universität
wien

Bachelorarbeit

Debugging Console for VECS

Verfasserin

Marlene Lucia Kasper

angestrebter akademischer Grad

Bachelor of Science (BSc)

Wien, 2025

Studienkennzahl lt. Studienblatt:	A 521
Fachrichtung:	Informatik
Betreuerin / Betreuer:	Univ.-Prof. Dipl.-Ing. Dr. Helmut Hlavacs

Abstract

In the Vienna Vulkan Engine (VVE), starting with Version 2, a Entity Component System was introduced. This Vienna Entity Component System (VECS) opens the doors for data driven game design using the VVE. Since a running application using VECS can possibly handle hundreds of thousands of entities, debugging by simply printing the values can become difficult. In this work, a debugging Console for VECS was implemented. Using C++ and DearImgui for the Console and TCP and JSON for data transfer, it is possible to directly get access to the data and display the entities in tabular form to inspect their properties and even monitor the program in a live View to watch the changes in the data while they happen. During evaluation, the Console showed good usability for Entity counts up to a million, depending on the complexity of the used components. The Console proves to be a useful debugging help for typical game development use cases, but also shows potential to be sped up more in future work, especially in the data parsing.

Contents

1 Introduction **4**

2 Related Work **5**

3 Foundations of Entity Component Systems **8**

4 Implementing a Debug Console and UI for VECS **11**

4.1 Adaptations for VECS 12

4.2 Console 15

5 Evaluation and Discussion **21**

5.1 Simple Entities 21

5.2 Complex Entities 23

6 Conclusions and Future Work **24**

1 Introduction

The Vienna Vulkan Engine is a Gaming Engine developed at the University of Vienna for educational purposes for the students at the Department of Computer Science. This Engine is based on Vulkan [1] and C++. In the new version of the Vienna Vulkan Engine, an Entity Component System was introduced, the Vienna Entity Component System [19]. The Entity Component System is a architectural pattern that allows for dynamic data composition and meets the requirements of modern game development, using entities, represented by handles and components. The system is using this structure to iterate over the entities efficiently. This ECS opens the way to data oriented design in the game development for the students of the university. But using an ECS also brings along new challenges, especially in debugging. When the used data is saved in entities and their archetypes, the number of entities can grow fast. If thousands of entities are used, then printing them to a console for debugging purposes is not feasible anymore.

In this bachelor's project, a debug console and UI for the Vienna Vulkan Engine was created. This poses the following research questions:

- What Data needs to be displayed in a debugging Console for an Entity Component System (ECS) to ensure a structured and detailed debugging process?
- Is the Console efficient enough to accurately display the data in a timely manner to aid debugging, in both local and remote Use Cases?

Based on these research questions we arrive at the following hypotheses:

- A debugging Console using visualization of data helps in debugging and relevant runtime metrics can help in finding data leaks and other problems.
- Using optimization in data aggregation and network protocol, a console can be used in both remote and local context to debug small and big applications using the VECS.

By addressing these research questions and hypotheses, in this bachelor's project a debug console and UI for the Vienna Vulkan Engine was created. The console allows all data to be displayed as a tabular snapshot and also to display the current entities in a live view to show possible correlations in the application. This is intended to facilitate debugging of larger VECS projects. Due to caching in the Console, displaying big snapshots with over hundred thousand entities and scrolling over them to inspect them is possible without lag. The Console is implemented in C++ with ImGUI for the UI, using a TCP connection and JSON for bidirectional data transfer between the running VVE application and the console. The VECS application will connect automatically, but only if the application is started in debug mode. In release mode, the Debug Console is not used, since that data aggregation and transfer inevitably affects the performance. If necessary, however, it can be activated by adding a single code line.

