Seminar

Seminar: UI-Rendering Systems

Dominik Völkel

Graz, WS 2022

Contents

List of Figures

List of Tables

1 Abstract

User interface development has become a crucial part of game development. The user interface is the main way for the player to interact with the game and therefore has to be designed with care. This paper will give an overview of the different approaches to user interface rendering and implementations in game engines which are currenlty on the market. Special focus will be put on the rendering aspects of the different user interfaces and their interoperability. This paper suggests a standardised user interface rendering system for use in game engines based on established web technologies.

A prototype will be presented realizing a modular and engine-agnostic user interface rendering and input system using standardised web technologies and their systems/tools.

Sections prior to the prototype serve to show its requirements and establish a baseline for comparison against existing UI systems.

2 Literature

This section establishes a common terminology and gives an overview of the most important concepts and techniques in the field of game-engines, game-frameworks and game-libraries. Additionally established game-engines and their approach to User-interface rendering/implementation are listed and briefly discussed.

2.1 Definition of game-engine, game-framework and game-library

An important distinction to make is the difference between a game and a game-engine. Game engines, for the purpose of this paper, are mainly data-driven [12, Chapter 1.3] pieces of software that enable authors to create several games without designing the entire software architecture again from the ground up. A typical game engine consists of several components handling different aspects of its functionality, typically a game engine has components/modules concerning themselves with [7]:

- Scripting component: provides simple engine related control frequently used for debugging.
- Rendering component, responsible for rendering the game world and its entities.
- Animation component: concerned with movement and deformation of objects in fixed animations.
- Artificial intelligence (AI) component: responsible for 'smart' behaviour of the game world, mainly its entities (e.g. enemies, allies etc.).
- Physics component: responsible for the simulation of physical interactions between objects in the game world.
- Audio component: responsible for the generation and playback of audio.
- Networking component: responsible for the communication between different instances of the game or some kind of server or online entity.

Depending on the genre or the type of game, some of these components might not be needed. For example, a game that does not feature any kind of multiplayer or online component would not need a networking component.
For faster development game engines may also provide pieces of software to aid the authors which together with the game engine form a game development framework.

These frameworks are usually more opinionated than game engines and provide a more rigid structure for the game development process, but in exchange allow for much faster editing and creation of games. The choice of the framework is highly important as designers are usually comitted to using the engine/framework and cannot easily switch between two different frameworks. Additonal pieces of software are usually related to distinct components inside the engine and change their behaviour:

- Level editors: allow the author to create levels for the game without having to edit raw files describing the world to the engine. These editors usually provide a graphical user interface to create the levels and allow for a more intuitive and faster creation of levels. This consequently influences components for: Rendering, Physics, Animation and AI.
- Script editors: for quick editing of object behaviour. These editors mainly influence the AI, physics, audio, scripting and/or networking components.
- Material editor: for editing the look and feel of an object which mainly influences their rendering.
- Sound editor: for editing the sound of an object which mainly influences the audio component and its behaviour.

All of these components and their respective editors are usually provided by the game engine [7]. Large-scale commercial game-engines typically provide all of these tools like the unity engine[21] or unreal engine [9]. Game-libraries are implementations of subsets of game-engine components. The allegro 5 'game programming library'[18] provides cross-platform components for rendering and audio, its' set of features allows Allegro to produce a full game but in most practical cases the library requires further libraries/components. These libraries are usually more lightweight than game engines (since they only implement their sub-components) and are usually used in combination with other libraries to create a game.
Establish a common definition for these terms has been a common issue in past works. Cowan [7] quotes Sherrod [16] defining a game-engine as "a framework comprised of a collection of different tools, utilities, and interfaces that hide the low-level details of the various tasks that make up a video game" which Cowan concludes to the definition: " framework includes a game engine in addition to external tools and resources that simplify the process of game development". This paper will use the following definition based on the above quotes:

- **Game libraries**: focus on a subset of components of a typical game engine and can be used in combination with other libraries to create a game but cannot do so by themselves. Example: Allegro 5 [18], ASSIMP [5].
- **Game engine**: a software framework that provides a set of components required to create a game. The engine may be accompanied by development tools making

it into a game framework. Examples: G3D innovation engine [14], Quake Engine, Doom Engine, Source Engine, Frostbite Engine

- **Game frameworks**: includes a game engine in addition to high-level editors for faster development. Example: Unity [21], Unreal Engine[9], Godot engine[17]

This definition stands in direct conflict with the community-established definition as seen on the popular site gamefromscratch. The site defines game-libraries similarly but the definition game-engine and framework are swapped.

2.2 Analysis of existing game-engines

The following sections will discuss game-engines which are currently in use in further detail and their approach to UI-Rendering.

2.2.1 Unity

In their documentation unity currently supports three different UI-renderng systems:

1. The Unity UI package (uGUI), is the current standard system for UI-creation in the games themselves.
2. IMGUI: mainly used to create UI elements in the unity editor itself (for plugins and similar purposes).
3. UI-toolkit: using an UI-Asset system 'inspired by standard web formats such as HTML, XML, and CSS.' [21]

Unitys'documentation provides a direct comparison of the three different systems. The exact technical implementation is not documented. The documentation states: 'Unity intends for UI Toolkit to become the recommended UI system for new UI development projects, but it is still missing some features found in Unity UI (uGUI) and IMGUI.', which clearly indicates that unity wants to move away from the uGUI system for in-game UI-creation and towards the UI-toolkit. With UI-toolkit the system will be *inspired* by web-standards but further details remain to be seen.

2.2.2 Godot

Godot is an open source, community-driven 2D and 3D engine [17] founded by Juan Linietsky and Ariel Manzur. Its interface and project structure and development process have similarities to unity. The engine features one UI-system which is asset-based similar to unitys'uGUI system mentioned in 2.2.1. The Documentation does not mention its technical implementation explicitly but the engine itself is open-source . The engine is written in C++ and uses the OpenGL API for rendering. The engine is currently in version 3.2.1 and is still in active development.

