Introduction to WebAssembly

for BIS Multimedia students

By Hannah Koorbanally

U23587832

## Part 1: An introduction to WebAssembly for Multimedia students

In a fast-paced field of web and multimedia development, new technologies emerge, others evolve, and weaker technologies fade. These technologies grow and reshape the manner in which technologists build digital experiences. WebAssembly (WASM) is one of these reshaping technologies that creates lasting innovations.

*What is WebAssembly?*

WebAssembly is a low-level, binary instruction format that has been made to run close to native speed in a web browser (webassembly.org, n.d.). WASM differs from other languages, such as JavaScript, in that they are initially interpreted. WASM offers an alternative by providing a compact binary format which browsers execute directly through a virtual machine. Despite this, WASM is not intended to replace JS but rather work in conjunction with it, allowing for high-performance code such as gaming, graphics, and real-time simulations to work on-site (W3.org, 2019).

*Why is WebAssembly essential?*

As a Multimedia student or as a general developer, WASM can serve as a great tool in that it bridges the gap between traditional software development and web platforms. Imagine rendering, hosting, and gamification made simple: bringing interactive 3D visuals, minigames and other media types, such as AR, onto the browser without plugins. Performance-intensive applications are made simple by allowing developers to write in languages such as C, C#, C++ and Rust, thus bringing these languages into the browser ecosystem (MDN Web Docs, 2025).

*What relevance does WebAssembly have to the degree BIS Multimedia?*

Within this degree, multimedia concepts such as game design, web development, and interaction design have been explored and practised through numerous assignments, tests and practicals. WASM ties these areas together by allowing the user to create interactive experiences on the browser in a performance-oriented manner. Here are some examples:

1. From IMY 220 – integrating WASM alongside JavaScript frameworks for web programming
2. From IMY 300/310 – we can create high-performance games and simulations on the browser
3. From COS 330/ other Computer Science modules (COS) – applying knowledge of programming directly to writing code that compiles into WASM.

General users can no longer ignore how web technologies have been growing in popularity and evolving in their functionality, so much so that traditional desktop applications are no longer the only tool available (Mads Soegaard, 2015). It is through the use of WASM that such power-hungry, multimedia experiences, such as 3D tools, real-time collaboration and VR/AR prototypes, that such growth of web technologies occurs. For Multimedia students, this means that delivering projects seamlessly across different devices whilst retaining the performance of native apps has become easier and more efficient.

Overall, WebAssembly (WASM) is not just another trend – it is the way of supporting high-performance applications on the web, backed up by your trusted browsers. For you, an upcoming Multimedia specialist, it poses as an opportunity to merge your creativity with good performance: delivering games, interactive art and dynamic work has never been easier.   
  
(456 words)

### Bibliography

**Reference list**

MDN Web Docs. (2025). *WebAssembly concepts - WebAssembly | MDN*. [online] Available at: <https://developer.mozilla.org/en-US/docs/WebAssembly/Guides/Concepts>.

W3.org. (2019). *WebAssembly Core Specification*. [online] Available at: <https://www.w3.org/TR/wasm-core-1/#design-goals%E2%91%A0> [Accessed 15 Sep. 2025].

webassembly.org. (n.d.). *WebAssembly*. [online] Available at: <https://webassembly.org/>.

Mads Soegaard (2015). *Native, Web or Hybrid App: Which One is Better?* [online] The Interaction Design Foundation. Available at: <https://www.interaction-design.org/literature/article/native-vs-hybrid-vs-responsive-what-app-flavour-is-best-for-you?srsltid=AfmBOor47k7_z5CmPtRkILPIi7VA7CYuCAgcmOWQzeclqzbY0zfsofMJ> [Accessed 15 Sep. 2025].

## Part 2: Creating a Dino Jumper game on WASM using C#(The how)

1. Install .NET SDK

* Go to <https://dotnet.microsoft.com/en-us/download/dotnet/thank-you/sdk-8.0.414-windows-x64-installer>
* Download the LTS version (should be on the right)
* Click on the downloaded installer file and wait for it to install.
* To confirm successful download, type “*dotnet --version*” on your Command prompt: a version number should appear.

A black background with white text

AI-generated content may be incorrect.

1. Creating a Blazor WASM project

* Type the following into your command prompt:  
  *dotnet new blazorwasm -o DinoHop*

*cd DinoHop*

* This creates the folder for the DinoHop game, which has a Blazor WASM project. Note that this file will have a premade project, which can be changed and adjusted accordingly. Details on that to follow.

