Research into engines for developing 2D video games in web environments

Felix Hernandez Muñoz-Yusta

September 6, 2024

ESNE: University School of Design, Innovation and Technology
Degree in Video Game Design and Development
Programming Specialty
Course 2023/2024

ESNE Escuela Universitaria de Diseño, Innovación y Tecnología

Index

Abstract

Nowadays, there are multiple tools capable of developing video games that are accessible to anyone. Additionally, it is true that there is currently a trend towards the development of games that run directly from the browser, meaning no installations are needed and allowing users quicker access to play these games.

The objective of this research is to gather and compare the video game engines that are currently available and select the most suitable ones for this study. After the selection, the strengths and weaknesses of each engine were compared, and a small game was created as a test for the research. This research aims to analyze how well the engines are adapted for web game development. The engines studied are Unity, Godot, Construct 3, Phaser.js, and Defold.

web video games, Unity, Godot, Construct 3, Phaser.js, Defold, analysis.

Introduction

Today, there are a vast number of engines available for video game development. These engines are used by both large companies and small projects. To achieve the most widespread use, they strive to be as versatile as possible, as this increases the possibilities for use in many more areas. Online gaming is currently trending, as the most popular games are multiplayers, making online video game development a field worth investing in.

The idea for this research came from the news that came out in September, in which one of the most widely used game engines, Unity, changed its pricing policy for developers. Previously, it used a subscription model with different plans: Unity Personal (free for revenues under $100,000/year), Unity Plus ($399/year for revenues under $200,000/year), Unity Pro ($2,040/year with no revenue cap), and Unity Enterprise (for large companies with negotiable prices).
However, Unity announced a new plan that generated controversy: they intended to charge developers a fee for each install of a game using their engine, as long as they exceeded certain revenue and number of installs ($200,000 in revenue and 200,000 installs for Unity Personal and Plus, and $1 million in revenue and 1 million installs for Unity Pro and Enterprise). The fees would start at $0.20/install and could decrease to $0.01 depending on the number of downloads.
This change was poorly received by the developer community, as it could lead to unpredictable and disproportionate costs, especially for free-to-play or indie games. As a result of all the developer anger, the company behind Unity, Unity Technologies, decided to reverse these changes and leave everything as it was before.

Following this news, many developers began looking for alternatives to develop their games in an engine that would give them more freedom. However, the problem they encountered was that there are currently a huge number of engines, each focused on a specific field, a different programming language, a different interface, etc.

The problem to be solved is a combination of two themes. The first is the rise of online games, or at least those available in web format. The second is the search for a graphics engine to understand the existing varieties, in addition to those already established as the most widely used on the market.

To solve this problem, a series of selected engines that meet the requirement of being compatible with the creation of web games will be studied and compared. The engines ultimately selected are Unity (to compare what many consider the "standard" for a development engine), Godot (the engine considered to have gained the most popularity after the events with Unity's tariffs), Construct 3 (a completely web-based engine), Phaser.js (a JavaScript library that includes a graphical editor version), and Defold (known for being compatible with all platforms).

The results of this research provide an overview of the advantages and disadvantages of each engine and an explanation of which engine is considered most suitable for developing a game for web format.

General objectives

This research is organized according to the following objectives:

1. Investigate whether there are other investigations similar to the one being proposed.
2. Find out which development engines are the most used.
3. Research the tools and uses of the selected engines.
4. Begin simultaneous development on engines.
5. Take notes of the differences between the engines (during development).
6. Compare the results obtained.

Conceptual and contextual framework: the state of the art

This research is a comparative study in which, from all the currently available video game development engines considered most suitable for the study, in addition to being compatible with the creation of web games, will be filtered and analyzed individually.

It's important to emphasize that, for the purposes of this research, when referring to a web format, we're referring to the game's ability to run directly from a web page, meaning no installation is required to play. This doesn't mean the engine has the ability to offer multiplayer services, as that would require external services, such as one or more servers, to support the game.

To begin, it's necessary to understand the definitions of the topics covered in this research. The definition of a game engine is:

> *A game engine is a software program designed to facilitate video game development by providing a set of reusable and extensible components. These engines allow developers to focus on creating art, laying out game worlds, and defining game rules, while the engine handles essential aspects such as 3D graphics rendering, collision detection, object physics, character animation, and audio. The emergence of the concept of a "game engine" date back to the mid-1990s, particularly with first-person shooters like "Doom," which implemented a clear separation between engine components and specific game content.* (Gregory, 2014)

It is also important to know the definition of what is known as a web game, that is, a game that supports web pages:

> *Web-based games are those designed to be played exclusively in web browsers, using technologies such as HTML5, CSS3, and JavaScript, without the need to install additional plugins or applications on the user's device.* (Makzan, 2011)

It's worth noting that this definition fits a basic concept of web-based games, as there are currently engines that allow for small executables to be placed within the website itself.

However, it's important to note that technological developments have allowed web-based games to go beyond simple script-based applications or multimedia content.

It was not until the creation of the Java language in the 90s and the beginning of the use of applets (small applications that ran within a program) when it was possible to start including small games within web browsers.

Later, in the 2000s, Flash technology (Adobe Flash, formerly known as Macromedia Flash) emerged, allowing developers to create interactive animations and applications. This technology enjoyed tremendous popularity because it didn't require any downloads or installations to work.

Despite its huge success, Flash technology had its limitations, and it wasn't until 2010 that HTML5 arrived, introducing new graphics, multimedia, and on-screen element manipulation capabilities.

Later came WebGL, a JavaScript API for 3D graphics rendering, which allowed games to be created without additional plugins. Furthermore, since it's based on OpenGL, it's compatible with most browsers.

As web game development gained importance, development engines began attempting to make their engines capable of running games on web pages. Previously, these engines focused mostly on trying to maximize the computer's potential and performance, allowing for the creation of games with improved graphics. However, this wasn't the case in web format, as, due to its limited capacity, ways had to be found to compress the engine's executables to meet these limits. On the other hand, engines focused exclusively on the web format began to emerge, such as Phaser.js, which is based on HTML5.

Today, new technologies such as WebSockets and WebRTC have been developed, enabling the creation of real-time multiplayer web games, which previously required complex server deployments.

The arrival of new environments, such as mobile devices, has further expanded the audience for web games, as these devices allow users to connect to the internet and play games from anywhere. This has led many games to adapt their environments to mobile devices, as game play shifts from keyboard and mouse controls to touch controls on a mobile device.

Web technologies

Among the technologies that will be discussed is HTML5, which is defined as:

> *HTML (HyperText Markup Language) is the most basic building block of the Web. It defines the meaning and structure of web content. In addition to HTML, other technologies are commonly used to describe a web page's appearance/presentation (CSS) or functionality/behavior (JavaScript).* (MDN contributors, 2024)

Specifically, HTML5 has been key to enabling the development of web games, as it is now capable of adding APIs (mechanisms that allow two software components to communicate with each other) that handle graphics, audio, and video very efficiently and without the need for plugins, something that was mandatory with Adobe Flash. Another very important aspect is browser compatibility, including those for mobile devices.

On the other hand, there is CSS3, which can be defined as:

> *Cascading Style Sheets (CSS) is the style-setting language used to describe the presentation of HTML or XML documents (including several XML-based languages such as SVG, MathML, and XHTML). CSS describes how the structured element should be rendered on screen, on paper, in speech, or in other media.* (MDN contributors, 2024)

This style sheet, in this context, is used to style user interface (UI) elements such as buttons, menus, text, and information displays. It is also capable of creating animations or transitions.

It is also worth highlighting the use of JavaScript, whose definition is important to know:

> *JavaScript (JS) is a lightweight, interpreted, or just-in-time compiled programming language with first-class features. While it is best known as a scripting language for web pages, and is used in many non-browser environments, such as Node.js, Apache CouchDB, and Adobe Acrobat, JavaScript is a prototype-based, multi-paradigm, single-threaded, dynamic programming language with support for object-oriented, imperative, and declarative (e.g., functional) programming.* (MDN contributors, 2023)

This language is used in multiple web game engines and is essential for web game development, as it is capable of modifying the DOM (Document Object Model), managing user events (input detection), and executing game logic in real time. It is also capable of handling APIs that allow drawing within the canvas, modifying 2D and 3D graphics (with the help of WebGL), playing audio, and supports the use of WebSockets, enabling the creation of multiplayer games, among many other features.

Finally, it is important to understand WebGL technology since; despite having been mentioned previously in the context of JavaScript, it is relevant to this research:

> *WebGL allows web content to use an OpenGL ES 2.0-based API to perform 3D rendering on an HTML canvas in supported browsers without the use of plug-ins. WebGL programs consist of control code written in JavaScript and special effects code (shaders) that run on a computer's graphics processing unit (GPU). WebGL elements can be mixed with other HTML elements and blend into other parts of the page or the page background.* (MDN contributors, 2023)

The addition of WebGL to development environments has allowed games running in browsers to have graphics comparable to those of a console.

Development engines

Another vitally important aspect is the existence of game engines. This research will focus on the following engines:

First, we have one of the most famous engines, Unity, used for both 2D and 3D development since its launch in 2005 by Unity Technologies. Among its main features, we can see that it uses the C# programming language for development and has a graphical editor with multiple built-in tools to assist developers in their work.

