

Master thesis

March 15, 2021

Elemental UI

Portable and performant solution for modern
GUI rendering

Luka Lapanashvili

of Tbilisi, Georgia (13-934-062)

supervised by

Prof. Dr. Harald C. Gall

Dr. Pasquale Salza



University of
Zurich^{UZH}



Master thesis

Elemental UI

Portable and performant solution for modern
GUI rendering

Luka Lapanashvili



University of
Zurich^{UZH}



Master thesis

Author: Luka Lapanashvili, luka.lapanashvili@uzh.ch

Project period: 15.09.2020 - 15.03.2021

Software Evolution & Architecture Lab

Department of Informatics, University of Zurich

Acknowledgements

First, I would like to thank my supervisor Dr. Pasquale Salza for accepting my proposal and agreeing to guide me through this difficult endeavor. Your continued support and encouragement reassured me to keep pushing forward, even when the task seemed impossible.

A special thanks goes to Mariam Lapanashvili for proofreading the thesis and for providing constant motivation. Furthermore, I would like to thank my friends and family for their continuous support. Finally, I want to thank Prof. Dr. Harald C. Gall from the Software Evolution and Architecture Lab research group for giving me the opportunity to write this thesis.

Abstract

Interacting with software is commonplace in modern society. Music, video, images, and text are routinely consumed on smartphones, laptops, smart wearables, stationary workstations, and embedded devices. Simultaneously, the number of network-enabled devices per person rises, which increases the demand for the preferred media playback software or the social media application to be available on any device in such a way that a podcast paused on a laptop can be resumed on a smartphone. However, rarely is it possible to use the same frontend code or even the same programming language to create an application that can be run on different devices. Application developers often have to adapt implementations for every single platform or even write bespoke implementations for individual operating systems. Naturally, such a fragmented codebase is difficult to maintain and to uphold feature parity across all the devices. Having one unified solution, where one codebase can target a large set of devices, would be highly beneficial. In this thesis, we discuss the difficulties of developing a cross-platform application and evaluate available solutions. Finally, we introduce Elemental UI, our cross-platform solution for modern GUI application development, and discuss its strengths and shortcomings compared to other established frameworks.

Zusammenfassung

Die Interaktion mit Software ist in der modernen Gesellschaft alltäglich. Musik, Video, Bild und Text werden regelmässig auf Smartphones, Laptops, Smart-Wearables, Arbeitsplatzrechnern und Embedded-Devices konsumiert. Gleichzeitig steigt die Anzahl der netzwerkfähigen Geräte pro Person, wobei die Anforderung immer grösser wird, dass die bevorzugte Wiedergabesoftware oder die Social-Media-Anwendung auf jedem Gerät verfügbar sein muss, sodass ein auf dem Laptop pausierter Podcast auf dem Smartphone fortgesetzt werden kann. Allerdings ist es selten möglich den gleichen Frontend-Code oder sogar die gleiche Programmiersprache zu benutzen, um eine Anwendung zu entwickeln, welche auf verschiedenen Geräten ausgeführt werden kann. Entwickler müssen ihre Implementierungen oft für jede einzelne Plattform anpassen oder sogar massgeschneiderte Lösungen für einzelne Betriebssysteme schreiben. Verständlicherweise ist es schwierig, mit einer solch fragmentierten Code-Basis die Funktionsparität auf allen Geräten aufrechtzuerhalten. Eine einheitliche Lösung, bei der eine Code-Basis auf eine grosse Anzahl von Geräten abzielt, kann viele Vorteile bieten. In dieser Arbeit erläutern wir die Schwierigkeiten bei der Entwicklung einer plattformübergreifenden Anwendung und evaluieren vorhandene Lösungsansätze. Schliesslich stellen wir Elemental UI, unser plattformübergreifendes System für die Entwicklung moderner GUI-Anwendungen vor und erläutern die Stärken und Schwächen im Vergleich zu anderen etablierten Systemen.

Contents

1	Introduction	1
2	Background	3
2.1	High-Level Concepts	3
2.1.1	Cross-Platform	3
2.1.2	Rendering API	3
2.2	Technical Glossary	4
2.2.1	Shader	4
2.2.2	Buffer Objects	4
3	Related Work	5
3.1	Requirements	5
3.2	State of the Art	6
3.2.1	Rendering Libraries	6
3.2.2	GUI Toolkits	7
4	Approach	9
4.1	System Architecture	9
4.2	2D Rendering Library - Elemental Draw	10
4.2.1	Cross-Compatibility	10
4.2.2	Rendering	19
4.3	GUI Library - Elemental UI	23
4.3.1	Node Hierarchy	23
4.3.2	Styling	24
4.3.3	Event System	25
4.3.4	Transitions	26
4.3.5	Dynamic Framerate	27
5	Case Studies	29
5.1	Painting Application	29
5.2	Breakout Game	30
5.3	Calculator	31
5.4	Messaging Application	32
5.5	Conway's Game Of Life	33
6	Evaluation	35
6.1	RQ1 - Performance Evaluation	35
6.2	RQ2 - Performance Comparison	37

7	Future Work	41
7.1	Metal Implementation	41
7.2	Path Rendering	41
7.3	Custom OS Interface	42
7.4	Binding Generation	43
8	Conclusion	45

List of Figures

4.1	Elemental UI architecture	9
4.2	Supported primitives in Elemental Draw	21
4.3	Event propagation	26
5.1	Painting application	30
5.2	Breakout game	31
5.3	Calculator application	32
5.4	Messaging application	33
6.1	Benchmark application	36
6.2	Performance benchmark	38
6.3	Performance comparison	39

List of Tables

3.1	2D Rendering Libraries	6
3.2	GUI Toolkits	7
4.1	Graphics API compatibility	11

Introduction

It is difficult to create cross-platform graphical user interfaces, especially if the requirements demand for the application to be compatible across a wide range of platforms and operating systems. This is due to the fact that operating systems expose unique proprietary interfaces to their own systems, forcing a developer to write unique implementations for every targeted operating system. The complexity is increased by the fact, that some operating systems can provide multiple backends. For example, the Windows operating system alone exposes four different application platforms, which were accumulated over time: Universal Windows Platform (UWP), Windows Presentation Foundation (WPF), Windows Forms, and Win32 [1].

Furthermore, if high performance is important, the complexity of the resulting system further increases drastically, as low-level languages as well as Graphics Processing Unit (GPU) acceleration need to be used, implying manual memory and state management.

Naturally, this is not a new problem and numerous libraries and frameworks already exist, which can aid in interfacing with the various operating systems in a unified manner. Unfortunately, many such libraries are rather dated and employ old technologies. Although a high age of a library is not necessarily a negative attribute, as typically such libraries are much more robust and well tested, often they also prove to be rigid, which makes adapting to emerging design trends much more difficult. New design concepts such as Glassmorphism [2] are difficult to replicate with older libraries, as these rendering systems were never designed for such use cases.

Simultaneously, web technologies are pushing the envelope and provide the basis for new design ideas and emerging trends. Furthermore, artists invent new interactive experiences that run inside the web-browser [3] [4]. Combined with the almost frictionless portability and availability of a web page on virtually any device, it is not surprising to observe an avalanche of new web applications and the increasing dominance of the JavaScript programming language since 2014 [5]. While web technologies might provide diverse features for creative designs, the client code is always written in JavaScript which is slower than lower-level languages like C++ [6]. Therefore, the question arises as, why such great user experiences should not be replicable utilizing native technologies which take full advantage of the hardware to deliver a smoother experience.

The goal of the thesis is to identify critical requirements that are necessary for a portable and high performant graphical user interface solution, that enables creative design choices. Furthermore, we will evaluate the pre-existing state-of-the-art libraries and toolkits on the basis of the predetermined standards.

In addition, the thesis will propose two new libraries (Elemental Draw and Elemental UI) by employing all the requirements as best as possible to explore the following two research questions:

RQ 1: How does the proposed system perform on different hardware? As hardware can majorly influence the performance of an application, we will perform a benchmark on a wide number of devices to identify the specific performance characteristics of the proposed architecture.

RQ 2: How does the performance compare to similar frameworks? To confirm the validity of our proposed system, a preliminary comparison has to be performed to verify whether our system meets the performance standards set out by other existing software.

The thesis is structured as follows. In Chapter 2, we introduce background concepts, relevant to the technical approach of this thesis. Chapter 3 presents the identified requirements and provides an overview of the existing state-of-the-art GUI toolkits. In Chapter 4, we introduce our proposed framework and elaborate on the main design decisions and how they relate to the designated requirements. Subsequently, in Chapter 5, we conduct case studies for possible applications of the introduced system and select a suitable benchmark for performing evaluations in Chapter 6 related to our formulated research questions. Finally, we list the shortcomings of our solution alongside ideas for future improvements in Chapter 7 and conclude the thesis in Chapter 8.

Background

This chapter establishes conceptual and technical topics and terms to provide a clear overview before they are used in the context of this thesis. The two main categories are high-level concepts and a technical glossary. The concepts are introduced in order of their appearance in the thesis in the individual categories.

2.1 High-Level Concepts

2.1.1 Cross-Platform

A cross-platform or portable code base refers to software that can be run on different platforms with the same source code. To achieve portability, special care must be given to the choice of language, the version of the language standard, use of operating system (OS) features, the dependencies, and the build configuration. For example, C++ code can be compiled and run on a vast number of platform targets through compilers such as Visual C++ [7], GCC [8], Clang [9] and others. However, the C++20 standard, although already published, is still not fully supported by all the compiler vendors and, therefore, might not work on all the platforms [10]. Furthermore, the OS-specific calls, such as window creation, networking, and so forth, must be segregated and implemented separately for every single platform. In addition, dependencies must be carefully selected, as they also have to comply with the same restrictions. Finally, the build system must be set up in a way that different instruction set architectures and bit register ranges can be targeted during compile time [11, p. 3-9]. Common architecture targets for Intel-based chips are x86 with 32-bit or 64-bit.

2.1.2 Rendering API

A rendering API, also referred to as graphics API or rendering backend, is an application programming interface that exposes procedures and commands for interacting with a rendering system [12, p. 21]. In the case of Vulkan [13] and OpenGL [14], the interface describes how to access the graphics processing unit (GPU) and perform graphics and compute operations. As the interfaces only describe how to interact with the GPU, the actual implementation of said interfaces has to be done by the GPU vendors through drivers. Therefore, discrepancies between different GPUs can occur, caused by driver bugs or different implementations leading to different states in cases of undefined behavior [15, p. 36].

2.2 Technical Glossary

2.2.1 Shader

A shader, in the context of computer graphics, is a program that runs on the GPU [12, p. 150]. Typically, the output of a shader are pixels (fragment shader). However, there are different types of shaders (vertex, geometry and compute) that can be chained together to perform varying operations on user-defined data. In the case of compute shaders, the resulting output can be fed back to the main memory and accessed by the application. This type of hardware acceleration can lead to great performance improvements for highly parallelizable algorithms.

2.2.2 Buffer Objects

As shaders run physically separated from the remainder of the application code, a mechanism is required to share data from the application memory to the GPU. This can be done through different mechanisms provided by the graphics API [12, p. 186].

Uniform Buffer Objects (UBO)

A uniform buffer can, for example, be used to rapidly pass data to the GPU by storing the data in the registers of the GPU. This approach allows the shaders fast access to said data; however, the size of the buffer is limited. Typically, only a size of 16 KB [16] is guaranteed to be available through the OpenGL standard.

Shader Storage Buffer Objects (SSBO)

In contrast, in cases where the UBO does not provide sufficient space, SSBOs that allows the buffer to be up to 128 MB [16] in size can be used. However, the fast GPU registers cannot be utilized for this approach, so the access can be considerably slower than with UBOs. Furthermore, unlike UBOs, SSBOs are read and write enabled [12, p. 186].

Related Work

Considering the motivation and goals of this thesis, the following chapter provides an overview of the requirements necessitated by a modern and portable UI system. Thereafter, the currently available frameworks are presented, categorized, and evaluated according to the stated requirements.

3.1 Requirements

Following are five core requirements corresponding to the categories: portability, flexibility, and performance, based on the initial motivation for this thesis.

Low-Level Exposure While high-level languages can offer some convenience factors during development, these conveniences come with a runtime cost. Therefore, while high-level language bindings can be beneficial in some applications, a low-level API should still be exposed for performance-critical code.

GPU-Based Rendering To ensure high-performance rendering and smooth framerates, the rendering system ideally should be utilizing the GPU to its fullest. While a software renderer can be beneficial to overall compatibility, it should not be established as the main component but instead as a complement to a high-performance GPU renderer.

Flexible Styling Separating style from content and offering a wide range of styling options can enable creative use cases and provides the basis for new design concepts. Therefore, it is important that styling is not only enforced as a means for configuring preexisting UI widgets but instead offers room for creative design choices.

Composition Based Although the object-oriented paradigm of inheritance would seem as an obvious choice for modeling graphical user interface widgets and components, it entails a rather rigid design. Alternatively, a composition-based approach would allow for runtime generated objects with varying properties, contributing to the flexibility of the system.

Responsive Design Portability in the context of graphical user interfaces does not only entail the capability to run the application on different platforms but also to present the visual content. As device monitors vary in resolution and shape, the system has to be able to adapt and offer the developer the means for optimizing the presented content regardless of the available screen dimensions.

3.2 State of the Art

Following, we introduce the state-of-the-art libraries, which we divided into two categories for better comparison: Rendering libraries and GUI toolkits. While rendering libraries solely provide necessary means for drawing on a color buffer with simple lines, arcs, and shapes, GUI toolkits provide a much more diverse tool-set for rendering and interacting with user interface elements. However, this division might not always be clear-cut and might not apply for certain libraries which provide both: low-level rendering commands and high-level widget elements.

3.2.1 Rendering Libraries

While there may certainly exist many more rendering libraries, the following Table 3.1 only focuses on the most popular options currently available. Native OS rendering solutions such as Quartz2D or DirectDraw are omitted from the table as they do not provide cross-platform capabilities.