2.2.3 Unreal Engine 5

Unreal Engine 5 is a commercial game engine developed by Epic Games. It is currently in version 4.25 and is used in many AAA games. The engine features a UI-system called Slate which is asset-based similar to unitys uGUI system mentioned in 2.2.1. The Documentation does not mention its technical implementation explicitly and the engine itself is closed-source. The engine is written in C++ and uses the OpenGL API for rendering.

2.3 Analysis of existing game-libraries for UI-Rendering

This paragraph will discuss game-libraries and their approach to UI-Rendering. Special focus is put on libraries that are disconnected from game engines and can work independently. From the previous section 2.1 it is clear that in game-engines ui-systems are mostly 'baked into' the system itself. Unity is the only outlier in this regard since their upcoming system 'UI-toolkit' focuses on building upon an existing standard, namely HTML5/CSS. With their inherently more modular design game-libraries are more flexible in their approach to UI-Rendering. The following section will discuss the most popular game-libraries and their approach to UI-Rendering. This chapter will highlight if an UI-systems uses existing standards like the HTML5/CSS webstandards, referred to as *webstandard-descriptive-system* from those that implement their own description of the user interface.

2.3.1 Coherent labs: Gameface

Coherent labs [6] focuses on a *webstandard-descriptive-system* with their product Gameface. This system has, according to their website, been used in several games which are listed here. Among these are well known titles such as 'Outriders' and 'Control'. Though further research can neither confirm nor deny this to be true. Possibly the system is

also only used for prototyping purposes but this could also not be confirmed without directly inquiring the game-creators. Major features, which are listed here, of Gameface include full support for the entire JavaScript language ecosystem inside a WebAssembly core engine. In general any instruction running in javascript/react/typescript will also run correctly in this system according to their documentation. It is closed source and further information of the technology and its library/source code requires an inquiry at the firm. In their technical documentation, which can be found here, is available openly but the samples are not fully listed and therefore further asessment of inner functionality and engine integration can not be made. It is not apparent if the system is still actively in development because of its closed-source nature.

2.3.2 sciter

Sciter [19] is published via gitlab, which can be found here in pre-compiled binary snapshots. Getting full access to all platforms requires a license. Therefore the project is closed source. The main purpose of this software is to be embedded in existing c++ engines/systems as a sub-module similar to Gameface Referencessec:Coherent labs: Gameface and is also an *webstandard-descriptive-system*. According to the projects github page it is still actively being developed and improved. The largest software product using skiter is 'War Thunder' a videogame.

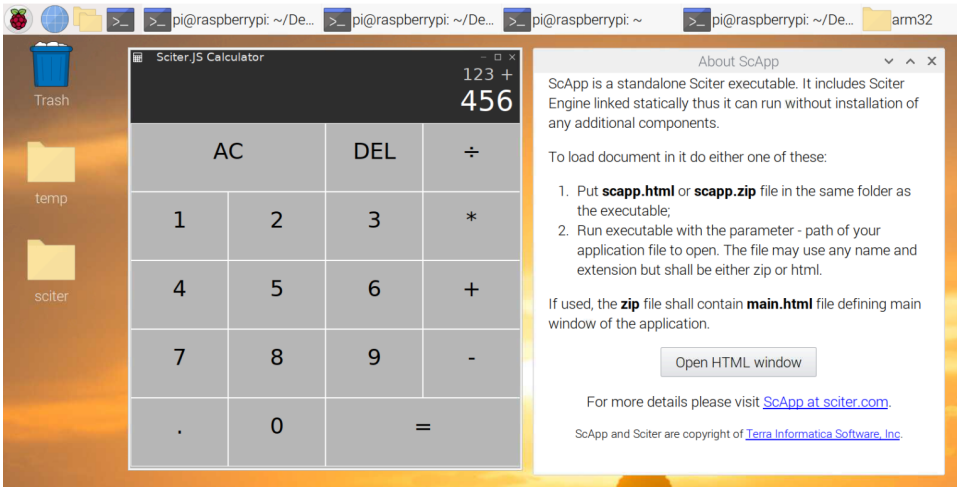

Figure 1: Sciter based UI source

2.3.3 ultralight

Ultralight [20] uses the javascript core engine used by webkit according to their documentation which can be found here and mainly advertises itself as being 'very low

weight' and having a small memory footprint. These claims are not backed up by any benchmarks or other sources since the project is not entirely open-source and there have not been studies done confirming or denying its performance. The repository, found hereincludes samples and licensing information; the most recent update to their repository and their most recent statement is over 2 years old which makes it unclear if it is still in active development. According to the projects page it mainly supports C/C++ engines but also has bindings from the community for other languages. Ultralight focuses on single-process support compared to the other platforms multi-process rendering as is common in modern browsers where each tab/website represents a process. Ultralight uses the same JavaScript engine as WebKit/Safari and therefore also supports react/Vue.js/Angular.

2.3.4 QT

QT is a cross-platform framework for developing applications and user interfaces. It is written in C++ and is open-source. It is used in many applications and games. The QT framework is not a game-library but a framework for developing applications, therefore not a direct competitor to the game-libraries discussed in this section. However, it is a good example of a framework that is used in games and has a modular UI-system. The QT framework has a UI-system called Qt Widgets which is a C++ based system, which does not attempt to be based on any existing webstandard. Therefore the description of the UI is bound to the QT platform.