It's one of the most versatile engines available today, capable of exporting games to consoles, computers, mobile devices, and web browsers via WebGL. It features an in-engine store called the "Asset Store," where developers can upload their own Unity-compatible objects, models, scripts, or other resources to facilitate development for other users.

Another key aspect of this engine is its large community and extensive documentation. Being such a widely used engine, its forums and documentation are full of useful information for troubleshooting any issues that may arise during development.

Secondly, we have Godot. This engine has gained popularity among small game studios, as it is very accessible and highly flexible. It was released in 2007, and its development was led by Juan Linietsky and Ariel Manzur. However, being open source, it has received the support of thousands of contributors, allowing the creation of an engine perfectly capable of developing the most demanding games. It can support both 2D and 3D projects and uses its own programming language, GDScript, similar to Python. It is completely free to use.

The architecture of this engine is node-based, meaning the entire project structure can be organized based on nodes, which simplifies its structuring. It also features a visual editor that allows you to modify nodes and scripts, all within the editor. It's a cross-platform, meaning it can export to computers, mobile devices, and, of course, to the web via OpenGL.

It's important to note that Godot gained considerable momentum following the controversial news of Unity's pricing change, and despite a subsequent retraction, Godot became known to many developers who preferred to give this engine a chance.

On the other hand, we have the Construct 3 engine. This is the only engine in this research that runs directly from its website, meaning no installation is required to use it. It's also an engine that doesn't require programming knowledge, as it uses a system of "boxes" or events to manage all the game logic. It was developed by Scirra, and although the initial version of this engine was released in 2007, it wasn't until 2017 that the third version was released, which works with HTML5 and JavaScript.

While it's true that it's not capable of developing games as demanding as other development engines, its ease of use and easy access allow it to position itself as a competent engine for 2D game development. The engine has a paid version, but there is a free version that's sufficient for making a personal game.

The fourth engine discussed in this research is Phaser.js, which is specifically a library that uses the JavaScript language. Although the library can be used for free through a code editor, there is a graphical editor that acts as an engine and can be used for a monthly fee. Developed by Photon Storm in 2013, this graphical editor is called Phaser Editor and is currently in its fourth version. This library stands out for its simplicity and effectiveness when creating 2D games.

Being based directly on HTML5 and JavaScript, it's designed for creating web games, making its implementation simple. It also supports technologies like WebGL. Throughout its history, this library has built a large community, allowing for the development of active forums with advanced users. Finally, we find Defold, an engine developed by King employees Christian Murray and Ragnar Svensson. They have worked on this engine since 2008 as a side project for in-house use, but in 2016, it was decided to make it open source. Finally, in 2020, the engine's management became completely independent from the company, becoming the Defold Foundation.

This engine stands out for being completely free, and its main feature is its ability to export games to any console, computer, mobile phone, and even web browser. It uses Lua as its programming language, and while it supports 3D, it is primarily designed for 2D games.

All these engines meet the requirements set out in this research, which are:

- Have an IDE, that is, an integrated development environment within the engine itself.
- Be compatible with the creation of 2D games.
- That they are compatible with the possibility of creating games for the web.

Finally, it can be added that, after conducting research on other studies that refer to comparisons between different development engines, it has been observed that there are multiple and varied comparisons of tools, evaluating aspects such as cost, functionality or ease of use, according to the author's criteria.

However, it is noteworthy that no research has been found that, in addition to comparing these engines, simultaneously focuses on their export capabilities to web format.

While this research is not intended to be innovative or disruptive, it is true that, after analyzing the work published on various research platforms related to this topic, no study has been identified that is similar to the present research.

Research methodology

The objective of this research is to compare the engines selected for study. For comparison, the engines themselves will be evaluated, from the visual aspect, such as the IDE (integrated development environment) offered by each engine, the tools that facilitate development for users, the difficulty of the learning curve, the documentation available to understand their use, to the programming languages used by each engine.

The engines to be studied are Unity, Godot, the paid version of Construct 3, the paid version of the Phaser.js library (Phaser Editor), and Defold. GitHub was used as the repository for project creation and maintenance.
The research will develop as the projects progress within each engine. The first step is to create the project within each engine and create a GitHub repository to allow for project backups. The project will then be adapted to a web format if necessary.

As a first step in the project, the character will be created and adjusted to the specific object type of each engine (node, object, object, scene, etc.), analyzing what type of component each one uses.

Next, the character's movement animations will be added, either through sprite sheets if the engine allows it, or through separate images if it can't understand them. It's essential to add animations to the character so that the correct animation is displayed at all times, depending on the action performed.

The next step will be to add the floor and platforms that will make up the level. Tools that can facilitate the creation of scenarios will be investigated, if possible, as many engines currently include functionality for this purpose.

Next, collisions will be added between level elements. After this, gravity will be set for the objects so that it applies to the necessary elements.

The next step is to create the movements (*inputs*) of the character so they can move around the level by pressing the assigned keys. Similarly, animations will be integrated so that, depending on the action the character performs (standing still, moving, jumping, or falling), the corresponding animation is displayed.

An important aspect is that the camera follows the character throughout the level, as this is a necessary feature for the correct development of a game.

Although it can be considered a simple decorative element, adding a background to the level improves the quality of the product and allows you to know if there are tools inside the engine that facilitate its use.

Finally, a small text will be included within the canvas or UI (user interface) to display relevant information or instructions to the player.

After completing these objectives, we intend to compile all the information obtained, as well as the notes and results from the research, into this document. This will provide a summary of the main advantages and disadvantages of each engine.

In addition to developing the level to test the features of each engine, we aim to analyze the external elements surrounding them, namely their official documentation, their community and active support, compatibility with other libraries, and the software architecture they support.

There's no attempt to find a definitive solution or a winner among all the selected engines. Each engine was chosen for its particular characteristics, and each has specific strengths that may make it more suitable for the type of game or project being developed. In this case, a 2D side-scrolling game was chosen, as it's a widely known genre and can be easily run from a web browser.

All character art, scenery, and textures are free of any kind and do not have a license for commercial use.
It is also important to mention that ESNE University has provided me with an access code to use the full version of the Construct 3 engine.

Project development

The development of the projects in the engines used in this research will be described below.

It's important to note that the Unreal Engine is not part of this research, despite being another well-known and widely used engine, because it's not capable of natively exporting to a web format. It's true that there are plugins created by users outside of Epic Games (the creators of Unreal Engine), but these aren't widely used and haven't proven to be particularly effective.

On the other hand, it is also important to emphasize that all games have been designed within the 2D field and while it is true that several of the engines to be investigated are compatible with the 3D format, in this research it has been decided to limit it only to the two-dimensional (2D) field of vision.

The final result is expected to be a level with a character that has gravity applied to it, collides with the ground and platforms, and is able to move around the stage.

Figure 1 below shows the expected result during development on the engines:

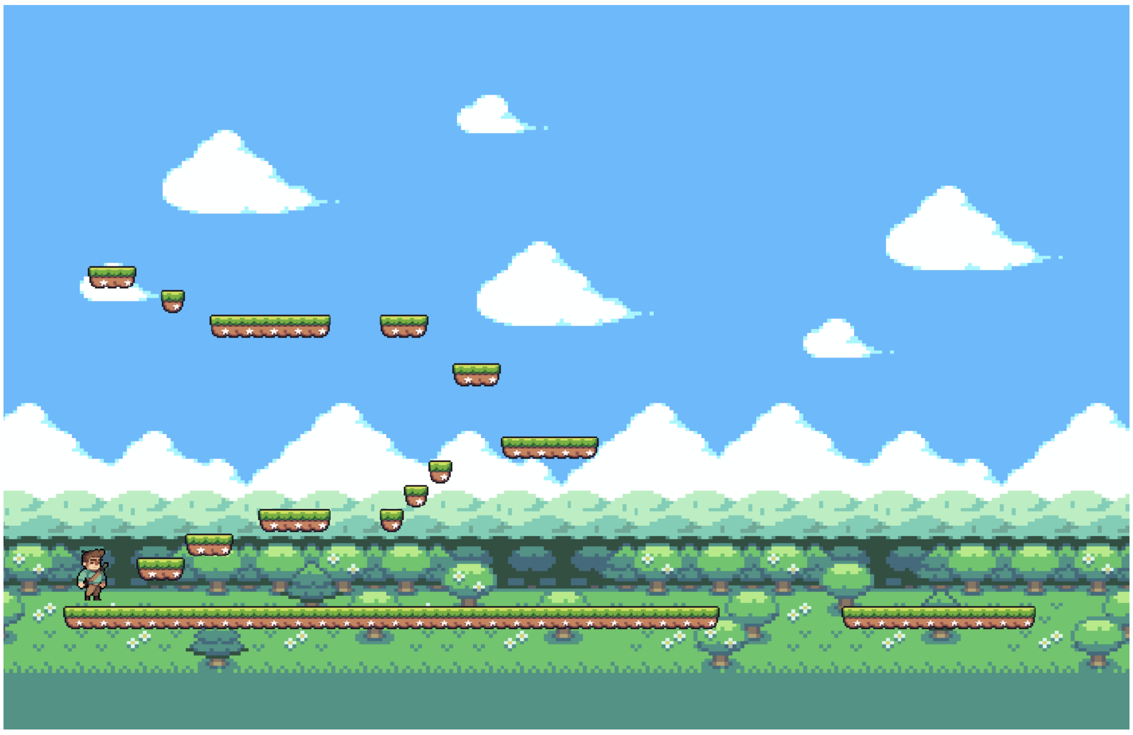


Figure 1 - Outline of the level to be developed

Unity

The research began with the creation of the project in Unity, one of the most well-known and widely used engines today. Unity requires the installation of a manager called Unity Hub, which allows you to create new projects, select the engine version to use, and manage the type of project you want to develop.

