**UDC 004.946**
**DOI: https://doi.org/ 10.17721/3041-2323.2025.377-387**

**Ivan SOKALSKYI, Master**
**ORCID ID:0009-0000-4213-6633**
**e-mail: ivansok110503@knu.ua**
**Taras Shevchenko National University of Kyiv,**
**Kyiv, Ukraine**

**Olena FENDO, PhD(Engin.), Assoc. Prof.**
**ORCID ID: 0000-0001-9197-6399**
**e-mail: olena1.fendyo@knu.ua**
**Taras Shevchenko National University of Kyiv,**
**Kyiv, Ukraine**

# DEVELOPMENT OF A GAME ENGINE USING THE ENTITY COMPONENT SYSTEM ARCHITECTURE

***The article discusses a relevant topic – the development of a data-oriented game engine based on the entity-component system architecture. Given the high resource demands of modern games, data-oriented architectures, particularly the entity-component system, are of particular relevance, as they significantly reduce the load on the central processing unit (CPU). The use of this software allows rendering scenes with the engine's built-in renderer, compiling the game for multiple platforms, and linking the user's game code to the engine through a DLL.***

**Keywords:** ***game engine, entity component system, data-oriented design, DLL, WPF.***

## Introduction

Nowadays, video games are massive applications that contain hundreds of thousands of lines of code, textures, meshes, and audio files. Advancements in real-time rendering, combined with the constant consumer demand for higher graphical fidelity, result in video games consuming a tremendous amount of CPU resources – a trend that is expected to worsen over time.

Video games are applications that constantly operate with an enormous amount of data, so to mitigate the issue of CPU resources usage, developers can use game engines that are built using data-oriented architectures (entity component system). Game engines are

enormous pieces of software, that can include elements from every field of computer science (e.g. computer graphics, artificial intelligence, systems programming). Game engines usually consist of an editor part (user interface that allows to interact with an engine), an engine, that contains all the components of an engine (renderer, specialized frameworks, math functions, etc.) and also it can have a DLL, if, for example, an editor is wrote in C# and an engine is wrote in C++.

The main purpose of game engines is to enable development teams, consisting of both programmers and non-programmers, to collaborate effectively on game production. It is essential to note that a modern game engine must be designed not only for programmers but also for game designers and artists. Moreover, several contemporary game engines provide opportunities for developing game logic even without professional programming skills through the use of visual scripting tools, such as Blueprints in Unreal Engine.

That being said, the primary responsibility for optimizing a game engine – and consequently, the video game built upon it – rests with the engineering team. During the development process, numerous complications may arise. For instance, improper texture loading into the renderer can place excessive strain on the GPU, while simultaneously loading a large number of assets into the scene may lead to CPU or RAM bottlenecks. These issues necessitate the implementation of asset streaming functionality within the engine. Game engines typically consume substantial system resources across all their components, making it critically important to optimize resource usage wherever possible in order to achieve maximum performance.

Unfortunately, developing a software in a data-oriented way is a gigantic toll on the development, as it is usually quite different from other approaches, that developers are used to, and also has its own set of serious problems. The fact that out of all popular game engines none were originally developed to support the entity component system speaks for itself. However, all is not lost here. For example, Unity, one of the most popular game engines on the market nowadays, sevrral years ago developed a dedicated framework for writing a game code in the engine using the ECS. Bevy, most popular game engine for

developers, that use the Rust programming language, was from the ground developed to support the ECS.

**Results**

The game engine described in this article consists of three main components: an editor developed in C# using WPF, a core engine written in C++, and a DLL that links the editor to the engine (Fig. 1). Although an editor implemented in C++ might offer greater performance, developing it with WPF is significantly less time-consuming. Moreover, the editor's performance is not critical, as it does not affect the final product – the video game itself.

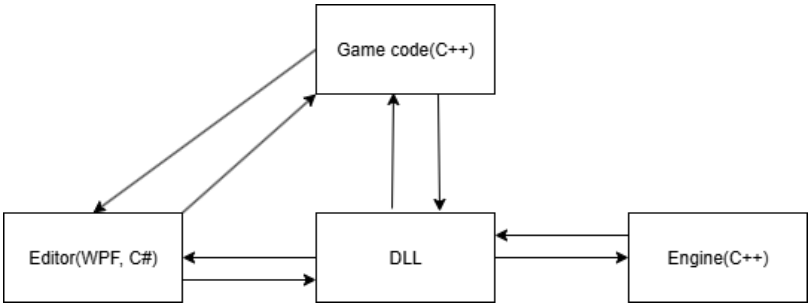


***Fig. 1.*** *Architecture of a game engine*

The game code is authored by the user, but the editor may provide several predefined templates to facilitate interaction with the engine. Additionally, it is possible to allow the game project within the editor to be modified by the game code through reflection techniques. For instance, Epic Games developed a C++ reflection system for their Unreal Engine to support similar functionality.