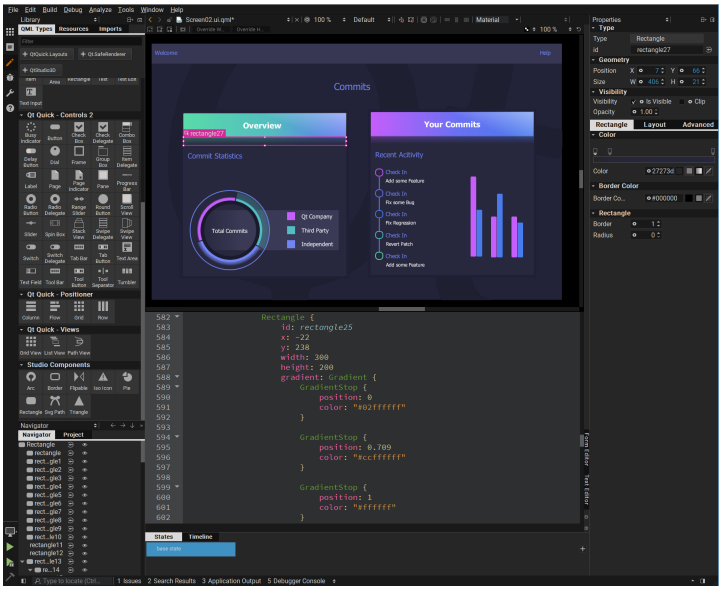

Figure 2: QT designer and an example UI inside of it source

3 Approach

In this section I will elaborate on the existing approaches and discuss my own approach and the potential advantages and disadvantages of it.

3.1 Existing approaches

Existing approaches can be roughly divided into two categories:

- *webstandard-descriptive-system*: The UI is described in a webstandard inspired way. This means that the UI is described in HTML/CSS/JS. This is the approach taken by Gameface 2.3.1, Sciter 2.3.2 and Unity UI-toolkit 2.2.1.
- *non-webstandard-descriptive-system*: The UI is described in a non-webstandard compliant way. This means that the UI is described in a proprietary way or in some specific language. This is the approach taken by QT 2.3.4, Unity immediate-GUI System 2.2.1, Unreal Engines' UI-System and Godots' UI-System 2.2.2. All these systems are due to their specific description methods not interoperable and development has to be done in the specific language/environment.

In my research there was no established approach to the problem of a modular UI-system using fully web-standard compliant formats for UI-description. Most established game-engines use their own proprietary UI-systems which are not interoperable in any way. Committing to a single engine also means commiting to its UI-System as the engine and accompanying frameworks are developed in tandem with the UI-system. This brings up a few development issues for non web-based UI-systems:

- Developing in an engine and needing to switch to another engine because of a specific feature or limitation will mostly entail a complete rewrite of the UI-system thus far. This creates significant friction and may make the engine switch unfeasable or at least very costly in temrs of time.
- The UI is 'locked' to the engine and cannot be reused in other engines or even projects on the same engine. As an example Unity 2.2.1 immediate system uses in-engine objects to bind to which makes the UI very hard to reuse without also reusing large parts of the internals of the game. This makes reusability possible inside an engine but makes it very rigid.
- The capabilities of the UI-System are bound by the capabilities of the UI-System of the engine itself.
- Development for these UI-Systems is very specialized and mostly require specific training for the engine. This makes it hard to find developers for these UI-Systems and also makes it hard to find developers that can work on multiple UI-Systems.

- For these UI-Systems there is an ongoing challenge to change the UI on a runtime-level. Since the engine and UI are tighly interconnected outside modification and customization (especially during runtime) is very difficult and often times requires a lot of custom code and large 'menu' screens that edit each element/color individually. This is a very tedious and time consuming process that is very unfriendly for developers and users and is additionally very hard to maintain and extend.

The above disadvantages are mainly due to the custom nature of the UI-systems. From the perspective of the engine devopers this 'locked in' behaviour of the UI might be a *desired* feature as it requires developers to stay with the engine.

Web-based systems address some of these issues like:

- Reusability within the engine as html/JS events are more easily bound.
- Finding developers for UI-Systems as web development is a much more common skill than UI-Design in a specific engine.
- Reusability even between engines as long as the UI system is web-standard compliant.

Currently no 100% web-based UI-system that have been tested in an open-source manner exists. UI-Webkit from Unity 2.2.1 does attempt to use this technology, but is by design not fully web compliant as it will use their custom XML/HTML hybrid system, locking developers in the ecosystem just as much as previous approaches.

3.2 A fully modular approach

All the above approaches to UI-Design also do not consider *parallel development*, as in separate development teams for the game logic and the game-UI/Interface. The web-based approaches allow this to some extend as there is a common interface between UI and the program itself. I will suggest a fully modular event-based system that decouples the development processes of the UI and the Game logic itself. The below diagram tries to illustrate the 2 distinct parts and how they communicate with each other.

From the viewpoint of the Game Engine or the UI system the other compontent should be treated as a "black box". From a design perspective the two system should know as little as possible from one another in order to make them as modular as possible. They communicate via primitive-datatype (numbers/names) defined events in order to keep communication as simple as possible and reduce necessary custom implementation for different platforms. It can be assumed that modern systems have a common type definition in string/numbers without needing any lower level communication like endian-ness or similar issues that might occure on low-level systems/communication. The internal communication can be acheived with a socket communication structure. decoupling the