The version used for this project is 2022.3.22f1, an LTS version (*Long Term Support*), which stands out for offering extended support and is recommended for long-term projects due to its stability and lower risk of errors. Unity Hub allows you to change the version of Unity used for a specific project, which can be useful for accessing new features or performance improvements. However, upgrading to a newer version may cause compatibility issues that require adjustments to your code or project resources. It's important to note that it's possible to upgrade a project to a newer version of Unity, but it's not possible to revert to a previous version without the risk of losing data or functionality.

Within the Unity interface, it can be seen that it is highly customizable, allowing you to adjust not only the tabs of the elements to be displayed, but also through the tab *Layou*t It's possible to completely change the layout of the tab groups. The default layout allows for seamless access to all the engine elements.

Unity allows you to group assets into folders according to your developer's preferences, making it easier to organize your project during development. Within the engine, scenarios are grouped into files called *Scenes*(scenes). These files store all the objects that will be displayed in that scene when they are running; these are called *GameObjects* and there are many types, such as characters, cameras, among others.
Next, Figure 2 shows an image of the Unity editor with the game project that was developed for this research.

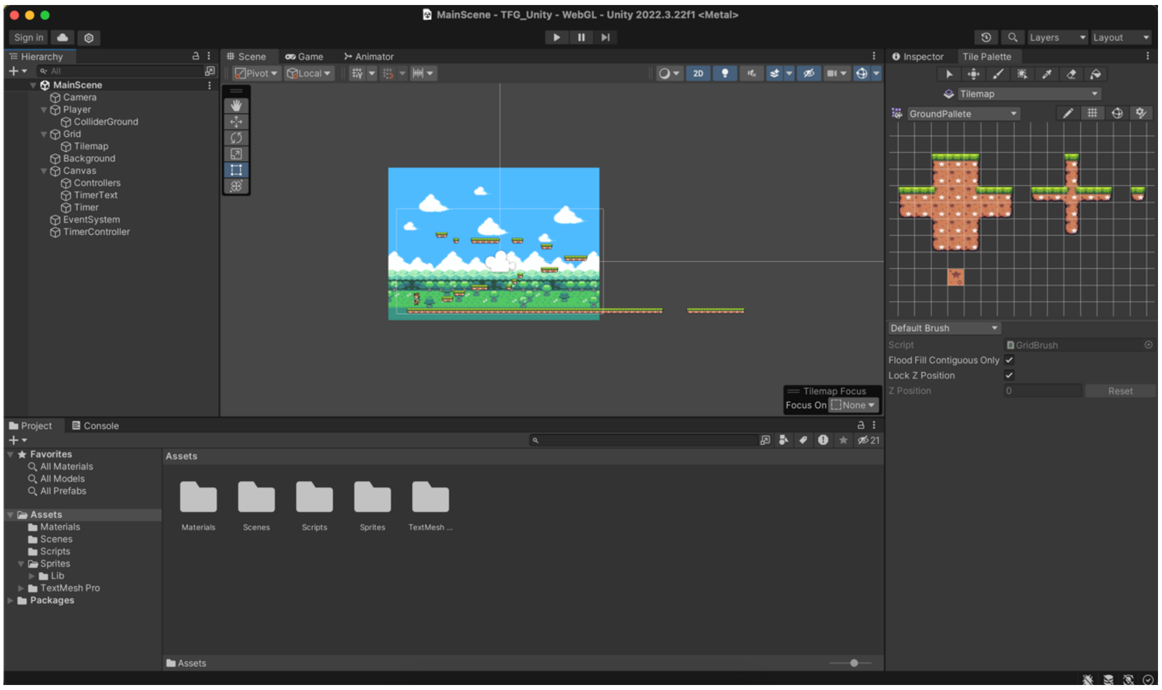


Figure 2 - Unity Editor Screenshot

Within this image, multiple elements can be observed. While the organization of folders is free for each developer, there is generally a tendency to maintain certain standard structures, such as the one shown in the image, where scenes are stored in one folder, scripts (code sheets) in another, sprites (objects) in another, and so on. In the upper left window, you can see that the *Main Scene* (main scene) contains all the elements that compose it: the player with his collisions, the stage, the background, the camera and the canvas elements.

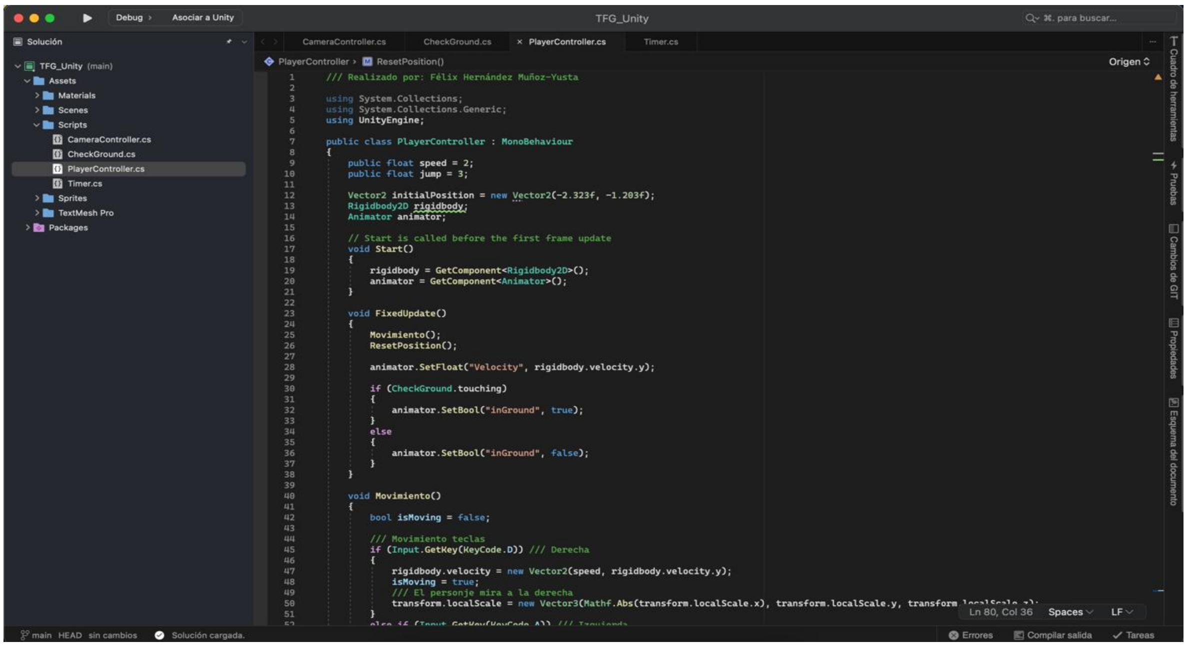
Normally, when creating a scene, Unity includes by default a camera, which is how the player will see what is displayed in the scene, and a light to illuminate the objects in the scene.

For animations, Unity has a window that allows you to create an animation based on a sprite sheet or a set of separate images and can be assigned to an object in the scene. Additionally, when these animations are assigned to a *GameObject*, like the character, Unity offers the ability to switch between animations based on specified conditions, which the developer must adjust based on their project.

On the other hand, Unity includes a tool to create scenarios based on clippings of an image, called *Tilemap* as you can see in Figure 2, the top right window displays a terrain cut into multiple image cubes. These cubes can be placed throughout the scene to facilitate level creation. The images used for this purpose are stored along with their cropping data in files called *Tiles*, and every cube added to the scene already has collisions and all the necessary elements, like any other object.

To write your project code, Unity installs the Visual Studio development environment by default, although other text editors can be used as long as they support C#.

The Visual Studio editor with a snippet of the character controller script is shown in Figure 3 below.



```
/// Realizado por: Félix Hernández Muñoz-Yusta

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class PlayerController : MonoBehaviour
{
    public float speed = 2;
    public float jump = 3;

    Vector2 initialPosition = new Vector2(-2.323f, -1.203f);
    Rigidbody2D rigidbody;
    Animator animator;

    // Start is called before the first frame update
    void Start()
    {
        rigidbody = GetComponent<Rigidbody2D>();
        animator = GetComponent<Animator>();
    }

    void FixedUpdate()
    {
        Movimiento();
        ResetPosition();

        animator.SetFloat("Velocity", rigidbody.velocity.y);

        if (CheckGround.touching)
        {
            animator.SetBool("inGround", true);
        }
        else
        {
            animator.SetBool("inGround", false);
        }
    }

    void Movimiento()
    {
        bool isMoving = false;

        /// Movimiento teclas
        if (Input.GetKey(KeyCode.D)) /// Derecha
        {
            rigidbody.velocity = new Vector2(speed, rigidbody.velocity.y);
            isMoving = true;
            /// El personje mira a la derecha
            transform.localScale = new Vector3(Mathf.Abs(transform.localScale.x), transform.localScale.y, transform
        }
```

Figure 3 - Screenshot of the Visual Studio editor

Once the project is created, the first step is to choose the format to which you'll export. By default, the option for computers (Windows, macOS, and Linux) is enabled, but since the project will be focused on the web, the ideal option is WebGL.

