# Porting a Unity 3D application into the Godot Engine

Bruno Lönne
*TU Berlin*
Berlin, Germany
loenne@campus.tu-berlin.de

Eloi Sandt
*TU Berlin*
Berlin, Germany
eloi.sandt@campus.tu-berlin.de

Benedikt Utsch
*TU Berlin*
Berlin, Germany
b.utsch@campus.tu-berlin.de

***Abstract*—Due to sudden license changes unfavourable to consumers, the popular game engine Unity has painted itself in a poor light. This raises the concern of finding a suitable alternative game engine operating under a safer license. Godot may be such an engine but many developers may already have invested a lot of resources into their Unity projects, fearing the amount of effort needed to port their applications to another engine.**

**This paper aims to shed light into what porting a Unity 3D application into the Godot engine might entail. It documents the porting process of a 3D game, shares tools that were helpful and highlights encountered problems. Finally, it discusses these aspects under the circumstance of working as a team with version control.**

**We believe that Godot could be a suitable alternative to Unity. In the course of this project, a 3D game was ported to Godot, resembling the original Unity game to a fair degree. Nevertheless, we found it important to understand that porting should not be approached with the expectation of achieving an exact replication of the original Unity application.**

## I. Introduction

Unity is a popular game engine, offering many resources and functionalities that can be used to develop video games as well as interactive simulations and other experiences. [3, p. 162] The engine can deploy applications to a variety of platforms, including web, mobile, and desktop devices. [5, p. 18] Both free and paid licenses are available. [5, p. 24]

For the beginning of 2024, the company behind Unity tried to change the Unity license, charging developers for up to $0.20 for every install of a game on a client device. [1] These plans were since then partially retracted after public outcry among Unity developers. Nevertheless, this has raised the concern of using a game engine with a safer license.

A popular alternative to Unity is the Godot game engine[2]. It tries to offer a similar amount of features while being free and open source. Godot offers programming language support for its native language GDScript, C/C++ and C#[3]. Since Unity applications are also mainly written in C#, this makes Godot an attractive engine to switch to.

This paper explores the prospect of porting an existing Unity 3D application to Godot in order to prevent any current or future licensing issues. It describes and evaluates the porting workflow, highlighting tools that were useful and problems that presented themselves during the porting process while comparing the capabilities of both engines.

## II. Related Work

As the issue at hand has only recently attracted a lot of attention, there does not exist much related work on the subject yet. However, Dr. Bryan Duggan from the TU Dublin details his experience in porting a game AI course from Unity to Godot in [4].

Furthermore, some game developers have documented the process of porting their games from Unity to Godot online (see for instance [4]).

## III. Approach

Firstly, this section begins with how a suitable 3D application was selected and which Godot version was chosen to port the application to. Then, it proceeds by looking at how the application was ported. This includes porting scenes, assets and the code base.

### A. Choosing a 3D Application to Port

We chose the Karting Microgame Template[5] provided by Unity. It is an already working and implemented 3D racing game, utilising many aspects of Unity's game engine. This includes character models and animations, visual effects, audio, plugins (e.g. Cinemachine plugin for the camera following the player), a custom implementation of the racing car physics ("math" code) and finally menus and a user interface. A wide variety of functionalities was used, thus making the game very suitable to highlight the capabilities of Godot when porting it.

The game template already having the basic game logic implemented meant that we could focus on the porting process, not having to implement a 3D application in Unity

[1] Kyle Orland. Wait, is unity allowed to just change its fee structure like that? https://arstechnica.com/gaming/2023/09/wait-is-unity-allowed-to-just-change-its-fee-structure-like-that/, 2023. Accessed: 25 February 2024.

[2] Godot official website. https://godotengine.org/. Accessed: 25 February 2024.

[3] Godot's scripting languages. https://docs.godotengine.org/en/stable/getting_started/step_by_step/scripting_languages.html. Accessed: 26 February 2024.

[4] Gamefromscratch. Switching to godot... how it went? https://www.youtube.com/watch?v=SktBNRVUHKM. Accessed: 27 February 2024.

[5] Karting Microgame Template. https://learn.unity.com/project/karting-template. Accessed: 26 February 2024.

from scratch. Nevertheless, we implemented some additional functionality to the game to get accustomed to the sample project in Unity. Firstly, we altered the game mode to count laps and the time needed for each lap. We also added speed boost power-ups to make the game more fun and complete. Furthermore, we modified the race track by adding slopes, roads with ridged surfaces and a free falling section. This proved to be fairly useful for testing the ported game in Godot later, as it allowed us to identify differences in how the physics of the kart behaved.

