Modulith: A Game Engine Made for Modding

Daniel Götz
goetz@bii-gmbh.de
Building Information Innovator GmbH
Würzburg, Germany

Sebastian von Mammen
sebastian.von.mammen@uni-wuerzburg.de
Julius-Maximilians-Universität
Würzburg, Germany

ABSTRACT

A modular game engine facilitates development of game technologies and game contents. The latter includes content creation by the player in the form of modding. Based on a requirements analysis to meet the needs of developers, gamers and modders, we present a novel concept for a fully modular game engine. This concept includes a module system, which allows additional code to be loaded at runtime, and various support systems that simplify the use of the engine and the creation of modules. We present a proof-of-concept implementation, underline its fulfillment of the unearthed requirements, and demonstrate its use.

CCS CONCEPTS

• **Software and its engineering** → *Requirements analysis*; Object oriented development; • **Information systems** → *Multimedia content creation.*

KEYWORDS

game engine, modding, entity component system, user-generated content

ACM Reference Format:
Daniel Götz and Sebastian von Mammen. 2023. Modulith: A Game Engine Made for Modding. In *Foundations of Digital Games 2023 (FDG 2023), April 12–14, 2023, Lisbon, Portugal.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3582437.3582486

1 INTRODUCTION

The source code of computer games can be organized in a modular fashion to support extensive reuse. Examples are sub-systems for rendering, physics calculations, or input handling, sometimes referred to as sub-engines. They can be wrapped into an execution framework referred to as game engine, which drive the games created with them [1, 12, 27]. As game technology undergoes fast adaption, even the sub-engines, as the corner stones of game engines, need to be frequently updated and sometimes even replaced. For instance, several current game engines would still heavily benefit from changing their rendering sub-engines to the Vulkan application programming interface (API) [26]. As a result, game engines are usually designed to be inherently *modular* at a low level to make the process of re-writing and replacing sub-engines as easy as possible.

Similarly, modular design also plays a significant role in game design and development—on top of the underlying game engine. Instead of offering the same modular API of the sub-engines, a different solution is typically used: Engines expect the developers to utilise high-level coding patterns such as object orientation or component aggregation, oftentimes in combination with a scripting language for writing the gameplay code that interacts with the lower-level systems. This infrastructure aims at faster development times, e.g. by using an interpreted language that does not need to be compiled for less performance-critical parts of the game logic. However, this also results in a decrease in performance as the code might not run from optimised, native binaries on the target systems. Also, the need for an interfacing layer with the engine and individual (sub-)systems increases code complexity and maintenance overhead. This can significantly diminish the relative benefits of scripting languages, if the designers/developers aim at complex or performance-intensive gameplay features. However, without the additional scripting layer, low-level API changes would immediately break existing code bases, whether of games or code extensions such as dedicated libraries or plugins: Backwards compatibility would quickly get lost.

Lastly, games themselves can also be the platform for adding modular pieces. Examples include developer-made additions in form of downloadable contents (DLC) or player-made additions referred to as modifications (mods), both being especially prevalent in singleplayer games. The creation of mods, or modding, can be greatly facilitated by game engines [23]. At the same time, if an engine does not inherently support modding, game studios oftentimes do not consider it worth their investment. As a result, games may miss out on establishing a modding community, which could otherwise increase the lifetime and the sales of a game for years to come. Apart from sales, modding communities can also serve as a recruitment pool for developers. For example, both of the aforementioned advantages are highlighted by [11] to further the success of the popular Elder Scrolls game series.

In this paper, we present the requirements-based engineering concept, design and development of a proof-of-concept game engine that is inherently modular at every level, even supporting hot-loading of code during runtime[1]. We highlight its flexibility in light of the gathered requirements and give three complementing examples that allow one to trace the amount of work required for the respective mods. We briefly outline seminal works that our project was built on in the next section. Section 4 will present both the results of the conducted requirements analysis and the design we inferred. In Section 5, we will present exemplary use cases that

[1]Link to the Git-Repo of the developed game engine Modulith: https://github.com/Eregerog/Modulith.

comprehensively demonstrate the flexibility of the engine. In particular, we briefly present the modular composition of graphics and physics sub-engines, the modular composition of game contents, and, finally, mods that alter the game contents. We conclude with a short summary and an outlook on potential future work.

2 RELATED WORKS

There are numerous advantages for developers and players to rely on modding [23]. From a creator's perspective flexibility through modularity might offer smoother and faster development throughout the whole life cycle of a game. For sake of longevity, platform-independence, specialisation, and many more, developers may want to swap lower-level sub-engines, e.g. for rendering or physics [12], whether externally provided for fast prototyping or developed in-house for perfect customisation [10]. When creating game content such as a map navigation technique or combat mechanics, multiple iterations are often needed to ensure both great playability and code quality. Concurrent development and easy-to-use integrated tests, as made possible by a modular engine, would provide the greatest insights and allow for fast advancements. Lastly, after a game has been released, developers will generally need to develop post-release patches and content additions. Players heavily invested in a specific title may be motivated to introduce minor improvements, e.g. the visual quality by changing post-processing effects or textures. They may also seek a different gameplay experience, by re-balancing the game's mechanics or creating new maps. Larger teams of such modders may also work together to create new high-quality content, such as new quest-lines with voice-acted dialogue or sprawling worlds. There are even rare cases of new games built with original assets and stories using the modding tools of another game, such as the total-conversion mod Enderal [25]. Accordingly, modders might want to create individual modules that contain everything necessary from code to assets to have their creations stand for themselves. They might also, however, want to collaborate with others and contribute to a greater goal.