Name	Language	Published	Last Release	GPU	Platforms
Allegro ^[17]	C	before 2000 ^[18]	5.2.7 (2021)	OpenGL, D3D	Windows, Linux, Mac OSX, iPhone, Android
cairo ^[19]	C++	2003 ^[20]	1.17.4 (2020) ^[21]	partial ^[22]	Win, Linux, Mac, BeOS
JavaFX ^[23]	Java	before 2008 ^[24]	v16 (2021) ^[25]	OpenGL, D3D ^[26]	Win, Linux, Mac
Skia ^[27]	C++	2008 ^[28]	2021 ^[29]	OpenGL, (Vulkan) ^[30]	Win, Linux, Mac, iOS, Android

Table 3.1: 2D Rendering Libraries

While JavaFX provides great features and flexibility in terms of portability and rendering, it was discarded as only the best performing rendering libraries were considered. Skia and cairo are both widely adopted and mature graphics engines. Other technologies and GUI toolkits build upon these renderers as shown in Table 3.2. In addition, Skia is the renderer used in the Chromium browser which is the basis for Google Chrome and recently Microsoft's Edge browser [31]. It is not surprising that Skia is highly optimized and can run on all major platforms. However, its Vulkan implementation is still new and unstable [30]. Finally, Allegro would have been considered due to its various platform targets, nevertheless, it was discarded, as it lacked modern backend support.

3.2.2 GUI Toolkits

Similar to the rendering libraries, only a subset of all available GUI toolkits were considered because not all relevant information about technical details could be obtained on all options, due to public availability.

Name	Exposed Language	Last Release	Renderer	Platforms
Avalonia ^[32]	C#, XAML	0.10.0 (2021)	Skia ^[33]	Windows, Linux, macOS
CEGUI ^[34]	C++, XML	0.8.7 (2016)	custom	Windows, Linux, macOS
Electron ^[35]	JavaScript, HTML, CSS	12.0.1 (2021)	Skia	Windows, Linux, macOS
FLTK ^[36]	C++	1.3.5 (2019)	custom	Windows, Linux, macOS
Flutter ^[37]	Dart	v2 (2021)	Skia	Android, iOS, (Windows, Linux, macOS)
GTK ^[38]	C, (many bindings)	4.0.3 (2021)	cairo ^[39]	Windows, Linux, macOS
imgui ^[40]	C++, (many bindings)	1.81 (2021)	custom (GPU)	Windows, macOS, Linux, Android
nana ^[41]	C++	1.7.4 (2020)	custom (CPU)	Windows, Linux, FreeBSD
JUCE ^[42]	C++	6.0.7 (2021)	custom (CPU)	Windows, Linux, macOS, Android, iOS
Nuklear ^[43]	C	4.06.2 (2020)	custom	Windows, Linux, macOS
Qt ^[44]	C++ (many bindings)	v6 (2020)	custom	Windows, Linux, macOS, Android, iOS
wxWidgets ^[45]	C++ (many bindings)	3.1.4 (2020)	native/GTK/Qt	Windows, Linux, macOS

Table 3.2: GUI Toolkits

CEGUI was initially designed to provide user interface functionality for game applications. Therefore, it provides several OpenGL and Direct3D backends as well as integration for some game engines [34]. Unfortunately, it has not been maintained since 2016 and therefore lacks modern backends implementation such as Vulkan.

Whilst FLTK is a mature and lightweight technology, the offered styling options did not meet the set requirements of a modern and creative environment, therefore, the library was not further evaluated.

GTK and nana both mainly utilize the CPU for rendering. While this is not necessarily a negative aspect, the specified requirements demand a highly performant renderer. Therefore, both were not further considered.

While wxWidgets is a versatile platform leveraging native OS renderers and offering multimedia playback capabilities, the 1992 founded technology was discarded due to its rigid nature and lack of hardware acceleration.

imgui and Nuklear both operate as immediate mode GUI libraries. In contrast to retained mode, immediate mode entails that no information is stored from one rendering to the next. The entire user interface has to be reevaluated and re-rendered. This approach is commonly used in computer graphics applications due to its simplicity of integration. However, it was deemed not suitable based on the specified requirements.

JUCE and especially Qt provided a diverse toolset for authoring rich graphical user interfaces. However, due to a more strict licensing, the technologies were not deemed suitable for further investigation.

Avalonia, Electron, and Flutter all use Skia either directly or indirectly through other frameworks. As already elaborated, Skia is an exceptional renderer driving many GUI applications, however, we believe that a dedicated GPU renderer can still outperform a general purpose renderer.

Approach

The following chapter describes the approach, the architecture, and design choices made for constructing the envisioned UI system named Elemental UI according to the specified requirements.

4.1 System Architecture

The Elemental UI architecture is built on the concept of layers, abstracting and unifying lower-level functionality for higher layers. Figure 4.1 depicts the architecture of the framework.

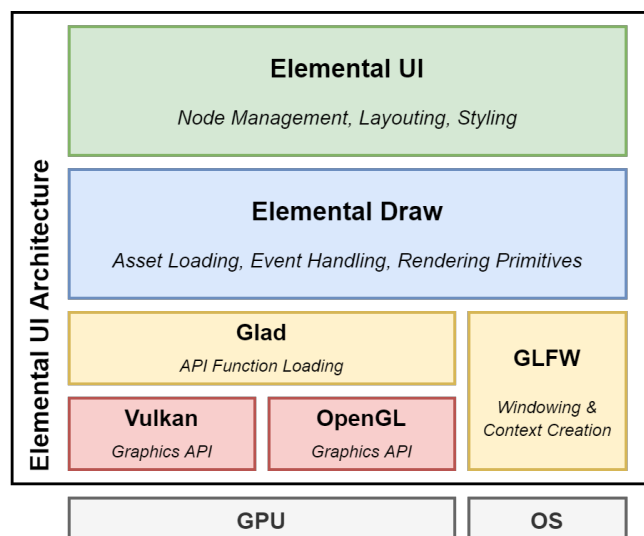


Figure 4.1: The architecture of the Elemental UI Framework (Own source).

Glad At the lowest level, Vulkan and OpenGL (marked in red in Figure 4.1) expose an API to communicate with the GPU and issue a variety of commands. To access said API (especially for Vulkan), the functions have to be dynamically loaded on run-time. This is typically done through a loader library, which is commonly auto-generated from the API specifications. Glad [46] was the preferred loader for the Elemental UI architecture, as it can generate both a Vulkan and an

OpenGL loader, is minimal, and integrates well with GLFW.

GLFW Apart from accessing the GPU, the system also requires the means to create a window frame provided by the operating system, create a context for the appropriate backend API, and handle user input. These tasks are highly platform-specific and require individual approaches and implementations for every operating system. Writing windowing implementations for every OS is beyond the scope of this thesis. Therefore, the widely used GLFW [47] library was embedded into the architecture, enabling simplified access to window creation and events. Unfortunately, GLFW currently does not support mobile platforms, hence limiting the portability of the Elemental UI framework onto handheld devices. Possible solutions and an outlook regarding this issue is presented in Chapter 7.

Elemental Draw The Elemental Draw library (marked in blue in Figure 4.1) sits on top of the previously mentioned components and ties them into one coherent interface for window and context creation, event handling, and, most importantly, rendering primitives. Furthermore, Elemental Draw loads and manages resources such as images and fonts through `stb_image` [48] and `FreeType` [49], respectively. The library only provides the bare minimum functionality for rendering a predefined set of primitive shapes as well as text. Nevertheless, the provided capabilities are sufficient for many applications and can be combined to render much more complex shapes and UI elements, as will be demonstrated by Elemental UI. Furthermore, Elemental Draw can be directly used by a developer to build any type of application on it, be it a 2D game or a visualization software. A more detailed breakdown of the Elemental Draw library is provided in Section 4.2.

Elemental UI Elemental UI is the topmost layer in the architecture stack. It provides a tree structure, where every node specifies its styling, positioning, and events, similar to the document object model (DOM) that is utilized in web-standards such as HTML [50]. The library provides all the necessary interfaces for managing said node structures and responding to events. In addition, it utilizes the rendering capabilities of Elemental Draw and transforms the user-defined tree structure into primitive draw calls that are then issued down through the stack. In contrast to Elemental Draw, the Elemental UI library fully exposes the underlying Elemental Draw library, so that custom rendering code can be developed for cases where the provided default node rendering logic is insufficient. Further elaborations of the Elemental UI system are provided in Section 4.3.

4.2 2D Rendering Library - Elemental Draw

The choice of the programming language has major implications about the range of performance that can be achieved. Therefore, as interpreted languages are inherently slower, the choice was narrowed down to low-level compiled languages. Ultimately, C++ was selected as the main language for Elemental Draw, due to its portability, raw speed, and ability to natively interface with most operating systems.

4.2.1 Cross-Compatibility

As already mentioned, cross-compatibility is one of the core pillars of our system. If the proposed framework cannot provide a large target-set, there is no need for it to exist in the first place, as

every platform has its own development tools. Therefore, multiple design considerations have been made to ensure cross-compatibility.

Modular Backend

Elemental Draw utilizes exclusively the GPU for rendering. As shown in the architecture in Section 4.1, a graphics API is required to access the GPU. Conventionally, the hardware vendors decide which specific graphics API the hardware supports. However, in some cases, a graphics API can be unavailable on certain operating systems, even though the underlying hardware is compliant with the graphics API specifications. For example, the Apple 16" MacBook Pro - AMD Radeon Pro 5300M [51] variant utilizes the AMD Radeon Pro 5300M graphics card that is Vulkan 1.2.133 compliant [52]. Nonetheless, the macOS operating system and Apple devices, in general, do not support Vulkan [12, p. 372]. Although, there are initiatives to enable Vulkan on macOS through translation layers such as MoltenVK [53], the technology has proved to be not practical in the context of Elemental Draw and was, therefore, omitted.

Evidently, a well-considered choice of the graphics API is crucial when portability is important, especially when rendering is the primary task of the library. Therefore, the five common graphics APIs were examined.

	Windows	Linux	macOS	iOS	Android	Xbox	PlayStation	Web
OpenGL [54]	✓	✓	(✓) [55]		✓			
Vulkan [56]	✓	✓			✓			
DirectX [57]	✓					✓		
Metal [58]			✓	✓				
OpenGL ES [59]	✓	✓	✓	(✓) [60]	✓		✓	✓

Table 4.1: Graphics API compatibility with platforms. (✓) denotes deprecated support.

As can be seen from the Table 4.1, there is no single API that is supported across all targets. The following is a breakdown of the APIs in respect of their supported platforms:

OpenGL OpenGL, OpenGL ES, and Vulkan are graphics APIs managed by the Khronos Group consortium [15, p. 35] [12, p. 376]. As an open standard, these APIs benefit from widespread adoption and, therefore, strive for compatibility across a wide range of devices. OpenGL is supported on all three major desktop operating systems and Android. However, the support on macOS is deprecated and only versions up to OpenGL 4.1 [55] are currently supported.

OpenGL ES OpenGL Embedded Systems (OpenGL ES) is similar to OpenGL but, as the name suggests, is designed for embedded systems where hardware resources are sparse, as, for example, in mobile devices [15, p. 735-736]. Therefore, it covers all the major desktop systems, mobile operating systems such as Android and iOS, and PlayStation. Additionally, it also enables hardware acceleration for web platforms through WebGL, which is semantically based on OpenGL ES 2.0 [61].

Vulkan Vulkan is officially regarded as the successor to OpenGL. The new API has the same goals as OpenGL to reach as many platforms as possible, in some cases with the help of compatibility layers. In contrast to OpenGL however, Vulkan shifts the responsibility of resource management into the hands of the programmer, thus providing considerably more control over

the exact behavior of the GPU [12, p. 16]. A more in-depth overview of the strengths of Vulkan, as well as comparisons to OpenGL, is presented in Section 4.2.2.

DirectX DirectX is a Microsoft-owned set of APIs that provide different functionalities. DirectX is comprised of Direct2D and Direct3D for 2D and 3D rendering, respectively. DirectX also encompasses more APIs, such as DirectWrite for font rendering, DirectXMath for accelerated math and linear algebra operations, XAudio, and XInput for signal processing and controller input events [57]. As a Microsoft product, the DirectX APIs target Windows and Xbox platforms only.

Metal Similar to DirectX, Metal is the proprietary graphics API for macOS, iOS, iPadOS, tvOS, and Apple devices in general. It is, therefore, the favored interface for communicating with Apple hardware [58]. In an attempt to force out foreign APIs from their ecosystem, Apple has deprecated support for the open standard API OpenGL [55] and is not planning to support Vulkan, therefore leaving Metal as the only target API for Apple hardware moving forward.

It is evident from the compatibility Table 4.1, that no one graphics API can provide full native compatibility over all platforms. Therefore, the Elemental Draw library was designed in a way that the rendering backends could be interchanged and new implementations added at any time. This was achieved by introducing a new rendering API comprised of a set of carefully chosen primitive draw commands. An exact breakdown of the introduced API is presented in Listing 4.2. In the context of this thesis, Vulkan was implemented as the first backend, which establishes compatibility with Windows, Linux, and Android. Afterwards, an OpenGL backend was introduced to cover macOS-capable devices. Further sensible extensions could be a Metal backend for native support on hardware from the Apple ecosystem, OpenGL ES for web rendering, and, finally, a software rasterizer in cases where no discrete GPU is present on low-power devices.

Build System

As already alluded to in Section 2.1, many factors can influence the portability of a project, and a major one is the compiler, which is responsible for turning the source code into an executable. There are several compilers available, some only on select platforms. The most popular ones are Microsoft's Visual C++ for Windows platforms and GCC for macOS and Linux [11, p. 3].

To reach as many platforms as possible, it has to be ensured that first, the source code does not utilize exclusive compiler features that are not present elsewhere and, second, a simple means of generating a project configuration is provided. For example, the set of inputs required for the Microsoft Visual C++ compiler to convert source code into an executable is different from the inputs required for GCC.

Typically, a build system is then introduced to automate among other tasks, the process of supplying the necessary compiler and linker flags for generating an executable [11, p. 9]. Generally, a build system is simply a script that abstracts a set of commands into simpler, more common commands, for the convenience of the developer. There are several build systems available, one of the more popular being GNU make. It automatically detects a developer-generated script called the `makefile`, which typically states all the necessary steps for successfully compiling the project at hand [62].

Unfortunately, a make file alone does not provide a C++ project sufficient flexibility when it comes to setting up a development environment. Code development is commonly performed inside an integrated development environment (IDE) [11, p. 9], which provides alongside a code editor, productivity features to speed up development. Often the build system is directly integrated into the IDE for a more streamlined experience. It is not uncommon for platform vendors

to also provide native IDEs that are appropriately set up to work on native applications for that platform. One example is Microsoft's Visual Studio IDE, which works on native Windows applications [11, p. 67].