A screen shot of a computer

AI-generated content may be incorrect.

1. Running the project:

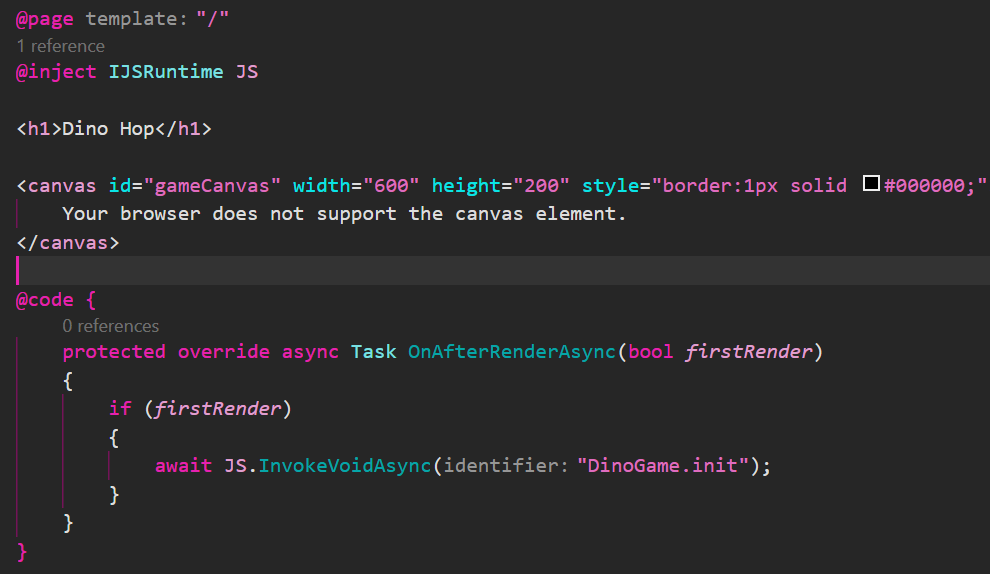
* Command Prompt: dotnet run
* Allow it to build and Ctrl+click on the link given within the returned text. An example:

A screen shot of a computer program

AI-generated content may be incorrect.

1. Open Pages/Home.razor and replace with code needed for the game.

* See GitHub for changes made to Home.razor. Copy the contents of this file and paste it into your project’s Home.razor file.
* NOTE: The file contains a reference to a method that will be created in a file called game.js (do not be alarmed, follow the steps as per normal, and this will make sense in step 5)
* Copy the code from GitHub’s file and paste it into your project’s file.



1. Create a file called game.js (DinoHop/wwwroot/game.js)

* Copy the contents of DinoHop/wwwroot/game.js from GitHub and paste it into your newly created game.js file.
  + This file has the logic for the initial layout of the game, including the start screen, game loop, obstacles, dino character, gravity, game logic and mechanics.
* Attach this file as a script tag to index.html (which exists in the same directory). See index.html on GitHub for reference (DinoHop/wwwroot/index.html)
* NOTE: This code calls on images that are used as sprites for the application. To do this, create a folder called “Images” so that your path looks like the following: *DinoHop/wwwroot/images/*
* Add the images inside the folder. Take note of their name and their image type. If you would like to use your own images, follow the following steps:
  + The GitHub folder will contain the image files used for this tutorial. If you wish to change these images, see comments inside of game.js to change the images to your choosing. Replace image names and ensure that the right file type suffix is used.

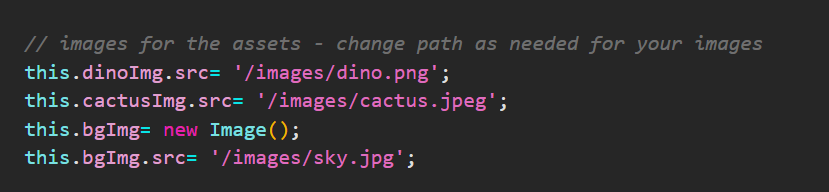
A screenshot of a computer

AI-generated content may be incorrect.