ARGoS is an example of an engine with modular architecture [21]. It is an efficient and flexible multi-robot simulator which allows for loading sub-engines at run-time, e.g. the physics engine or robot controllers. Data components are used to communicate between these "plug-ins" and to aggregate various state information in model entities, e.g. the transformation state as well as sensor and actuator capabilities of a robot. Swapping in compiled C++ code, as shown by [6], allows for even greater performance than only relying on well-organised data structures of such entity component systems (ECS) [12]. To generate machine-readable binary code, C-code files are translated by a compiler. The resulting object files contain the memory addresses to compiled code blocks, such as functions or variables. Swapping in code can still be achieved by compiling code changes, or even entire "dynamic" classes [16] into dynamically linked libraries (DLLs), which can be loaded at runtime [24]. While static libraries will be embedded in the executable binary of an application, the code blocks contained by dynamic libraries may be shared by numerous applications. Their code blocks can be accessed from the outside since their exact location within the file is exposed. Dynamic libraries' contents can, thus, be changed without the need to recompile dependent applications. Access to the dynamic libraries' contents is either established when the respective application is loaded (load-time dynamic linking) or later, on-demand (run-time dynamic linking). For the latter, the operating system typically provides four functions: One function for establishing access to a dynamic library, one that retrieves a pointer to specific code blocks (e.g. functions), one that unloads a library, and another one that captures and reports any errors.

Subtype polymorphism as provided by class inheritance in C++ allows us to override behaviors [7]. Implementing virtual functions, obsolete ones can be replaced by their newly-compiled versions due to pointers referencing the underlying data [21]. For performance-critical applications, this flexibility may be turned off for release builds to save the referencing overhead. Data-oriented programming, i.e. ensuring that data is laid out in memory for efficient handling, and building on the ECS pattern is currently considered the ideal strategy for efficient and yet flexible model designs [12]. Folmer points out that component-based development also supports high-quality software development and easier integration of cooperative works [10].

In the popular game engine Unity [28], low-level components can be added though "packages" which may contain high-performance native code, C# source files, as well as game assets. They are installed and updated through a graphical user interface (GUI) in the editor. The Creation Engine, used for games like Skyrim [4] and Fallout 4 [5], provides the Creation Kit-editor for modding. Edits and additions to game code are saved as plug-ins, whereas assets such as scripts or textures are archived separately. Several popular game engines have been compared with respect to their modding capabilities in [23].

While, popular game engines generally provide flexible asset management and script loading, none of them provides modding capabilities to the extent proposed and realised in the work presented in the remainder of this paper. In particular, our approach supports modding of native C++ code snippets and low-level modding of (sub-)engines. As a result, our proposed engine just consists of the bare minimum, i.e. a module and resource system for loading other mods, whereas the engine's expected core systems are implemented as interchangeable mods, themselves. Hence, upon execution, the engine's functionality is bootstrapped by loading one mod after the next, rather than being tightly and unalterably integrated.

3 REQUIREMENTS ANALYSIS

Both, the developers' and the players' modding needs would be addressed by a fully modular game engine, which, in addition to modular loading of assets, allows to load sub-engines and gameplay code during runtime. Several according sub-goals G can be defined. First, the engine should allow its sub-engines to be interchangeable modules, potentially built on modules themselves (G1) to meet the need for modularity during game development and modding. Furthermore, it should be able to compose games on top of numerous sub-engine modules (G2), allowing the same low-level modules to be re-used between games, without having to modify them on a game-by-game basis. Lastly, the engine should offer the possibility to add additional content to deployed games, to make them easy to mod. This has to be considered from the perspective of the modders wanting to realize their creative vision, who depend on tools for easy mod creation and the ability to modify as much of a game's assets as possible (G3). Additional perspectives that need to be considered are those of the players, who want an easy installation of mods (G4) and the perspective of the game developers who want to still be able to update their game with fixes and provide additional content without breaking mods with every update (G5).

Considering the outlined goals, we can infer several concrete system requirements R. Integral to all goals concerned with module development (G1 to G3) is the engine's capability to load additional code at runtime to maintain fast iteration times and inversion of control. To achieve runtime loading, there must be a specification on how to structure and compile the code of a module that developers can follow, so that the engine can detect the module as a loadable component (R1). Furthermore, the engine should offer an API to load these modules on-demand or on engine startup (R2). Since the engine itself has control over the game loop, loaded modules need to be able to register callbacks for various game loop events (R3), allowing them to execute their behavior. Additionally, modules should also be able to provide their own game loop events to other modules (R4). However, to register callbacks, the compiled module code needs to have access to the engine's data structures and functions API (R5). It is equally important for all goals that a module can use the API of another module. In the most simple case, the created games will generally depend on the sub-engines' APIs, such as the rendering and physics systems. But in more complex examples a higher-level sub-engine may depend on the API of a lower-level sub-engine. The requirements for such dependencies are as follows: A module needs to be able to define dependencies to other modules (R6). It should also have access to the API of all its dependencies (R7), allowing it to build higher-level code on top of them. The module system also needs to be designed in a way that ensures that modules are loaded in the correct order (R8) to make sure that dependencies are resolved when loading and maintained when unloading the respective modules. Modules' dependencies to external libraries also need to be resolved (R9). Furthermore, the exchange of data between modules should be realized by means of an efficient, so-called glue layer (R10) in order to avoid a potential bottleneck [10]. Providing a GUI (R11) can make the engine easier to use and allow one to define dependencies in a visual manner. From the GUI, the build process can be automated by creating build files for every module (R12). Generally, mods need to be able to override the assets of modules they depend on (R13), to override the behavior of a dependency (R14). The latter would, for example, allow a mod to provide its own camera controller. Lastly, the deployment of mods should not require writing any code (R15), and it should be possible to deploy updates in the form of fixes and content additions to a game, while minimizing the risk that mods could break as a result (R16).

4 CONCEPT & DESIGN

The architecture of the game engine as a whole has to consider the *engine runtime*, the *engine* and *modules* as visualized in figure 1. We first elaborate about their relationships, then we present the ECS as a generic means to organise data and communication in Modulith, and finally we shed some light on its editor—all to address the requirements outlined in the previous section.