The development of a game engine typically begins with the creation of an editor, as it facilitates subsequent testing and provides the programmer with a clearer understanding of the components required within the engine. In this article, the editor is implemented using Windows Presentation Foundation (WPF) a free and open-source user interface framework for Windows-based desktop applications. WPF is built on the .NET platform and primarily utilizes C# and XAML for development. A developer can use any programming language or UI frameword to create an editor, as long as

it's possible to link it with the core engine code. For example, Epic Games developed their own C++ UI Framework "Slate", which was used to create the editor for their "Unreal Engine" game engine.

Each game engine editor features a unique interface and set of functionalities, which reflect the components available in the engine itself. However, almost all game engines include a dedicated window for rendering the game scene in real time. In WPF, developers can host a Win32 window using built-in mechanisms and employ it to display the game scene via the engine's rendering system. All professional game engines have settings for their "Scene view" (rendering window). For example, the user can change the perspective (Top, Bottom, Orthogonal, Perspective, etc.), change the field of view of the virtual camera or have some special settings (for example, render only meshes).

Also, a very important thing to bear in mind, during development of a rendering system, is a choice of a graphics API, that will be used in the development of a renderer. The choice of API directly influences platforms, that the engine will be available for and games developed on this engine will be availabe for (Tab. 1). However, this can be mitigated by the fact, that it is possible to use several APIs in the rendering system and let the user to switch between them. Of course, one significant disadvantage of this approach is the fact, that it will consume gigantic resources to develop a game engine for use with several APIs (Wikipedia contributors, 2025).

***Table 1***

**Latest Graphics Library usage across different platforms**

| OS | Vulkan | Direct X | GNMX | Metal |
|---|---|---|---|---|
| Windows 10 | Free, Nvidia and AMD | Free, MS | no | no |
| Mac | Free, MoltenVK | no | no | Free, Apple |
| Linux | Free | no | no | no |
| Android | Free | no | no | no |
| iOS | Free, MoltenVK | no | no | Free, Apple |

**Continuation of Table 1**

| OS | Vulkan | Direct X | GNMX | Metal |
|---|---|---|---|---|
| Tizen | in Development | no | no | no |
| Sailfish | in Development | no | no | no |
| Xbox One | no | Free | no | no |
| Orbis OS (PS4) | no | no | Free | no |
| Nintendo Switch | Free | no | no | no |
| HarmonyOS | Free | no | no | no |
| OpenHarmony | Free | no | no | no |

In most cases editors commonly include a list of objects (noting that the term “object” may vary in definition across different engines) currently present in the scene, along with a panel for modifying these objects (Fig. 2). This panel typically allows users to add new components, delete or rename objects, and adjust their properties. Also, certainly developers would want to create a dedicated “Output” window, which will facilitate “communication” between developers of a game engine and a game developer, that uses an engine to create a videogame.

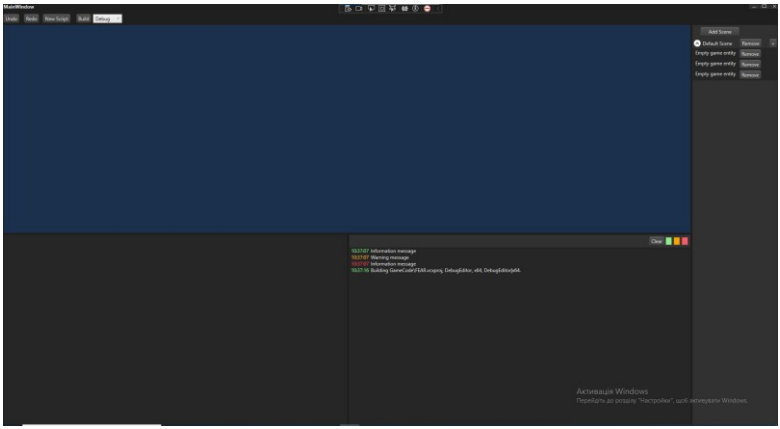

***Fig. 2.*** *Example of a game engine’s editor*

A very important requirement that game engine developers must bear in mind when developing an editor for their game engine, is that the interface will be used not only by developers with a programming background, but also by game designers and artists. For example, Unity and Unreal Engine allow programmers to set some variables in a game code as “Open to editor”, which means that this variable can be viewed and modified (it depends on parameters that were set by a programmer in a game code) in an editor, most certainly in a dedicated “Object’s Properties” panel. This allows for a quicker development time and takes some tasks off an engineering team, which allows them to focus on other tasks. However, it also means additional time for developing an engine’s architecture and developing tools, that allow for interaction with a non-programming staff to happen. Developing (or using an existing) a reflection system may considerably facilitate these interaction, but this is a very serious task which requires huge resources and hard to find knowledge among the team (Khitami, 2019).