It was stipulated that the game should run in the web browser as a means of providing an accessible avenue of comparison between the original[6] and the ported game[7]. The scope of the modified karting template is small enough to be performant in a web browser while still implementing a diverse assortment of functions and game logic.

### B. Choosing Godot 3 versus Godot 4

The second step was choosing a befitting Godot version to port the Unity game to.

There effectively exist two major Godot versions, Godot 3 and Godot 4[8]. At the time of beginning to work on the project, Godot 4 was still in its infancy, with a stable version of Godot 4.1.3 just released. We anticipated Godot 4 having less fleshed out features and less user-generated content (such as helpful plugins and learning resources) than Godot 3. Since it was our first time using Godot, we preferred Godot 3 for this reason.

On the other hand, more porting tools (e.g. for asset translation from Unity to Godot) seemed to be developed for Godot 4 (see for example [9] and [10]). Section III-D1 will detail a similar tool we found and used for Godot 3 (with some drawbacks).

In the end, the requisite of the ported application having to run in a web browser alleviated the decision. Godot 4.1's documentation [2] states several restrictive issues concerning web support. For instance, "[p]rojects written in C# using Godot 4 currently cannot be exported to the web". This posed a problem for us, as we aimed to use C# as the main scripting language when porting to Godot (see our reasoning in section III-E). Furthermore, the documentation says that "Godot 4's HTML5 exports currently cannot run on macOS", which was not ideal since one member of our team used macOS. We experienced further problems when trying to deploy test applications to the web in Godot 4.1. Firstly, there is an overhead setting up a hosting server because additional headers have to be served for an application to run without errors. Secondly, applications are rendered using the Compatibility rendering method which lacked support for several visual effects (e.g. volumetric fog), causing graphical errors.

In contrast, we could host Godot 3.5 games with a simple python http server (`python3 -m http.server`) without the above mentioned Godot 4 issues. This conclusively lead us to choose Godot 3.5 for this project.

### C. Project Setup

Using C# in Godot requires the .NET version of the engine, which can be downloaded from the official website. The documentation provides a more in-depth guide [1] on how to set up the engine with C# support. C# scripts have to be created via the Godot editor, otherwise an application using them will not work correctly.

### D. Porting Scenes

A Unity game usually consists of multiple scenes, each incorporating game objects, scripts and assets such as for example character models and audio files. This subsection details how the visual parts of the scenes of the racing game can be ported to Godot (i.e. the race track, user interface elements, etc.).

*1) UnityGLTF Asset Translation Tool:* Unity and Godot use different formats for storing data. This means that Unity assets have to be translated to the correct format in order to be able use them to recreate a game in Godot. Note that assets from the Unity Asset Store may be restricted to non-commercial use outside of Unity due to licensing.

Fortunately, automatic asset translation tools exist. For our project, we used a tool called "UnityGLTF"[11]. It is capable of exporting Unity prefabs, assets (e.g. 3D Meshes) or even entire scenes into the GLTF format at once, which can be imported in Godot. That being said, the tool only supports some data types and does not (correctly) port scripts and engine specific functionality such as physics, particle systems, user interfaces, etc..

We used the tool to port the whole race scene at once. While the positions of the objects in the scene are correctly ported and retained, there is a significant drawback to porting whole scenes. All objects in the now ported Godot scene are not instances of prefabs anymore. This means for example that each rock of the same type (which was one prefab in Unity) is now its own object. Instead of changing a single prefab and the change broadcasting to every instantiated object of the prefab, modifications have to be applied to every object of the same type manually. In the case of the race track for the racing level, this meant reapplying collision physics to all relevant objects, as physics were not ported by the tool. Furthermore, while materials of game objects were ported to the correct format in Godot, they were removed from their respective game objects. They had to be reattached to every single game object in the scene manually to enable correct lighting.

[6] Hosted Unity game. https://eloinoel.github.io/UnityRacingGame/.

[7] Hosted Godot game. https://eloinoel.github.io/awt-pj-ws23-24-porting-unity-to-godot-3/.

[8] Godot engine versions. https://godotengine.org/download/archive/. Accessed: 26 February 2024.

[9] Scene translation tool. https://github.com/barcoderdev/unitypackage_godot. Accessed: 26 February 2024.

[10] Asset translation tool. https://github.com/V-Sekai/unidot_importer. Accessed: 26 February 2024.