A screen shot of a computer program

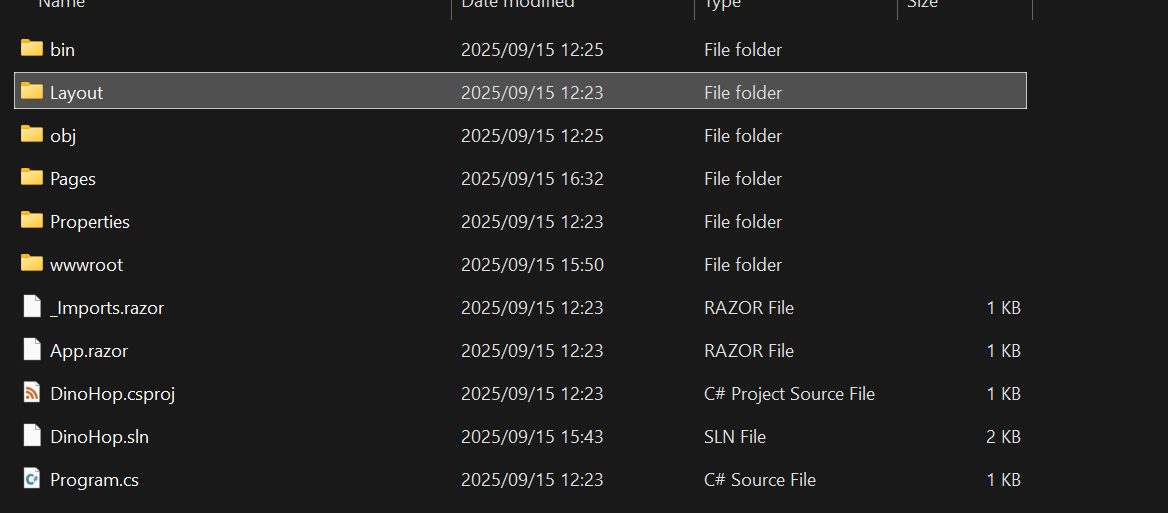
AI-generated content may be incorrect.

Figure 1- Note: not the entire file



1. Delete unused pages

* Counter.razor, and Weather.razor: DinoHop/Pages
* Delete navbar in MainLayout.razor (located in Layout folder – project’s root)



1. Save and run files:

* In the command prompt, use *Ctrl+C* to stop the initially running project.
* Ensure all changes are made and saved
* Use *dotnet run* to re-run the project and see changes.

A screenshot of a video game

AI-generated content may be incorrect.

A blue sky with clouds

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

1. Enjoy your new minigame!

## Part 3: Explanation of why these steps are needed for the tutorial (what and why)

***What have we just created?***

The tutorial has shown you the step-by-step instructions on how to create Dino Hop, an interactive runner minigame.

***How have we done this?***

If followed correctly, the tutorial would have taken you through how to use C# and Razor to give structure to the project. JavaScript handles canvas drawing and the game loop, which is hosted in the game.js file. Blazor compiles to WASM, enabling C# to run in a browser. Assets are loaded from the wwwroot folder just like any normal static site.

***Why do we need to make use of Blazor WASM?***

We use Blazor WASM to compile the .NET files into WebAssembly, allowing for C# to run on the browser. When we call dotnet run on the command prompt, the Blazor project is compiled into WebAssembly bytecode, which spins a lightweight server to host the app. When we separate the game logic (game.js) and the UI (Home.razor), we are bridging between low-level execution and the web’s DOM.

***The initial setup process, conversion and final publication of the tutorial***

1. Initial setup:

* Downloading the LTS version of .NET SDK and creating a Blazor project (steps 1-2). This gives us the compiler and runtime environment needed to build the C# code. This supports Blazor WebAssembly projects.
* Using dotnet new blazorwasm to create a new Blazor WASM project, which contains a skeleton of a complete project structure using Razor pages, wwwroot for static files and build settings that automatically compile C# into WebAssembly.

1. Conversion/compiling

* Using dotnet run, we convert the C# into WebAssembly bytecode. Blazor uses the Mono runtime compiled into WebAssembly, which means that the C# code that was written can be executed inside the browser with no plugins.
* Razor components, like the pages, are compiled into C#, which then becomes a part of the WASM build.
* The game.js and wwwroot folder remain plain static assets, yet WASM allows C# to interact with them via JavaScript interop.

1. Publication

* After the needed changes to the files, running dotnet run starts the lightweight server and provides a local URL, which is clicked on to load the Blazor app.
* The browser initially downloads the blazor.webassembly.js loader script, which will then fetch and run the compiled .wasm files.
* The browser will serve the static assets (game.js and wwwroot)
* Game.js will run inside the canvas and Blazor will deal with the app’s structure, lifecycle and UI.

## Part 4: GitHub