To retain freedom of choice regarding the IDEs and, thus, widen the development potential on different platforms, all the necessary script, make, and solution files have to be provided. However, it is a time-consuming task to create and maintain these. Therefore, another abstraction layer has to be introduced. By creating a manifest file that contains detailed descriptions of where the source files are located, as well as configurations and other preferences, we can ask a build system generator to generate a solution of choice, according to the manifest. There are a number of such generators available, such as premake, which is based on a Lua manifest file [63] or the more popular CMake [64], which uses its own language for the manifest files.

Furthermore, by allowing the manifest files to not only contain descriptive content on the whereabouts of the source code files but also logical operators, conditional includes are made possible according to predefined switches and flags. For example, the elemental draw project generator provides a flag for the rendering backend, whereby the preferred API can be chosen. Due to the individual backend implementations being segregated into separate folders, entire source code modules can be excluded from the build system, effectively decreasing compilation times.

In the context of this thesis, CMake was selected as the main build system generator due to it being the preferred system for interacting with vcpkg.

Implementation Separation

The header file `context.h` defines the public rendering interface in the `Context` class (the exact list of methods are provided in Listing 4.2). Its constructor is kept private, so no foreign function can construct it independently. Instead, a static method `create()` is exposed, returning a pointer to the context object. Next, a Vulkan, as well as an OpenGL-context were created by inheriting from the `Context` class and implementing their respective behavior (Listing 4.1). Therefore, invoking the exposed `create()` method creates a new Vulkan or OpenGL context and returning it as a `Context` pointer.

This approach, referred to as PImpl (Pointer to implementation), not only decreases compile times by reducing header file includes, but also effectively hides all the implementation details, which would otherwise have been exposed [65, p. 147-148]. Furthermore, from the consumer side of the API, the interaction with the interface is streamlined and consistent, regardless of the target platform.

```
// context.hpp
class Context
{
public:
    static Context* create();
protected:
    Context() = default;
    virtual ~Context() = default;
}

// context_impl_vulkan.cpp
Context* Context::create()
{
    return new ContextImplVulkan();
}

// context_impl_opengl.cpp
Context* Context::create()
{
    return new ContextImplOpenGL();
}
```

Listing 4.1: Plmpl implementation of Context Class

Dependency Management

A considerable effort went into reducing the necessity for foreign code inside the project as dependencies might utilize platform-specific code or rely on specific compilers features, hindering the portability of the entire project. Nevertheless, reinventing the wheel for every project is time-consuming; therefore, a select list of established libraries were considered to expedite the development time.

As already established in Section 4.1, the dependencies were as follows:

- GLFW [47] - For window and context creation and input handling.
- glslang [66] - For shader compilation.
- stb [48] - For reading and writing image files.
- FreeType [49] - For font loading and glyph generation.

In C++, the integration of dependencies is a surprisingly challenging task. In contrast to the more modern languages, such as JavaScript or Rust, there is no one dominant package management system for C and C++. Six possible approaches were evaluated for managing the dependencies in a clean and structured manner. The main focus was once more the portability of the entire system.

1. Including Source - An obvious approach would be to include the required library in its entirety into the project. However, this has a few downsides. On the one hand, the project's size increases drastically, as many unnecessary files will be included, such as demos and documentation. On the other hand, custom glue scripts have to be made to interface with the build systems, which the dependencies might be using. Furthermore, the licensing of the dependencies might prevent redistribution or uploading of the source code.
2. Git Submodules [67] - Instead of manually downloading the sources for the libraries, git submodules can be utilized, which allow the declaration of dependencies in a git repository. The main benefit from this approach is the reduction in the size of the repository, as the dependencies are not duplicated inside every repository, depending on the library. Furthermore, the process of updating dependencies is facilitated as only the declaration has to be adjusted accordingly. Git tools automatically detect the changes and pull in the desired commit of the dependency. However, the manual work of coordinating the exotic build systems that might arise inside dependencies still has to be done by hand.
3. Manual Restructuring - An alternative to including the dependencies in their entirety is to consider only certain modules of interest for copying. By selecting only the relevant code base, the size of the dependency can be somewhat reduced. Furthermore, a unified folder structure can be employed for every dependency, improving the readability of the project as a whole. Unfortunately, a considerable downside to this approach is the significant workload beforehand to convert all dependencies into a unified format. Reordering the source code will most likely negatively affect the build system, as it may no longer be able to find the source files in their predetermined locations. Finally, accepting updates from the original branch might be impossible if the changes are too comprehensive.
4. Forked Submodules - The fourth approach can be identified by combining the manual restructuring approach with submodules. By forking the dependency, a separate development branch can be opened where all the restructuring can be applied. The fork can then be utilized as a submodule for the main project, inheriting all its benefits. Unfortunately, the same problems regarding the inherent difficulty in updating the dependency still apply; however, it being on a separate repository increases the separation and might simplify the process of updating from the original branch.
5. Pre-built Binaries - Instead of providing the source code and instructions to compile binaries, the binaries themselves can be distributed, eliminating the compilation process. The dependency folder can then be considerably simplified, containing only header files and the binaries.

Although this approach might appear reasonable at first glance, it still entails some significant issues. First, the file size of the generated binaries is typically considerably larger than their source code counterparts, therefore, notably increasing the size of the entire repository. Regardless of the size, there is still the issue of obtaining the pre-built binaries in the first place. Although binaries are sometimes generated by the developers of the libraries and made publicly available on binary repositories, it is still not a given that all the necessary platform and architecture binaries will be provided, as, for example, is the case with GLFW, where only Windows 32/64 and macOS 64 are provided [47].
6. Package Manager - The responsibilities of a package manager are downloading, installing, updating, and removing packages or modules. It is a convenient and centralized way to address dependencies and can alleviate all the shortcomings of the previously stated approaches. An additional benefit of a good package manager is the ability to notify of security alerts in dependencies when they arise. As an example, npm [68], a popular Node

package manager for JavaScript developers, requires the declaration of dependencies in a so-called `package.json` manifest file [69]. The packages can, therefore, be downloaded directly from a common repository by the package manager and the developer can be notified of any vulnerabilities in their dependencies.

As illustrated, it makes sense to utilize a package manager as it can simplify setting up the project on different platforms by automatically managing the dependencies. Therefore, during the development of Elemental Draw, `vcpkg` [70], a package manager for C++ libraries was utilized. It was specifically chosen for the following reasons.

As C and C++ libraries are structured, built, and hosted completely differently from one another, `vcpkg` offers a flexible and decentralized approach to packaging libraries. A port contains a manifest that describes the library it is packaging as well as instructions on where to download the source for the specific library and the changes that are necessary to modify the library to comply with the `vcpkg` interface. Although there is a considerable amount of work to be done beforehand to set up a port, the benefits are the simplified maintainability of the preexisting ports, as libraries can be updated rather quickly by adjusting the download repository. `Vcpkg` contains 1'610 prepackaged ports [71] and provides a system for introducing custom ports, called ports overlay [72], for cases where a dependency might not already be prepackaged.

In a manner familiar to other package managers, `vcpkg` allows the downloading and installation of libraries. During the installation process, `vcpkg` applies the predefined changes to the library and compiles it to the desired target triplet. Triplets are a set of target configurations that contain information about the target architecture, platform, and library linkage. The officially supported targets are as follows [73]:

- arm-uwp
- arm64-windows
- x64-linux
- x64-osx
- x64-uwp
- x64-windows-static
- x64-windows
- x86-windows

In addition, there are community triplets as well as custom triplets that can be defined to further increase portability. However, unofficial ports are not tested and can, therefore, not be guaranteed to work with all ports.

The patches specified in the ports allow for the modification of the library, which, in some cases, can improve the portability of the library, which was originally not set up to be compiled to certain target platforms. This dedication to portability was one of the main reasons why `vcpkg` was chosen as the package manager for the development of Elemental Draw.

Continuous Integration

In summary, there are numerous possible points of failure when it comes to cross-compatibility. Starting with the dependencies to the build system and even the source code for Elemental Draw can introduce unintended side effects that might cause incompatibility with a platform. Therefore, it was important to closely track the compatibility of the project during development to

detect possible violations immediately.

Continuous Integration (CI) is an automated system that performs integration tasks [74, p. xx]. Although often described as a useful system for larger teams, the scripts run by the automation system are defined by the developer and can, therefore, perform any generic action necessary. Thus, the CI system was used to integrate the entire technology stack of Elemental Draw, by producing compiled binaries on every commit.

The CI system utilized in the context of this thesis was GitHub Actions [75]. The choice of this CI solution was natural since the Elemental Draw project was hosted on GitHub. The setup comprised of setting up a `.yaml` script detailing what actions were to be executed when. For the library introduced in this thesis, the following actions were specified:

1. Download Repository - On every commit to the master branch, the repository is downloaded to a GitHub Actions runner machine.
2. Setup Environment - The environment is prepared by downloading or updating necessary build dependencies.
3. Generate Solution - Through invoking CMake, the appropriate project solution is generated according to the current platform and the linking parameters.
4. Build - The project is then built via the appropriate compiler from the platform.
5. Package - If the build step is successful, the generated binaries alongside the header-files are packaged into a compressed folder.
6. Upload - The generated artifacts are uploaded from the runner to a database where they can be accessed later.

This process is performed six times for three platforms (Linux, Windows, and macOS) and two linking configurations (static and dynamic). Altering the architecture would have also been possible but was omitted, as it was deemed unnecessary.

Unfortunately, the current CI system is only able to detect static issues, such as misconfiguration or syntax errors. Runtime errors, especially those caused by loading unsupported backends, cannot be discovered because the binaries are not executed in the CI pipeline. Although it is technically possible to execute generated binaries on the remote CI systems, the runners are typically set up in a virtual environment where accessing hardware, especially the GPU, is not possible. Therefore, the CI system could be further improved by adding an additional step, with the binaries transferred to a trusted machine where they are executed.

Language Bindings

The low-level programming language C++ was chosen for Elemental Draw not only because of its high performance but also with the intent to provide portability across platforms and languages. It is understandable that, in some cases, the speed of development might be preferred over the speed of the resulting application. In such cases, C++ code might prove difficult to work with, as the low-level language requires more attention regarding memory management and strict type declaration. Commonly, higher-level languages provide the means for loading and interfacing with lower-level C shared libraries, to communicate with the operating system, to benefit from preexisting low-level libraries, or to shift performance critical code into a lower-level language due to performance concerns.

The language C provides a great standard interface for types and functions, and, therefore, there are many dynamic library loaders available for different higher-level languages or, in some cases, such loaders are already provided with the language.

Elemental Draw was written in C++ and, therefore, naturally allows being interfaced by another C++ codebase directly. In addition, a C wrapper was introduced on top of the C++ API to expose the library's functionality in a more accessible interface. This was achieved by introducing a new file called `c_bindings.cpp`, which wraps all the public calls to the Elemental Draw API, including the constructors and destructors, as the library was written in an object-oriented style while the C language does not have the concept of classes. Furthermore, small adjustments were made to the new C bindings, where complex C++ classes, such as strings and vectors, were adapted to their C counterparts. The adaptation process was performed rapidly, as great care had been taken early on, to utilize C types as much as possible for exposed classes as well as public methods. Still, for the implementation itself, the full range of C++ types was used as it was fully encapsulated through the PImpl approach.

C# Binding The exposed C API allowed for a wide range of new language bindings to be introduced. For example, the C# language directly provides the means to load libraries through the annotation "DllImport" [76].

```
using System;
using System.Runtime.InteropServices;

// load
[DllImport("elemd.dll", CharSet = CharSet.Unicode)]
public static extern void fill_rect(IntPtr ctx, float x, float y, float w, float h);

// call
fill_rect(ctx, 10, 10, 20, 20);
```

When the declared function is called, the program executes the implementation from within the shared library, effectively making Elemental Draw accessible through C#.

Java Binding Similarly, the Java language also provides the means to load a library directly from within the language with the system call "System.loadLibrary" [77].

```
// load
static { System.loadLibrary("elemd"); }
private static native void fill_rect(int ctx, float x, float y, float w, float h);

// call
fill_rect(ctx, 10, 10, 20, 20);
```

Some additional wrapping has to be performed for the Java binding, as Java expects certain prefixed names and does not provide the concept of pointers. Therefore, the interface should be adjusted to utilize `uint64_t` types to support 32-bit as well as 64-bit pointers. In the context of this thesis, language bindings were not a major focus. Therefore, there is potential for future improvements in this aspect. Possible refinements to the binding system will be presented in Chapter 7.

Python Binding For the programming language Python, the library loading is exposed through the standard library "ctypes" [78].

```
from ctypes import *

# load
lib = cdll.LoadLibrary("elemd.dll")
lib.fill_rect.argtypes = [POINTER(Context), c_float, c_float, c_float, c_float]

# call
lib.fill_rect(ctx, 10, 10, 20, 20)
```

A reference Python language binding was created in the context of this thesis, to demonstrate what a more advanced binding could look like, by reintroducing the concept of classes.

4.2.2 Rendering

Leveraging the GPU for rendering has the potential to drastically increase performance, especially when it comes to large-scale bitmap filtering operations. However, by shifting the rendering operations to a separate hardware, the task at hand inherently increases in complexity since, besides rendering, new responsibilities arise. Among these, resource allocation has to be addressed on the GPU; communication barriers have to be established between GPU and CPU; and, most importantly, a rendering strategy has to be devised for issuing rendering commands with as few state changes as possible.

In general, performing rendering operations is much more involved on the GPU than on the CPU. Furthermore, the degree of difficulty also depends on the chosen rendering API. While immediate mode in OpenGL allows for rendering points, lines, and vertices with little effort or setup, the approach suffers from poor performance [15, p. 23]. In contrast, Vulkan requires a considerable amount of work upfront, even for the simplest rendering operation, but it provides opportunities for optimizations on the lowest level.

Despite these difficulties, to achieve the best possible performance, it was decided to utilize the GPU for all rasterization tasks. Furthermore, Vulkan was selected as the main backend target, as it offers the highest performance potential compared to older graphics APIs, due to its exhaustive configuration capabilities.

Vulkan

GPUs can perform a vast number of simple operations in a highly parallel manner. However, in order to communicate with the GPU, so-called synchronization barriers have to be established. Performing too many synchronizations on the GPU can lead to idle time and wasted performance. Therefore, special care has to be taken to strategically sequence the calls to the GPU to avoid unnecessary synchronization between the host and application [12, p. 322].

