WebAssembly and WebGPU

A New Approach to Cross-Platform Application Development

Dawid Błaszczyk
International Journal of
Computer Science Research

Filip Kolasa
International Journal of
Computer Science Research

Abstract

In modern times, there are many possibilities for creating multiplatform applications that implement the famous Java language assumptions *"Write once, run anywhere"*. However, they require using various external applications or performing many compilations for selected operating systems. The solution to this problem are web applications, which, unlike native applications, run in web browsers. Despite the security they provide, these environments have so far been limited. They required using a specific set of tools and did not allow for creating highly efficient applications. The reason for this state of affairs was the way web browsers and their applications were perceived, which evolved over time and effectively eliminated their flaws. The latest technologies used in these environments enable the creation of applications that do not differ in performance from native applications and are compiled based on their source codes. Examples of these technologies are WebAssembly and WebGPU, which are discussed and presented in this article.

Keywords: WebAssembly, WebGPU, cross-platform development, web applications, native applications

1 Introduction

One of the most popular technologies used in Internet browsers was Adobe Flash Player, the first version released in 1996. It was installed as a separate application that worked as a plug-in and allowed the creation of interactive animations, games, and multimedia playback.

Although this technology was a breakthrough then, it was not perfect. From the very beginning of its existence, it had many problems, which deepened with the appearance of mobile devices and their dynamic growth in use on the Internet to stationary devices. Web applications began to adapt more and more to the changing market, which Adobe Flash Player did not do. Apart from temporary support for Android devices, it did not support mobile devices, which meant losing many customers. In addition, the appearance of other technologies, such as HTML5, increasingly marginalized its role.

Ultimately, Adobe Flash Player, a closed-source creation of a private company whose business model assumed free playback of content and paid creation, was officially abandoned in 2017. Its full support was ended in 2020, and the ability to play content in it was blocked in 2021. The reason for such an end of the groundbreaking technology from the 90s consists of many factors, but one of the main ones is the security problem. The characteristic of Adobe Flash Player was to run content intended for it in its environment, which, in the form of a plug-in in web browsers, was the target of attacks.

Many other technologies, such as Java Applet or Microsoft Silverlight, used in web browsers over the years have suffered a similar fate. Despite many different reasons for abandonment specific to a given technology, one of the main ones was always the security issue resulting from similar characteristics of operation, as in the case of Adobe Flash Player. The danger for web applications was posed by independent runtime environments installed in web browsers in the form of plug-ins. They introduced another layer of vulnerability to attacks that the web browser did not control.

These technologies were eventually replaced by new open-source alternatives developed by organizations such as the W3C. The critical difference between the earlier solutions and those currently in use is that all client-side web application code is executed exclusively directly in the web browser environment, which is considered safe and controlled.

This article aims to present relatively new technologies such as WebAssembly and WebGPU, extensions of currently used standards that enable the creation of highly efficient cross-platform web and native applications.

2 WebAssembly

2.1 Properties

WebAssembly is a binary instruction format designed to enable the creation of cross-platform executables for high-level programming languages in web browser environments. The following figure (see fig. 1) shows the WebAssembly logo.

Figure 1: WebAssembly logo [1].

WebAssembly aims to enable the compilation of native application programming languages, such as C++ and Rust, into a web application format. Such code is executed in a web browser environment, independent of the operating system. It is also characterized by speed similar to native applications because it uses hardware resources provided by the web browser. This also means the code is safe because it runs within the web browser environment.

Another feature of WebAssembly is its compatibility with other technologies used in web browsers. An example of this is running WebAssembly modules in JavaScript to perform a given task faster. In addition, the modules' code, despite being written in binary format, can be read in text form, making it easier to analyze during debugging.

2.2 Modules

2.2.1 Usage

As a result of the WebAssembly compiler compiling the source code, a WebAssembly module is obtained, which provides all the functionalities of the source code. The module is a binary file with the extension *".wasm"* (WebAssembly Binary Format). However, after selecting the appropriate compiler configuration, it is possible to generate an additional file with the extension *".wat"* (WebAssembly Text Format), which is its text equivalent. In order to analyze the module, generating *".wat"* files is unnecessary because tools supporting module analysis provide a text preview of the code based on the *".wasm"* file itself. Initially, modules were intended for use in web environments, but with the development of technology, their use was extended to native environments.