[11] KhronosGroup. Unitygltf. https://github.com/KhronosGroup/UnityGLTF. Accessed: 27 February 2024.

One could also use the tool to only port single game objects and make prefabs. However, then one has to rebuild the scene and place the objects at their correct positions.

*2) Skybox:* The skybox (surrounding the race level) is an engine-specific functionality and was not correctly ported by the UnityGLTF tool. In Unity, a cubemap of 6 images is used to render the skybox. Godot utilises a panorama image instead. The cubemap images can be converted to a panorama image using the following tool: [12].

*3) Light Sources:* The sun and light sources of the race scene had to be redone manually. The UnityGLTF tool ported the glow effect of the sun but it did not look correct. Light sources are engine specific functionality and were not correctly ported either.

*4) Menus and User Interfaces:* Menus and user interfaces of the Unity game had to be rebuilt using Godot's node system. It is fairly intuitive to use and learn. Each individual node has editable properties and functionality. Functionality can also be added by attaching a script to a node. Together, nodes can be organised to form a tree to produce more complex behaviour (see Fig. 1).


Fig. 1. Example of a node tree

*5) Animations and Visual Effects:* Similarly to user interface elements and other engine specific components, animations and visual effects have to be rebuilt using Godot's native systems or be imported from plugins or tools such as Blender.

As this project is our first time using Godot, we were dependent on guides for how to create visual effects. A second option is to look for plugins. It is to be noted that it can be difficult finding appropriate resources in the Godot community ecosystem because the Godot version utilised (Godot 3 or 4) may differ from the one used in your project, rendering them mostly unusable.

We could recreate the exhaust fumes particle effect coming from the kart using Godot's "Particles" node to adequate similarity. Furthermore, the Unity game utilises a Trail Renderer component to depict trails behind the kart when it receives a speed boost. We did not find such built-in functionality in Godot 3.5. Instead, we discovered and used a plugin[13] for it.

Some animations in the Unity game were steered using scripts. For example, Unity's Coroutines are utilised to gradually depict the game's objectives at the beginning of the game in a scheduled, asynchronous manner. This can be done similarly in Godot, using either the `yield` keyword in GDScript or `await` in C#, coupled with signals (see Fig. 2).

```
// Unity Coroutine
yield return new WaitForSecondsRealtime(1f);
playAnimation();
```

```
// Godot C#
await ToSignal(GetTree().CreateTimer(1f), "
    timeout");
playAnimation();
```

Fig. 2. Play an animation via script after waiting 1 second

Other animations like the player tilting its head when steering to the left or right were also redone with scripts in Godot.

### *E. Porting the Code Base*

When porting code it should be considered how to minimise the number of locations in which code behaviour can diverge between engines. In the best case, the syntax, execution environments etc. for the original and ported code respectively are as similar as possible. Then it becomes most tenable to achieve an accurate and complete translation of the original code.

Godot 3.5 offers language support for C#, C++ and GDScript. As the Unity code is written in C#, the natural choice is to prefer porting from C# to C# instead of GDScript or even C++. Nevertheless, we also decided to set out and port some more isolated parts of the code base to GDScript (why it is important for the code to be isolated, is covered in section IV-2).

For C# specifically, a major goal was to ascertain whether it is possible to largely copy and paste unity C# code with minimal changes like replacing engine specific calls. However, as it turned out this generally still involves a lot of work, since many minor differences between the usage of C# in Unity and Godot exist, as is described in the following subsections.

*1) Engine Features:* Although Godot 3.5 and Unity both offer C# language support, the game engine interfaces differ.

For example, the Unity game to port uses `onEnable` and `onDisable` callbacks to execute conditional code behaviour when a game object is enabled or disabled. These callbacks do not exist in Godot. This is because in Godot 3.5, disabling and enabling nodes (game objects) is not possible. Nodes can only be instantiated, deleted or made invisible. Deleting nodes and re-instantiating them later in Godot is not the same as disabling and enabling nodes in Unity, as temporary properties are lost when deleting a node. For this reason, we built a workaround helper class (`DisabilityManager.cs` in our project files) to make disabling and enabling nodes possible while also providing the needed callbacks. The persistent DisabilityManager node stores other nodes-to-disable in a list and deletes them from the scene tree. When a node is enabled again it is removed from the list and added back to the scene

[12] danilw. Cubemap to panorama. https://danilw.github.io/GLSL-howto/cubemap_to_panorama_js/cubemap_to_panorama.html. Accessed: 27 February 2024.