Considering this fact, we identified an opportunity for accelerating the rendering, by delivering all the necessary data upfront and issuing only one rendering command. Hereby, only one synchronization barrier would have to be established as well as only one data upload would be performed. This approach is known as instancing and is used in games to render vast numbers of objects in a performant manner [12, p. 247-248]. While different properties can be defined that vary from instance to instance, the main limitation of this approach is that the instances have to consist of the same geometry. Therefore, instancing can only be used, if a common shape between

the instances can be identified.


In the case of GUI rendering, we discovered that the most common shapes in UI applications can be described as axis-aligned rectangles. Circles, rectangles, rounded rectangles, squares, outlined rectangles, font glyphs, and image bitmaps can all be expressed within a rectangle. Therefore, by constructing a list containing positions, dimensions and contents of individual rectangles, the instancing approach can be utilized to render the entire user interface in one draw call. Figure 4.2 depicts how all primitives supported by Elemental Draw can be rendered as rectangles. The list of rectangles is generated by invoking shape draw calls exposed in the Elemental Draw API (Listing 4.2). The desired shapes and their positions are stored in the list and only submitted to the GPU as one instanced draw call as soon as the `draw_frame()` method is invoked. To ensure that all rectangles can be fitted in one draw call, SSBOs are utilized due to their higher capacity. Furthermore, to allow for alpha blending, an operation, where none opaque colors are blended according to their alpha value, the correct order of rendering has to be employed.

Unfortunately, not all primitives can be expressed with this approach. Bespoke shapes or free form paths cannot be realized directly.

While it is technically possible to construct unique meshes to allow for custom shapes to be rendered, the added shapes can potentially interfere with the ordering of elements, causing a split in the instance list. For example, issuing six rectangle shapes and one complex shape call in-between will cause three rendering calls. One for the first three rectangles, one for the complex shape, and a third one for the three remaining rectangles. In this case, omitting the complex shape would have made it possible to combine the six rectangles into one instanced draw call. The restriction to a limited set of shapes was done deliberately to prevent such breaks in the instance chain. Possible improvements regarding this issue are given in Chapter 7.

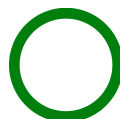
Rendering Primitives

Given the information regarding the color, corner radius, and line width, the desired shape can be rendered utilizing shaders. As an example, a simple circle can be rendered, by evaluating the vector length of every x and y coordinate pair inside a rectangle and only coloring those pixels, that lie within the radius r of the circle as visualized (Equation 4.1).



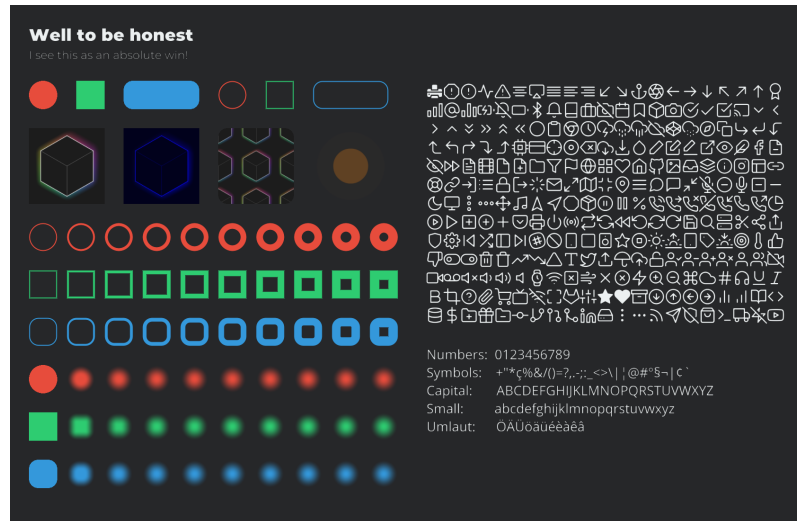
$$\sqrt{x^2 + y^2} < r \quad (4.1)$$

Outlines can be achieved, by introducing a second circle with a smaller radius r_2 and carving out the first circle, by enforcing that both conditions, smaller than r and bigger than r_2 are met (Equation 4.2).

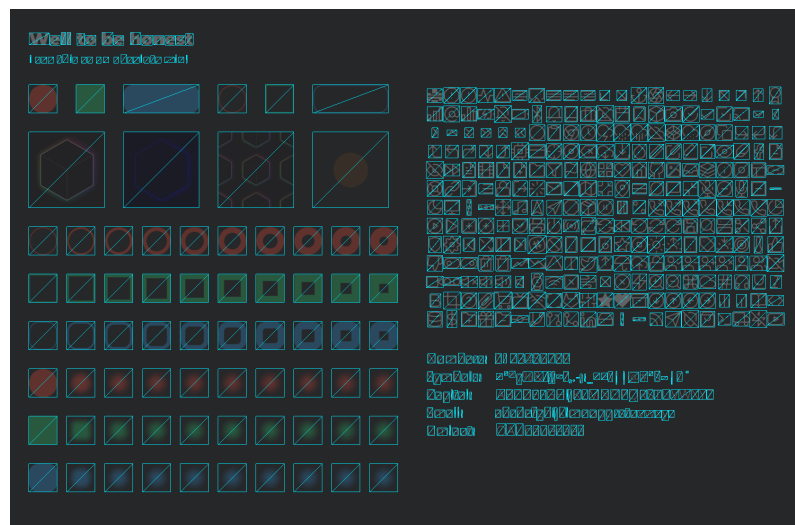


$$\begin{aligned} \sqrt{x^2 + y^2} &< r \\ \sqrt{x^2 + y^2} &> r_2 \end{aligned} \quad (4.2)$$

Applying this concept, one shader was created providing the ability to render all the shapes depicted in Figure 4.2 and provided in Listing 4.2. The base equation for the shapes was derived



(a) Overview of supported primitives in Elemental Draw



(b) Wireframe view of the supported primitives in Elemental Draw

Figure 4.2: An overview of all supported primitives, font, and bitmap rendering provided by Elemental Draw (a). Figure (b) depicts the generated wireframe mesh used for the simultaneous rendering all elements. (Own source).

from Inigo Quilez's `sdRoundedBox` algorithm [79].

```
// Stroke
void stroke_rect(...);
void stroke_rounded_rect(...);
void stroke_circle(...);

// Fill
void fill_rect(...);
void fill_rounded_rect(...);
void fill_circle(...);

// Draw
void draw_text(...);
void draw_image(...);
void draw_rounded_image(...);

// Shadow
void draw_rect_shadow(...);
void draw_rounded_rect_shadow(...);
void draw_circle_shadow(...);

// Mask
void set_rect_mask(...);
void remove_rect_mask();

// State
void set_clear_color(...);
void set_fill_color(...);
void set_stroke_color(...);
void set_line_width(...);
void set_font(...);
void set_font_size(...);

// Submit
void draw_frame();
```

Listing 4.2: Rendering API exposed by Elemental Draw

Text Rendering

Text rendering was achieved, by first loading a font file utilizing the FreeType [49] library and converting all available glyphs from said font file to image bitmaps for texture mapping. To avoid creating individual glyph images, one texture atlas was created, encompassing all glyphs, laid out in a grid structure. Alongside it, a mapping was created, encoding individual characters to positions inside the texture atlas as well as their dimensions. The text can be then rendered, by constructing rectangle meshes for individual glyphs and filling them with the appropriate texture regions.

In most cases, this is a passable approach for font rendering, however, it suffers from aliasing artifacts on small font sizes and blurry edges on big font sizes. Increasing the bitmap sizes of the glyphs can improve overall quality but at the same time also increases the memory footprint.

To avoid increasing the bitmap sizes, a technique called signed distance fields (SDF) can be employed. By encoding distance information of the glyph in a bitmap, a clean separation between the glyph and background can be produced at virtually any scale, effectively eliminating the blurring of the edges. The original SDF approach was later improved by Viktor Chlumsky, by introducing a multi-channel approach that eliminated several artifacts common to SDFs [80]. A reference implementation was also provided by Chlumsky [81], which was integrated into Elemental Draw. However, due to practical reasons, the multi-sample distance field font rendering implementation was not included in the submitted version.

4.3 GUI Library - Elemental UI

Whilst Elemental Draw performs all the heavy lifting by communicating with the operating system and providing a window with a context ready to be drawn to, Elemental UI builds on top of the provided system and once again abstracts it for a more convenient use. In particular, Elemental UI abstracts away the primitive rendering calls and provides functionality for defining media content, styles, layouts, and interaction logic. Many concepts and implementation details were mainly influenced by web technologies, such as HTML, CSS, and JavaScript's event system, and the document object model.

4.3.1 Node Hierarchy

In Elemental UI every entity is represented as a node. Every node can have one parent and multiple child nodes. This allows for a tree structure of nodes to be generated. Nodes are implemented as an abstract class to provide their composition capabilities to other, more specialized nodes, such as the `element` class. Elements represent generic container entities mostly utilized for layouting purposes. In contrast, the `heading` class is used specifically for displaying text. Therefore, it is only used as a leaf node and does not provide any layouting capabilities or the ability to accept child nodes. Lastly, the Elemental UI implementation provides a `text_field` class that is similar to the `heading` class but allows the displayed text to be modified during runtime. The three classes define implementations, which, according to the current style of the node, issue draw commands to the Elemental Draw renderer. This system can be easily expanded if the desired functionality is not provided. For example, a slider element could be implemented by deriving from the node class and implementing its visual appearance with the primitive rendering commands exposed through Elemental Draw.

A node tree can be created by instantiating individual nodes, providing the appropriate contents and style configurations, and chaining them by calling the `add_child(node* child)` methods on the parent nodes. Once a node tree is declared, the root node can be passed to the Elemental UI context. The context automatically traverses the tree structure, performing layouting instructions and caching all the calculated positions of the nodes. Finally, a second traversal is executed, however, this time calling all rendering commands from the nodes with their respective cached positions. If more operations have to be done after the tree is rendered, as is the case with transitional effects, another traversal is requested, to be executed immediately after.

4.3.2 Styling

In an effort to separate content from styling, the Elemental UI implementation closely follows the style properties defined by the Cascading Style Sheets (CSS). However, liberties were taken during the implementation to diverge functionality from the CSS specification where it was deemed necessary.

In addition, some new generic types were introduced, such as `measure_value` to assist in representing measurement values in pixels or percentages. Similarly, the `maybe<T>` template structure was introduced to help to specify a default state of a property where the type itself cannot express an appropriate default value. The implemented properties are as follows:

- `width`, `height` - These are of type `measure_value` and provide information regarding the dimensions of the element.
- `min_width`, `min_height`, `max_width`, `max_height` - The minimum and maximum properties specify the dimensions to which an element is allowed to grow or shrink, whilst the type is `measure_value`.
- `padding` - The `padding` property is used to insert whitespace in the four main cardinal directions. The whitespace is added to the inside of the element and can therefore, increase the size of the element. The type is defined as a `float [4]` to represent the four directions, north, east, south, and west.
- `margin` - Similar to `padding`, `margin` is also used for adding whitespace. Contrary to `padding`, however, the whitespace is inserted outside the element, which can cause the element to be re-positioned. The type is again `float [4]`.
- `border_width` - The property specifies the width of the border to be stroked. As a `float [4]` type, it provides individual widths for the four main cardinal directions.
- `border_color` - `border_color` should be used in conjunction with the `border_width` property. It is of type `color` and specifies the color of the border stroke.
- `border_radius` - The property of type `float [4]` specifies the roundness of the four corners of the rectangle. The four float values correspond to the four diagonal cardinal directions: northwest, northeast, southwest, and southeast.
- `background_color` - Of type `maybe<color>`, the `background_color` specifies the background color of the element. Although a black color with zero transparency could be seen as an appropriate default value, this property can conflict with `background_image`. The additional information stored in the `maybe` struct provides a clear distinction, whether the background is supposed to be rendered transparent or if the values are not set and the `background_image` property should be considered for providing the appropriate background decoration.
- `background_image` - A type of `maybe<image*>` that specifies whether a bitmap should be rendered as the background to the current element.
- `color` - The `color` property specifies the font color and is of type `color`. In this instance, no `maybe` type is required as black is always an appropriate default color. Font-related properties are only utilized by node classes that handle text rendering such as `heading` and `text_field`.

- `font_family` - The property of type `font*` specifies the font type.
- `font_size` - The property of type `float` defines the size of the font.
- `display` - The `display` property defines the layout properties of the current element. The enumerated type `Display` with options `{BLOCK, INLINE}` specifies whether the element should stretch horizontally (`BLOCK`), filling the entirety of the width, provided by its parent element, or whether the element should shrink (`INLINE`) to its minimum width according to its child elements. The `BLOCK` property is mostly used for container elements, whilst `INLINE` is more appropriate for leaf nodes, which define their own width, as is the case with the node type `heading`.
- `overflow` - The `overflow` property enumerates three options `{SHOW, HIDDEN, SCROLL}` and is of type `Overflow`. In cases where an element exceeds the size of its parent, a strategy has to be devised on how the element should react. While `SHOW` does not act at all and lets the content overflow, the option `HIDDEN` cuts off the overflowed content. The third option `SCROLL` converts the element into a scrollable item and renders scroll bars.
- `scroll_bar_color` - Related to the `overflow` property with option `SCROLL`, the displayed scroll bar color can be adjusted with the `scroll_bar_color` property of type `color`.
- `transition_time` - The `transition_time` of type `float` specifies the transition duration in seconds.

Compared to the CSS specifications, the implemented style properties are only a small subset. Still more properties have to be introduced; however, only the presented properties were considered within the scope of this thesis.

4.3.3 Event System

The interaction system implemented in Elemental UI is based on events. Thus, interaction resolution logic has to be written as a lambda function and added to nodes as an event-listener. The following event types are exposed on the node objects:

node_click_event The node click event is triggered, providing the left mouse button was pressed, while the mouse cursor was positioned within the bounding box of a node element. The event object provides a reference to the node which emitted the event as well as a struct containing x and y coordinates of the click origin.

Due to the inherent kd-tree structure of the nodes, where the parent node always fully encompasses all its child nodes, a fast search algorithm can be devised to identify the exact leaf node, where the click event should be triggered on. Given a click event with global coordinates, we can start a search on the root node and, according to the x and y coordinates, decide which child node encompasses those coordinates. Once a child node is found, the search can be continued recursively, ignoring all other siblings, as shown in Figure 4.3, steps 1 to 3. Once a leaf node is reached, all event listeners can be triggered on that node. Finally, the node returns the recursive call, allowing the parent node to employ its event-listeners if present. The events will, therefore, automatically be bubbling from leaf node to root node, as depicted in Figure 4.3, steps 4 to 6. This behavior can be interrupted by returning `false` in the lambda function, signaling the parent node that bubbling is not desired.