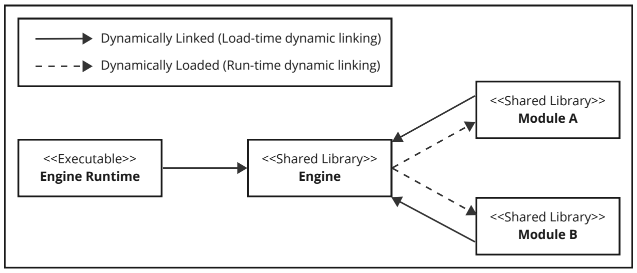

Dynamically Linked (Load-time dynamic linking)
Dynamically Loaded (Run-time dynamic linking)
<<Executable>> Engine Runtime
<<Shared Library>> Engine
<<Shared Library>> Module A
<<Shared Library>> Module B

Figure 1: Module A has a dependency to B (demonstrating R7). The runtime and both modules are dynamically linked against the engine (R5). The engine has the capability to dynamically load modules A and B on demand or on startup (R2).

4.1 The Modular Foundations

Aligned with Section 2, the engine itself as well as all modules are compiled as shared libraries to be able to use run-time dynamic linking and dynamic loading. As a result, modules have access to the engine's API (R5), as well as to those of other modules (R7), allowing one to modify their behaviours (R14). Another point of consideration in the engine's architecture is how the engine and modules expose their API in order to facilitate R5 and R7. In order to share the declarations of functions and signatures with dependants, the headers of a module can be deployed alongside its library. Then the depending modules can link against its library and include its headers.

Since the engine is compiled as a library itself, a separate application, the engine runtime, is needed for loading and starting it. It is compiled as an executable, is dynamically linked against the engine at load time, and hands over execution to the game loop of the engine. The engine also offers an API for modules to register callbacks for various game loop events (R3). On top of that, it is also responsible for loading, initializing, shutting down and unloading modules and exposing an API for modules to (un-)load other ones. Which modules are loaded upon startup is determined by a text file (R15). The engine and the engine runtime have dependencies to several third-party libraries: *crossguid* allows for OS-independent generation of unique identifiers used by the module system [15]. *boolinq*, a header-only library, allows for functional programming when modifying and iterating over data structures [8]. *spdlog* is incorporated for logging [18], as well as *yaml-cpp* for parsing and emitting YAML files [3]). And lastly, *glm* library is used for the mathematical data structures and functions useful for graphics and game logic [13]. Should a module depend on an external library, it can simply link against it to use its API (R9). If the external library is a shared library, it has to be deployed alongside the module. Using an external static library, however, might result in excessive space usage and, even worse, conflicting code, e.g. when embedding different versions to resolve the dependencies of different modules. A module has to implement a given specification to be recognized and loadable by the engine (R1). All files related to a module must be contained in a subfolder relative to the executable containing a configuration file. It contains the module's name, author, version, and, most importantly, its dependencies to other modules (R6). The subfolder may also contain game asset files, the module's shared library, and other shared libraries. The dependencies of modules are specified in a text file (to address R8) and stored as a directed acyclic graph (DAG), nodes representing modules, edges dependencies. Its reverse/forward traversal yields the loading/unloading order of the modules, respectively.

Search paths of external libraries are kept in memory and automatically considered for the built process. After loading a module, its initialization function is called to perform any necessary start-up logic. Analogously, a shutdown function with clean-up logic is called when a module is unloaded. The implementation of these methods is mandatory (R1). Re-compiling a library that has already been loaded is made possible as well, by flagging it *hotloadable* which makes sure that a copy of the originally compiled DLL is created and loaded, and the original can still be worked on. To avoid pointers to unloaded modules and to allow modules to create their own abstractions for callbacks (R4), modules (un-)register their data when (un-)loading, respectively. Modules can, e.g., register callbacks in the game loop (R3) by means of the so-called *Subcontext* abstract class. Modules can derive from this class and register an instance with the engine, which will then call various virtual methods in the subcontext when called from the game loop. Default callbacks happen for initialization and shut-down of the subcontext, at various stages of a frame, and when modules are loaded and unloaded. Custom callbacks can be implemented as well, e.g. for drawing GUI elements or responding to click events as part of a GUI module.

Since modules should also be able to override the assets of dependant modules (R13), each module should register its assets with the engine, whereas the last identical one persists. Its identity would be inferred from identical paths of the respective files[2]. In Modulith's current version, the asset management functionality is not completed, yet. Instead, assets are directly loaded by the gameplay code. An example of dependencies and the envisioned override behaviors is depicted in figure 2.

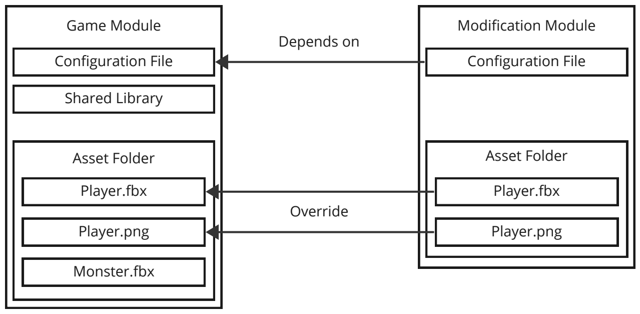


Figure 2: In the example, the Game Module provides assets, which are overwritten by the Modification Module (R13). The underlying dependency is specified in the Configuration File (R6).

[2]The paths would be defined relative to the respective module's subfolder placed in the root directory of Modulith.