[13] Oussama Boukhelf. Godot trail system. https://github.com/OBKF/Godot-Trail-System. Accessed: 27 February 2024.

tree. All original properties of the previously disabled node are retained.

Another example is that Godot's `_Ready` method for executing code at node startup replaces both of Unity's startup methods `Awake` and `Start` for MonoBehaviours. As detailed above, MonoBehaviours also possess additional disabling functionality and callbacks, which Godot's nodes do not. Unity generally appears to offer a little bit more detailed functionality. More instances of diverging functionality occurred during the porting process, requiring close attention to Unity's and Godot's documentations while porting and subsequent workarounds.

*2) Porting Math:* In this section, we will share some of the challenges we encountered while porting mathematical code and how we dealt with them.

Let us begin by considering linear interpolation. The linear interpolation method offered by the Mathf library has the exact same signature in Godot and Unity, but different behaviour (see Fig. 3).

```
// Unity Linear Interpolation
Mathf.Lerp(0.0f, 1.0f, 2.0f); // => 1.0f

// Godot Linear Interpolation
Mathf.Lerp(0.0f, 1.0f, 2.0f); // => 2.0f
```

Fig. 3. Mathf.Lerp in Unity vs. Godot

Whereas Unity constrains the output value to be between the linear segments endpoints, Godot's implementation allows extrapolation. Thus we ported Unity's `Mathf.Lerp` calls by constraining the domain of the interpolation calls blending value $\alpha$ to the interval $[0, 1]$ by using `Mathf.Clamp` and thereby preventing extrapolation (see Fig 4).

```
// Ported Linear Interpolation
float a = Mathf.Clamp(2.0f, 0.0f, 1.0f);
Mathf.Lerp(0.0f, 1.0f, a); // => 1.0f
```

Fig. 4. Ported Mathf.Lerp

More instances of diverging code behaviour emerged. This required us to pay close attention and often compare the outputs of function calls line by line in both engines.

Spherical interpolation between vectors does not work properly in Godot 3.5 and may yield errors or incorrect results in some circumstances. Thus we met the need for a workaround by transforming the vectors in question into quaternion space, performing the spherical interpolation there via the Godot `Quat.Slerp` method and mapping the result back into the vector space (see Fig 5).

*3) Porting the Kart Gameobject:* The recreation of the Unity kart in Godot proved to be a big challenge as the UnityGLTF asset porting tool only translates the kart's node structure, meshes and textures correctly. Colliders, animations, audio sources, the camera and the physics scripts controlling

```
// Spherical Interpolation Workaround
Vector3 QuatSlerp(Vector3 from, Vector3 to,
    float slerpRatio)
{
    Quat fromQuat = new Quat(from);
    Quat toQuat = new Quat(to);
    fromQuat = fromQuat.Slerp(toQuat,
    slerpRatio);
    return fromQuat.GetEuler();
}
```

Fig. 5. Spherical interpolation workaround

the behaviour of the kart had to be rebuilt from scratch. This section will focus primarily on the physics (e.g. collision) of the kart which turned out to be the greatest hurdle.

Because Godot 3.5 does not support the use of multiple collision shapes as part of the same dynamic physics object (e.g. `RigidBody`), the implementation of the Unity kart which employs multiple colliders (one for the fuselage and each wheel of the kart) cannot be recreated exactly within Godot. An exploration of possible approaches to achieve similar physics behaviour to the Unity kart yielded the following.

The first approach is to build a kart that is made up of separate physics objects (i.e. the individual wheels and the kart's main body) that are held together by joints. These physics objects are inserted into the scene separately to circumvent Godot's previously mentioned collider restriction. The kart behaviour is entirely handled via code. As a natural consequence, it is difficult to constrain the behaviour of the disjointed kart as its constituents have a high degree of freedom. In our testing it frequently happened that the Kart wheels fell off, flew off to infinity or showed other erratic behaviour. This is in part also due to the fact, that amongst the joint types offered by Godot 3.5 the only one capable of modeling the desired suspension behaviour is the `Generic6DOFJoint` which permits the joint physics objects full range of motion unless constrained and is far more generalised than is necessary for this application.