2.2.2 Web environment

In order to use WebAssembly in web browsers, a particular piece of code that loads the WebAssembly module in JavaScript called *"Glue Code"* needs to be written. This is the only way to use WebAssembly modules in web applications. However, there are plans to allow them to be loaded directly using the *"script"* HTML tag, similar to JavaScript files, or *"import"* statements in JavaScript files, similar to JavaScript modules.

The *"Glue Code"* is the connecting code that enables interactions between WebAssembly modules, JavaScript modules, and the rest of the web application. It is based on several concepts, which are described below.

Compilation and Instantiation: Compiling and instantiating is the process required to load and use a WebAssembly module in JavaScript. During compilation, the module's binary code is converted into a Module object, but its instantiation as an Instance object enables its use. Multiple instances can be created from a compiled module.

Memory: Memory is a class whose objects are responsible for managing the memory of WebAssembly modules that use its low-level concept. These objects represent an untyped block of bits that allows for common memory operations such as pointer arithmetic. Unlike traditional programming languages, WebAssembly allows for memory separation, meaning each module can have its Memory object, thus increasing the overall security and isolation level.

Table: Table class is a table of references, which, unlike Memory class, does not allow direct modification of the contents of its memory and is typed. Currently, the Table supports only one type of reference, function references, but it is planned to be expanded to include other types of references. Access to Table is possible from both JavaScript and WebAssembly, and its interface allows operations to be performed on it, such as introducing new function references.

Globals: Global variables are variables that can be accessed from both JavaScript and WebAssembly. They allow information to be exchanged between multiple WebAssembly modules.

2.2.3 Native environment

Using WebAssembly in native environments allows the creation of cross-platform native applications. They require particular runtime environments for their operation, which, compared to similar solutions, do not negatively affect the application's performance. In addition, they allow complete use of the *"WASI"* interface (WebAssembly System Interface), which allows the module to directly access system resources as in the case of traditional programming languages. In order to use the module in a given native environment, the instructions of the given environment must be followed.

2.3 Emscripten

Emscripten is a LLVM to WebAssembly compiler. For C/C++, it is a complete tool that allows direct code compilation. Additionally, for projects implemented using CMake, it allows easy integration by adding it as a parameter *"toolchain"*.

3 WebGPU

3.1 Interface

WebGPU is an abstract API that allows operations on the GPU to be performed using native APIs. The following Figure (see fig. 2) shows the WebGPU logo.

Figure 2: WebGPU logo [2].

WebGPU was initially designed for web applications, but as the project evolved, it was realized that it could also be used in native applications. It is the only widely developed and accepted attempt to create a standard interface for all modern graphics APIs.

In the case of web applications, the WebGPU interface is available as a JavaScript API, which uses the implementation of the standard provided by a given browser. Another more advanced way to use WebGPU is to use the header file *"webgpu.h"* directly, describing the interface itself and its selected implementation. This method allows for the creation of native applications. However, additionally, using the Emscripten tool based on the same source code in general, it is possible to perform compilation to a web application.

The general principle of operation of WebGPU is based on abstraction. The interface provided as part of the standard does not allow independent use of the GPU but does it using native APIs available in a given device. WebGPU, through *"adapter"*, obtains information about available hardware implementations of WebGPU in the device (supported native APIs) and, on its basis, creates *"device"* constituting logical devices representing hardware resources of the GPU. Only such created *"device'* enables using the WebGPU interface, which acts as an abstraction layer referring to the selected native API. The diagram (see fig. 3) below shows the model of WebGPU.