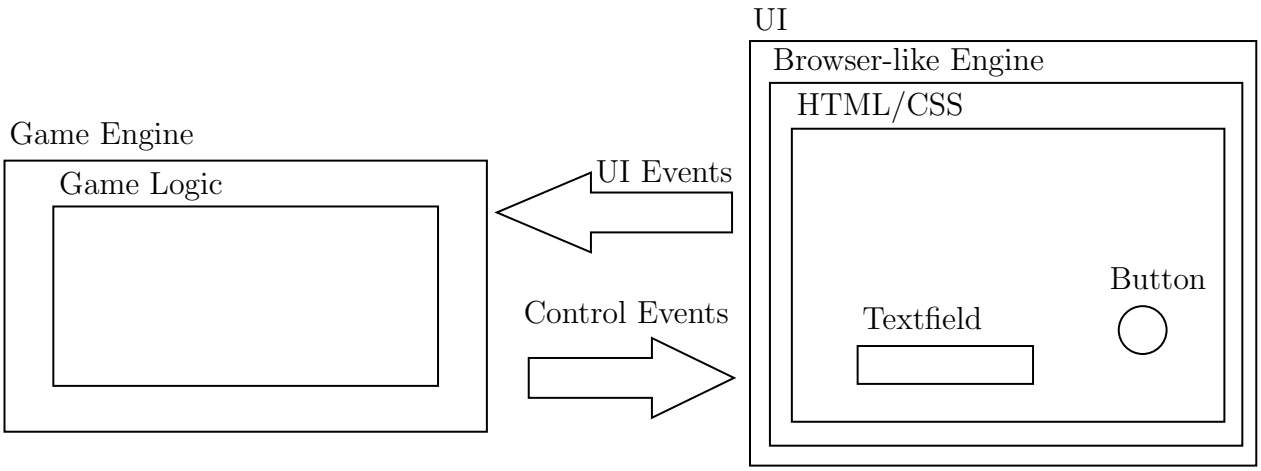

Figure 3: Basic Design

entire system even further and also enabling applications that might want to run UI-Rendering and Game logic on two completely separate machines, leaning into the area of Server Driven UI.

3.3 Component: Browser-like Engine

There are a lot of different Browser engines which are currently in use and activiely developed, though for the purposes of this paper there exist a few important requirements which must be met:

- The engine should be open-source and free to use. The main idea of this system is to make testing results easy to verify; using a closed source engine would hinder development and debug-ability immensly.
- Offscreen rendering: In this context ”Offscreen rendering” means that the main UI that is rendered by the engine needs to be rendered to a texture or a buffer that can be used by the game engine. This is critical as to accomodate as many different rendering pipelines and approaches as possible.
- Cross platform: Most browser engines are cross-platform and can be compiled for Windows, Linux and Mac. As to not restrict platform usage the engine itself should support as many as possible out-of-the-box.

The following paragraphs will examin an engine in more detail and argue if an engine would be fit for this modular purpose.

3.3.1 Webkit Engine

Webkit can be broadly categoriezed into 2 versions: Webkit V1 and Webkit V2. Webkit V2 is the sucessor to Webkit V1 and is the current version of the engine. This document mainly concerns itself with Webkit V2.

Webkit V1 / Webkit Legacy API Webkit V1 is based on the KHTML[2] and KJS[3] engine which was developed by the KDE [1] project, it is said that the project started in 1998 and the Webkit project itself was started by Don Melton in 2001 (no primary source found).

Webkit V1 is considered legacy software and is severely outdated. The Webkit 1 version was a single-process architecture which was abandoned in favor of a multi process architecture described in section 3.3.1 (source bugtrack). The main reason ,stated on webkits wiki, was the incompatible API switch from V1 to V2.

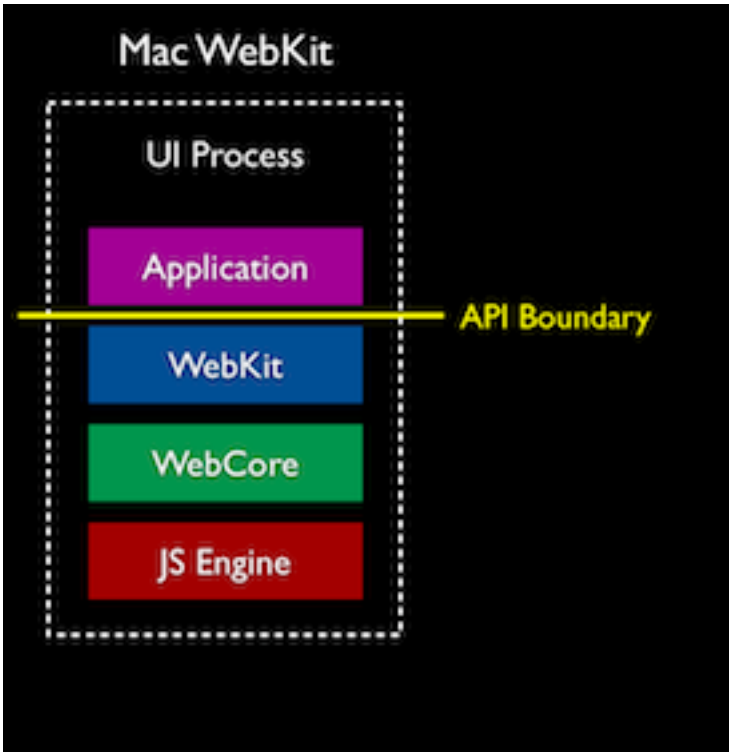

Figure 4: Webkit 1 stack source

Essentially figure 4 shows that layouting, JS processing and rendering all happen in the same process and there is no need for interprocess communication. This makes it suitable

for Game-Engine applications as it is a single process and can be easily integrated into the a game engine process. Tools like 2.3.3 are based on Webkit V1. Though its outdated nature imposes obvious limitations on the engine and its capabilities as well as diverse security risks as it is not maintained anymore.