The entity-component system (ECS) architecture is born at the editor level. Each object within a game scene must possess a unique identifier (ID), which the engine uses to distinguish it from other entities in the scene. Initially, an object is merely a shell containing an ID and a name; however, developers can subsequently attach various components to it, thereby extending its functionality. If necessary, initialized objects can contain more than just an ID and a name, if the engine requires so. Commonly, game object is required to have a “Transform” component, which contains information about object’s position, rotation and scale, as without it a game object just doesn’t make any sense in most situations.

Naturally, the editor must also support a range of auxiliary features beyond those directly related to ECS. These include project creation and saving, keyboard shortcuts, user interface customization, and other general-purpose utilities. This article does not delve into these aspects in detail, as they pertain more broadly to software development practices rather than to the specific challenges of game engine architecture.

Once a basic version of the editor has been implemented, development of the game engine can begin. For the entity-component

system (ECS) to function properly, each object within the engine must be assigned a unique identifier (ID). However, a challenge arises when an object is deleted: its ID is permanently lost and cannot be reused. While this approach is simple, it carries a significant risk – over time, the engine may exhaust the available pool of IDs. The possibility of this disaster in modern 64-bit systems is extremely low, but a responsible development team always has to handle critical bugs.

Fortunately, there is a solution. For example, if the ID is 4 bytes long, it is possible to make n number of bits the "generation" bits and 4 * 8 - n number of bits the "index" bits. Every time the object is deleted generation of its index is increased by 1, when index bits remain the same. This way it is possible to reuse IDs for an enormous amount of times (Autodesk Stingray, 2014).

After that it's possible to create an entity class, which will represent all the objects in a scene, and a transform component, which most certainly all the objects in a scene are required to have. This component-based system is a huge advantage of the ECS, as it lets minimize the amount of cold data that objects (in its original computer science meaning) have, because if some objects need particular components, a programmer can just add them to the object. In OOP, for example, it is not a rare sight to see an inherited class, that has fields or methods that it certainly doesn't need, but its closest relatives depend on it. One may say that it is more of a sign of a bad architecture decision, not innate OOP problem, but "good architecture" is easier said, than done, especially in software of gigantic complexity, that video games are, and ECS certainly helps minimize the risks here.

The second advantage is that ECS is operating with fundamental types, which helps to minimize cache misses. But at the same time it is also a disadvantage. For example, if the function is reading from Transform and AI components and modifies the Health component, there is a potential for three cache misses instead of one, if components were tightly packed in a single object.

The third advantage, is that the ECS basically forces programmers to design a product in a data-oriented way. It is certainly possible to do without the ECS, but it requires a lot of discipline and knowledge from a programmer.

As any architecture, ECS has its own set of problems. Firstly, it is very hard to debug. As an engine is operating with identifiers that only point to objects, instead of straightly working with objects. This decreases the amount of information a programmer can get from IDE debuggers and increases amount of time required to debug an engine to find out a problem.

Secondly, ECS is a complicated architecture and may require some time for programmers on the team to get used to, as it is very different from the standard OOP. Taking into account that time and money are very scarce at the game development industry, this actually becomes a very serious disadvantage (Gustafsson, 2025).

When shells of future editor and engine have been developed, it is now possible to link them through a DLL. A Dynamic-link library (DLL) is a library that contains code and data that can be used by more than one program at the same time. For example, in Windows operating systems, the Comdlg32 DLL performs common dialog box related functions. Each program can use the functionality that is contained in this DLL to implement an Open dialog box. It helps promote code reuse and efficient memory usage (Microsoft, 2025). The game engine's DLL is a standard DLL that is developed for any other kinds of software. Programmer needs to write functions in the DLL that will call functions from a C++ game engine and in an editor code write functions, that have the same signature as DLL functions. After that, an editor and an engine are linked and are ready to work together (Fig. 3).

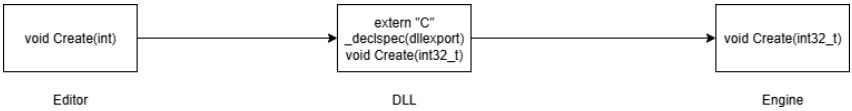


***Fig. 3.*** *Linkage between an editor and a DLL*

When all the required three components have been done, it is basically a matter of creating a new system interface in an editor (e.g. AI framework using GOAP architecture), actually implementing the system's internal logic in an engine and link them through a DLL.