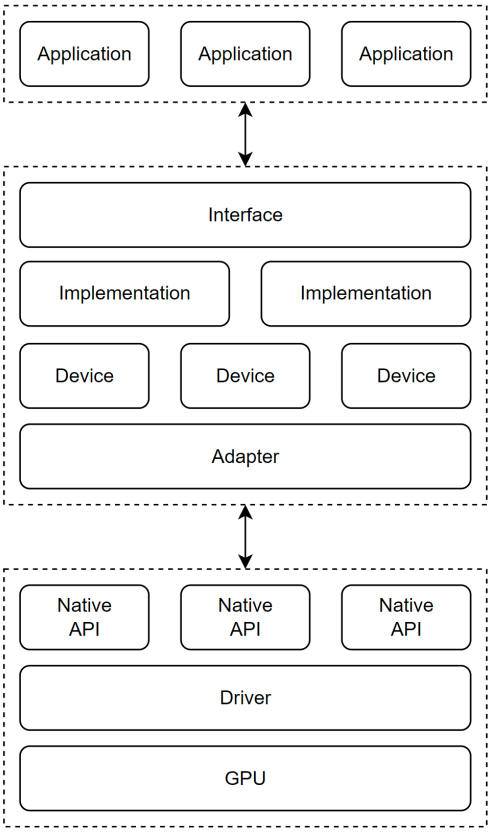

Figure 3: WebGPU model.

3.2 Implementation

3.2.1 Current state

The currently used libraries implementing the WebGPU interface are *"Dawn"* and *"wgpu"*. In both cases, for native applications, they extend the interface with functions that are not currently part of the official specification. Therefore, the documentation of the individual implementations should be considered when creating native applications.

3.2.2 *"Dawn"* library

The *"Dawn"* library is an open-source C++ implementation of WebGPU developed by Google and used in Chromium. It natively supports all modern graphics APIs and includes a C++ wrapper for the *"webgpu.h"* header file that allows you to use C++ syntax when using the library. A WGSL shader language compiler called *"Tint"* is included with the whole thing.

Due to the use of C++ language and native support of all modern graphical APIs, the *"Dawn"* allows for the broadest possible cross-platform native compilations. Unfortunately, due to the complexity of the library, it takes up a relatively large amount of space and requires downloading additional dependencies and more advanced project configuration.

3.2.3 *"wgpu"* library

The *"wgpu"* library is an open-source implementation of WebGPU in Rust developed by Mozilla and used in Firefox. It natively supports all modern graphics APIs and, in addition, in the case of the compiled version of the library, includes several wrappers for other programming languages.

Due to the Rust language and native support for all modern graphics APIs, the *"wgpu"* library allows for the broadest possible cross-platform native compilations. Unfortunately, when using wrappers for other programming languages, cross-platform compilation is limited to the systems for which the library was compiled.

3.3 Shading language

WGSL (WebGPU Shading Language) is a statically strongly typed shader language developed for WebGPU, with a syntax similar to Rust. It was designed to simplify the code and make it easy to convert to other shader languages.

Like WebGPU, WGSL relies on abstraction. The language described in the specification is not directly applicable to GPUs. Depending on the native API used by WebGPU, code written in WGSL is appropriately converted to the shader language supported by that native API.

4 Example application

4.1 Description

In order to present the capabilities of WebAssembly and WebGPU, an application was made in C++ that renders an interpretation of the symbolic model *"Utah Teapot"* [3] in OBJ format and then compiled it into a native and web version. It was decided to use the libraries *"Dawn"*, *"SDL"*, *"GLM"* and *"tinyobjloader"* in the form of source codes, which allow for a broad cross-platform compilation of the application in the native version. The main tools used to compile the application are *"CMake"*, *"Clang"*, and *"Emscripten"*. Since the application is to be a simple example, its structure is not extensive, and the rendered image is limited only to displaying the model and illuminating it using normal mapping.

4.2 Structure

4.2.1 Main

Due to the use of the C++ language, it was decided to create a *"FWebGPU"* class, which in its constructor initializes all fields to selected default values, and, in its destructor, deallocates all resources allocated by the *"SDL"* and *"WebGPU"* libraries. The *"main"* function, therefore, comes down to calling two public methods *"Setup"* and *"Loop"* on an object of the *"FWebGPU"* class (see fig. 4).

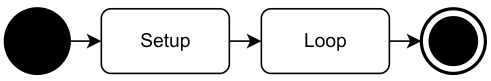

Figure 4: Function *"main"*.