Developers need to be able to release updates and content additions for their deployed game modules (R16), which affords *backwards compatibility*. Without, a dependant module might not build any longer, not load after an update, or behave incorrectly. Removing, renaming or changing the signatures of functions is only possible for internally used ones. Implementation changes of functions work in general, but they are not considered by dependent modules that have previously defined versions in their sources due to the functions being declared as 'inline' code. Similarly, adding data structures is safe, whereas any other changes result in incompatibility. As a general guideline, we recommend exporting a minimal set of functions and data structures [24] and avoiding 'inline' code to improve maintainability. Instead of modifying existing functions, new ones can also be added and old corresponding ones labelled as 'deprecated' rather than being removed. If incompatibility cannot be avoided, developers can always fall back on semantic versioning schemes for their modules, to properly communicate backwards compatibility [22].

4.2 Modulith's Entity Component System

To facilitate efficient inter-module communication (R10), we provide an ECS inspired by the property-centric approach which separates data and logic and optimizes the memory layout for better cache utilization. It consists of four components: *Entities* are identifiable wrapper objects which aggregate data *components* and which are registered with an *entity manager*. The components are registered with different *systems* that process their data, i.e. they read or write component data when receiving callbacks from the engine's game loop.

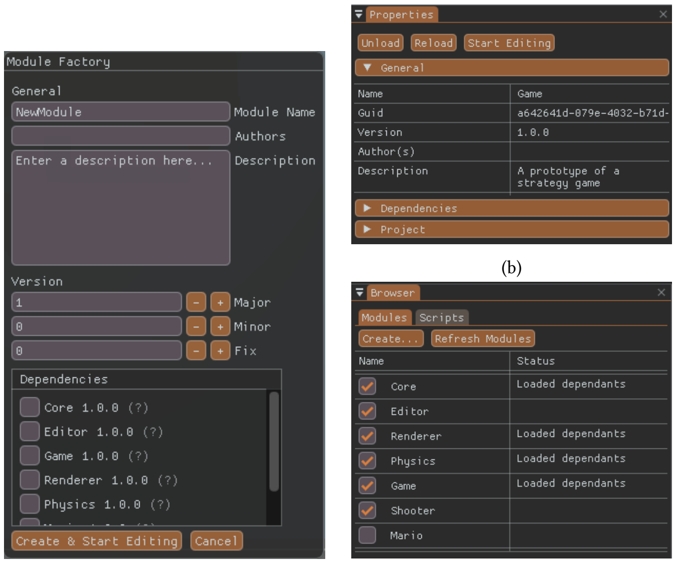
In order to keep the inner-most loops small and to improve instruction cache utilization, we support bulk operations, i.e. that systems read/write from/to large numbers of components at once. To this end, we provide a *query system*, making use of C++ templates to implement various querying constraints, to select sets of specific components, as well as a custom memory allocator, an *entity chunk*, which makes sure that the respective components are laid out sequentially in memory. Each chunk allocates a fixed size buffer of memory on the memory heap in which a certain amount of entities alongside their components are allocated. When a chunk is filled with entities a new chunk is created and memory of empty chunks is released again. Gaps due to removal of entities are filled by entities moved from the end of the chunk.

The ECS and the asset management system are outsourced into the so-called *Core Module*, which also provides an application window and asset importers. It relies on the *glfw* library for creating the application window [9] and on the OpenGL API of *glad* [14]. *Dear ImGui* is integrated to draw debug UI for developers [20]. Lastly, the *assimp* library [17] and *stb_image* header [2] from the stb repository have been integrated for loading models and textures, respectively.

4.3 Modulith's Editor

An important feature of the engine is to provide a GUI for easy creation of modules that integrate with IDEs (R11 and R12). To this end, we provide an editor module that can optionally be loaded and allows for control of the loaded modules and creation of new ones.

The idea is that developers and modders can load the editor while they are developing to allow for fast iteration cycles. The editor is realized as a separate module depending on the core module's provision of *Dear ImGui*'s API to draw its debug windows. The editor's overlay windows can be toggled on or off to not interfere with gameplay. In its *module factory* window (figure 3 (a)), the user can enter the name, author, description, version, and dependencies of a new module, which can be viewed in the module's *properties pane* (figure 3 (b)). Subfolders and configuration files are automatically created beneath the targeted executable. A C++ project containing minimal default source code files (including mandatory start-up and shutdown code) and headers, as well as CMake build files for CMake for use with popular IDEs such as Visual Studio or CLion are stored in a user-chosen directory. On this basis, the user can work on modules right away. In the *browser window* window (figure 3 (c)), all the available modules can be viewed at one glance and be (un-)loaded by single click. It offers the 'Scripts' tabs to also dive into the source code of the respective modules (figure 4).



NewModule

Enter a description here...

Version: 1 (Major), 0 (Minor), 0 (Fix)

Dependencies
- ☐ Core 1.0.0 (?)
- ☐ Editor 1.0.0 (?)
- ☐ Game 1.0.0 (?)
- ☐ Renderer 1.0.0 (?)
- ☐ Physics 1.0.0 (?)

(a)

Name	Game
Guid	a642641d-079e-4032-b71d-
Version	1.0.0
Author(s)	
Description	A prototype of a strategy game

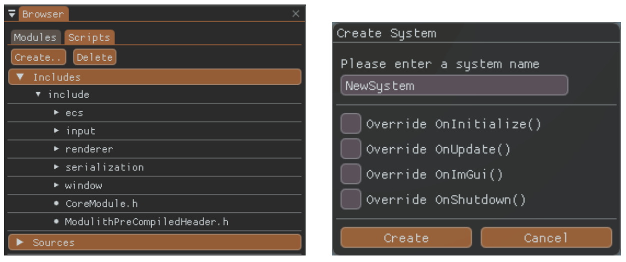
(b)

	Name	Status
✓	Core	Loaded dependants
✓	Editor	
✓	Renderer	Loaded dependants
✓	Physics	Loaded dependants
✓	Game	Loaded dependants
✓	Shooter	
☐	Mario	

(c)

Figure 3: (a) shows the module factory window to define new modules, (b) the properties that can be displayed for a selected module, and (c) the browser window listing the available modules. Together, these GUI elements implement R11 and R12.