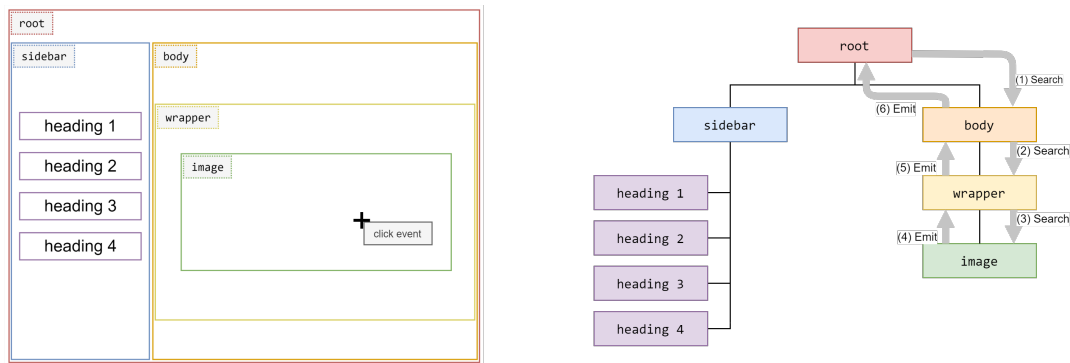


Figure 4.3: Event propagation diagram. Fictive layout (left), event propagation along node tree structure (right). (Own source).

node_scroll_event Similarly to the node click event, the node scroll event is invoked on a mouse scroll event. Hereby, the mouse coordinates are checked against the node tree and the leaf node triggers its event listeners. The bubbling behavior in the scroll event is dependent on the scroll state of the child node. In a scenario, where one scroll node is inside another, it is not desirable, for the child node to propagate the scroll event upwards, which would cause a double scroll translation. Therefore, the bubbling only occurs if the child node has reached a scroll limit in either direction. The `node_scroll_event` object provides a reference to the target node as well as a scroll delta in the x and y direction.

node_key_event Elemental UI allows for one node to be focused at any given time. Focusing a node is performed by clicking on it. The node key event is triggered upon any keypress from the keyboard and is triggered on the currently focused node. The `node_key_event` object has a reference to the focused object and provides the key scan-code as well as potential modifiers. The event has bubbling as the default behavior.

node_char_event Finally, the node char event acts similar to the node key event. The main difference being the contents of the `node_char_event` object. It contains, Unicode character instead of a key number. `node_char_event` is therefore mostly useful for events, where the literal character value is required instead of a key code. As an example, the `text_field` element utilized the `node_char_event` for inserting characters, as `node_key_event` would retrieve ambiguous results due to differences in keyboard layouts and language settings.

4.3.4 Transitions

The concept of transitions was also significantly influenced by web technologies. It entails a visual transformation from one form into another via a user event or through any other means. Hover effects are a typical example of transitions. Upon a user mouse move input, a `hover` event is triggered on the elements directly underneath the cursor. The elements can act upon the event as specified by the developer, for example by changing the background color or adding a border. To ease the transition for a smoother experience, we introduced a `transition_time` variable,

controlling the duration of the transition. The default value being 0, executes transition instantly from one frame to the next. Increasing the value spreads the transition over the specified amount of time.

The transition is implemented on all numeric style properties of a node. By defining the starting value x_0 and the desired end value x_1 , we can evaluate any value along the path from x_0 to x_1 given a factor t with the linear interpolation formula:

$$(x_1 - x_0)t + x_0 \quad (4.3)$$

These intermediate values can then be presented during the transition according to the transition time and transition progress.

The same is true for color properties, as color properties are stored as a four-component vector, corresponding to r red, g green, b blue, and a alpha channels.

$$\begin{aligned} (r_1 - r_0)t + r_0 \\ (g_1 - g_0)t + g_0 \\ (b_1 - b_0)t + b_0 \\ (a_1 - a_0)t + a_0 \end{aligned} \quad (4.4)$$

Again, a source and destination color can be specified and linearly interpolated to obtain mixed color values according to the mix factor t .

4.3.5 Dynamic Framerate

Graphical user interfaces usually tend to be static most of the time and typically change with user input. Therefore, it would not make sense to employ a constant render loop, as is the case for games. Re-rendering the presented UI without there being any visible change would waste GPU and CPU cycles and, therefore, produce unnecessary heat and waste energy.

The render loop created in Elemental UI is a mix of an event-driven and animation-based loop. In general, the framework awaits input events and issues a render call, as soon as one is given. Therefore, every event can directly lead to one render call. However, in the case where animations or transitions are expected to be played, components can request a subsequent render call, regardless of the input. On every render call, all the appropriate components update their progress and evaluate their new transition states, requesting further frames, if necessary. The renderer performs rendering calls, as long as individual components report being in such a transition state. This approach, therefore, allows for a high framerate rendering during transitions and animations and completely halts rendering as soon as the UI reaches a steady state.

As is presented in the performance evaluation Section 6.1, on middle to high tier GPUs, Elemental Draw can provide high-frequency rendering, in some cases even surpassing the refresh rate of a monitor. Providing higher framerates than the monitor can present does not grant any additional benefits, therefore, Elemental UI provides means to limit the framerate, in cases where consequent rendering calls are issued without any throttling, as is the case with animations.

Case Studies

To evaluate the viability of the Elemental UI framework, five case studies were conducted, each covering different aspects and capabilities of the Elemental Draw and Elemental UI libraries. Three applications were made directly, utilizing the rendering library Elemental Draw, to demonstrate the adaptability of the system. The two remaining applications were built with Elemental UI to demonstrate the layouting and animation mechanics.

5.1 Painting Application

The painting application provides a canvas as well as a color picker with ten predefined colors. The brush is a bitmap, which can be replaced by the user to achieve different brush stroke styles. The brush size can be modulated by the scroll wheel.

This application, though not designed for the creation of artistic masterpieces, helps to demonstrate the flexibility of the renderer, despite Elemental Draw being mainly tailored towards geometric user interface rendering.

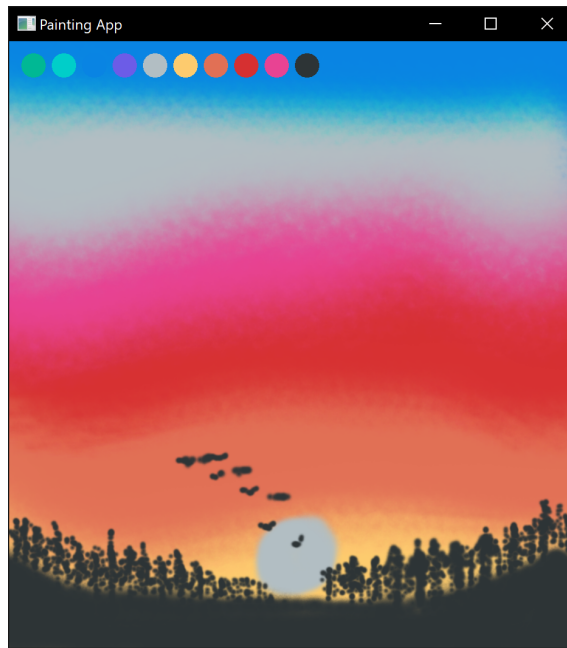


Figure 5.1: A sunset scene created with the painting application running on top of Elemental Draw. (Own source).

The technical implementation can be greatly improved since every single brush stroke is stored in memory and re-rendered on every input. Due to the employed instanced rendering technique, re-rendering as many as thousands of brush strokes does not degrade the performance. However, in future work, it is advised that the presented painting is stored as a whole in a texture rather than re-rendering individual strokes.

5.2 Breakout Game

Showcasing the real-time capabilities of the Elemental Draw renderer, a generic breakout game was created. Using the arrow keys as controls, a player can steer the paddle on the horizontal axis to bounce the ball back into the bricks. Upon colliding with a brick, it either gets destroyed, or depending on its type, its hit points get counted down. Once minimal health is reached the brick is destroyed and an extra life is spawned, falling down towards the paddle. The level is completed by destroying all bricks.

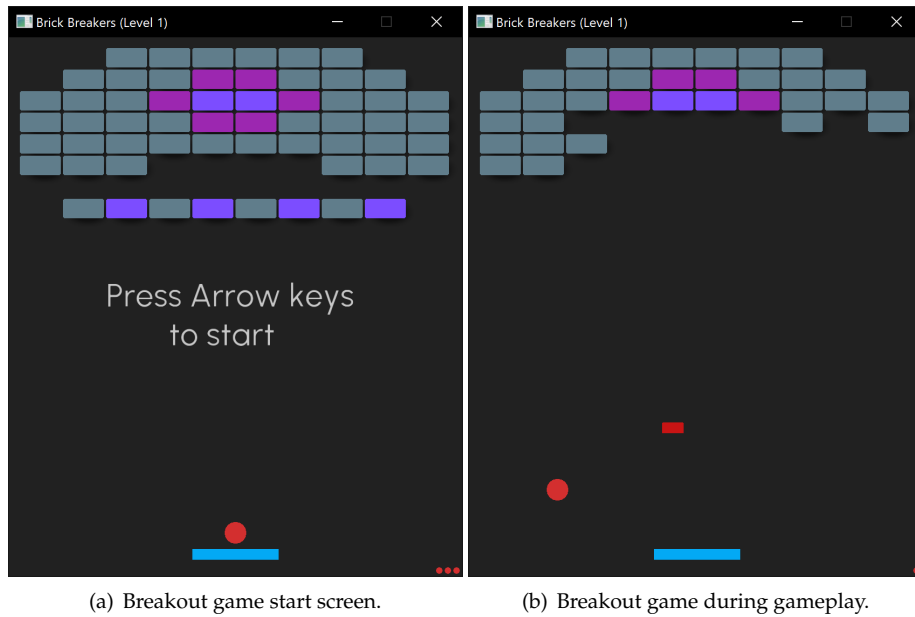


Figure 5.2: Breakout game application. (Own source)

In contrast to the painting application, the breakout game runs on a constant loop, independent of the user input.

5.3 Calculator

A visual calculator app was created on the Elemental UI framework, directly utilizing layouting and animation features. On mouse hover over the buttons, a transition plays, animating the background color of the buttons. Such an effect is commonly applied to elements to signal intractability with the visual element. Furthermore, the buttons were laid out automatically by specifying the desired widths in proportion to the width of their parent container. In this case, at 25%, four buttons are allowed to appear adjacent to each other. Providing relative measures for the dimensions of the elements allows for responsive reactions of the layout for resizing events. This behavior can be observed in Figure 5.3.

In addition, character events were mapped to allow for the inputting of numbers and operators directly from the keyboard. Finally, the calculator features a scrollable number display. This was done by restricting the height of the display, causing the layouting algorithm to horizontally overflow characters in the event that there is no more room to expand. The overflowed content is then accessible via a scrollbar or by scrolling with the mouse wheel.

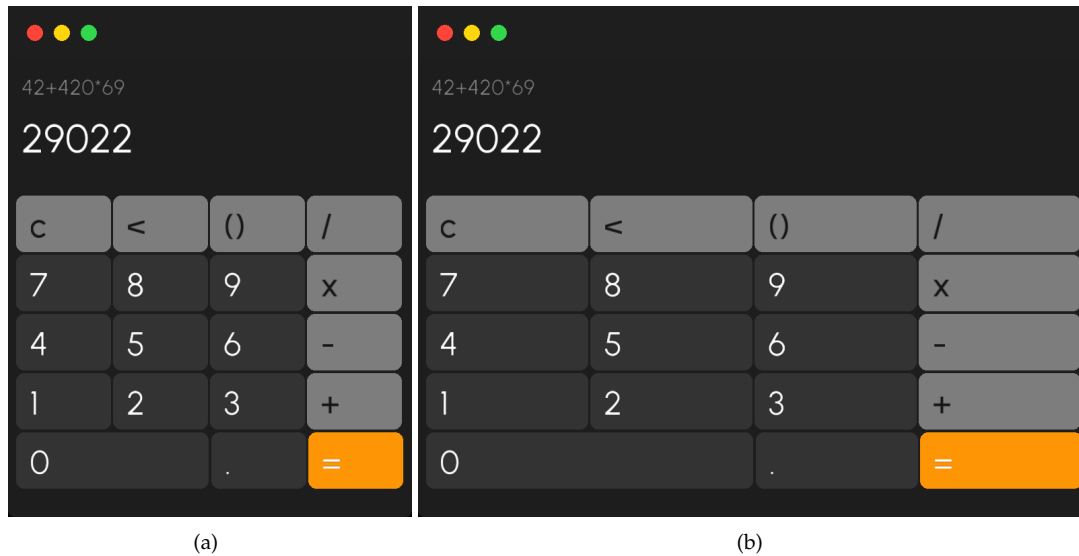


Figure 5.3: Calculator application in initial size (a) and resized (b). (Own source)

5.4 Messaging Application

Preceding, a mock messaging app was created, demonstrating more complex layout possibilities as well as special icon and media rendering. This layout was achieved by dividing the content into containers. The left sidebar was placed in a container with a 23% width whilst the main content body was assigned a 77% width, resulting in a side-by-side arrangement. The individual elements were then laid out similarly, utilizing margin and padding properties to provide enough whitespace in-between the elements. Furthermore, the demo features hover effects and a scrollable main body.

Although the title bar appears similar in style to the macOS, the image was taken from a demo application running on a windows machine. This effect was achieved through disabling the native frame and reimplementing the title bar appearance and functionality, including the three title bar buttons.