All visual elements and textures to be used are imported. Unity is able to detect sprite *sheet and* automatically split them into the necessary frames, but if this is not done correctly, it can be edited manually. On the other hand, Unity has a built-in store called *Asset Store* where you can get *scripts*, visual elements (*sprites*) or objects (*GameObjects*) created and developed by other users who have uploaded their work to this store and are permitted to be used in exchange for a fee.

The next step is to create, even if only in a basic way, the character using cubes to define its movement and collisions. To add these components, it is necessary to include within the characteristics of the *GameObject* the elements *Rigidbody* (which manages the gravity and physics of the object) and *Box Collider* (which defines the shape and collision limits of the object, usually square in shape but adjustable in size).

Once this is completed, the next step is to create the animations using the already imported images and separate them into different animations through the window *Animation*. Once the necessary animations are saved, in the window *Animator* all previously created animations are organized into a controller that manages the logic for alternating animations based on variables adjustable by the developer to adapt to any possible scenario. These variables can be modified from the Unity editor or through a code script.

To create the scenario in this Unity project, a tool composed of two main components was used: *Tiles* and *Tilemap*. The *Tiles* are generated from images, which are divided into symmetrical cubes. These *Tiles* represent small portions of the game environment, such as parts of the floor or walls. On the other hand, the *Tilemap* is the organizational structure that allows you to place and manage these Tiles in a grid, thus forming a complete map or level. Since the *Tilemaps* commonly used for floors and other scenery elements, they already have built-in collisions and physics, making it easy to recreate a coherent and functional world.

With the character and the setting already created, a story is developed *scrip*t which is responsible for capturing user input from the keyboard (*inputs*) to apply movement and jumping forces to the character. Additionally, if the developer deems it appropriate, they can have the character's animations modify in response to user input, providing a more dynamic and responsive visual experience.

A fundamental element for the proper development of a game is the ability for the camera to follow the character throughout the level, ensuring that the player always remains centered on the displayed plane. Although many games do not use this logic or employ variations of it, this project decided to implement it for a more comprehensive analysis. Camera tracking is achieved through a script that allows the camera to be centered on any selected object, in this case, the character.

Another important aspect, although more aesthetic in nature, is the inclusion of a background image in the stage. To achieve this, it was decided that, like the camera, the background would follow the character through the same *scrip*t from the camera. This ensures that the user's view is always present, improving visual immersion and continuity of the setting during the game.

Finally, to complete the level, information was added within the user interface layer. Although limited to basic instructions on how the game works, this latest addition allows for the completion of a fully functional level.

After completing the development of the first game, it's clear that Unity has multiple built-in tools that facilitate smooth development. While it's true that researching these tools is required to understand their functionality and compatibility, once you've overcome this learning curve, game development in Unity becomes quite straightforward.

It is important to highlight the wide range of information and documentation available; an example is the official Unity page as in the *Unity Documentation. This* documentation is available in multiple languages and is updated based on the engine used. There's also an extensive catalog of active forums where users post problems and questions and seek help from more experienced developers.

In terms of compatibility, Unity can be installed on any operating system, be it macOS, Windows, or Linux (although with less support for the latter). When it comes to exporting projects, the engine is compatible with a wide range of platforms, including consoles, computers, mobile devices, servers, and, as in the case of this research, web formats.

During development, it's worth noting that if you don't have prior knowledge of Unity, you'll need to learn how to use the engine and become familiar with C# to work effectively. To date, the company behind Unity has kept the original costs unchanged, so as long as these licenses are maintained, Unity will remain an affordable development environment for small businesses.

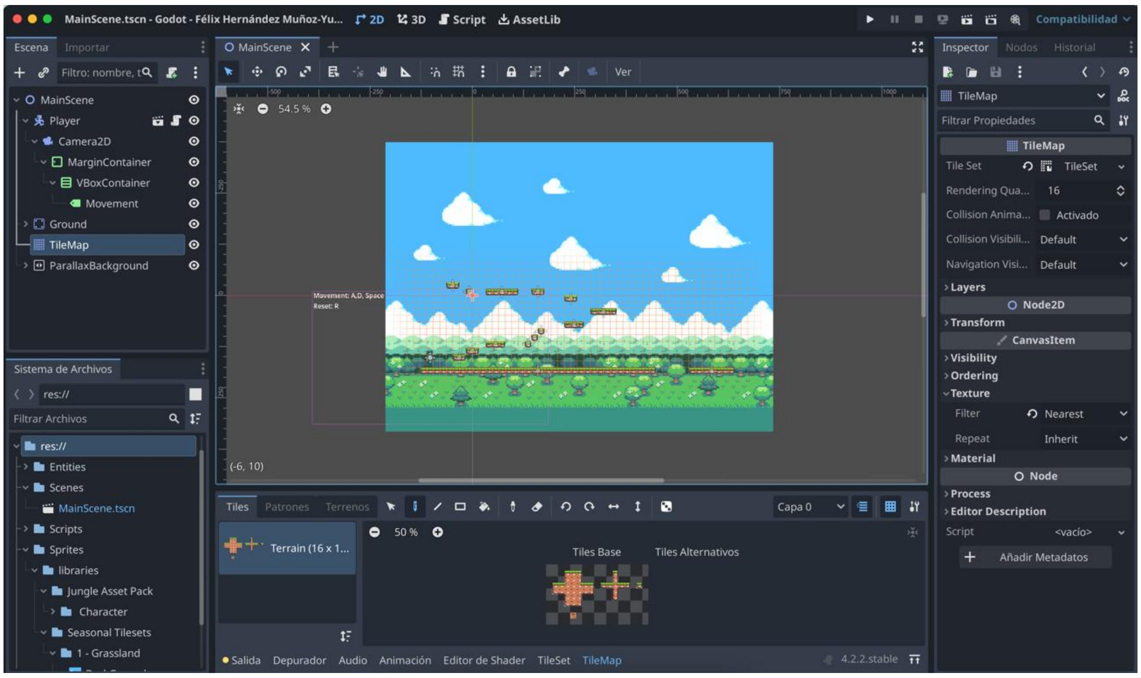
Finally, once the project is exported, a folder is generated containing the basic elements for the website: an HTML file, a CSS file, and several JavaScript files, including the game logic translated for WebGL compatibility. Also included are compressed files containing all the artwork provided by the developer, which are subsequently decompressed during execution when the project is loaded from a web page.

Godot

This engine has gained a lot of popularity in recent years, as it is presented as a free-use engine and *open-source* This means that any developer can create modified versions of the engine at no cost, although proper credit must be given to Godot if modifications are made.

The engine has a start menu that allows you to select between different projects and even offers predefined templates as a base. There are different versions of the engine, with two main types: the LTS version (*Long Term Support*), which is considered the most stable and is recommended for long-term projects, and the latest published version, which incorporates more new features and is constantly updated, which could lead to possible errors or instabilities.

The interface of this engine is highly customizable and features a preconfigured tab system, as shown in Figure 4. These tabs can be modified according to the developer's needs.



Tile Set: TileSet
Rendering Qua...: 16
Collision Anima...: Activado
Collision Visibili...: Default
Navigation Visi...: Default
Filter: Nearest
Repeat: Inherit
Script: <vacío>

Figure 4 - Screenshot of the Godot editor

This engine organizes its components into nodes, which are very versatile. Nodes can serve as the main scene that groups other nodes, or they can represent specific entities such as characters, images, and others. Godot allows you to add nodes through a menu that displays all available and pre-created nodes within the engine.

In Figure 4, the right panel shows all the configurable elements for each node. These elements are generally common across nodes, but there are also specific configurations depending on the node type and its particular function.

On the left side of the interface, there's a hierarchical tree showing the organization of nodes into groups. Below is the file explorer, where you organize and manage your project.
In terms of code development, this engine supports several programming languages, including C#, C++ (the language in which the engine itself is developed), and its own programming language, called GDScript, which bears a strong resemblance to Python. The code editor is integrated into the engine itself, eliminating the need for external programs for editing and developing scripts.

When starting project development in Godot, it's essential to determine the format to which the project will be exported. Godot offers three main options:

1. Forward+: This option is designed for games that require high-quality graphics and larger scale, primarily targeting PCs and consoles. It's ideal for projects that require high graphics performance.
2. Mobile: This option is designed for less graphically demanding games, making it easier to export mobile devices, as well as computers and consoles. It's a good choice for developers who want to reach a wider audience on mobile platforms without compromising quality.
3. Compatibility: This option is limited to less demanding graphics, but is compatible with a wider range of devices, including computers, consoles, mobile devices, and the Web. It offers a more versatile solution for projects that require multi-platform access.

Given the focus of this research, we decided to use Compatibility format. This will allow the project to be accessible on a wide range of devices, including the possibility of running in web format, aligning with the goals established for the game's development.

Once the project has been created and initialized in Godot, a scene is generated with a "root" node. This node acts as the starting point from which new "child" nodes can be created. As shown in Figure 4, the upper left window shows the organization of the nodes that make up the level.