Webkit V2 On the surface webkit V2 [4] is the most promising html rendering engine for a modular UI system. The Webkit Engine is hosted on github and has (at the time of writing) almost 6k stars, 865 forks and over 900 contributors. This chapter will mainly focus on the structure of webkit, its sub components and its history. It is well maintained, has a large company, Apple [4], behind it and is used in many cross-platform scenarios. One of its main design philosophies is "hackability" which means that it is easy to extend and modify the engine to fit any needs. This is a very important aspect for this project as it will be used as a base for many different projects and will need to be easily extendable and hackable. The list of components creating webkit (cited from their github page) consists largely of:

- bmalloc: A custom malloc implementation with security features.
- WTF: "Web Template Framework" which lays the foundation in C++ datastructures for the rest of the engine.
- JSC: "JavaScriptCore": The Javascript engine.
- WebCore: Rendering engine (which is the most interesting component for our purposes)

The main difference to Webkit V1 is that it is a multi-process architecture.

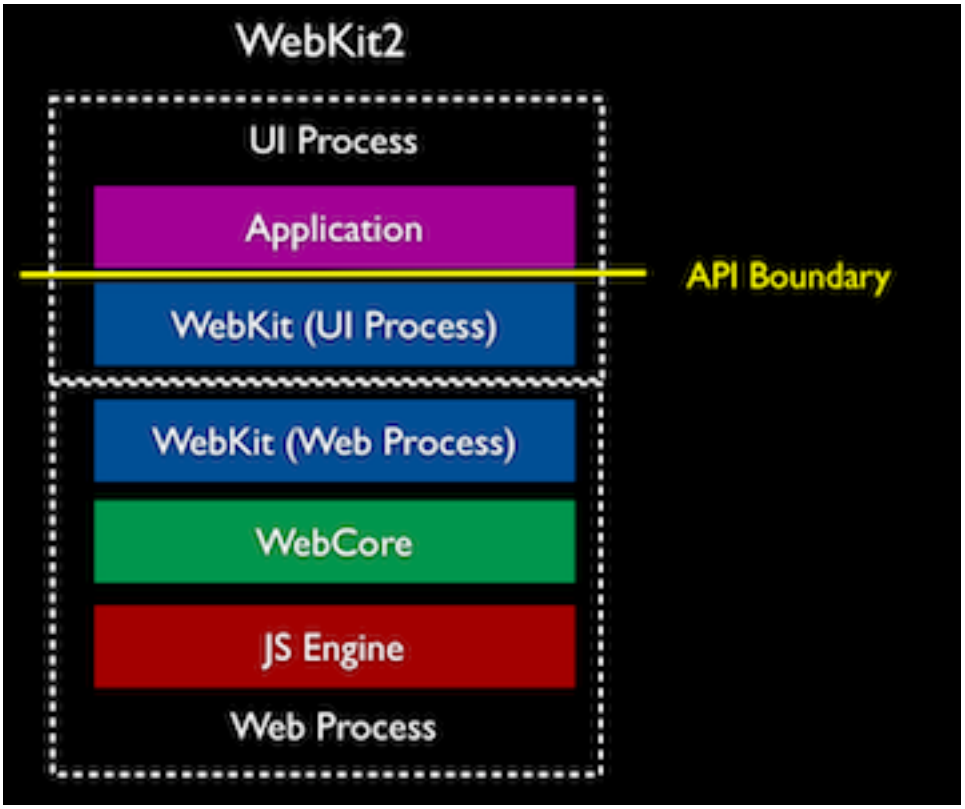

Figure 5: Webkit 2 stack source

Figure 5 shows that the actual host Application is outside of the previous stack. With Webkit V2 interprocess communication (IPC) is necessary which is the main reason the ports exist mainly for Linux based operating systems. The abstraction Webkit tries to provide with this is called "Core IPC" which uses mach messages on OSX and named pipes on windows. On Linux the implementation is port dependant.

The main advantage of Webkit is its large array of features and its open-source status. It is field tested and used as the rendering engine for many popular browsers and therefore has considerable support and a large community. Most notibly Safari uses Webkit and with apples high standard for security and feature implementation makes it a good choice. Additionally is Webkit cross-platform compile-able making it additionally suitable for this kind of modular application. Webkit also shares most of the goals of this project listed here including: Performance, Portability, Usability, Open Source and Open Source ;and is therefore a good fit for this application. Most notibly is the project goal of "Web Content Engine" stating: "The project's primary focus is content deployed on the World Wide Web, using standards-based technologies such as HTML, CSS, JavaScript and DOM. However, we also want to make it possible to embed WebKit

in other applications, and to use it as a general-purpose display and interaction engine.", which is exactly what this project tries to use it for.

Webkit Ports There exist 2 large "platform ports" of webkit next to the Safari technology preview (seen on webkits website).

- WPE: "Webkit Port Embedded" is the official Webkit port for embedded platforms. Its main design goal was to be independent of specific user-interface toolkits, as explained on their website. it requires a libwpe backend for rendering on each platform.
- "Webkit GTK": is a port created by gnome.org using GTK UI library as a rendering backend.

Technically the only requirement is WebCore itself as a rendering engine. The component is very deeply embedded into the rest of the software with no documented way to extract it or using it as a standalone component. Therefore a project using Webkit would require producing a port that implements all the necessary backend components for the *e*ntire webkit project.

Both options of ports are linux only. A project that runs on windows as well would require a new port to be developed. Creating a new port is a very large undertaking and would require a lot of time and personel. Each platform target would require some IPC for the specific platform as well as some rendering backend to handle the actual rendering.

Additionally the documentation of creating a port is very sparse and does not provide a lot of information without having someoen on the team with experience in porting webkit.

For confirmation on the scale of such a project "Mario Sanchez Prada <mario@igalia.com>" who works at igalia[13] the company behind the WPE port. On 13.04.2023 he replied with:

> Working with a Web engine like WebKit is a very complex task on its own, even if it's just to create an application that uses it as a library, without modifying the engine itself... but creating a port of WebKit is of course way more complex than just "consuming" it as a library. Unless you are Andreas Kling and have multiple years of experience and knowledge on both Operating Systems and Web engines, it's usually a multi-person and multi-year effort to build a working port even if you only target one OS, so it's indeed a very complex task..."