4.2.2 Setup

The *"Setup"* method prepares all the required fields before rendering the image. The following activity diagram (see fig. 5) shows the individual activities performed within the *"Setup"* method.

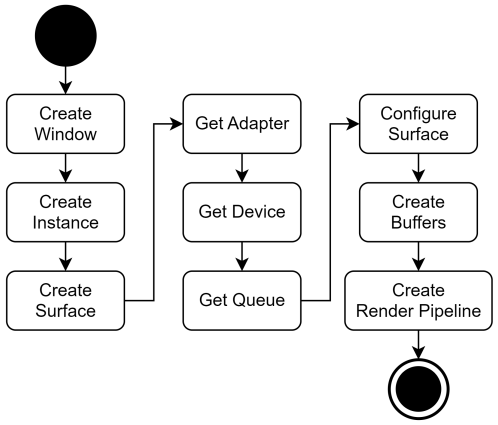

Figure 5: Method *"Setup"*.

All actions performed within the *"Setup"* method are described below.

Create Window: Method responsible for creating a window managing the surface in the case of a native application, using the *"SDL"* library for this purpose.

Create Instance: Method responsible for creating a WebGPU instance.

Create Surface: Method responsible for creating a surface on which an image is rendered. In the case of a native application, the surface is created within a created window, while in the case of a web application, the surface is created within the HTML tag *"canvas"*.

Get Adapter: Method responsible for obtaining information about available hardware WebGPU implementations on the device.

Get Device: Method responsible for creating a logical device representing the GPU hardware resources.

Get Queue: Method responsible for obtaining the rendering queue of a given logical device.

Configure Surface: Method responsible for configuring the surface, including the selected logical device.

Create Buffers: Method responsible for creating a Vertex Buffer used to send Vertex Data to the GPU describing the rendered models and a Uniform Buffer used to send Uniform Data to the GPU used by shaders. The method also loads the Vertex Data of the *"Utah Teapot"* [3] model into RAM, using the *"tinyobjloader"* library for this purpose.

Create Render Pipeline: Method responsible for loading the shader into RAM and creating the rendering pipeline.

4.2.3 Loop

The *"Loop"* method is responsible for reading and responding to user input and rendering the image. The following (see fig. 6) activity diagram shows the individual activities performed within the *"Loop"* method.

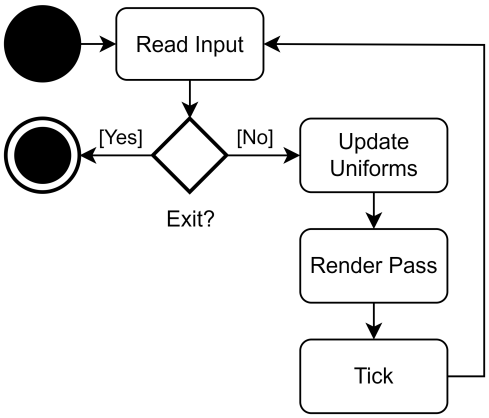

Figure 6: Method *"Loop"*.

All actions performed within the *"Loop"* method are described below.

Read Input: Method responsible for reading user input data, using the *"SDL"* library for this purpose.

Update Uniforms: Method responsible for updating the Uniforms value by the Uniform Buffer.

Render Pass: Method responsible for rendering an image on a surface.

Tick: Method responsible for updating the internal state of the GPU. In the case of a web application, this is done directly through the web browser.

4.3 Result

Depending on the toolchain configuration and *"CMake"* parameters, it is possible to compile a native and web application in general from the same source code.

The screenshot below (see fig. 7) shows the compiled native version of the application.

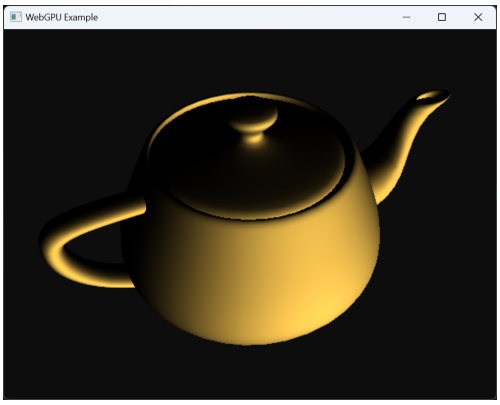

Figure 7: Native application example.