The tests show good usability for VECS applications using up to a million entities, when tested with simple entities. The send and parse time are low enough to ensure accurate data for debugging and monitoring the data in the live View. For up to 100,000 Entities transferred, the lag while receiving a snapshot is negligible. For complex entities, the total time rises, but poses good usability up to a million entities. Gathering the data and parsing it in the Console takes longer when the entities all contain complex data types. It is also possible to use the Console over a network connection to debug from a different device. This was tested with complex entities and has shown that it is possible for small projects, but not advisable for projects with over 100,000 entities. In future work, the data accumulation process on the VECS side could still be optimized, to minimize lag and ensure accurate data also for large program dimensions. Also, the translation of incoming data from JSON into the data structures maintained internally by the Console has potential to be sped up further.

2 Related Work

In this chapter we will explore the following questions: What is an Entity Component System? How much scientific research was done about them, what are their usual applications and to which extent can they be used outside of classical game development?

Getting an overview of what an ECS is and where it came from is the first step to understanding ECS programming. Finding literature on this topic is not easy, since it is rather new and niche. In *Advantages and Implementation of Entity-Component-Systems*, Toni Härkönen provides this overview, having researched the history and different possible implementations of the Entity Component System pattern [17].

While researching more papers on the topic of ECS and its applications, one paper that stood out was *Mapping Study of the Entity Component System Pattern*. In this paper, a mapping study was done to gather more scientific papers about ECS and to produce an overview of state-of-the-art literature. By doing this, they show a lot of different possible applications for the ECS pattern and it was a valuable source in finding more papers about ECS [35].

When looking for scientific papers about Entity Component Systems, it is common to come across papers about implementations of different styles of ECS. As a good example, in 2014 Vittorio Romeo built a multi threaded compile-time C++14 library called *ECST* after analyzing common entity encoding techniques used in Entity Component System programming. The goal is to make the programmers' life easier by letting them define the high level program logic in a declarative way. Like many other Entity Component Systems, the ECST is header only, making integration in programs easy. Benchmarks and analysis also shows the benefits of multithreading in ECS[32].

In ECS, there are multiple possible approaches to use. The VECS used in this bachelor thesis is an Archetype ECS. In a paper by Cox et al., the run time performance between a sparse set ECS and Archetype ECS was compared. For this comparison, the architecture of each ECS was developed in C++20 and showed that the sparse set approach allows for cheaper modifications, but runtime worsens upon scaling whereas archetype based ECS show higher cache efficiency but also higher costs while changing compositions [5].

To improve the knowledge and understanding of the ECS pattern, the *Core ECS* formal model was built. It abstracts away from the implementation details to reveal the essence of the ECS Pattern. Using Core ECS as a point of comparison, several real-world ECS frameworks are surveyed and found that they all leave opportunities for deterministic concurrency unexploited [31].

A basis of Entity Component Systems is Data Oriented Design. DOD is not just a set of optimizations, but a programming paradigm of its own. Richard Fabian describes this in *Data-Oriented Design*. The goal is to lead to more efficient, testable, and maintainable software by reducing complexity and make the program defined by its data[8].

The beginning of data oriented design dates back to as early as 1980. Since efficiency in programming has always been a big topic in Computer science, a new approach was presented as an alternative to the traditional von Neumann computing to increase this efficiency with the changing demands. This new approach was called Data Oriented Design, with the intention of increase processor efficiency and memory utilization [33].

The ECS pattern is one of the most common design patterns using Data Oriented Design. At its core, DOD is about designing programs around data input, the transformations performed on the data, and the data output. For developing Game software, using Data Oriented Design offers a lot of benefits, like easily accomplished data parallelization and improved hardware usage[2].

Data Locality always has an influence on the efficiency of a program. In many cases, a data oriented approach will yield better results than an object oriented approach. This was shown also in a paper by implementing an Entity Component System using the open source library Ophidian. In this case, it was shown that the data oriented ECS approach delivered a significant speed up in comparison to

the object oriented approach [12].

The most common area of application for ECS is game design. The following papers all examine the uses of the pattern in game development in different areas and specific uses.

The ECS pattern offers more flexibility for programming, and these advantages are what makes it popular in game design. In *ECS Game Engine Design*, these advantages are shown with the examples Cupcake and Artemis, two popular Entity Component Systems with different architectures. Based on these two architectures, the author proposes a new ECS Design to use the strengths of both these ECS frameworks [15].

