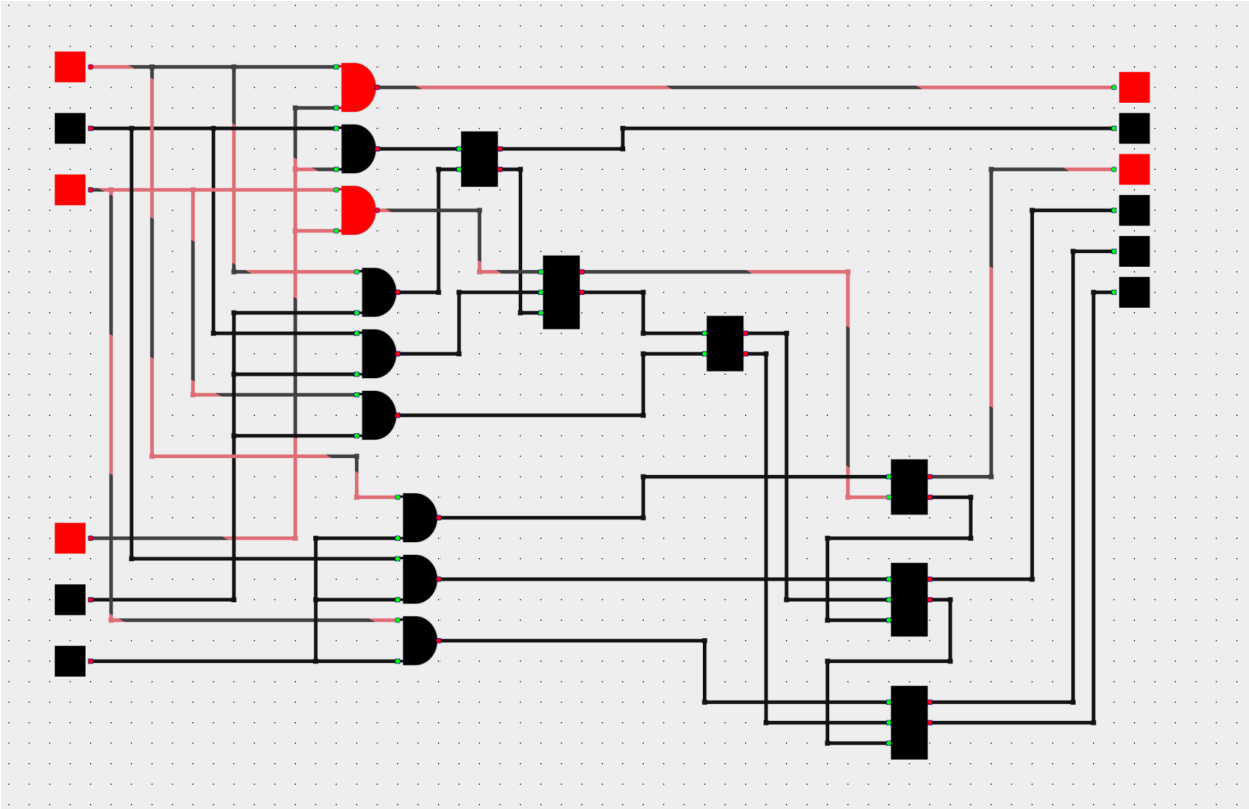
4. Wires - Connectors



(image of a disconnected, connected-inactive and connected-active wire)

In a logic gate simulator, wires are virtual connections that represent the pathways for electrical signals to flow between different components and logic gates within a digital circuit. These virtual wires serve as the medium through which binary information (0s and 1s) is transmitted, allowing for the creation and testing of digital logic circuits without the need for physical hardware.

5. Chips



(image of a 3-bit binary multiplier on a chip)

Using the previously mentioned elements of a logic gate simulator, chips allow us to plot and layer the logical circuits and the creation of **ChipCompoments** (chips in chips).

6. Logic Gate Graph Traversal

The components (logic gates, switches) and wires form a graph. Components have pins which can be of input and output type. We traverse the graph starting from the Input category type (Switch component) and traverse the graph.