The screenshot below (see fig. 8) shows the compiled web version of the application.

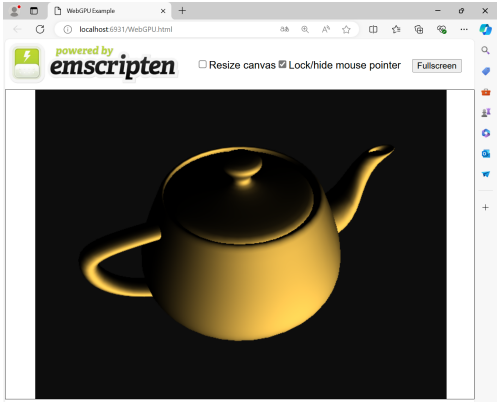

Figure 8: Web application example.

The native and web versions of the app behave the same, and there was no noticeable difference in performance when updating Uniforms with Z-axis model rotation based on frame render time.

The ability to compile a native and web application in general based on the same source code outside the *"Emscripten"* tool has been enabled by the use of macros that change active code fragments depending on the target system for which the application is being compiled. However, it should be taken into account that in the case of a native application, these fragments only concern the creation of a window and surface, i.e., aspects that typically depend on the target system, and in the case of a web application, additionally the method of updating the internal GPU state, which is implemented directly by the web browser.

5 Summary

The WebAssembly and WebGPU technologies discussed and presented in this article show how web browsers have developed over the last few years and the direction of their future development. Both technologies enable the creation of high-performance web applications, which was not their goal at the beginning of the development of web browsers.

WebAssembly is a JavaScript supplement that enables more efficient execution of parts of demanding code. In practice, however, it enables creating web applications based on languages typically used in native applications. This completely solves the problem of multiplatform because the code compiled into the WebAssembly module does not require any additional programs other than a web browser to run, and the compilation itself is performed once and is not dependent on the target operating system. It should be noted, however, that the security of WebAssembly resulting from the runtime environment of the web browser operating in the form of a virtual machine imposes certain limitations typical of virtual machines. An example of such a limitation is the lack of direct access to the local file system, which can effectively hinder the implementation of specific native applications in web versions.

WebGPU enables GPU operations to be performed within the browser environment, thus eliminating the problem of external runtimes and their associated plugins. In practice, however, it is a response to a much broader problem of lacking a modern cross-platform graphical interface, which Vulkan has tried to solve. It was created as a successor to the OpenGL standard and was not implemented on Darwin systems, which focused on their Metal interface and, at the same time, withdrew support for OpenGL. Apart from the MoltenVK compatibility layer enabling Vulkan to run on Darwin systems, there is currently no native graphical interface directly supporting all modern operating systems. In this situation, the solution turned out to be an abstraction layer over the currently used native interfaces introduced by WebGPU, which was the only widely accepted one.

In summary, WebAssembly and WebGPU are undoubtedly groundbreaking technologies that, when combined, enable the creation of efficient web applications based on the source codes of native applications. Thanks to them and other technologies, web browsers will become increasingly crucial as environments for cross-platform applications over time. It is also worth noting that WebGPU is currently the only graphical interface that supports all operating systems and web browsers. This may be of particular importance for application development tool manufacturers who can quickly provide native cross-platform support with the additional advantage of being able to compile web applications.

References

[1] Carlos Baraza (2017).
"Web Assembly Logo Icon".
`https://github.com/carlosbaraza/web-assembly-logo`
[Accessed on 30 September 2024].

[2] Brandon Jones (2021).
"W3C WebGPU Logos".
`https://www.w3.org/2023/02/webgpu-logos.html`
[Accessed on 30 September 2024].

[3] Martin Newell (1975).
"Utah Teapot".
`https://users.cs.utah.edu/~dejohnso/models/teapot.html`
[Accessed on 30 September 2024].