Whole scene graphs of entities can be inspected in the *entity view* window (figure 5 (a)), whereas the *properties window* (figure 5 (b)) displays component data of selected components only. These windows have been inspired by the editor of the Unity Engine [28]. Lastly, the editor's *profiling window* (figure 6) displays the statistics of the engine's built-in profiler. These include the overall frames per second (FPS) and milliseconds needed to execute various game loop callbacks. As a result, this window can be used to profile and optimize any modules that might slow down gameplay.



- include
  - ecs
  - input
  - renderer
  - serialization
  - window
  - CoreModule.h
  - ModulithPreCompiledHeader.h

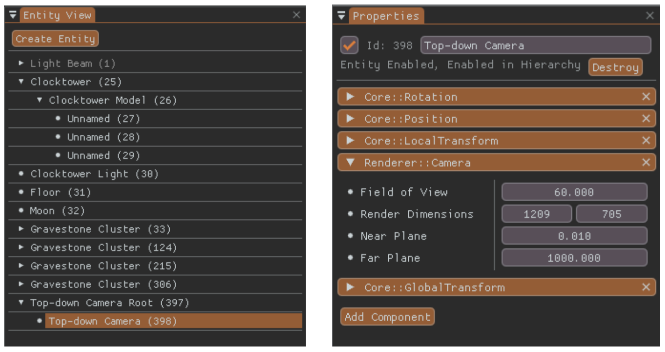
(a)

NewSystem

- ☐ Override OnInitialize()
- ☐ Override OnUpdate()
- ☐ Override OnImGui()
- ☐ Override OnShutdown()

(b)

Figure 4: (a) shows the scripts tab of the browser window revealing the file and folder structure of the edited module's code files, (b) a popup where the user can enter information for creating an ECS system (our solution for R10).



- Light Beam (1)
- Clocktower (25)
  - Clocktower Model (26)
    - Unnamed (27)
    - Unnamed (28)
    - Unnamed (29)
- Clocktower Light (30)
- Floor (31)
- Moon (32)
- Gravestone Cluster (33)
- Gravestone Cluster (124)
- Gravestone Cluster (215)
- Gravestone Cluster (306)
- Top-down Camera Root (397)
  - Top-down Camera (398)

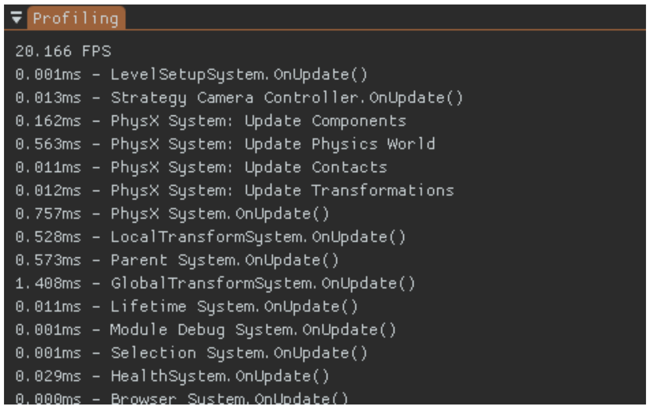
(a)

✓ Id: 398 Top-down Camera
Entity Enabled, Enabled in Hierarchy

Field of View	60.000	
Render Dimensions	1209	705
Near Plane	0.010	
Far Plane	1000.000	

(b)

Figure 5: (a) shows the entity view window can be seen, which visualizes the scene graph formed by the entities, (b) the properties that can be displayed for a selected entity.



```
20.166 FPS
0.001ms - LevelSetupSystem.OnUpdate()
0.013ms - Strategy Camera Controller.OnUpdate()
0.162ms - PhysX System: Update Components
0.563ms - PhysX System: Update Physics World
0.011ms - PhysX System: Update Contacts
0.012ms - PhysX System: Update Transformations
0.757ms - PhysX System.OnUpdate()
0.528ms - LocalTransformSystem.OnUpdate()
0.573ms - Parent System.OnUpdate()
1.408ms - GlobalTransformSystem.OnUpdate()
0.011ms - Lifetime System.OnUpdate()
0.001ms - Module Debug System.OnUpdate()
0.001ms - Selection System.OnUpdate()
0.029ms - HealthSystem.OnUpdate()
0.000ms - Browser System.OnUpdate()
```

Figure 6: The profiling window of the editor module shows the current FPS at the top, while the times for each game loop callback to various gameplay systems are listed below.

Figure 7 shows the default workflow of the editor. One starts the engine with at least the "Editor" module loaded. Pressing F3, one switches into the editor's UI where one can create a new module in the browser window. After setting the new module's name, version, dependencies and storage path, it is automatically loaded and activated for editing. At the same time, all the required CMake files and default code stubs are generated for the automated build process. The files can be accessed through the IDE, where the build process can also be triggered. In the editor's module browser, new script files for systems and components can be added to the new module. Typically one would add at least one system and several components. According files are generated and added to the build files as well. Arbitrary gameplay, low-level (sub-)engine code or modding code can be introduced in the new script files, which need to be manually registered in the module's code file, next. Assets such as 3D models, textures, animations, sounds, would be manually added to the mod's folder structure and loaded in code. The new version of the module can be compiled from the IDE and reloaded from the editor's browser or properties windows. One would repeatedly add more assets, systems, components, change their concrete interplay and implementations, recompile and reload to arrive at the desired mod.

5 CREATING A GAME, MOD-BY-MOD

In this section, we create and change a game mod-by-mod, implementing some of the mods outlined in Section 2, in order to demonstrate the flexibility of our proposed engine. In particular, we load the engine core, mod it with the required rendering and physics sub-engines, load a game, mod the game's view and mechanics, and assets.

The foundational *renderer module* implements ECS components and an ECS rendering system built on rendering abstractions provided by the core module. Entities can have point light, directional light, or mesh components attached and the rendering system will collect and render all of this data each frame. The *physics module* wraps the external PhysX library [19] to provide ECS physics components and a simulation system. Our basic implementation provides support for rigid bodies, box colliders, collision events, and character controllers.