When creating a character, it's a good idea to start with a new scene specifically for managing all the elements and changes associated with the character. This allows for cleaner organization and makes it easier to reuse the character in different parts of the game. To achieve this, create a new scene with the root node of type*CharacterBody2D*This node is ideal for playable characters, as it allows you to adjust essential properties such as gravity and collisions that will affect the character.

To manage the character animations, a node is added*AnimatedSprite2D*as a child of the character's root node. This node is responsible for handling all of the character's animations. It allows you to load different image sequences, splitting them into the number of frames corresponding to each animation. The animations will be executed based on specific calls within the *script* that controls the character's behavior.

Within the main scene, Godot provides a node that encapsulates the tool *Tilemap*, which allows you to use this tool to build the game environment efficiently. To use the *Tilemap*, it is necessary to create a resource *Tile* and divide the *sprite* of the terrain to be used, specifying the dimensions of the cubes into which it will be divided. This facilitates stage construction, as it allows multiple cubes to be placed to form the level.

In this particular case, collisions are not automatically added to the cubes in the *Tilemap*. Therefore, it is necessary to incorporate a node*CollisionShape2D*. This node allows you to create square-shaped collisions, adjusting their size as needed. To better manage collisions, you can create a node group. This involves using an empty node that acts as a container, grouping multiple collision nodes for all the platforms present in the level. Likewise, within the character scene, a node is added. *CollisionShape2D*to define the character's specific collision area.

Regarding user interactions in Godot, it's necessary to configure the keys to be used in the project. This is done through the project options, specifically in a tab called "Input Map." In this section, you must assign a name to each action or effect you intend to implement and, subsequently, associate the keys that will activate that action. For example, to allow movement to the right, you could create an action called "Right" and assign the 'D' and right arrow keys to execute it.

Once all the actions have been assigned in the Entry Map, we proceed to create a *script* associated with the character node. This *script* will contain all the functions necessary to handle movement based on keyboard input and will also control the animations that will be executed based on the character's movement. This approach allows animations to automatically change based on the player's actions, providing visual feedback consistent with the movements.

After setting up and to test the *script*, you can instantiate the character in the main scene, ensuring that all elements are properly linked and working together.

Another fundamental aspect of level design is having the camera follow the character. Thanks to Godot's hierarchical node structure, this can be easily achieved by dragging the camera node into the character instance node. Doing so will automatically follow the character's movements, keeping them centered on the screen.

To add a wallpaper that stays aligned with the user's view, Godot provides specific nodes such as *Parallax Background*. This node allows you to configure a background that moves at a different speed than the foreground, creating a parallax effect. When using *Parallax Background*, ensures that the background is always present and visible to the user, enriching the visual experience of the game.

In Godot, the user interface (UI) is organized using specific nodes. To manage the layout of elements in the UI, nodes such as *Margin Container*, which determine the relative position of the elements with respect to the screen margins. Within this node, other nodes can be added such as *VBoxContainer*, which allow you to organize UI elements in a vertical list format. Finally, to display text, the node is used *Label*, which is inserted into containers and allows you to add the desired text to be displayed on the screen.

With all these elements implemented, the level's development is complete. It can be said that Godot is an alternative that offers multiple tools, streamlining the developer's experience and allowing for much more agile work.

It is important to note that Godot has official documentation called *Godot Docs*, which is a valuable resource for developers. In addition, the Godot user community actively contributes to the development of the engine, allowing users to modify and add their own ideas. Although the engine's resource store, known as *AssetLib*, is not as popular as other engines, it offers a variety of resources that can be downloaded for free or purchased, all of which are shared by the community.

Godot is compatible with multiple operating systems, as it can be installed on Windows, macOS, and Linux computers. It also has an app for Android mobile devices, allowing developers to work directly from their devices. It's also possible to run a version of the engine directly from a web browser, further expanding its usability.

While the node structure proposed by Godot can be seen as a limitation when creating games that require specific elements that are not available, this same structure facilitates access and learning of the engine, making it more intuitive for new users.

On the other hand, the engine's own programming language, GDScript, is very similar to Python. While this similarity can be an advantage, it's necessary to learn the details and specifics of the language to use it correctly within the engine.

Regarding project export, since it's designed for web format, Godot generates a series of files that include an HTML header, JavaScript files, and several compressed packages containing the engine's resources. Once compiled, these allow the game to run directly from the web.

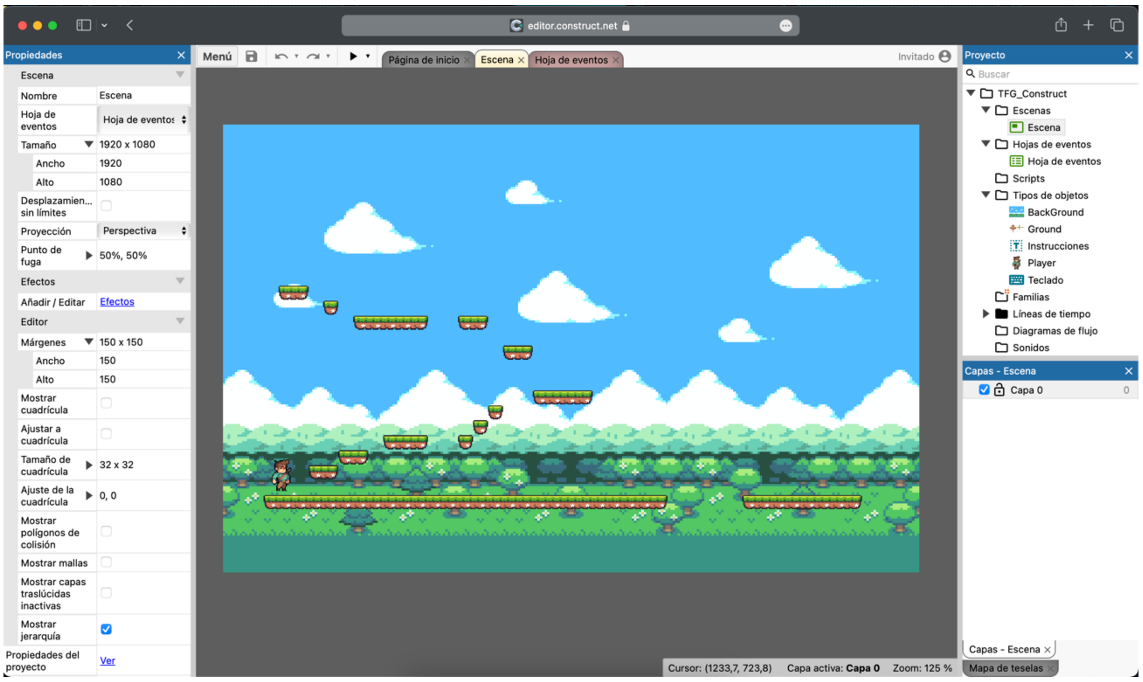
In conclusion, despite its limitations, Godot is a highly capable engine to compete with other engines in the industry. Its free licensing model, which imposes no costs for publishing games developed with this engine, makes it a very attractive option for independent developers and small businesses.

Construct 3

This engine is characterized by its accessibility, as it requires no downloads or programming. It runs directly from its website, allowing it to be used from any location or device with an internet connection.

The engine starts with a project selector, which allows you to create new projects with the necessary specifications. A notable feature is the ability to save projects to cloud storage services such as Google Drive, OneDrive, or Dropbox. This facilitates automatic saving and ensures that the project is available anytime, anywhere. Additionally, Construct saves the entire project in a compressed file with its own extension (c3p).

The engine's interface isn't very configurable, beyond the ability to move the editor windows. However, it has several tabs that provide access to all the elements necessary for development. The engine editor can be seen in Figure 5.



Escena	
Nombre	Escena
Hoja de eventos	Hoja de eventos
Tamaño	1920 x 1080
Ancho	1920
Alto	1080
Desplazamien... sin límites	☐
Proyección	Perspectiva
Punto de fuga	50%, 50%
Efectos	
Añadir / Editar	Efectos
Editor	
Márgenes	150 x 150
Ancho	150
Alto	150
Mostrar cuadrícula	☐
Ajustar a cuadrícula	☐
Tamaño de cuadrícula	32 x 32
Ajuste de la cuadrícula	0, 0
Mostrar polígonos de colisión	☐
Mostrar mallas	☐
Mostrar capas traslúcidas inactivas	☐
Mostrar jerarquía	✓
Propiedades del proyecto	Ver

Cursor: (1233,7, 723,8) Capa activa: Capa 0 Zoom: 125 %

Figure 5 - Screenshot of the Construct 3 editor

The engine is organized with multiple options for developing a variety of games. However, it's important to mention that you can only use the tools and components provided by the engine itself. Although it offers a fairly extensive catalog of options, it doesn't allow for the integration of external or custom components.