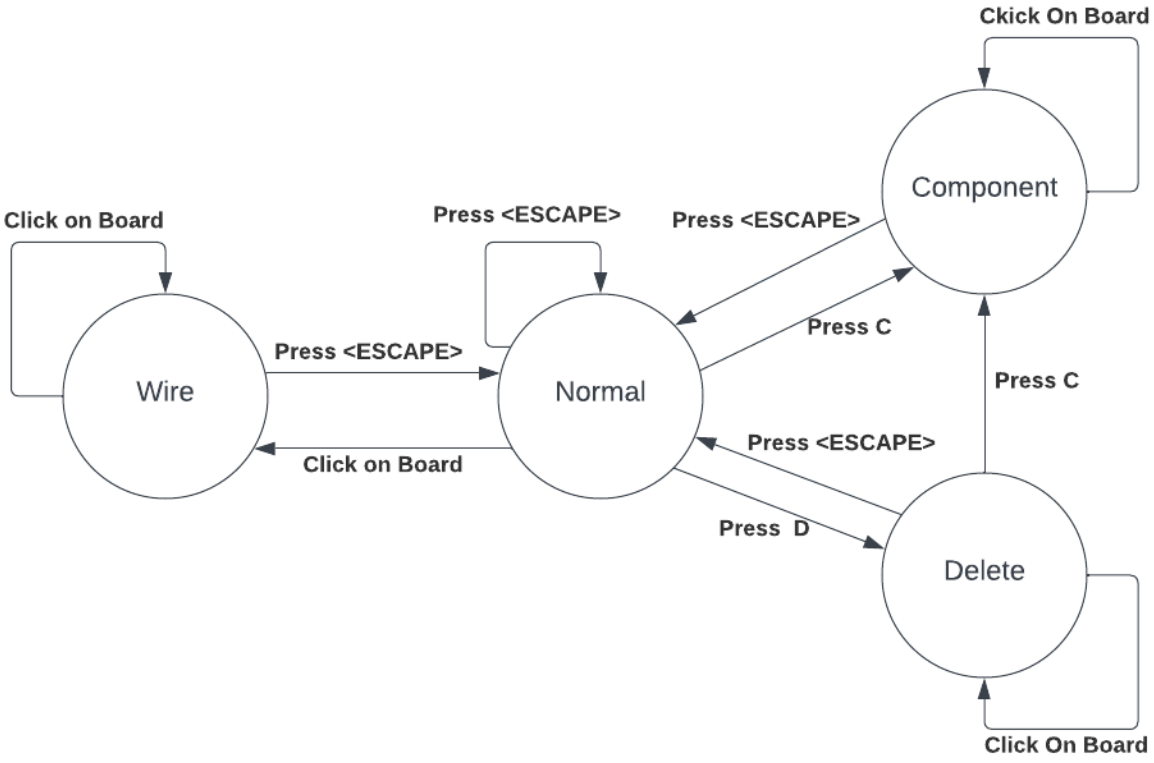
We have a priority queue in which we initially push all switch components (which are of category input). We pop an element from the queue, mark it as visited and if all input pins have been visited we call the update method of the component which does the component specific calculation. Then we push all the output pins connections to the queue. Not all of its pins have been visited. We mark it, push it in the queue with a low priority, if it has been visited but once and and not all of its pins have been visited then we do nothing (it most likely has unconnected pins) we repeat until the queue is empty.

This is called a tick and we tick only on change events.