Knowing this makes using Webkit directly unfeasable.

3.4 Chromium / Electron

This paragraph more closely investigates chromium-like systems. Electron [10] also falls into this category as it internally embedds chromium, this is documented as "What is Electron?". Chromium internally also depends on webkit, as can be seen in their documentation. The engine implements its own port for webkit in order to use it cross-platform.

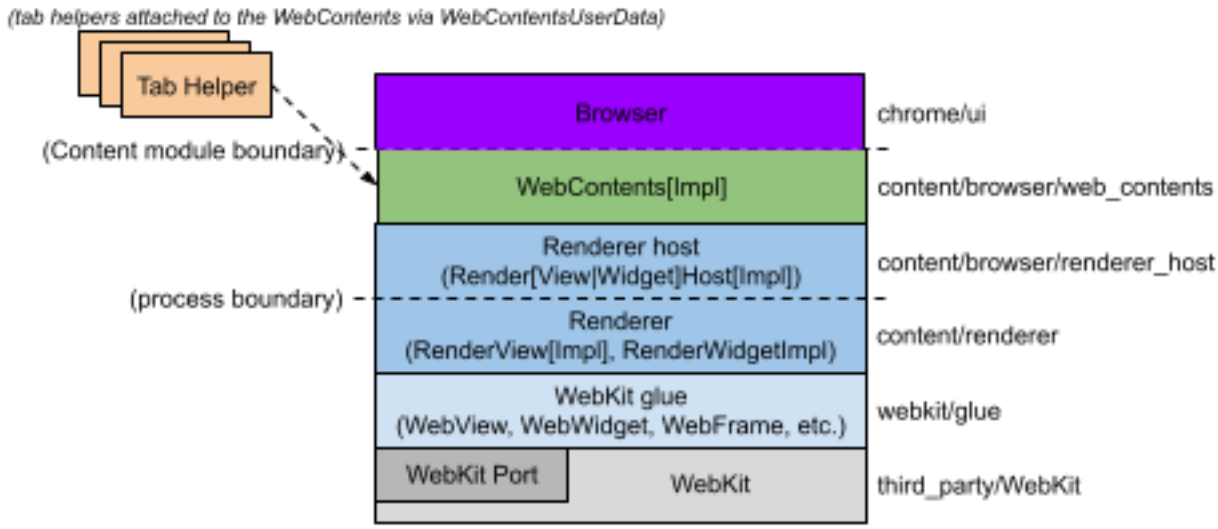

Figure 6: Chromium stack source

The conceptual image 6 shows the different components of chromium. The layer "Webkit Glue" translates the different types webkit uses to the types that google more commonly uses. Again the entire engine would implement too many features for the purposes of this project, as this project is interested in the rendering component only.

Electrons own documentation states that it works by embedding chromium and nodejs into its binary. This allows for large compatability with many platforms, but essentially means the application includes an entire browser as described in their documentation. Using electron a developer writes one nodejs codebase to run on many platforms, but this broad compatability comes with the cost of a lot of abstraction and losing hardware-features. For example multithreading is only possible via a browsers web-worker implementation. This would severely restrict performance of any game engine.

3.5 Chromium Embeddded Framework (CEF)

As mentioned in the previous section including the entire Chromium browser would be too much overhead, but the projects bring more minimal implementations to light. The Chromium Embedded Framework [11] aims for exactly this purpose. The CEF project can be embedded into other programs by starting up its own process for rendering and input processing. Most notibly the project is used by spotify to power their different clients across different operating systems [8].

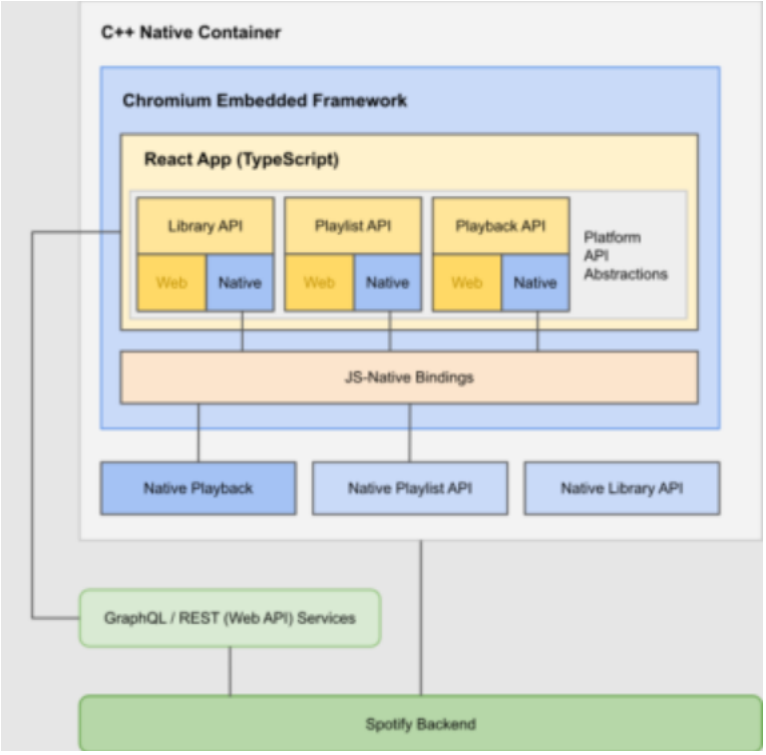

Figure 7: new architecture of spotifys desktop client source. (Sadly the original resolution is very low.)