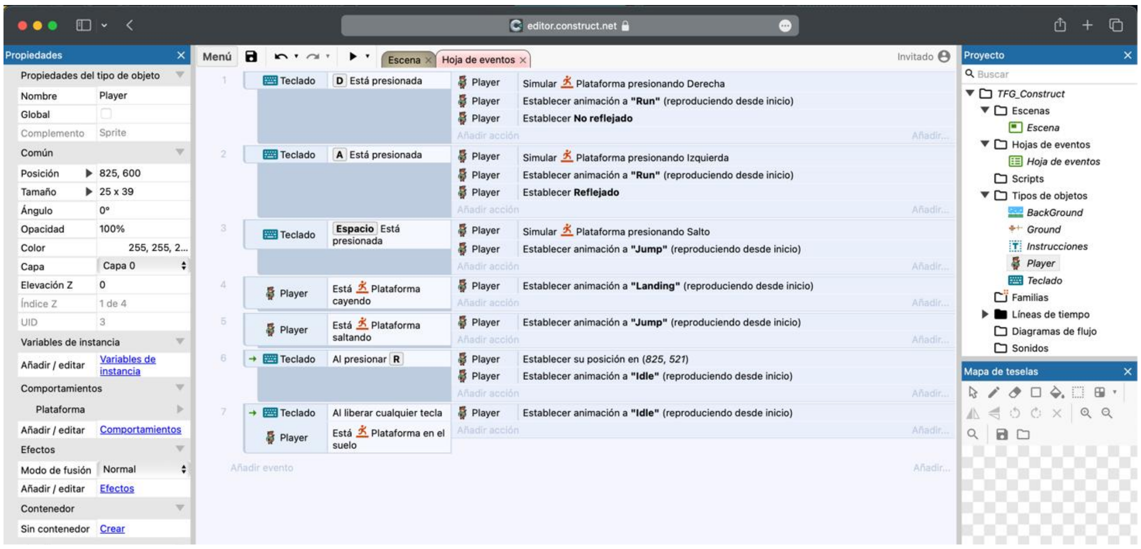
As shown in Figure 5, the left panel of the editor displays all the options and modifications available for each selected object. The right panel displays all the objects added to the project. At the top, there are a series of tabs that represent the current scene and provide access to other features, such as returning to the home screen or accessing the Event Sheet, where all the instructions programmed by the developer are stored.
Once the project has been created and the screen size ratio has been configured, the next step in this engine is to create the character. To do this, you need to add an object called *Sprite*. *When* instantiating this object, it's essential to assign it a specific behavior, depending on the role it will play in the game. In this case, since it's a character on a 2D map with lateral movement, it's necessary to configure the Platform behavior within the sprite's options. Doing so gives the object a series of properties specific to this type of game, such as maximum speed, acceleration, the ability to double jump, and more. All of these options become preconfigured in the engine, and the user only needs to adjust them according to their game's needs.

Additionally, when instantiating the character object (*sprite*), a small editor opens that allows you to add *sprite sheets* for animations. This editor makes it easier to manage character animations, as the character object itself is capable of dividing and storing its different animations.

For the creation of the floor and platforms, another tool called is used *Tilemap* This editor allows you to design the terrain on which the character will move. As with the character, you must select the Tile Map option within the engine options. Once the image to be used as a texture has been added, the level is designed within the scenario by adding the necessary blocks using the "paint" tool provided by the editor. As with the character, you can assign a specific behavior called Solid, which gives all added blocks collision protection and prevents them from being affected by gravity.

Thanks to the engine's predefined behaviors, both collisions and gravity of the character and environment are automatically configured.

To manage user input, character movement, and changing animations based on their movement, Construct offers a tool called the Event Sheet. This sheet is based on a series of conditionals that allow for the management of elements added to the project. Figure 6 shows the events recorded for this level.



Propiedades

Propiedades del tipo de objeto	
Nombre	Player
Global	
Complemento	Sprite
Común	
Posición	825, 600
Tamaño	25 x 39
Ángulo	0°
Opacidad	100%
Color	255, 255, 2...
Capa	Capa 0
Elevación Z	0
Índice Z	1 de 4
UID	3
Variables de instancia	
Añadir / editar	Variables de instancia
Comportamientos	
Plataforma	
Añadir / editar	Comportamientos
Efectos	
Modo de fusión	Normal
Añadir / editar	Efectos
Contenedor	
Sin contenedor	Crear

	Condiciones	Acciones
1	Teclado D Está presionada	Player Simular Plataforma presionando Derecha Player Establecer animación a "Run" (reproduciendo desde inicio) Player Establecer No reflejado
2	Teclado A Está presionada	Player Simular Plataforma presionando Izquierda Player Establecer animación a "Run" (reproduciendo desde inicio) Player Establecer Reflejado
3	Teclado Espacio Está presionada	Player Simular Plataforma presionando Salto Player Establecer animación a "Jump" (reproduciendo desde inicio)
4	Player Está Plataforma cayendo	Player Establecer animación a "Landing" (reproduciendo desde inicio)
5	Player Está Plataforma saltando	Player Establecer animación a "Jump" (reproduciendo desde inicio)
6	Teclado Al presionar R	Player Establecer su posición en (825, 521) Player Establecer animación a "Idle" (reproduciendo desde inicio)
7	Teclado Al liberar cualquier tecla Player Está Plataforma en el suelo	Player Establecer animación a "Idle" (reproduciendo desde inicio)

Figure 6 - Capture Construct 3 Event Sheet

As for camera tracking, Construct offers a basic functionality called *Move to* This behavior allows the camera to automatically focus on a specific object, such as the character, ensuring that the player always has it in the center of the screen.

On the other hand, for background management, Construct allows for the generation of backgrounds using tiles, creating an infinitely repeating effect that is useful for long or endless levels. However, because these backgrounds are static, it's not possible to animate them or interact with them in any advanced way. The background image must be placed at the end of the level, limiting its dynamic nature.

For inserting information into the user interface, Construct provides an object called *Text*. This object allows you to add and display text directly in the interface, without any additional configuration, making it easy to create a *UI* or informative menus quickly and easily.

Having completed level development, it can be concluded that Construct, although limited in terms of creative freedom, is an excellent tool for developing small games, especially for those without programming experience. The lack of need to write code is a significant advantage for beginners, although the engine also allows the use of JavaScript scripts for those who prefer to further customize their project. At this level, it was not necessary to use custom scripts.

Construct has official documentation, called *Manual & Documentation*, available on its official website. Although its user community is relatively small, it's active, making it easy to find answers to questions the documentation can't resolve.

In terms of compatibility, Construct runs directly in the browser, meaning it can be used on any device with internet access. However, Construct is primarily designed for creating web-based games, making it ideal for devices with browsers, such as desktop and mobile devices. It also allows games to be exported as minigames in the form of web ads.

Although the engine is relatively easy to use once you master the available tools, its closed nature can require an initial learning curve. Once you understand its features, Construct allows for rapid and agile development.

Regarding project export, selecting the HTML5 format generates a compressed project that includes HTML and JavaScript files. These files contain the Construct runtime engine along with images and other resources uploaded by the user.

In short, Construct is an excellent choice for inexperienced developers looking for an accessible and easy-to-use tool that can be run from anywhere. However, the fact that a paid license is required to remove the engine watermark or unlock more features may limit its appeal to those looking for a completely free solution.

Phaser Editor (Phaser.js)

Phaser is a popular 2D game development engine that offers two main ways to develop a project. The first is through direct use of the Phaser.js library, which can be downloaded from the official Phaser website. This library can be used in any code editor, such as Visual Studio Code, and allows developers to write game code manually.

On the other hand, there is a more visual tool called Phaser Editor. This development environment was created to make Phaser easier to use, allowing developers to build games in a more graphical and intuitive environment. Phaser Editor is currently in its fourth version and is designed to offer a more accessible development experience, especially for those who prefer a visual interface to working directly with code.

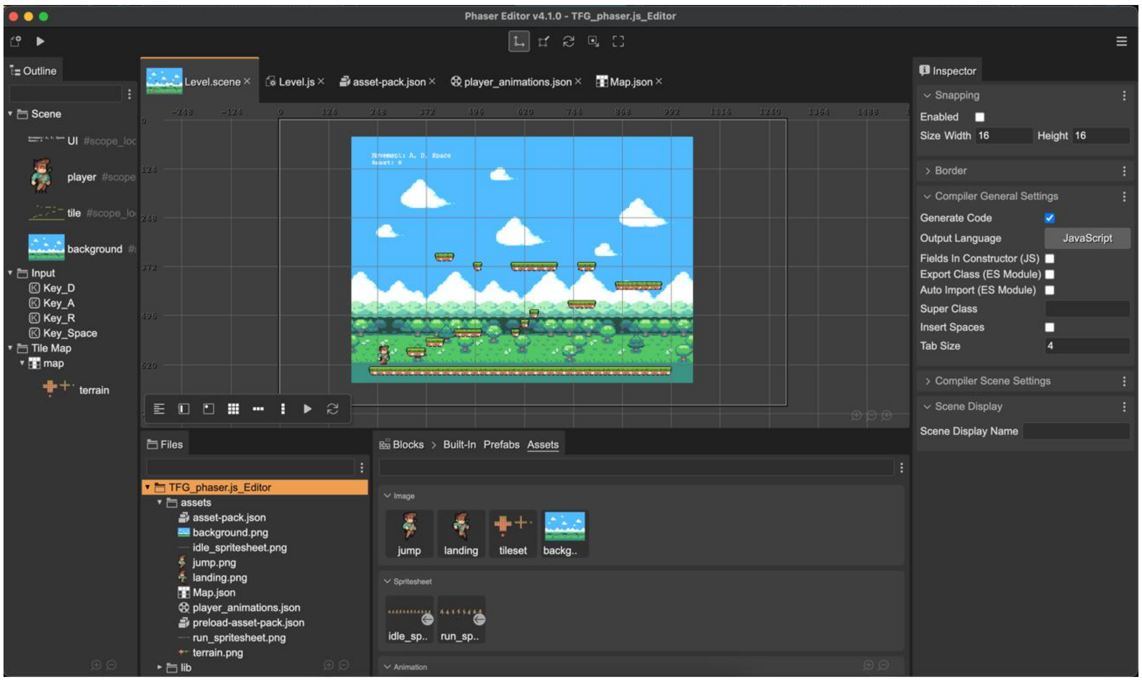
When you open Phaser Editor, the first thing that appears is a list of all previously created projects, along with the option to create new ones. When creating a new project, the developer has several options:

- Completely empty project: Allows you to start from scratch without any default settings.
- Project with starting bases: Offers basic templates that provide an initial structure.
- Complete examples: Projects pre-developed by the Phaser team that serve as a reference or starting point.
- Empty Phaser projects with structure for *Frameworks*: Empty projects but preconfigured to be compatible with *frameworks* external ones such as Angular, React, among others.