Another approach is to utilise Godot's `VehicleBody` node, which is a more specialised RigidBody with utility functionality for creating vehicles. Its movement and steering work by applying forces to the vehicle body and are inherently different from the Unity kart implementation (where properties of a Rigidbody are directly modified via script). The VehicleBody node has one collision shape for the body of the vehicle and none for the wheels. Instead of using rigid wheel colliders, invisible spring forces are applied to the kart to keep the kart afloat (push it out of "collisions") and emulate a suspension system. This means that, as opposed to the Unity kart, the wheels may sink and clip slightly into the ground. Furthermore, the kart acquires a high degree of bounciness. These issues can be mitigated with parameter tuning, but could not be solved to our complete satisfaction. The VehicleBody node also does not offer a way of setting the center of mass of the vehicle independently of the main body, which came up

because it was specifically set in the Unity kart. Because of these issues we also assessed whether the VehicleBody node could be mixed with other approaches (e.g. adding an invisible weight node under the kart to change the center of mass). However, most of these mixed approaches proved to worsen the overall driving behaviour.

Finally, another approach of realising the kart is to apply forces to an invisible rolling sphere RigidBody and follow the center of that sphere with the kart mesh. However this approach blatantly falters in the face of tilted terrain. Driving up a slope, the initial speed of the sphere decays and reverses, such that the kart is pushed back down the slope. In this case, gravity affects the kart qualitatively different than the Unity kart, such that trying to model it this way is futile.

In the end, the final version of the Godot kart is based on the VehicleBody approach. Mixing this approach with a slightly adapted version of the physics script controlling the kart in Unity yielded a result that felt closest to the Unity kart.

*4) Porting the Kart camera:* An interesting question is how plugins could be ported. The Unity game uses a Cinemachine plugin for an immersive camera following the kart. Just like any other code, plugin functionality would have to be adapted to the Godot engine.

As the Cinemachine plugin only exists in Unity, we decided to implement a custom camera imitating the effects of the cinemachine plugin. This includes following the kart while smoothing any sudden movements (interpolating to the position of the kart with an offset). Furthermore, the camera should face the forward direction of the kart. For this we only consider the kart's rotation on the y-axis. Thus, the camera will not cause a nauseating experience for the player when the kart is flipped upside down.

### F. Porting the Audio

The audio of the Unity Kart game can in large parts be ported well to Godot 3.5. Unity audio bus setups translate to Godot without any issues and global sound settings are accessible via Godot's global AudioServer class.

Audio assets (.mp3, .ogg, .wav) can be used without modification. Within scripts it is necessary to use the appropriate audio sample data type within Godot scripts. C# offers AudioStream for .wav files, AudioStreamOGGVorbis for .ogg files and AudioStreamMP3 for .mp3 files.

Unity's Audiosources can be replaced with Godot's AudioStreamPlayer for global and AudioStreamPlayer3D for spatial sound emission. For spatial sound Unity offers the feature of adjustable audio attenuation curves that is missing from Godot 3.5. Godot 3.5 merely provides three standard audio attenuation models (logarithmic, inverse distance, inverse square distance), but the more specific curves, that are used in the Unity Kart game can not be perfectly translated without calculating attenuation manually in code and raising and lowering an Audiosource's volume explicitly.

Let us also mention, that Unity most commonly represents the game volume as a linear factor that ranges between 0.0 and 1.0, whereas Godot uses decibels. To remedy this, Unity sound volume values need to be converted to decibels when used within Godot. Calculators for this conversion can be found online[14]. In case such a conversion needs to be made within Godot C#, one can use the `GD.Linear2Db` method.

At last, there remain some slight differences in sound. It might be the case that there are some engine specific intricacies, that we are unaware of.

## IV. Evaluation/ Discussion

Summarising, we have discussed how to first port visual assets from Unity to the correct Godot formats via tools. Engine-specific components (e.g. visual effects) mainly have to be rebuilt using Godot's native systems. Secondly, we ported the code as faithfully as possible to the original, exploring the viability of a copy and paste approach. We can recommend this approach, although close attention has to be paid to whether Godot methods that seem equivalent to Unity methods actually function the same way. In the following, we discuss our experience working with Godot as a team and which programming language to choose for porting.

*1) Group Work in Godot:* For this project, we used Git for version control. We encountered two issues when working with Godot in a team.

Firstly, Godot 3.5 will frequently and automatically overwrite changes pulled from Git from another person. These automatic changes can be undone via Git but are frustrating to deal with.

Secondly, we often experienced problems when two people worked on the same scene independently. Merges would almost always result in merge conflict that are difficult to resolve. In some cases, these merges would result in corrupted (unusable) scene files, due to how Godot keeps track of the game objects in a scene. In other cases, some changes would go missing after a merge. To avoid merges, we often found that one person pulling changes and then redoing his entire work again after pulling was the least troublesome option.

*2) GDScript versus C#:* In the process of porting the code base from the Unity game to Godot, we used both GDScript and C#. Consequently, we got to know advantages and disadvantages of using either language for porting.