For illustration purposes, we built a simple 3D tower defense game module which relies on the core, renderer, and physics modules. The player views the world from a bird's eye. Ghosts spawn on graveyards at the edges of the map. They move towards a clock tower at the center of map and try to destroy it. It is the player's task to prevent this from happening by using special abilities or by placing obstacles. Both activities cost resources, which results in the game's challenge of finding an optimal strategy to fight the ghosts. Screenshots of this game can be seen in figure 8.

In the Tower Defense module, every rendered object is an entity with components containing data for meshes and shaders. The ghosts also use the character controller components provided by the physics system to handle movement behaviour and collision logic. Additional ECS components and systems were created for managing the health of the ghosts and killing them when their health drops to zero. However, the information related to the game state, such as available resources, were not stored in an ECS component. Instead, these were managed by a custom *Subcontext* implementation, a class provided by the core module implementing a singleton pattern, as it was easier to access from other places in the code than a component. Lastly, the GUI that shows the available resources was implemented using the functions of Dear ImGui, also provided by the core module.

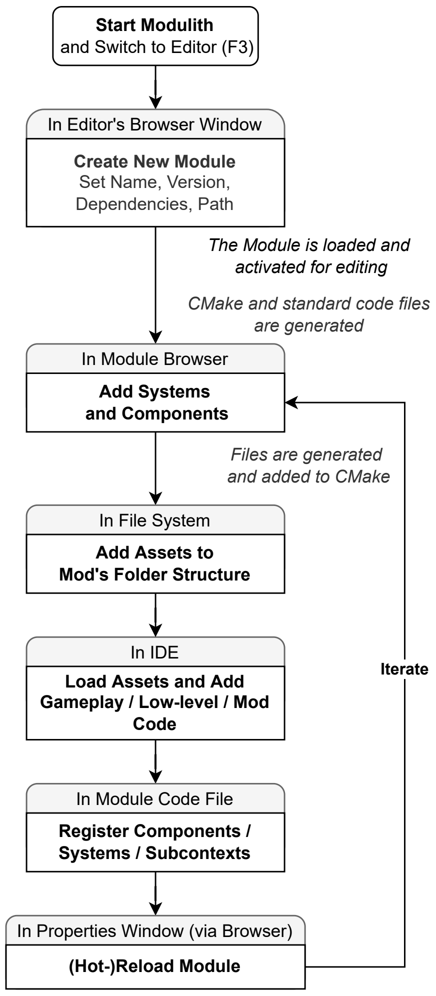

Start Modulith and Switch to Editor (F3)
In Editor's Browser Window
Create New Module Set Name, Version, Dependencies, Path
The Module is loaded and activated for editing
CMake and standard code files are generated
In Module Browser
Add Systems and Components
Files are generated and added to CMake
In File System
Add Assets to Mod's Folder Structure
In IDE
Load Assets and Add Gameplay / Low-level / Mod Code
In Module Code File
Register Components / Systems / Subcontexts
In Properties Window (via Browser)
(Hot-)Reload Module
Iterate

Figure 7: The default workflow for creating mods in Modulith.

We also created a mod for the tower-defense game, called the *shooter module*. It adds the ability for the player to take control of a character and fight the ghosts using a projectile weapon from the third-person view. The weapon has limited ammunition and needs to be reloaded. Additionally, the player can rely on a telescopic sight for better shooting accuracy. Screenshots of the character and its weapon can be seen in figure 9.

Similar to the game's module, the player character is also an entity with render components and a character controller component attached to it. Furthermore, ECS systems and components for the weapon were implemented. The projectiles utilize rigid body physics and interact with the already present health systems to deal damage to ghosts. Lastly, the ability for the player to switch between the bird's eye view and controlling the character was added by enabling or disabling the respective entities.