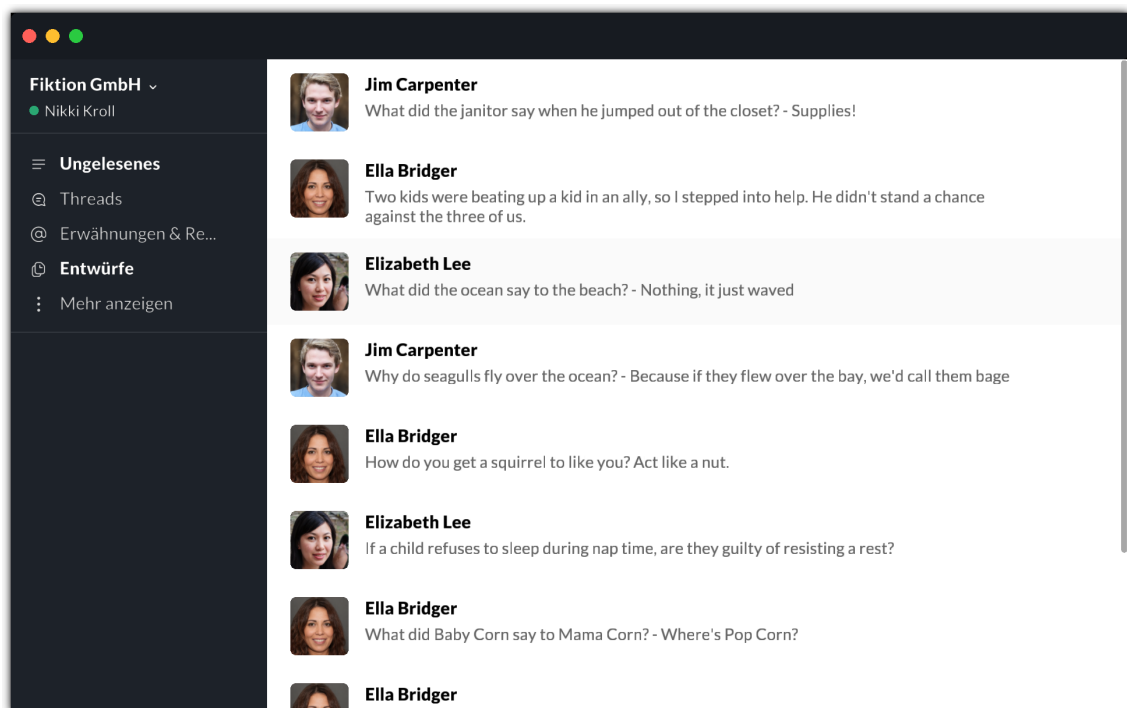


Figure 5.4: Messaging application demonstration with main focus on complex layouting. (Profile images from external source²)(Own source).

It should be noted that the application as such is non-functional and only serves to showcase visual results for demonstration purposes.

5.5 Conway's Game Of Life

Finally, Conway's Game of Life cellular automaton was implemented extensively utilizing the `draw_rect()` command provided by the Elemental Draw renderer. This automaton regulates the life and death of cells in a grid according to a predefined rule set [82]. The number of columns and rows, as well as the cell size, can be varied for different simulation domains. Therefore, the provided implementation can serve as a basis to study performance characteristics of the rendering system. Precisely due to these properties, the Game of Life application was subsequently utilized, though in a slightly modified state, for performing performance evaluations in Chapter 6.

For interaction purposes, a halt and resume functionality was introduced via the spacebar keypress. Furthermore, new cells can be created via a left mouse click in the desired grid cell. This functionality serves as a means to find interesting configurations of cell structures.

²<https://thispersondoesnotexist.com/>

Evaluation

The performance of the renderer was the main focus of the research questions. Thus, a benchmark application was devised for evaluating the exact performance characteristics of the Elemental Draw library (i.e., RQ1) as well as to compare its performance against other state-of-the-art frameworks (i.e., RQ2). The preliminary results from devising this benchmark are provided in the following section.

6.1 RQ1 - Performance Evaluation

The benchmark builds on the notion of a repeatable simulation, which can provide variable loads on the CPU and GPU. Hereby Conway's Game of Life as introduced in Section 5.5 was used in a slightly modified way, removing all interactivity to ensure consistent outcomes. A dense initial configuration was chosen, with approximately 60'000 individual cells (Figure 6.1 (a)) dispersing down to approximately 15'000 cells throughout 8'200 generations (Figure 6.1 (d)), upon which the cells reach a steady state. The cells were colored according to their current state. Red signaling death, green signaling birth, and white corresponding to an unchanged state of life. The initial state being rather dense causes a lot of state changes on the entire grid due to overpopulation. Therefore, more load is generated in the initial few seconds of the simulation, rather than towards the end where the cells reach an equilibrium.

We believe that such a synthetic benchmark is still a good measure of performance, as it simulates a significant load with many changing elements in addition to a slow and steady scene. Furthermore, the test can be repeated due to the deterministic nature of the selected cellular automaton.

The performance benchmark was conducted on several machines with different hardware components and operating systems. The hardware ranged from mid-tier laptops given the mobile graphics chips (GeForce MX250, Intel UHD Graphics 620), to high-end gaming stations (GeForce RTX 2080Ti, GeForce GTX 1080Ti). To provide comparable results, the application was fixed at a native resolution of 1800 pixels width and 1200 pixels height. Resizing was disabled. The benchmark ran automatically and measured the elapsed time during each generation in milliseconds. The measured frame-times were then averaged. The benchmark was run approximately five times on each device. Figure 6.2 presents the quartiles and medians of the tests. The x-axis enumerates the milliseconds per frame while the y-axis lists the machines the benchmark was conducted on. The green, blue, and red dotted vertical lines mark the 144Hz, 120Hz, and 60Hz limits, respectively. These are common monitor refresh rates and can be used as performance targets. Although overshooting the targets can lead to wasted performance and screen tearing [15, p. 764], it is still preferred, as the framerate can be throttled down artificially. However, if the target cannot be reached, it means that the application is underperforming. Stuttering and delayed response

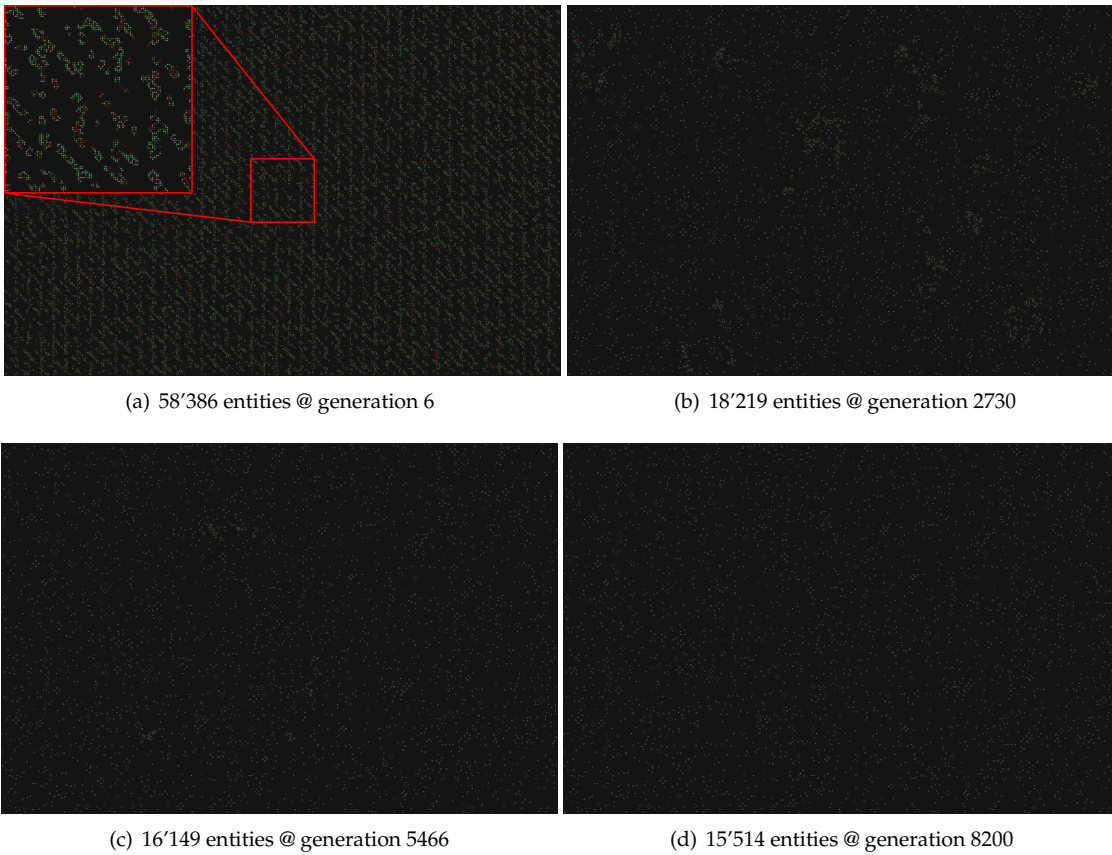


Figure 6.1: Benchmark application running Conway's Game of Life from the initial configuration (a) to a steady state (d). Red rectangle (a) marks zoomed in region. (Own source).

times can be artifacts of an underperforming application leading to a bad user experience.

Figure 6.2 shows that most of the devices performed well, far above the 144Hz mark. Furthermore, the midrange laptops also managed to perform above the 120Hz threshold. The only exception was the MacBook 16", performing the worst at an approximate 13 ms average. This was to be expected due to the OpenGL implementation not being able to utilize SSBOs. In addition, the macOS build was generally less optimized than its two counterparts due to limited accessibility during development. However, considering all the limitations, an average frame-time of under 16 ms can still be viewed as sufficient.

An additional observation was made on the spread of the quartiles. The spread from the lower to the upper quartiles appears to be more pronounced on laptops with poor thermal properties, where subsequent execution of the benchmark would lead to worse results due to heating issues that throttled the overall performance of the device. In contrast to the portable devices, the high-performance workstations managed to deliver highly consistent scores.

6.2 RQ2 - Performance Comparison

To evaluate how the achieved performance compared to other available solutions, the Conway's Game of Life performance benchmark was ported to JavaScript, utilizing the HTML5 Canvas API [83]. The web contents were then run on Google Chrome, Electron, and Firefox. The resolution was again fixed to 1800 pixels in width and 1200 pixels in height to ensure a comparable result. The generated output was confirmed to be visually identical to each other. All benchmarks were executed on the same device and operating system. In both implementations, the frame-times were measured, stored in a pre-allocated array, and printed out at the end to ensure that neither the array resizing nor the console printing operation would distort the measurement.

Figure 6.3 (a) shows the achieved milliseconds throughout 8'200 generations of the simulation. As expected, due to a high density in the initial configuration with up to 60'000 individually painted cells, the performance was worse throughout all variants. It was approximately 110 ms in the case of the HTML5 implementation running on Chrome and Electron compared to a consistent 20 ms at the end of the simulation. The same JavaScript benchmark performed better overall on Firefox with a starting performance of approximately 20 ms, leveling off to 18 ms at the end. Similarly, our implementation also showed a small improvement throughout the simulation. It was at 6 ms at the outset, although, the system quickly recovered from the initial load and showcased a rather steady frame-time of approximately 3 ms during the entire remaining simulation.

Due to the drastic difference in the frame-times between the Elemental Draw and the HTML5 Canvas implementations, the resulting total simulation duration was considerably shorter in the case of Elemental Draw, with 22 sec compared to Google Chrome's 202 sec and Electron's 178 sec as shown in Figure 6.3 (b). Firefox proved to have a slightly more optimized renderer, completing the benchmark in 114 sec.

A similar performance between Google Chrome and Electron was to be expected, as Electron utilizes the Chromium browser [35] for displaying web contents. The slight improvement can be, therefore, attributed to the somewhat more streamlined Chromium browser.