Of course, when in principle it sound simple, developing any system for a game engine is an enormous taks, as any system in a contemporary professional game engine has a lot of requirements and expectations to it. In early videogame history a game engine team and a game programming team were basically the same entity, as requirements to games and subsequently to game engines were significantly simpler. But nowadays, due to big toll that developing an engine takes on a team, only the wealthies of game companies can allow themselves a luxury of having a proprietary game engine. Most other game companies license other companies' game engine (Unreal, LithTech, Unity, Aurora, etc.), modify them and develop their games on modified versions of those engines.

**Discussion and conclusions**

The game engine described in this article enables users to develop video games with optimized CPU utilization, achieved through data-oriented design reinforced by the entity-component system (ECS) architecture. While this engine may offer a significant advantage in optimizing the final product, it is important to recognize that such benefits come with increased software complexity. This complexity can lead to longer development cycles, higher financial costs, and a greater demand for experienced programmers who are capable of understanding the architectural principles and working effectively within them.

**References**

Wikipedia contributors. (n.d.). *Windows Presentation Foundation*. Wikipedia. Retrieved from https://en.wikipedia.org/wiki/Windows_Presentation_Foundation#cite_note-2.

Autodesk Stingray. (2014). *Building a data-oriented entity system (part 1)* [Blog post]. Bitsquid Development Blog. https://bitsquid.blogspot.com/2014/08/building-data-oriented-entity-system.html.

Khitami, A. (2019). *Game engine series* [Video playlist]. YouTube. https://www.youtube.com/watch?v=hRL56gXqj-4&list=PLU2nPsAdxKWQYxkmQ3TdbLsyc1l2j25XM.

Gustafsson, D. (2025). *Thoughts on ECS* [Blog post]. Voxagon Blog. https://blog.voxagon.se/2025/03/28/thoughts-on-ecs.html.

Microsoft. (2025). *What is a DLL*. Microsoft Learn. https://learn.microsoft.com/en-us/troubleshoot/windows-client/setup-upgrade-and-drivers/dynamic-link-library.

**References**

Wikipedia contributors. (n.d.). *Windows Presentation Foundation*. Wikipedia. Retrieved from https://en.wikipedia.org/wiki/Windows_Presentation_Foundation#cite_note-2.

Autodesk Stingray. (2014). *Building a data-oriented entity system (part 1)* [Blog post]. Bitsquid Development Blog. https://bitsquid.blogspot.com/2014/08/building-data-oriented-entity-system.html.

Khitami, A. (2019). *Game engine series* [Video playlist]. YouTube. https://www.youtube.com/watch?v=hRL56gXqj-4&list=PLU2nPsAdxKWQYxkmQ3TdbLsyc1l2j25XM.

Gustafsson, D. (2025). *Thoughts on ECS* [Blog post]. Voxagon Blog. https://blog.voxagon.se/2025/03/28/thoughts-on-ecs.html.

Microsoft. (2025). *What is a DLL*. Microsoft Learn. https://learn.microsoft.com/en-us/troubleshoot/windows-client/setup-upgrade-and-drivers/dynamic-link-library.

**Отримано редакцією журналу / Received: 10.09.25**
**Прорецензовано / Revised: 16.09.25**
**Схвалено до друку / Accepted:01.10.25**

**Іван СОКАЛЬСЬКИЙ, магістр**
**ORCID ID: 0009-0000-4213-6633**
**e-mail: ivansok110503@knu.ua**
**Київський національний університет**
**імені Тараса Шевченка, Київ, Україна**

**Олена ФЕНДЬО, канд. техн. наук, доц.**
**ORCID ID: 0000-0001-9197-6399**
**e-mail: olena1.fendyo@knu.ua**
**Київський національний університет**
**імені Тараса Шевченка, Київ, Україна**

# РОЗРОБЛЕННЯ ІГРОВОГО РУШІЯ НА ОСНОВІ АРХІТЕКТУРИ ENTITY-COMPONENT SYSTEM

***У статті розглядається актуальне питання розроблення ігрового рушія з орієнтацією на дані, який побудований на основі архітектури типу entity-component system (ECS). З огляду на зростаючі вимоги до сучасних відеоігор, архітектури з орієнтацією на дані, зокрема модель entity-component system (ECS), набувають особливої значущості, оскільки дозволяють суттєво зменшити навантаження на центральний процесор (CPU). Використання такого програмного забезпечення дає змогу рендерити сцени за допомогою вбудованого рушія, компілювати гру для кількох платформ і зв'язувати користувацький ігровий код із рушієм через DLL.***

**Ключові слова:** ***ігровий рушій, архітектура entity-component system (ECS), динамічна бібліотека (DLL), фреймворк WPF.***

Автори заявляють про відсутність конфлікту інтересів. Спонсори не брали участі в розробленні дослідження; у зборі, аналізі чи інтерпретації даних; у написанні рукопису; в рішенні про публікацію результатів.

The authors declare no conflicts of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript; or in the decision to publish the results.