Once the project type is selected, the editor displays its interface. This editor is not highly customizable, although it allows you to move some windows. The interface is organized as follows:

- Left side: The elements that make up the currently open scene are displayed.
- Bottom: It's divided into two sections: on the left, a complete view of the project's folder structure, and on the right, the items imported and created within the project.
- Right side: The properties and characteristics of the elements selected within the editor are presented here.
- Top: There are tabs that represent different groups of objects or open files, such as the level scene and the *script* of the corresponding JavaScript. Importantly, Phaser Editor features an integrated text editor, allowing developers to write and edit scripts directly from the interface.

All the detailed elements can be seen in Figure 7:



Enabled
Size Width 16 Height 16
Generate Code ✓
Output Language JavaScript
Fields In Constructor (JS)
Export Class (ES Module)
Auto Import (ES Module)
Super Class
Insert Spaces
Tab Size 4
Scene Display Name

Figure 7 - Screenshot of the Phaser Editor

Once the project is created, the first step is to develop the character that will be used. In this case, you can use a *sprite* or, for this specific level, a specific type called *Arcade Sprite*, which already has built-in features like gravity and collisions.

When importing art assets into your project, you need to add them to a specific file before they can be used. These files are called asset-pack. json and can be imported as images or as *sprite sheets*. Then, you need to create an object that encapsulates all the animations called *Animation File*, which allows you to combine different frames and create animations from them *sprite sheets* already imported. This *Animation File* is also stored inside the package of *assets* mentioned above.

Once the animations are finished, they can be called up to be played in the *sprite* of the character through a *script*.

For the creation of the floor and platforms, this editor does not have an integrated tool capable of doing so, but it allows you to add maps created with a structure of *Tilemap*. The recommended tool in the documentation for this is Tiled. Tiled allows you to create maps with the distribution structure of the *Tilemaps*; that is, an image is divided into cubes, and with these, the level can be "painted" based on these cropped images. Once the map is finished, the Phaser editor only supports a JSON-exported version of the entire level.

Once the map is finished in Tiled, it is added to the project, saved in the asset-pack. json and an object called is instantiated *Tile Map* which allows the map to be displayed.

In this project, by having gravity already present in the *sprite* of the character, it is only necessary to add the collisions with the level. With a few lines of code in the *script* of the level you can manage the collision between the *sprite* of the character and the platforms of the *Tilemap*.

Within the Phaser editor, you can configure the keys to which the Phaser *script* must react. Once configured in the editor, they are instantiated within the *script* of the level and the code that allows both movement and changing of character animations is developed.

In this editor, there is no option to modify the level's camera or background in any way other than through the level's code.

Finally, to add text to the user interface (UI), the editor offers a very simple object called *Text*, which allows you to write text directly from the editor.

This concludes the development of the level in this editor. We can emphasize that the idea of this JavaScript-based engine is innovative, but it still seems underdeveloped and needs more features to be truly useful.

The official Phaser.js documentation is quite extensive, and the developers' website includes multiple code examples, with the ability to view the code's execution on the same website. However, the Phaser Editor documentation is limited and chaotic, attempting to explain all of the editor's features without clearly detailing its uses. The community also reflects this disparity; there is much more activity around the library than the editor.

The Phaser library is compatible with any browser capable of running it, while the editor is only compatible with Windows, macOS, and Linux. As for exporting, Phaser is completely web oriented.

It's important to note that this editor has a steeper learning curve than other engines. You'll need to know JavaScript, understand the applications Phaser.js offers, and understand the editor's tools to use it effectively.

Furthermore, this editor is not free; a monthly subscription is required to download and use it.

Finally, as for the exported project, the editor maintains the structure of a Phaser.js project, with HTML and JavaScript files that the editor modifies based on the changes made.

In conclusion, this editor is not recommended for developing a project, unless you specifically need to use the Phaser.js library. The added cost, coupled with the lack of tools and documentation, can significantly hamper project development.

Defold

This engine has gained popularity in recent years thanks to its full release and the licensing change that has made it completely free. It is primarily designed for developing games requiring high-quality graphics, although it also supports web formats, allowing it to adapt to different scenarios.

Upon launching the engine, a menu appears with all the projects created by the developer and the option to create new projects. For new projects, you can choose to create an empty project, one already prepared for the platform you plan to port your game too. Predefined examples and more advanced templates highlighting engine-specific tools are also included.

The Defold engine is shown in Figure 8 below:

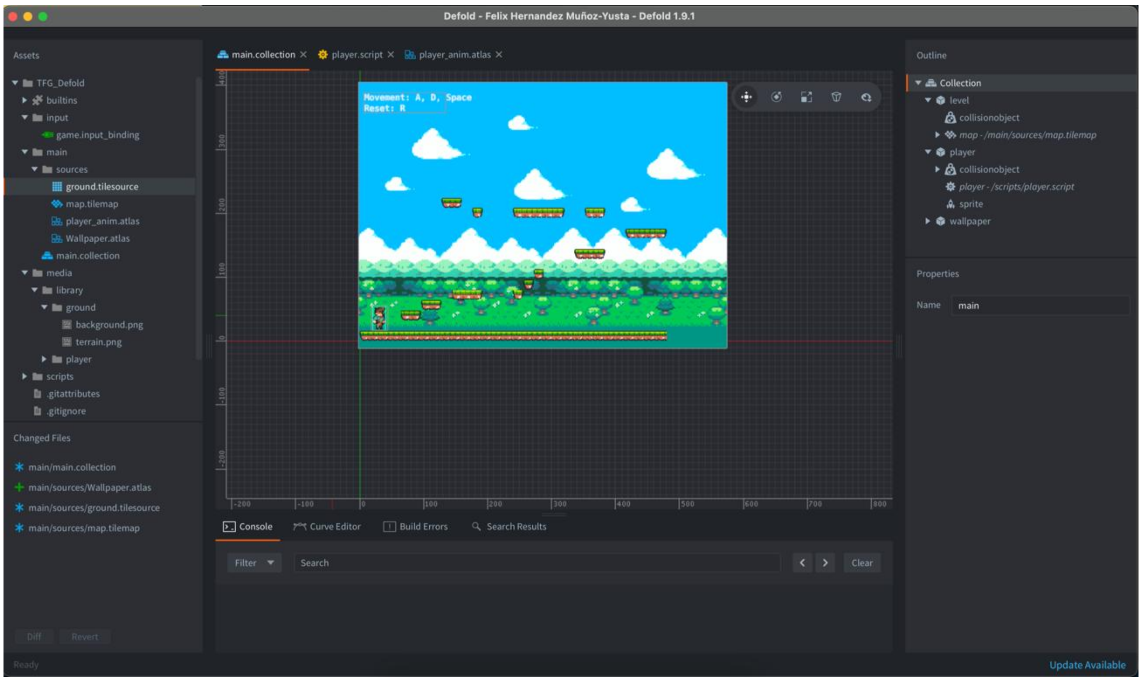


Figure 8 - Screenshot of the Defold editor

Although the interface is not configurable, it is intuitive and easy to understand. The interface is organized as follows: on the left side, the project folder structure is displayed, and below it, the changes made sense the last save in Git. At the bottom, there are various support tools within the engine. On the right, the level structure is displayed, or if a different object is selected, its characteristics and options are shown. At the top, the various engine objects that can be edited are listed.

The structure of this engine is organized around its various tools, encapsulated as objects within the engine. For example, one tab might represent the scene, another the project settings, a set of images (*atlas*), the tool to create terrain with *Tilemap*, among others.

Once you've created the project for the platform you want to export to, an interface appears with multiple elements already configured. The scenes within this engine are called *collections* and are responsible for grouping all the necessary elements.

The first object to be added is the character. To incorporate it into the scene, call *collection* in the engine, a is created *GameObject* empty, and within this the component corresponding to the character is added, which, at first, is nothing more than an empty component.

In this engine, the *sprite sheets* of the animations can be included within an object called *atlas*, which allows you to store all the frames of the different animations and group them according to their corresponding animation. Once the animations have been created and organized, they can be linked to the character component. To switch between animations, you can use a *script*.

When creating the level platforms, you can use the tool *Tilemaps* which includes the engine, as it allows you to create platforms efficiently. Through a *tile source*, the image from which the level will be designed is added, which is then saved in a file called *Tilemap* inside the engine.