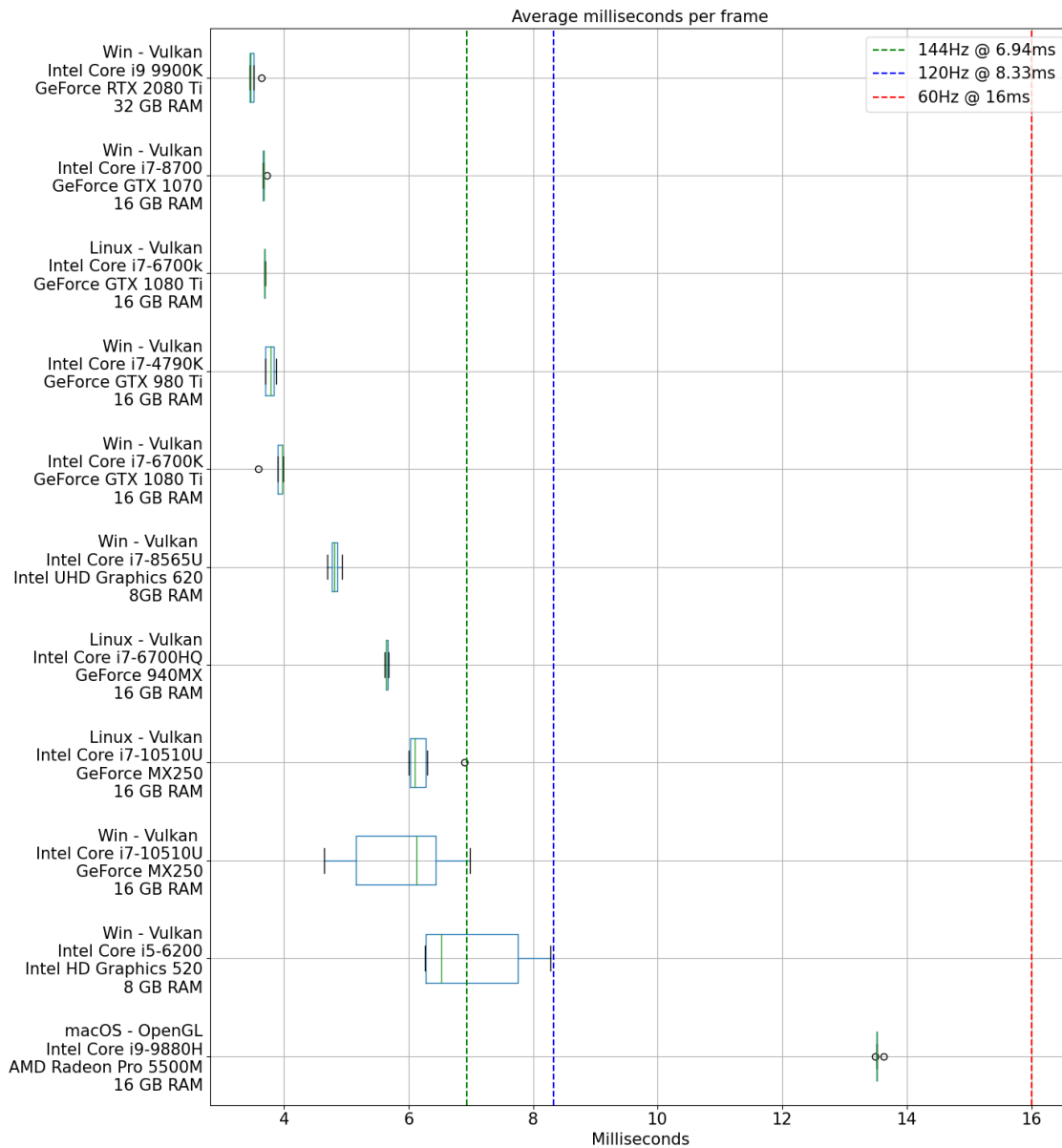
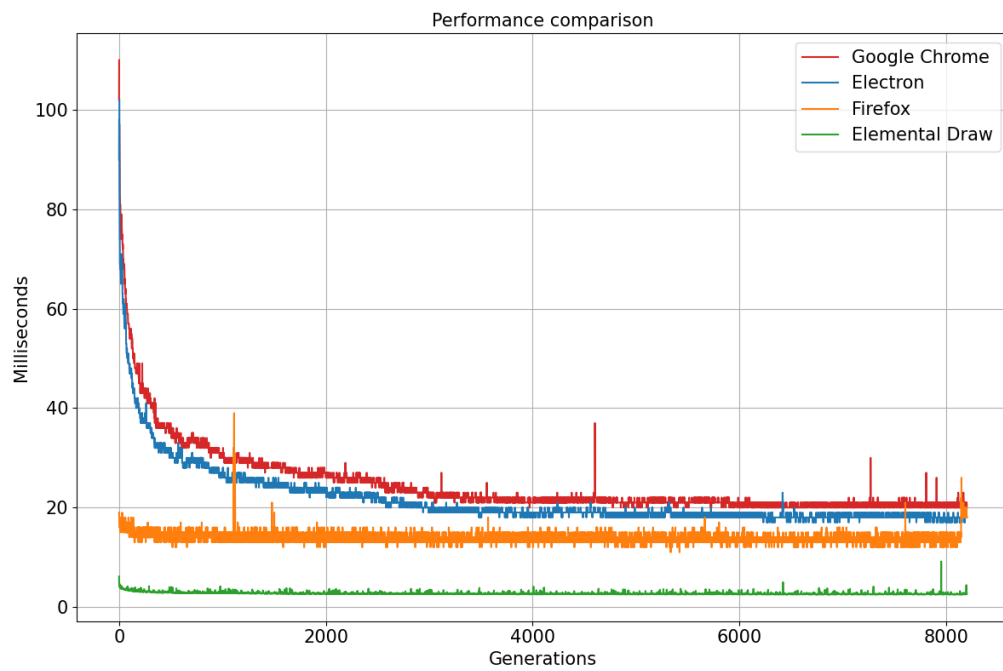
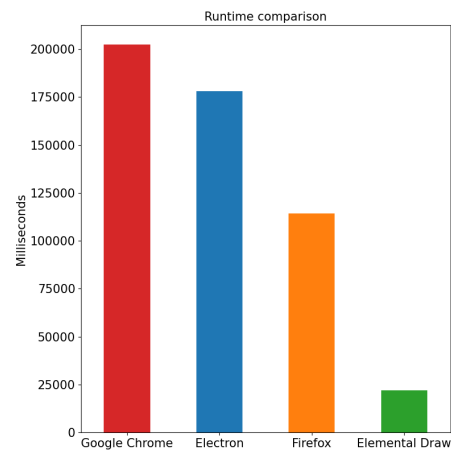


Figure 6.2: Conway's Game of Life performance benchmark on different hardware and platforms. Dotted vertical lines denote target refresh rate limits. Lower numbers are better. (Own source).



(a) Milliseconds per frame



(b) Total runtime

Figure 6.3: Conway's Game of Life performance benchmark comparison between Elemental Draw, Google Chrome (Version 88.0.4324.190 (64-bit)), Firefox (86.0 (64-bit)) and Electron (Electron: v12.0.0, Chromium: v89.0.4389.69, Node: v14.16.0). (a) Milliseconds per frame comparison, (b) overall runtime comparison. (Own source).

It has to be noted, that the comparison considerably favors the Elemental Draw implementation, as rendering a vast number of simple rectangles directly caters to the strengths of the presented system. Furthermore, the measured frame-times do not compensate for the inherent performance differences in the programming languages used. However, an argument can be made regarding the appropriate choice of a programming language in the correct setting; therefore, we believe that the comparison is still valid. After all, the C++ programming language was selected precisely for its performance capabilities.

Future Work

Considering all the technical challenges involved in building a cross-platform system, the evaluation showed that the Elemental UI framework provides a solid foundation for portable applications with a performant, though limited renderer.

A collection of possible issues that could be addressed in future work are discussed in the following section.

7.1 Metal Implementation

While discussing the rendering backends in Chapter 4, we presented the limited support for other backends on macOS in Table 4.1. Although some support remains for OpenGL, the preferred rendering backend for Apple devices is still Metal. In addition, the highest supported OpenGL version is 4.1 [55], which hinders the use of SSBOs that were only introduced in version 4.3 [16]. The OpenGL implementation is forced to invoke multiple draw calls because of having to resort to limited-sized UBOs. This is also partially the reason why we saw a poor performance on the otherwise well-equipped Mac machine running an OpenGL backend, compared to Vulkan supported platforms, as shown in Figure 6.2.

The OpenGL implementation was delivered solely as a stand-in backend in the context of this thesis, as a Metal backend proved to be difficult to implement because it required a unique ecosystem. First, the Metal implementation would have to be done in Objective-C with bindings from the main backend interface. Second, the shaders would have to be rewritten with the Metal Shading Language.

Therefore, it would still make sense to introduce a Metal backend in future work for Apple devices to replace the deprecated OpenGL implementation, as it would additionally open up the doors to target mobile platforms from the Apple ecosystem, such as iOS, iPadOS, and tvOS, as demonstrated in Section 4.2.1.

The OpenGL backend can still prove to be useful, as the current 4.1 version can be downgraded to OpenGL ES 2 with little effort to serve as a web renderer as illustrated in Table 4.1.

7.2 Path Rendering

We established in Section 4.2.2 that, due to performance concerns linked to breaks in the instance chain, it is currently not feasible to render lines, arcs, and arbitrary shapes with Elemental Draw. Although the text rendering system somewhat remedies the issue by providing a means to convert SVG shapes into glyphs that can be rendered as bitmaps, this approach is only a workaround

and does not solve the underlying issue. For example, dynamically generated and animated line graphs cannot be realized with the current set of primitives provided by Elemental Draw.

Although free form path rendering falls into the same realms as font rendering, as font files commonly also contain paths, there is a major difference between them, making font rendering much more practical. As glyphs are predefined within the font files, they can be pre-computed, either during compilation or on startup, to generate whatever data structure is most convenient. Alternatively, they can be directly converted into bitmaps and directly rendered. However, this is not possible with dynamically generated paths during runtime.

We do acknowledge that omitting such an integral set of primitives limits the usefulness of the library. Therefore, two techniques that could improve the rendering system in future work are presented.

Line Segments Paths can be expressed by short line segments chained to each other. By increasing the number of segments and reducing their length, somewhat smooth paths can be drawn. Naturally, it is desirable to reduce the amount of segmentation optimally without degrading the visual fidelity. A possible approach to optimize the number of segments is to carefully place the segmentation points on the path instead of dividing the path into segments of the same length. Favoring more rounded areas of the path when distributing the segmentation can lead to smoother curves [84].

Unfortunately, there are some issues with this method. Although it should be possible to pipe the generated segments through the existing pipeline of rectangle instances, this would drastically increase the instance count, as every single segment would have to be accounted for. Alternatively, breaking the instance chain could be explored to render a fully assembled path. However, this approach still only enables rendering paths. Filled shapes would still have to be accomplished through different means.

Mixed Rendering New opportunities could arise from combining existing software renderers with our presented GPU renderer. Components from the already well-established Skia graphics library [27] could be utilized for select pre-rendering applications, such as free form shape rendering, lines, and paths. Skia is optimized for real-time applications, so these primitives could be generated in a timely manner and stored to a bitmap. These bitmaps could then be directly used by the Elemental Draw rendering pipeline by queuing them in the instance buffer. The obvious benefit here is the ability to offload the GPU whilst maintaining the instance chain.

It is noteworthy that this particular issue of path rendering is still being researched and new concepts for GPU acceleration are being introduced, such as a proposed approach by NVIDIA to accelerate rendering, utilizing the stencil buffer [85], and later, a proposition utilizing scan-line rasterization [86].

7.3 Custom OS Interface

As already established in Section 4.1 regarding the system architecture of Elemental Draw, the core library significantly relies on GLFW for communicating with the operating system for common windowing tasks and events. While it is a robust and well-established library, there are some unfortunate embedded limitations in its design. GLFW mainly targets desktop-grade operating systems and, therefore, does not provide support for mobile platforms such as Android or iOS. Consequently, neither Elemental Draw nor its dependent library, Elemental UI, will be able to support mobile platforms in its current state.

One possible solution for this issue would be the introduction of the PImpl idiom, similar to the way it was done with the rendering backends. A new common API that encompasses all the

necessary interaction points between the OS and the application could be introduced. Thereafter, an alternative library for mobile platforms, such as GLFM [87], has to be embedded. Finally, the correct backend for interacting with the OS can be chosen, according to the configuration in the project generation step. Unfortunately, it is unclear whether both libraries will be able to provide substitutes for every single common API call.

Therefore, an alternative approach would be the implementation of a custom solution for interacting with the operating system, which is also platform agnostic including desktop and mobile systems. Although such an approach would provide a greater degree of flexibility and consistency for finding a common interface, it would also require a significant amount of time and effort to even attain feature parity with GLFW-supported platforms.

7.4 Binding Generation

Another area of improvement is in the language bindings. Currently, only partial bindings are provided for the Python language. This process is slow and error-prone. Automation can help to improve the quality and consistency, by automatically regenerating the bindings according to the latest API changes. An integration of such a system could be realized in the already present CI system.

In addition, the exported C interface is also only partially implemented and requires further changes. The exported C API utilizes pointers that are not supported by every language, as is the case with Java. A rewrite is required to convert all pointers into handlers to make the interface more accessible. Alternatively, instead of providing C wrappers on top of the C++ source code, the Elemental Draw library could be rewritten entirely in C. The value of such a drastic measure still has to be evaluated.

Conclusion

This thesis presents an alternative approach to user interface rendering, by focusing primarily on a robust hardware acceleration rather than on a complete feature set. Based on our formulated requirements, we presented a new cross-platform rendering library Elemental Draw, alongside its companion library Elemental UI.

The case studies presented in Chapter 5 have demonstrated the cross-compatibility capabilities of both the Elemental Draw and Elemental UI libraries. When compiled from source, the system allows for virtually all versions of Windows, macOS, and Linux distributions to be targeted, as it does not rely on specific OS rendering features, which can change over time. Hereby a majority of desktop-grade operating systems are covered. Though, it would be desirable to expand the portability to mobile platforms as well, the task at hand is difficult and requires modifications to the layer communicating with the operating system as well as new language bindings.

Taking a look at the preliminary results provided by the performance benchmarks in Section 6.1, it can be concluded that a major goal concerning high-performance rendering through GPU acceleration was realized. Though, there is still room for improvement considering the weaker performance delivered by the OpenGL implementation running on a MacBook. Finally, we demonstrated in Section 6.2 how much faster the rendering operations can be executed given a specific domain of primitive shapes if sequenced correctly. Although, it has to be reiterated that the comparison is focused on a very specific and narrow set of problems, which highly favors our implementation.

It can be concluded that the Elemental UI framework provides a solid foundation for rendering a set number of shapes on the GPU on the three most relevant platforms. Though, not complete in its feature set, it already provides essential UI capabilities in a performant manner. However, a lot more work needs to be done to consider our system over other, more robust, and established solutions.

Bibliography

- [1] M. Schofield, M. Wojciakowski, L. Blevins, K. Guo, L. Graham, S. Krsmanovic, G. Milener, D. Coulter, K. Bridge, J. Kennedy, A. Braden, and M. Lacey, "Choose your windows app platform - windows applications | microsoft docs," Accessed 11. March 2021 from <https://docs.microsoft.com/en-us/windows/apps/desktop/choose-your-platform#uwp>, 2021-02-03.
- [2] M. Malewicz, "Glassmorphism in user interfaces," Accessed 13. March 2021 from <https://uxdesign.cc/glassmorphism-in-user-interfaces-1f39bb1308c9>, 2020-11-22.
- [3] 14islands, "blobmixer," Accessed 13. March 2021 from <https://blobmixer.14islands.com>, 2021.
- [4] L. P. 5, "marseille," Accessed 13. March 2021 from <https://marseille.laphase5.com>, 2021.
- [5] C. Zapponi, "Github language stats," Accessed 03. March 2021 from https://madnight.github.io/githut/#/pull_requests/2020/2, 2021.
- [6] K. Stefanoski, A. Karadimce, and I. Dimitrievski, "Performance comparison of c++ and javascript (node.js -v8 engine)," 09 2019.
- [7] C. Robertson, N. Schonning, M. Jones, G. Hogenson, and S. Cai, "Msvc c/c++ compiler reference - visual studio | microsoft docs," Accessed 10. February 2021 from <https://docs.microsoft.com/en-us/cpp/build/reference/compiling-a-c-cpp-program?view=msvc-160>, 2018-10-12.
- [8] GCC Team, "Gcc, the gnu compiler collection- gnu project - free software foundation (fsf)," Accessed 11. February 2021 from <https://gcc.gnu.org>, 2021-01-18.
- [9] The LLVM Team, "Clang c language family frontend for llvm," Accessed 11. February 2021 from <https://clang.llvm.org>, 2021.
- [10] cppreference.com, "C++ compiler support - cppreference.com," Accessed 9. February 2021 from https://en.cppreference.com/w/cpp/compiler_support, 2021-02-04.
- [11] S. Logan, *Cross-Platform Development in C++: Building Mac OS X, Linux, and Windows Applications*, 1st ed. Addison-Wesley Professional, 2007.
- [12] G. Sellers and J. Kessenich, *Vulkan Programming Guide The Official Guide to Learning Vulkan (OpenGL)*. Addison-Wesley, 2016.
- [13] The Khronos Group Inc., "Vulkan overview - the khronos group inc," Accessed 9. February 2021 from <https://www.khronos.org/vulkan>, 2021.