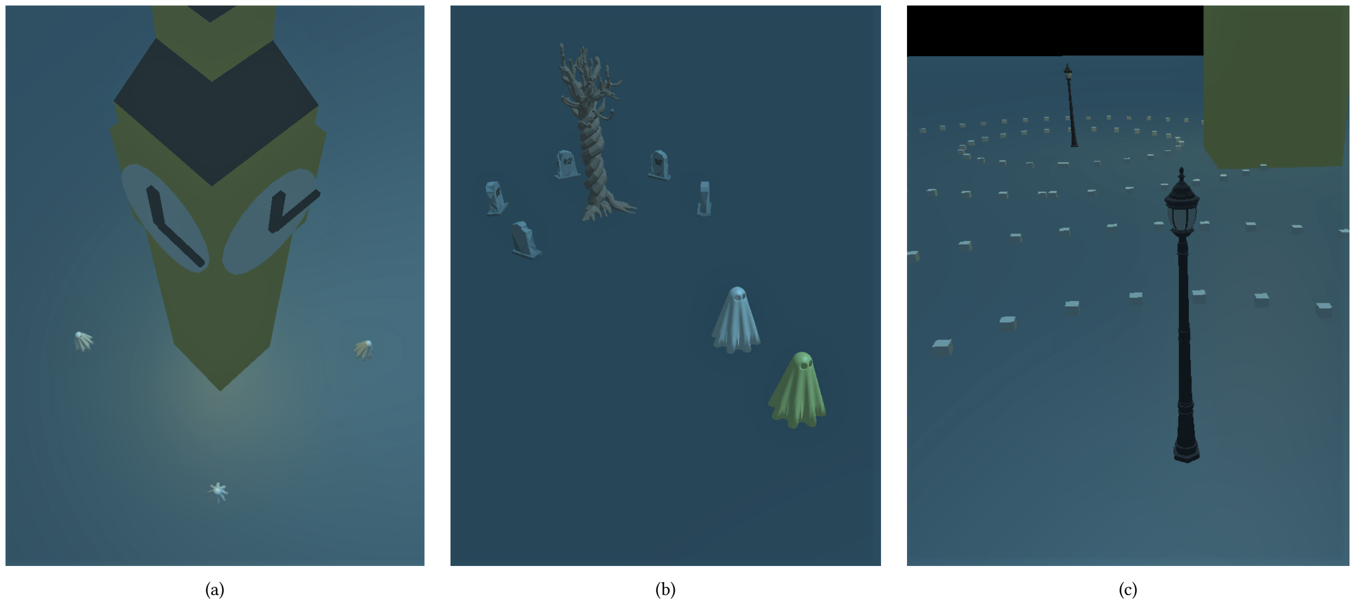


Figure 8: (a) shows a clock tower that needs to be defended from ghosts approaching from everywhere. (b) shows different types of ghosts (white ones are more robust, yellow ones faster) spawn on graveyards. (c) shows a lamp obstacle which damages close-by ghosts.

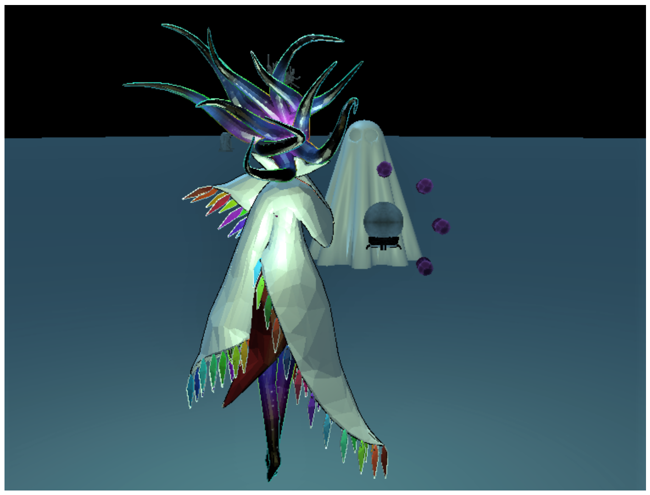


Figure 9: A player character seen in 3rd-person perspective, shooting (magical crystals) at a ghost.

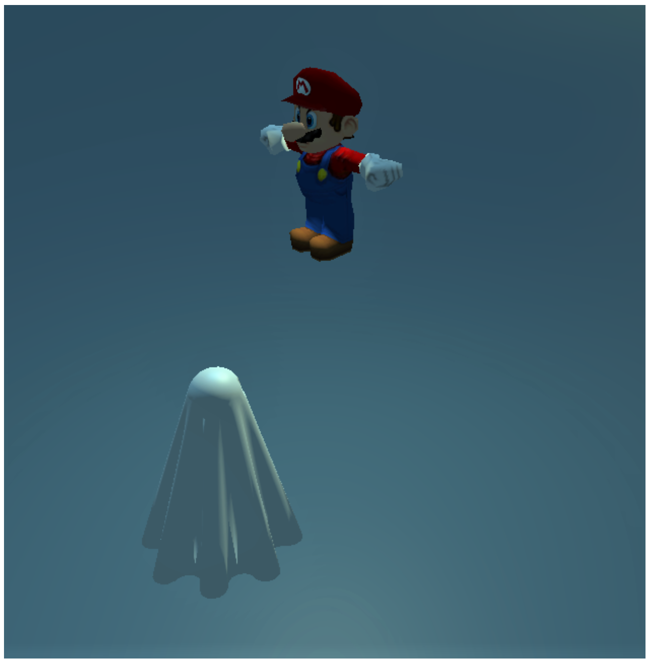


Figure 10: A Mario-like avatar overrode the previous player character of the shooter module. It can jump further and kill ghosts by hopping on their heads.

Finally, a mod for the tower-defense game with a dependency on the *shooter module* was created. It is called the *Mario module*, as it replaces the player's character added by the shooter module with a model looking like the famous jump and run hero Mario. It also increases the height of the player's jump and allows to kill ghosts by jumping on their heads, as can be seen in figure 10.

Just like the shooter module, the Mario module's gameplay is also implemented using the ECS by implementing a custom system that checks whether the player character collides with the heads of any ghosts. This module demonstrates that mods can even have a dependency on and modify the contents of another mod.

6 SUMMARY & FUTURE WORK

Based on developers', modders' and players' goals for modding, we formulated several requirements and developed, implemented and demonstrated the use of an according modular game engine, Modulith. It is open-source and freely available.

Modulith implements a powerful module system for loading assets and code at runtime, relying on dynamic loading of shared libraries. Modules may have dependencies to one another or to external libraries. To ensure backwards compatibility, we recommend to use non-inline code changes, additions to data and functionality, and deprecation warnings to fade out obsolete code over longer periods of time. Modules of arbitrary functional levels can be loaded—from low-level sub-engines over systems for game mechanics and game design specifications to small mods only introducing novel assets or changing minor gameplay aspects. Each of these modules can then be built independently, without having to recompile dependant modules. In the previous section, we explained and demonstrated the use of Modulith by providing two low-level modules for rendering and physics, a game module and two gameplay and asset modifications. These modules were built on top of the engine runtime that loads the engine module with its inherent extension infrastructure (e.g. dependency resolution of modules and provision of subcontexts) as well as the core module (e.g. provision of an ECS architecture and an asset management system). For fast modding cycles, Modulith's graphical editor hides a large amount of the underlying complexities of code generation, module integration and execution of the necessary build processes.