7. Editor

7.1. Editing Modes

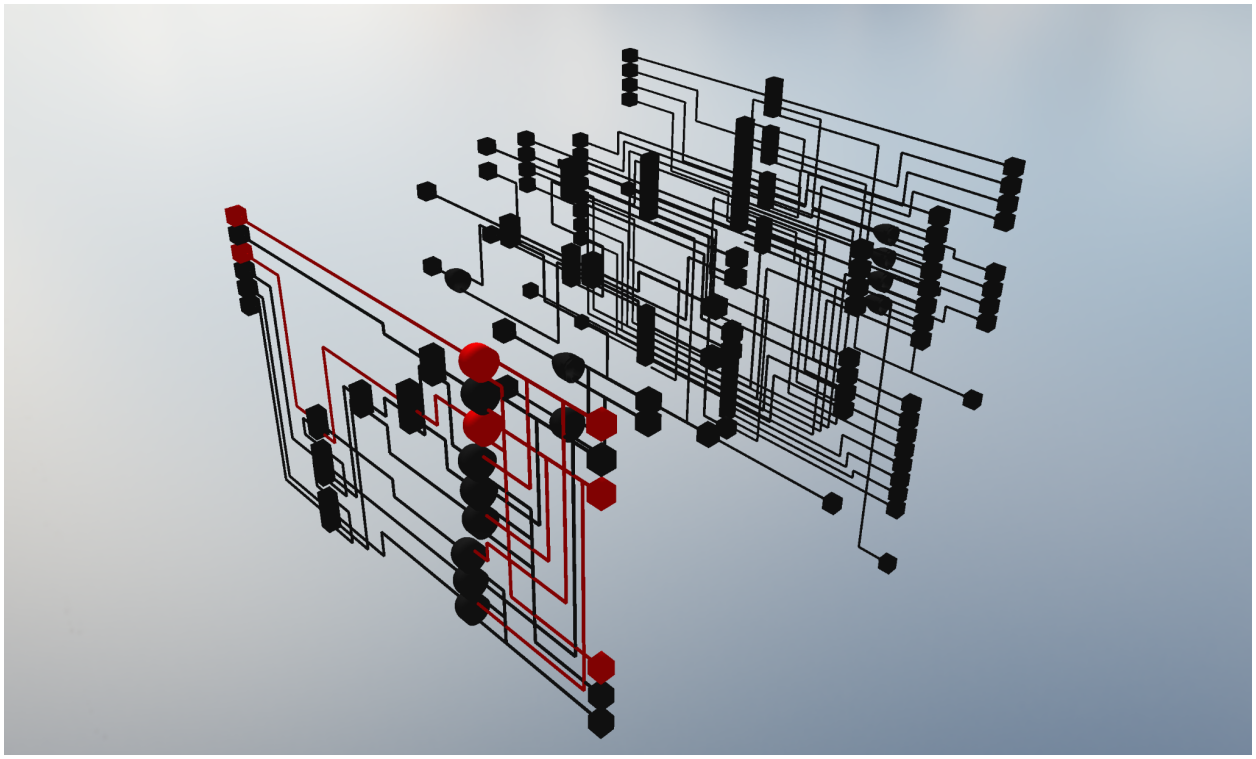
The editor is based on a finite state machine, where it is composed of modes and transitions. Initially we start in the normal mode, we can see the modes an triggers in the following diagram:



7.2. Render Modes

There are two modes of rendering 2D and 3D we can toggled between them with the R key. In the 2D mode we render the grid (dots) onto a texture and use that texture in subsequent calls. This is also done for the minimap in 3D.

This texture cache is invalidated on resize or board state change.



(image of a 3D view of full board)

8. WebGL abstractions

Many OpenGL concepts and constructs are abstracted into classes for easier use and resource management, such as textures, vertex buffers, vertex arrays, index buffers, etc. Eliminating many bugs during development.

Because the OpenGL resources like textures, buffers, and shaders are unique (identified with an unsigned int ***id***). Because they cannot be copied/cloned as this would be a bug and lead to double freed resources, for that reason their copy assignment and copy constructor are deleted to avoid these bugs. To be able to share these objects without making them global, they are put into a *std::shared_ptr<T>* class and once they are no longer referenced anywhere their destructor is called and the resources are freed. To stop the object from escaping the *std::shared_ptr<T>* the move assignment and move constructor have been deleted.

For brevity and to convey more meaning on each of these objects there is a *using typedef T::Handle, i.e. Shader::Handle, Texture::Handle, etc. Which is equivalent to std::shared_ptr<T>*.

In the majority of the objects we utilize the builder pattern for easier construction of objects.

8.1. Texture Abstraction

The 2D OpenGL textures are abstracted into the *Texture* class. This class utilizes the builder pattern to construct the object. There are three types of 2D textures.

The *Image Texture* type. This texture type is the more traditional use for textures as a place to put the image from a file. It is created with the *Texture::load()* static method that takes a path to the image on disk. Where it is parsed by stb_image and stored in OpenGL, It is also stored in a global hashmap where the key is the file path and the value is the Texture handle so subsequent calls will give the texture instead of creating a new one.