As can be seen in figure 7 the desktop client internally uses the CEF project. CEF provides JS-Native-bindings, which in practice means that the containing C++ application (here named "C++ Native Container") can interactive with the internal datastractures used in by the web app (or any other node application) through a layer with a similar purpose as the "webkit glue" mentioned previously in section 3.4.

This architecture allows spotify to quickly and reliably control native hardware, while graphically rendering the UI with a web-based technology. This is a very good example of how the CEF project can be used to achieve the goals of this project. Native hardware

interaction must be done manually in most cases as drivers vary widely between different operating systems.

In this system CEF is a child thread of the native program, eliminating the need for IPC (inter-process communication) which would also be implemented separately for the overarching operating system.

By describing the actual user-interface in a web-based technology the UI can be easily changed and updated without having to recompile the entire application. This further allows the usage of web-development infrastructure and testing mechanisms like unit tests and integration tests (e.g. jest [15]). Theoretically this would allow a complete decoupling of the development processes between a game running in an engine and the UI shown on top of it.

4 Prototype

The *prototype* (also called "WUI" or "Web UI") described in this paragraph facilitates a proof-of-concept and attempts to be a minimal working example for a user interface running over a 'game' running an arbitrary engine. As described in the above section 3.5 the CEF project is a good candidate for such a prototype.

For game logic the most minimal example would be a game library that can handle game logic but does not include any asset management out-of-the-box, with this in mind unity [2.2.1] or Unreal Engine [2.2.3] would be too "high level" for this purpose. The Allegro [18] game library is our arbitrary choice for such a purpose. The library provides window management and input management for all major operating systems out-of-the-box, while also providing nothing beyond this.

4.1 Goals

Generally this prototype aims to facilitate the common interface between the two components. A goal would be to create a minimally working example where an action on the UI can trigger some action inside the game/application logic itself via some generalized interface.

4.2 Intercommunication

Communication between the components would inevitably be realized via the "JS-Native-Binding" described in 7. As such the communication would use a predefined structure

that both sides understand trivially; a generic json structure or descriptive string/event id management would be suitable.

4.3 Modularity

The main process running the game (which in turn runs the UI CEF process), should include all necessary CEF features via a dynamic library. This would allow for maximum modularity and detachment between the UI and the game engine. To guarantee modularity the headers and definitions for some CEF events (e.g. mouse events and keyboard events) will need to be 'shadowed'; else the end user would need to link against an internal CEF header severely increasing difficulty of usage.

5 Prototype Implementation

A prototype would consist of two parts, the main library implementing "WUI" and an example project using said library. The prototype was developed on a linux system linking against X11 as a desktop manager. Most of the platform-specific logic is abstracted away by the Allegro library.

5.1 LibWUI

When considering the main goals of this library a "top down" approach was chosen. The library should be as simple as possible to use and compile with, while also not restricting the end-user too much. The approach will have to make reasonable assumptions of usage and provide interfaces to circumvent these assumptions.

5.2 Assumptions

Rendering any graphic (like the UI) will inevitably produce a bitmap representing the current 'look' of the UI. A callback-based approach cannot be easily considered due to frame syncing and FPS. As the library wants to interfer as little as possible with the rendering process it should just provide the bitmap in an "always valid" manner giving the user the freedom to decide when and what to render. For this the library pushes a pointer to the currently valid pixelbuffer by itself without developer interaction. Most of this benefits memory safety and the libray can always ensure validity before sending the buffer to the user. A disadvantage is the internal necessity of a double-buffer principle in order to have a 'hot' draw target and a currently shown 'stable' image. This would

duplicate memory usage for the display buffer and further triple it when the end user includes it in their own scheme. But even on large screens this is negligible and the memory usage is still very low.

System input events will be shadowed and forwarded from the CEF internal system to produce a common interface. Here an unexpected problem occures: It is not trivial to conclude when an event should be forwarded to the UI an when it should not. It is necessary to consider that html and the javascript engine does not expose any mechanism to check if any listener for an event was triggered or not. Even if it did there are cases where the user might want to click a part of the UI that does nothing but also does not trigger any event in the underlying engine.

To keep this complex problem as simple as possible any mouse-click or mouse-wheel events that are forwarded to the UI, have their coordinates checked against the currently shown pixel buffer. If the pixel on the events position is all 0 then it can be assumed that the event was not meant for the UI at all. These functions will then return ”false” and the click is ignored and not forwarded to CEF. A ”force” flag should be provided to circumvent this mechanic if desired (Though making an invisible button might not be the best UI design choice).

Knowing this WUI does a very simple approach for all 4 possible CEF events:

- Mouse movement events: Should always forwarded (otherwise hovering effects and similar cursor-modifications would not work).
- Mouse click events: Are forwarded to the UI, if the pixel at the events position is not all 0.
- Mouse wheel events: Work the same as mouse-click events; mainly considering ”scrollable” areas.
- Key events: Are ignored in this prototype (and example). There is no 'clean' way to deduce weather a key event was meant for the UI or not. This is a problem that needs to be solved in the future, either by checking if anything is 'in-focus' or a different 'enable' flag.

5.3 Example

An example mainly shows how to bind and use said library. The example shown in 8 consists of 2 main areas, the gray overlay on the top half of the display showns UI components. Any click on this area is forwarded to the UI. The contained html code is very simple, it shows a heading, some text, two interactive fields (one of which auto-increments each second) and another being incremented by the neighbouring button.

This also shows simple javascript and styling capabilies. The background color is deliberately chosen to be sem-transparent. The bottom half shows a ”blank” area. Right clicking this are will spawn a ball with a random color and diameter. The ball will bounce inside the screen and is rendered by the engine itself.