For potential future work, there are several important directions to work towards. First, there are numerous extensions and improvements of the current implementation of Modulith, as basic as the integration of a sound sub-engine module, advanced support for input modalities, or an improved support for data-oriented ECS organisation of sequential memory layouts. Given its inherent modularity, we could envision sharing different mod cascades and branches that might emerge from communities tailoring to specific needs such as high-quality rendering adventure games or very fast shooter games. Second, user experience and usability studies on Modulith could help to gain insights in how to further improve this system and what to specifically consider in next research steps in general. Third, albeit opposing the very principle of Modulith, we also see the necessity to proactively limit moddability for some aspects of games to ensure, for instance, that players enjoy the gameplay experiences as intended by the designers or that competitive multi-player games are fair. Fourth, we strongly believe that modding in itself deserves more research efforts both from a social and design perspective but also from a technology perspective. To this end, a more rigorous analysis and classification of different types of mods alongside their potential uses could be conducted. We hope that our research can contribute to these valuable potential future endeavours.

REFERENCES

[1] Michael Abrash. 1997. *Michael Abrashs graphics programming black book*. Coriolis Group Books, United States.

[2] Sean Barrett. Last Accessed Online: November 2022. stb: single-file public domain (or MIT licensed) libraries for C/C++. https://github.com/nothings/stb

[3] Jesse Beder. Last Accessed Online: November 2022. yaml-cpp: A YAML parser and emitter in C++. https://github.com/jbeder/yaml-cpp

[4] Bethesda Softworks. 2011. The Elder Scrolls V: Skyrim. https://elderscrolls.bethesda.net/en/skyrim

[5] Bethesda Softworks. 2015. Fallout 4. https://fallout.bethesda.net/en/games/fallout-4

[6] Doug Binks, Matthew Jack, and Will Wilson. 2019. Runtime Compiled C++ for Rapid AI Development. In *Game AI Pro 360*. CRC Press, Boca Raton, 155–171. https://doi.org/10.1201/9780429055058-12

[7] Grady Booch, Robert A Maksimchuk, Michael W Engle, Bobbi J Young, Jim Connallen, and Kelli A Houston. 2008. Object-oriented analysis and design with applications. *ACM SIGSOFT software engineering notes* 33, 5 (2008), 29–29.

[8] Anton Bukov. Last Accessed Online: November 2022. boolinq: Super tiny C++11 single-file header-only LINQ template library. https://github.com/k06a/boolinq

[9] Camilla Löwy et al. Last Accessed Online: November 2022. GLFW: Open Source, multi-platform library for OpenGL, OpenGL ES and Vulkan development on the desktop. https://www.glfw.org/

[10] Eelke Folmer. 2007. Component Based Game Development – A Solution to Escalating Costs and Expanding Deadlines? *Component-Based Software Engineering Lecture Notes in Computer Science* 4608 (2007), 66–73. https://doi.org/10.1007/978-3-540-73551-9_5

[11] Rob Gallagher, Carolyn Jong, and Kalervo A Sinervo. 2017. Who wrote the elder scrolls?: modders, developers, and the mythology of Bethesda softworks. *Loading... The Journal of the Canadian Game Studies Association* 10, 16 (2017), 32–52.

[12] Jason Gregory. 2019. *Game engine architecture*. CRC Press, Taylor & Francis Group, Boca Raton.

[13] Christophe Groovounet. Last Accessed Online: November 2022. OpenGL Mathematics. https://glm.g-truc.net/0.9.9/index.html

[14] David Herberth. Last Accessed Online: November 2022. Multi-Language GL/GLES/EGL/GLX/WGL Loader-Generator. https://glad.dav1d.de/

[15] Graeme Hill. Last Accessed Online: November 2022. CrossGuid: A minimal, cross platform, C++ GUID library. https://github.com/graeme-hill/crossguid

[16] Gisli Hjalmtysson and Robert Gray. 1998. Dynamic C++ Classes-A Lightweight Mechanism to Update Code in a Running Program. In *USENIX Annual Technical Conference*, Vol. 98. USENIX Association, New Orleans, 65—-76.

[17] Kim Kulling. Last Accessed Online: November 2022. The Open-Asset-Importer-Lib. https://assimp.org/

[18] Gabi Melman. Last Accessed Online: November 2022. spdlog: Very fast, header-only/compiled, C++ logging library. https://github.com/gabime/spdlog

[19] NVIDIA Corporation. Last Accessed Online: November 2022. NVIDIA PhysX: Open source scalable multi-platform physics simulation solution. https://developer.nvidia.com/physx-sdk

[20] Omar Ocornut. Last Accessed Online: November 2022. Dear ImGui: A bloat-free graphical user interface library for C++. https://github.com/ocornut/imgui

[21] Carlo Pinciroli, Vito Trianni, Rehan O'Grady, Giovanni Pini, Arne Brutschy, Manuele Brambilla, Nithin Mathews, Eliseo Ferrante, Gianni Di Caro, Frederick Ducatelle, and et al. 2012. ARGoS: a modular, parallel, multi-engine simulator for multi-robot systems. *Swarm Intelligence* 6, 4 (11 2012), 271–295. https://doi.org/10.1007/s11721-012-0072-5

[22] Tom Preston-Werner. Last Accessed Online: 2023. Semantic Versioning 2.0.0. https://semver.org/

[23] Lukas Schreiner and Sebastian von Mammen. 2021. Modding Support of Game Engines. In *The 16th International Conference on the Foundations of Digital Games (FDG) 2021* (Montreal, QC, Canada) *(FDG'21)*. Association for Computing Machinery, New York, NY, USA, Article 36, 9 pages. https://doi.org/10.1145/3472538.3472574

[24] Milan Stevanovic. 2014. *Advanced C and C compiling*. Apress, Berkeley, CA.

[25] SureAI. 2016. Enderal. https://sureai.net/games/enderal/?lang=en

[26] The Khronos Group. Last Accessed Online: November 2022. *Vulkan*. The Khronos Group. https://www.khronos.org/vulkan/

[27] Alan Thorn. 2011. *Game engine design and implementation*. Jones & Bartlett Learning, Sudbury, MA.

[28] Unity Technologies. Last Accessed Online: November 2022. *Unity*. Unity Technologies. https://unity.com/