With the platforms and the character already instantiated in the scene (*collection*), it is necessary to add collisions and gravity. To do this, a component called *Sprite* allows you to apply physical properties to the character's body. As for collisions, you need to include a component called *CollisionObject*, which allows you to add collisions using basic geometric shapes such as cubes, spheres or cylinders (for 3D environments).

When it comes to character movement, you must first configure the keys that will be used in the level. This is done in a default project file, called *game_input_binding*. In this file, you need to specify the keys the engine should detect and assign them a name for the corresponding action. Once configured, you need to create a script (written in Lua) that will manage everything related to the character's movement and the switching of animations based on the movement performed.

To make the camera follow the character, it is necessary, through a *script*, matching the character and camera coordinates along the x and y axes, with a difference that can be adjusted by the user. Similarly, if you want the level background to maintain a constant alignment with the camera plane, the background image must follow the camera coordinates.

Finally, to add text to the user's canvas (*UI*), a component called must be added *Label* within a *GameObject* empty or pre-existing, preferably the same one where the level's background image is located. Once this component is implemented, you can add the desired text to the interface.

In conclusion, developing a level in Defold is an accessible process. While the engine doesn't allow for great freedom in component modification, it offers enough tools to provide adequate and flexible control, even with its limitations.

Defold has official documentation accessible on its website, which makes it easy to understand the different tools and the engine's structure. However, because it's a relatively new engine, its community is still under development.

In terms of compatibility, Defold is capable of running on operating systems such as Windows, macOS, and Linux. One of its main strengths is the ability to export projects to various platforms, from next-generation consoles to online games on platforms like Facebook.

While it is necessary to learn both the engine and the Lua programming language, the documentation and ease of use of the tools make it quick and accessible.

In the context of this research, once the project is exported in web format, it is configured as a folder containing an HTML file and the necessary libraries in JavaScript format. Additionally, the required resources are presented in compressed files, ready to be used by the browser when the game is launched.

In conclusion, Defold is a viable option for video game development, especially notable for its cross-platform capabilities, which is very attractive to developers looking to cover multiple gaming platforms. However, the learning curve, although present, can be hindered by the limited community, which could make it difficult to resolve questions not covered in the documentation.

Programming Roles, Analysis and Task Relationship

Programming roles

Regarding programming roles, since this is an individual investigation, all aspects related to research, development, and comparison have been carried out entirely by the author of this Final Degree Project.

Analysis

Within the analysis, it's important to note that to view the developed games, you must run the compiled projects in a compatible browser, such as Microsoft Edge, Google Chrome, Safari, or Firefox. That is, any browser that supports HTML5, as this is the standard required for running the projects. For this, it's best to use a computer.

If you want to open projects in development engines, you must install them in compatible environments. Desktop operating systems such as Windows, macOS, or Linux is best. This applies to engines such as Unity, Godot, Phaser, and Defold. For Construct, simply access the editor directly from a supported web browser, such as Safari, Chrome, or Firefox.

As for the necessary peripherals, a computer keyboard is sufficient to run the developed levels, and to work on the engines, the use of a keyboard and mouse is sufficient.

List of tasks

Regarding the tasks carried out by the user, it can be concluded that they have been organized and managed effectively.

Regarding the game's development, the tasks were organized to allow for simultaneous development across all engines in parallel. To achieve this, the work was divided into numbered blocks as follows:

1. Create a project in the engine.
2. Create a character.
3. Add animations to the character.
4. Create the floor and platforms.
5. Apply gravity.
6. Add collisions between the ground and the character.
7. Add movement inputs to the character.
8. Make the camera follow the character.
9. Add a background to the level.
10. Add text to the canvas / UI (user interface).

Since this project was conceived as a research project, it was necessary to document all stages of development. Therefore, the entire development process for the different engines is detailed in the section. Project Development and subsequently specified in each engine analyzed.

As for the research tasks carried out, the study of the engines and their components has been fundamental, that is, the tools used in each engine, the language of *scripting* specific to each one, as well as the requirements necessary to export the project.

It is important to note that all engines have required the use of tools such as *Tilemaps*, which have been used to create the 2D maps by using a resource image (*Tile*), allowing you to design the level and subsequently implement it in the scene.

In addition, it has been necessary to investigate the different languages of *scripting* used in each engine. In the case of Unity, we delved into the C# language; for Godot, we studied its native language, GDScript; in Construct, we didn't need to learn a programming language as such, but we did need to understand the logic of "Event Sheets"; for Phaser, we worked with JavaScript; and finally, in Defold, we needed to learn Lua.

In handling the engines, it has been essential to learn their different configurations and structures, since, in engines such as Unity, Defold or Phaser, the objects may appear similar under the name of *GameObjects* However, in Godot this structure changes, representing objects as nodes.

Another relevant aspect is that each engine offers its own methods for importing, modifying, and displaying images. Each has its own way of managing graphic elements. On the one hand, some engines, such as Unity, Godot, and Construct, allow you to automatically divide the *sprite sheets* within the same engine. On the other hand, in engines such as Phaser and Defold, it is necessary to manually separate and modify the *sprite sheets* into individual images using external editors.

Results and conclusions

Results

The research focused on the analysis and comparison of five 2D game engines designed for web development: Unity, Godot, Construct 3, Phaser.js, and Defold. The results are divided into two sections:

Technical results:

- Web Compatibility: All selected engines meet the export requirements for web games. Each engine is capable of generating HTML and JavaScript files compatible with major browsers (Chrome, Firefox, Safari, etc.).

- Support platforms: Unity and Defold stand out for being multiplatform engines, allowing export not only to web formats, but also to consoles and mobile devices. Construction 3 is the most accessible in terms of use, as it doesn't require installation.

- Programming Language: The learning curve varies considerably between engines. Unity uses C#, Godot uses GDScript (its own language), while Construct 3 simplifies programming using event sheets. Phaser.js is based on JavaScript, and Defold uses Lua.

- Documentation and Support: Unity and Godot stand out for their extensive documentation and large support communities, which makes troubleshooting easier. Phaser.js also has good documentation, although its graphical editor (Phaser Editor) has some limitations. Construct 3 and Defold, while having smaller communities, have active users.

Video game development:

- Ease of use: Construct 3 is the most accessible engine for novice developers, as it doesn't require any programming knowledge. Unity and Godot, while requiring more preparation, offer greater flexibility and power. Defold and Phaser.js, on the other hand, offer more granular control but require greater familiarity with their respective languages.

- Graphical tools: Unity offers greater automation, such as automatic detection of *sprite sheets*, while Phaser.js and Defold require more manual intervention. The inclusion of a system of *Tilemaps* in several engines (Unity, Godot, Construct and Defold) made it easier to create 2D scenarios.

Conclusions

The main objective of this project was to research, compare, and evaluate web-compatible 2D video game development engines, which was successfully accomplished. Through the development of a small game in five engines (Unity, Godot, Construct 3, Phaser Editor, and Defold), the strengths and weaknesses of each were identified, allowing us to determine which is most suitable for the needs of the project and the developer.

Aspects such as ease of use, learning curve, and the ability to export to the web were evaluated. The results indicated that Godot and Defold offer flexibility without cost, while Construct 3 is simple but limited in customization. Unity and Phaser Editor proved to be powerful, although with greater complexity or cost.

In addition, technical knowledge of web engines and technologies was acquired, and the project provided experience in project management and comparative analysis. The initial objectives were fully achieved, and future plans include researching new engines or exploring multiplayer capabilities.

In conclusion, although all the engines evaluated are viable for video game development, Unity remains the most recommended option, even for web development. This is because it offers the greatest number of benefits for developers, both during the development process and in the subsequent stages.

Postmortem of development and future lines

In this individual investigation, it would have been beneficial to develop more complex levels with greater resources to push the development engines to the limits of their capabilities. This would have allowed us to evaluate how they manage texture compression and multiple event handling when compiling projects for web-based formats.

Furthermore, it was not possible to compare the resource consumption and performance of the compiled projects when they were published on an open-access website.

In future research, it would be interesting to include an analysis of development costs, both for companies and independent developers.

Attachments

In the folder where this memory is located there is a folder called “Programming Annexes” where you can find the developed projects of the various engines, for example “Project_Unity” and then the final compiled game called for example “Game_Unity”.

It is important to highlight the fact that the games developed are focused on the web format, that is, they are not an executable; it is necessary to open the file with HTML extension in the browser within the folder without modifying anything inside the folder once unzipped.

Literature

Gregory, J. (2014). *Game Engine Architecture* (Vol. 2nd Edition). Florida, USA: CRC Press.

Makzan. (2011). *HTML5 Game Development by Example: Beginner's Guide.*Birmingham, UK: Packt Publishing.

MDN contributors. (July 24, 2023). *JavaScript*Retrieved from MDN web docs: https://developer.mozilla.org/en-US/docs/Web/JavaScript

MDN contributors. (August 18, 2023). *WebGL*Retrieved from MDN web docs: https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API/Tutorial

MDN contributors. (2024 June 2024). *CSS*Retrieved from MDN web docs: https://developer.mozilla.org/en-US/docs/Web/CSS

MDN contributors. (July 28, 2024). *HTML*Retrieved from MDN web docs: https://developer.mozilla.org/en-US/docs/Web/HTML