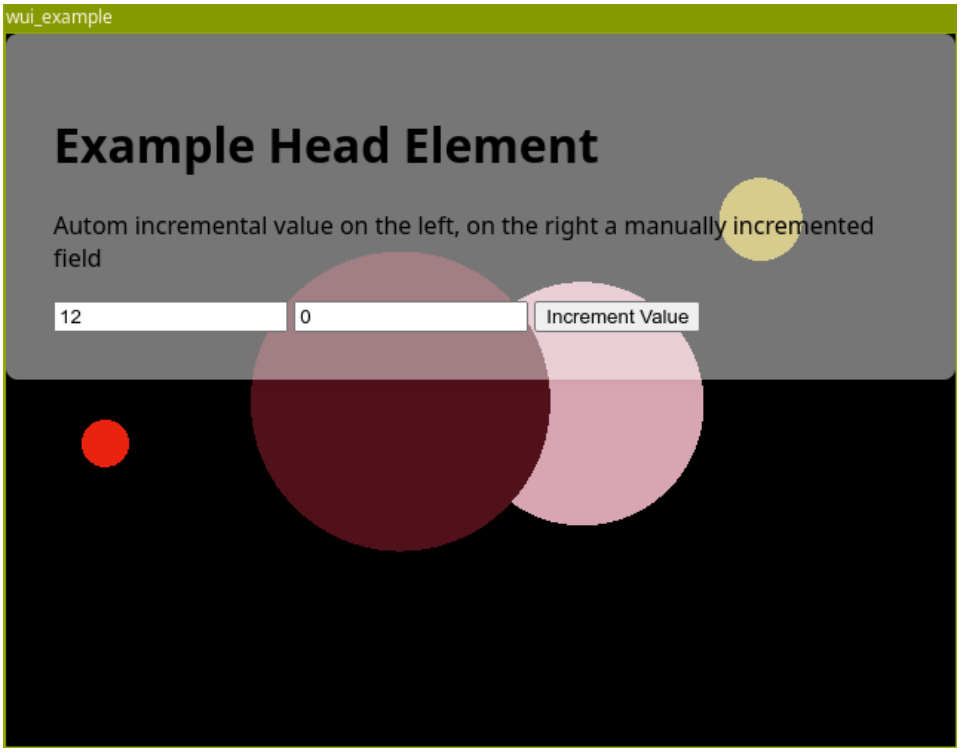

Figure 8: Example of the prototype

The program may be closed cleanly by pressing the ”ESC” key.

Code for this example can be found on here. The libray and compilation instructions can be found here.

6 Results

The prototype shows that the concept of a ”Web-UI” is possible and can be implemented with reasonable effort. Compilation of a release candidate shows that there is quite a significant memory overhead. Including all libraries the footprint of this example is around 1.2GB, which is quite subtantial, but considering the size of the libraries used (CEF, Sandboxing, ...) not unexpected. Additionally our example only shows a very basic pure html-written UI. Any additonal or higher feature using the CEF engine would

not increase the memory footprint further (meaning this is ”as big as it gets” regarding the CEF dependencies).

6.1 Further development

The current prototype has a lot of open questions that need addressing, some of which are listed below:

- How to handle key events?
- Test relative and file routing of assets.
- Add a more complex example for the UI itself.
- Add a build system for a more complex UI using Webpack and modern JS/Typescript systems (including Jest and modern testing systems).
- Bidirectional (possibly event based) Communication between the UI and the main program/engine. This will most likly make a custom npm package necessary.
- Test the entire system on windows/mac
- Profiling performance, memory and CPU usage compared to modern UI solutions or other frameworks (like unity 2.2.1 or unreal engine 2.2.3).
- Usability study of the UI system compared to other solutions.
- Security concerns using a web-based UI system (especially considering the possibility of using JS to access the internet).
- Usability study for system using separate team management between UI and engine development teams.

Nevertheless the prototype shows that a generalistic approach is possible and has potential to be implemented into a full product/library in the future.

References

[1] KDE e.V. - KDE Community/Non-Profit organization. KDE e.V., 2023.

[2] KDE HTML rendering engine. KDE, 2023.

[3] KJS - KDE JavaScript engine. KDE, 2023.

[4] Webkit engine. Apple Inc., 2023.

[5] Kim Kulling Alexander Gessler, David Nadlinger et al. Open asset import library, 2022.

[6] Coherent Labs. Coherent labs, November 2022.

[7] Brent Cowan and Bill Kapralos. A Survey of Frameworks and Game Engines for Serious Game Development. In *2014 IEEE 14th International Conference on Advanced Learning Technologies*, pages 662–664, July 2014.

[8] Spotify Engineering. Building the Future of Our Desktop Apps, April 2021.

[9] Epic Games. Unreal engine, April 2019.

[10] OpenJS Foundation. OpenJS foundation. https://www.electronjs.org, 2021.

[11] Marshall Greenblatt. Chromium embedded framework. https://bitbucket.org/chromiumembedded/cef/src/master/README.md, 2008.

[12] Jason Gregory. *Game Engine Architecture*. : A K Peters/CRC Press, an imprint of Taylor and Francis,, Boca Raton, FL, 3rd edition. edition, 2018.

[13] S.L. Igalia. Igalia. https://www.igalia.com/technology/browsers, 2009.

[14] Morgan McGuire, Michael Mara, and Zander Majercik. The G3D innovation engine, January 2017.

[15] Inc Meta Platforms. Jest. https://jestjs.io, 2023.

[16] A. Sherrod. *Ultimate 3D Game Engine Design & Architecture*. Charles River Media Game Development Series. Charles River Media, 2007.

[17] Software Freedom Conservancy. Godot engine, November 2022.

[18] Allegro 5 Development Team. Allegro 5 - A game programming library, January 2008.

[19] Terra Informatiaca Software, Inc. Sciter engine, November 2022.

[20] Ultralight, Inc. Ultralight, 2021.

[21] Unity Technologies. Unity engine, November 2022.