The next type is *Buffer Textures*. It is created with the *Texture::buffer()* static method that takes the width and height. This texture type only allocated space for a specified texture width, height, and format. It is used for frame buffer color attachments. They are not cached in a hashmap creating one will alway give you a unique one.

The last type is the *Color Texture* which contains one pixel (width = 1, height = 1) with format *RGBA8*. It is created with the *Texture::color()* static method that either takes 4 bytes (rgba) or that takes a single 32-bit number. I.e 0xFF0020FF. This type is used to keep a more uniform API for the renderer.

In all of there types the builder pattern is used, for example it we wanted a texture that is loaded from disk (“assets/MyTexture.png”) that has nearest filtering on minification and linear filtering with magnification and wrapping mode repeat and we what it to have no mipmap and to have an internal format of RGB8, we would do the following:

```
Texture::Handle texture = Texture::load(“assets/MyTexture.png”)
       .filtering({Texture::FilteringMode::Nearest, Texture::FilteringMode::Linear})
       .wrapping(Texture::WrappingMode::Repeat)
       .format(Texture::Format::Rgb8)
       .mipmap(Texture::MipMap::None)
       .build();
```

With these abstractions we do not need to call openGL functions directly, making it clearer and less bug prone.

Many such techniques are used for other OpenGL primitives.

8. WASM Porting Issues

This section describes some issues I encountered during the porting process of the native C++ codebase to WASM.

8.1. Emscripten

8.1.1. Memory allocations

By default Emscripten does not allow memory allocation through `malloc`, you need to specify the `-sALLOW_MEMORY_GROWTH=1` , otherwise allocation fails.

8.1.2. Accessing assets directory

We can’t access files with `fopen`, so the assets have to be bundled into a blob, solution was to use the `--preload-file "./assets@assets"` flag.

8.2. GLFW

The implementation of GLFW which is used to display the window, accept events, swap display buffers etc. But it is not fully compatible with the native version. Which requires conditional usage of native ant emscripten’s implementation, providing `-sUSE_GLFW=3`.

8.2.1. GLFW key listener blocking backspace and tab

Because of emscripten's implementation captures all keystrokes
There is an issue for this
https://github.com/emscripten-core/emscripten/issues/11221 which was closed, due to it being a non-fix issue.

Quote from the issue's last comment:
""
```
Not breaking existing codebases is critical, so changing up existing
function entry points is undesirable.
```
""

My solution for allowing backspace (which is needed for chip renaming and board saving which opens a html model), was to register a keypress event handler before every other handler is registered, then when we receive a backspace we don't propagate it other handlers registered on the `window` using `stopImmediatePropagation(event)`.

8.1.2. Missing functions like glfwGetError

There are some functions which are not implemented and it gives linker errors.

Issue: https://github.com/ocornut/imgui/issues/6240

Solution was to conditional compile it based on target (Native, Web) using `#ifdef`s, this pattern is used throughout the codebase.

8.4. Glad

The loading of the OpenGL functions on the web, was loading some and leaving the rest as NULL this caused some very hard to track bugs. The solution was to use the emscripten GLFW's OpenGL bindings which required defining the following marcos before including `#include <GLFW/glfw3.h>`

`# define GL_GLEXT_PROTOTYPES`
`# define EGL_EGLEXT_PROTOTYPES`

8.5. OpenGL to WebGL

8.3.2. No INTEGER_RED texture for color/picking

This was necessary for allowing you to click an object in 3D render mode. In native OpenGL we have a renderable `GL_RED_INTEGER` texture where each pixel is an unsigned 32-bit integer that allows us to easily paint a texture with each object's ID, when clicking we get the pixel and we determine which object was clicked.

The solution was using the `RGB8UI` type texture. We use bitwire operators to divide into separate bytes when we send to the shader and stitch it back together then we take the pixel from the rendered texture.

8.3.1. Missing functions and non-renderable textures

Because WebGL is a subset of the native OpenGL there are functions and features that I had to workaround, for example RGB8 texture cannot be rendered onto, or we can’t easily copy or clear the textures.