First of all, GDScript is the native engine language for Godot. Complementary resources such as guides, community discussions and downloadable addons/plugins are primarily written in and for GDScript. Code written in GDScript will not interact well with C# code, thus rendering most plugins mainly useless, if used in a C# code base instead. Another advantage of GDScript is that it has a very simple, python-like syntax. It is therefore relatively easy to get into.

On the other hand, choosing to use C# in Godot will allow for copy and pasting Unity code into the Godot code base, the code thus remaining closer to the original. This will work most of the time, because the underlying engine concepts are similar. Unity-native methods still have to replaced and checked

[14] Conversion of linear to db volume. http://www.takuichi.net/hobby/edu/em/db/. Accessed: 28 February 2024.

for diverging functionality. Most available GDScript methods also exist in C# and can be inferred from the documentation. Some mathematical C# functions are situated in Godot's Mathf library. The library does not exist in GDScript. This could sometimes lead to confusion when the documentation was not satisfactory. Furthermore, working with C# in Godot requires additional overhead setting the environment up. Strange engine behaviour or errors were sometimes hard to find a solution for as there was not an overwhelming amount of community discussion around the issues.

Overall, the performance of both languages seems to be comparable. Interaction between GDScript and C# scripts is not possible except with asynchronous signals (e.g. Fig. 6). It is recommended to choose and settle on one language.

```
// C# Script
public class SignalEmitter : Node
{
    //define the signal name
    [Signal]
    public delegate void customSignal();

    public override void _Ready()
    {
        // set up callback to send the signal
    to the GdScript method
        Connect("customSignal", GetNode("
    GodotScript"), "gd_script_method");
    }

    public void someFunction()
    {
        // ...

        EmitSignal("customSignal");
    }
}
```

```
// GodotScript.gd
extends Node

func gd_script_method():
    print("executed gd_script_method")
```

Fig. 6. Execute a GDscript method from C# code

## V. Conclusion

The Godot engine seems to offers a wide variety of functionality to create games. In this project, we explored the prospect of porting numerous applications of Unity's many features to Godot. We found it vital to understand that the space spanned by the multitude of adjustable parameters and other turning knobs is so large that one cannot approach porting an application to another engine with the expectation of achieving an exact replication of the original. Especially when the underlying system paradigms change (for instance for the kart), there are many opaque and minute differences that cannot be noticed right away and require in-depth knowledge of both engines in question to address. This means that aiming for a more "loose" port is probably a good idea. Further, we found that when porting to Godot, there is some extra work of implementing Unity-specific features by hand. Godot feels like a more lightweight engine than Unity, therefore creating the necessity to find workarounds for missing features in Godot (e.g. missing functionality to disable and enable objects). Furthermore, we found that ancillary resources like documentation, guides and community discourse were sometimes lacking in quantity (most likely due to Godot being a less popular engine than Unity according to recent statistics from a global game jam[15]).

In the end, we recreated a Unity 3D application with the Godot engine which holds a fair resemblance to the original. Comparing our porting experience with findings from other game developers detailed in Section II, we can recommend Godot 3.5 as an alternative engine, although workarounds will probably have to be found for some intricacies of envisioned projects. As Unity developers ourselves, we often found Godot's systems to be similar to Unity and intuitive to learn. The existence of porting tools such as the UnityGLTF package mostly boils down the porting problem to firstly, porting the assets with tools and secondly, recreating the functionality (and code) with the Godot engine while supporting this process with insights gathered from the already existing Unity application to port.

## References


[1] *Godot Engine 3.5 documentation in English*, 2023. https://docs.godotengine.org/en/3.5/tutorials/scripting/c_sharp/c_sharp_basics.html.

[2] *Godot Engine 4.1 documentation in English*, 2023. https://docs.godotengine.org/en/4.1/tutorials/export/exporting_for_web.html.

[3] I. Buyuksalih, S. Bayburt, G. Buyuksalih, A. P. Baskaraca, H. Karim, and A. A. Rahman. 3d modelling and visualization based on the unity game engine – advantages and challenges. *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, IV-4/W4:161–166, 2017.

[4] Bryan Duggan. Porting a game ai course from unity to godot: A case study. 2023.

[5] John K. Haas. A history of the unity game engine. Technical report, Worcester Polytechnic Institute, 2014.


[15] Game engine popularity in 2024. https://gamefromscratch.com/game-engine-popularity-in-2024/. Accessed: 28 February 2024.