- [14] —, “Opengl overview - the khronos group inc,” Accessed 11. February 2021 from <https://www.khronos.org/opengl>, 2021.
- [15] R. S. Wright, B. Lipchak, and N. Haemel, *OpenGL (R) SuperBible: Comprehensive Tutorial and Reference*, 4th ed. Addison-Wesley Professional, 2007.
- [16] The Khronos Group Inc., “Shader storage buffer object - opengl wiki,” Accessed 12. February 2021 from https://www.khronos.org/opengl/wiki/Shader_Storage_Buffer_Object, 2020-06-08.
- [17] liballeg.org, “Allegro - a game programming library,” Accessed 12. March 2021 from <https://liballeg.org>, 2021.
- [18] G. Foot, “Github - liballeg/allegro5 initial revision,” Accessed 12. March 2021 from <https://github.com/liballeg/allegro5/tree/d94899aa0381f1e1b2f2ba6a63ed6e9a1bd98626>, 2000.
- [19] cairographics.org, “cairographics.org,” Accessed 12. March 2021 from <https://www.cairographics.org>, 2014.
- [20] C. Worth and K. Packard, “Xr: Cross-device rendering for vector graphics,” Accessed 12. March 2021 from https://cworth.org/cworth/papers/xr_ols2003/html, 2003-05-17.
- [21] B. Harrington, “Latest cairo news,” Accessed 12. March 2021 from <https://www.cairographics.org/news>, 2020.
- [22] cairographics.org, “Using cairo with opengl,” Accessed 12. March 2021 from <https://www.cairographics.org/OpenGL>, 2016.
- [23] OpenJFX, “Javafx,” Accessed 12. March 2021 from <https://openjfx.io>, 2021.
- [24] J. Marinacci, “Javafx 1.0 is live : Javafx blog,” Accessed 12. March 2021 from https://web.archive.org/web/20081207095309/http://blogs.sun.com/javafx/entry/javafx_1_0_is_live, 2008-12-04.
- [25] Gluon HQ, “Javafx - gluon,” Accessed 12. March 2021 from <https://gluonhq.com/products/javafx>, 2021.
- [26] Oracle Corporation, “Javafx 2.2.3 system requirements | javafx 2 tutorials and documentation,” Accessed 12. March 2021 from https://docs.oracle.com/javafx/2/system_requirements_2-2-3/jfxpub-system_requirements_2-2-3.htm, 2012.
- [27] Skia Inc., “Skia graphics library,” Accessed 04. March 2021 from <https://skia.org>, 2021.
- [28] Google LLC, “Commits · google/skia · github,” Accessed 12. March 2021 from <https://github.com/google/skia/commits/1550a42d9647162edc4e6758fc2958fa4ab7f6ca>, 2008.
- [29] B. Osman, “refs/heads/master - skia - git at google,” Accessed 12. March 2021 from <https://skia.googlesource.com/skia/+refs/heads/master>, 2021.
- [30] Skia Inc., “Vulkan,” Accessed 12. March 2021 from <https://skia.org/user/special/vulkan>, 2021.
- [31] J. Belfiore, “New year, new browser – the new microsoft edge is out of preview and now available for download,” Accessed 13. March 2021 from <https://blogs.microsoft.com/latino/2020/01/15/new-year-new-browser-the-new-microsoft-edge-is-out-of-preview-and-now-available-for-download>, 2020-01-15.

- [32] AvaloniaUI OÜ, "Github - avaloniaui/avalonia: A cross platform xaml framework for .net," Accessed 12. March 2021 from <https://github.com/AvaloniaUI/Avalonia>, 2021.
- [33] —, "Avalonia docs," Accessed 12. March 2021 from <https://avaloniaui.net/docs>, 2021.
- [34] cegui.org, "Faq - cegui wiki - crazy eddie's gui system (open source)," Accessed 12. March 2021 from http://cegui.org.uk/wiki/FAQ#What_is_CEGUI.3F, 2014.
- [35] OpenJS Foundation and Electron contributors, "Electron | build cross-platform desktop apps with javascript, html, and css," Accessed 10. March 2021 from <https://www.electronjs.org>, 2021.
- [36] B. Spitzak, "Fast light toolkit - fast light toolkit (fltk)," Accessed 12. March 2021 from <https://www.fltk.org/>, 2021.
- [37] C. Sells, "What's new in flutter 2. by chris sells | mar, 2021 | medium | flutter | flutter," Accessed 12. March 2021 from <https://medium.com/flutter/whats-new-in-flutter-2-0-fe8e95ecc65>, 2021-03-03.
- [38] The GTK Team, "The gtk project - a free and open-source cross-platform widget toolkit," Accessed 12. March 2021 from <https://www.gtk.org>, 2021.
- [39] —, "The gtk project - a free and open-source cross-platform widget toolkit," Accessed 12. March 2021 from <https://www.gtk.org/docs/architecture/>, 2021.
- [40] O. Cornut, "Github - oconut/imgui: Dear imgui: Bloat-free graphical user interface for c++ with minimal dependencies," Accessed 12. March 2021 from <https://github.com/oconut/imgui>, 2021.
- [41] nanapro.org, "Nana c++ library - a modern c++ gui library," Accessed 12. March 2021 from <http://nanapro.org/en-us>, 2020.
- [42] E. Davies, J. Storer, and T. Poole, "Github - juce-framework/juce: Juce is an open-source cross-platform c++ application framework for desktop and mobile applications, including vst, vst3, au, auv3, rtas and aax audio plug-ins." Accessed 12. March 2021 from <https://github.com/juce-framework/JUCE>, 2021.
- [43] Nuklear, "Github - immediate-mode-ui/nuklear: A single-header ansi c immediate mode cross-platform gui library," Accessed 12. March 2021 from <https://github.com/Immediate-Mode-UI/Nuklear>, 2020.
- [44] The Qt Company, "Qt | cross-platform software development for embedded & desktop," Accessed 12. March 2021 from <https://www.qt.io>, 2020.
- [45] wxWidgets, "wxwidgets: Cross-platform gui library," Accessed 12. March 2021 from <https://www.wxwidgets.org>, 2020.
- [46] D. Herberth, "Github - dav1dde/glad: Multi-language vulkan/gl/gles/egl/glx/wgl loader-generator based on the official specs." Accessed 23. February 2021 from <https://github.com/Dav1dde/glad>, 2021.
- [47] glfw.org, "An opengl library | glfw," Accessed 23. February 2021 from <https://www.glfw.org>, 2021.
- [48] S. Barrett, "Github - nothings/stb: stb single-file public domain libraries for c/c++," Accessed 24. February 2021 from <https://github.com/nothings/stb>, 2020-07-13.

- [49] W. Lemberg, "The freetype project," Accessed 23. February 2021 from <https://www.freetype.org>, 2021.
- [50] WHATWG (Apple and Google and Mozilla and Microsoft), "Html standard," Accessed 17. February 2021 from <https://html.spec.whatwg.org>, 2021-02-11.
- [51] Apple Inc., "Macbook pro 16-inch - technical specifications - apple," Accessed 24. February 2021 from <https://www.apple.com/macbook-pro-16/specs>, 2021.
- [52] S. Willems, "Amd radeon pro 5300m - vulkan hardware database by sascha willems," Accessed 24. February 2021 from <http://vulkan.gpuinfo.org/displayreport.php?id=10099>, 2020-12-14.
- [53] The Khronos Group Inc., "Moltenvk is a vulkan portability implementation. it layers a subset of the high-performance, industry-standard vulkan graphics and compute api over apple's metal graphics framework, enabling vulkan applications to run on ios and macos." Accessed 11. March 2021 from <https://github.com/KhronosGroup/MoltenVK>, 2021.
- [54] —, "Conformant products - the khronos group inc," Accessed 01. March 2021 from <https://www.khronos.org/conformance/adopters/conformant-products/opengl>, 2021.
- [55] Apple Inc., "Mac computers that use opengl and opengl graphics - apple support," Accessed 01. March 2021 from <https://support.apple.com/en-us/HT202823>, 2020-07-28.
- [56] The Khronos Group Inc., "Conformant products - the khronos group inc," Accessed 01. March 2021 from <https://www.khronos.org/conformance/adopters/conformant-products/vulkan>, 2021.
- [57] J. Kennedy, K. Sharkey, D. Coulter, S. White, J. Kramer, D. Batchelor, and M. Satran, "Getting started with directx graphics - win32 apps | microsoft docs," Accessed 18. February 2021 from <https://docs.microsoft.com/de-de/windows/win32/getting-started-with-directx-graphics>, 2018-05-31.
- [58] Apple Inc., "Metal | apple developer documentation," Accessed 24. February 2021 from <https://developer.apple.com/documentation/metal>, 2021.
- [59] The Khronos Group Inc., "Conformant products - the khronos group inc," Accessed 01. March 2021 from <https://www.khronos.org/conformance/adopters/conformant-products/opengles>, 2021.
- [60] Apple Inc., "Opengl es | apple developer documentation," Accessed 24. February 2021 from <https://developer.apple.com/documentation/opengles>, 2021.
- [61] The Khronos Group Inc., "Webgl and opengl differences - webgl public wiki," Accessed 18. February 2021 from https://www.khronos.org/webgl/wiki/WebGL_and_OpenGL_Differences, 2014-04-13.
- [62] Free Software Foundation, Inc., "Make - gnu project - free software foundation," Accessed 25. February 2021 from <https://www.gnu.org/software/make>, 2020-01-19.
- [63] Premake, "Premake," Accessed 25. February 2021 from <https://premake.github.io>, 2021.
- [64] Kitware, Inc., "Cmake," Accessed 25. February 2021 from <https://cmake.org>, 2021.
- [65] S. Meyers, *Effective Modern C++*. O'Reilly, 2015.

- [66] The Khronos Group Inc., “Khronos-reference front end for glsl/essl, partial front end for hlsl, and a spir-v generator,” Accessed 24. February 2021 from <https://github.com/KhronosGroup/glslang>, 2021.
- [67] git-scm.com, “Git - submodules,” Accessed 25. February 2021 from <https://git-scm.com/book/en/v2/Git-Tools-Submodules>, 2021.
- [68] npm, Inc., “About npm | npm docs,” Accessed 25. February 2021 from <https://docs.npmjs.com/about-npm>, 2021.
- [69] F. Copes, M. Borins, F. Hemberger, O. Laru, J. G. Bousiquot, A. Miller, and A. Awais, “The package.json guide,” Accessed 25. February 2021 from <https://nodejs.dev/learn/the-package-json-guide>, 2021.
- [70] Microsoft Corporation, “Github - microsoft/vcpkg: C++ library manager for windows, linux, and macos,” Accessed 25. February 2021 from <https://github.com/microsoft/vcpkg>, 2021.
- [71] —, “vcpkg/ports at master · microsoft/vcpkg · github,” Accessed 25. February 2021 from <https://github.com/microsoft/vcpkg/tree/master/ports>, 2021.
- [72] T. Raj, “Vcpkg: 2019.06 update,” Accessed 25. February 2021 from <https://devblogs.microsoft.com/cppblog/vcpkg-2019-06-update>, 2019-07-19.
- [73] Microsoft Corporation, “vcpkg/triplets at master · microsoft/vcpkg · github,” Accessed 27. February 2021 from <https://github.com/microsoft/vcpkg/tree/master/triplets>, 2021-02-18.
- [74] P. M. Duvall, A. Glover, and S. Matyas, *Continuous integration improving software quality and reducing risk*, 8th ed., ser. The Addison-Wesley signature series; A Martin Fowler signature book. Addison-Wesley, 2013.
- [75] GitHub, Inc., “Introduction to github actions - github docs,” Accessed 28. February 2021 from <https://docs.github.com/en/actions/learn-github-actions/introduction-to-github-actions>, 2021.
- [76] Microsoft Corporation, “Dllimportattribute class (system.runtime.interopservices) | microsoft docs,” Accessed 01. March 2021 from <https://docs.microsoft.com/en-us/dotnet/api/system.runtime.interopservices.dllimportattribute?redirectedfrom=MSDN&view=net-5.0>, 2021.
- [77] Oracle Corporation, “System (java se 9 & jdk 9),” Accessed 01. March 2021 from <https://docs.oracle.com/javase/9/docs/api/java/lang/System.html#loadLibrary-java.lang.String->, 2021.
- [78] Python Software Foundation, “ctypes — a foreign function library for python — python 3.9.2 documentation,” Accessed 01. March 2021 from <https://docs.python.org/3/library/ctypes.html>, 2021.
- [79] I. Quilez, “Inigo quilez :: fractals, computer graphics, mathematics, shaders, demoscene and more,” Accessed 13. March 2021 from <https://www.iquilezles.org/www/articles/distfunctions2d/distfunctions2d.htm>, 2021.
- [80] V. Chlumsky, “Shape decomposition for multi-channel distance fields,” Master’s thesis, Czech Technical University, 2015.

- [81] —, “msdfgen - multi-channel signed distance field generator,” Accessed 13. March 2021 from <https://github.com/Chlumsky/msdfgen>, 2021.
- [82] M. Gardner, “Mathematical games the fantastic combinations of john conway’s new solitaire game “life”,” *Scientific American*, vol. 223, pp. 120–123, 10 1970.
- [83] Mozilla Foundation, “Canvasrenderingcontext2d - web api referenz | mdn,” Accessed 10. March 2021 from <https://developer.mozilla.org/de/docs/Web/API/CanvasRenderingContext2D>, 2021.
- [84] B. Ciechanowski, “Drawing bézier curves – bartosz ciechanowski,” Accessed 11. March 2021 from <https://ciechanow.ski/drawing-bezier-curves>, 2014-02-18.
- [85] M. J. Kilgard and J. Bolz, “Gpu-accelerated path rendering,” *ACM Transactions on Graphics (TOG)*, vol. 31, pp. 1 – 10, 2012.
- [86] R. Li, Q. Hou, and K. Zhou, “Efficient gpu path rendering using scanline rasterization,” *ACM Trans. Graph.*, vol. 35, no. 6, Nov. 2016. [Online]. Available: <https://doi.org/10.1145/2980179.2982434>
- [87] D. Brackeen, “Github - brackeen/glrm: Opengl es and input for ios, tvos, android, and webgl,” Accessed 02. March 2021 from <https://github.com/brackeen/glrm>, 2021.