Another good way to use the strengths of ECSs is to use the Entity Component System pattern in a Framework for game development. The *TRPGFramework* is based on Unity3D and built on the ECS principles; it utilizes some of the most effective elements of current Frameworks and is easy to control and modify. Using the pattern also increased the scalability, code reusability and modularity while coding [38].

ECS is mostly used in game development, where it is very popular for its expandability. In *Konzeption und Evaluation eines Entity-Component-Systems anhand eines rundenbasierten Videospiels*, the author wrote a game using the ECS *Ashley*. Ashley is a Java based Entity Component System Framework. The work showcases that an appropriate choice of ECS makes game development easier to accomplish [16].

Having a Debug Console for your Game Engine is a practical tool. Writing a custom one is also a good way to ensure all the needed data is displayed. In *Modulith: A Game Engine Made for Modding*, a paper describing the development of a modding game Engine, the author also wrote a custom Debug Console for this cause. For the Game Engine, the author also used an ECS pattern. Also using Dear ImGUI for the Debug Console, it does share some similarities with the Console presented in this work, but it differs greatly in the displayed data layer. The Modulith Debug Console shows more high level data, while the Console in this paper is more low level and shows the data values directly [14].

Another angle for game development that was examined is accessibility. On one hand, a paper found that the ECS does not bring added value to the accessibility of video games, but on the other hand it does very well with carrying out transformations on a large number of game objects. It is less likely to cause errors and the modularity of the ECS makes it easier for the programmers to identify the parts of the programming that need to be modified to increase accessibility for players with individual needs [24].

While the application of the ECS pattern in Game design and development is the most common occurrence, the pattern also holds value for other areas. The following papers show how the advantages of the ECS pattern can also enhance code not meant for video games.

For example, it was used to build the *Polyphony* Tool Kit for Graphical User Interfaces, making use of the polymorphism of elements due to the entities and utilizes the components to enhance the Selection. The Systems oriented programming helped to improve the management of interactive behaviors[30].

When using Object Oriented paradigms in a program, there is always coupling introduced. To help with this issue, it is possible to use the concept of semantic traits to enable reuse in the ECS context by decoupling objects from their class definition where possible. This approach uses ECS pattern techniques to make the programming stay flexible while also allowing detection of an entity's class affiliation at runtime [36].

The architecture of Real Time Interactive Systems (RIS) is determined by coupling and cohesion. These contradicting principles can both be satisfied by using ECS and in a paper, six semantic based approaches to expand the established ECS pattern were presented to pose a solution to this without causing the code to lose maintainability [10].

Real Time Interactive systems can have high memory demands. Using new memory management approaches for the used Entity Component System, it is possible to improve performance and efficiency.

This paper implements wait free hash maps, both centralized and decentralized, to the ECS pattern to efficiently reduce memory usage. In the centralized approach each system notifies a central hash map of changes and in decentralized each system manages its own hash map [21].

Entity Component Systems also find their place in VR usages. By using different techniques to extend the ECS patterns capabilities for Real-Time-Interactive Systems, their goal is to increase software quality. The techniques presented in the paper focus on semantic aspects and also decoupling [9].

In *Project Elements*, the goal was to build a lightweight, open-source computer Graphics framework for educational purposes. It uses the Entity Component System pattern to make use of its benefits like flexibility and memory usage and also offers the convenience of a Scenegraph-based pythonic framework. Using Python in this area is a good choice for teaching purposes [26].

There are also more abstract possible uses of the ECS pattern. It was, for example, used for a Cloud Based Simulation Service for 3D Crowd Simulations. Using the Entities as members of the crowd to be simulated, the simulation can work the transformations on the data and have simpler access to the data [27].

Vico, a co-simulation framework running on the Java Virtual Machine, was built on the ECS pattern. The flexibility and extensibility proved by the ECS showed good benefits for co-simulation purposes. The resulting framework Vico was proven to be effective compared to other open source co-simulation tools [18].

The Entity Component System pattern can even provide benefits for compiler and interpreter design. Here, while it does not offer the same consistent performance advantages like it does in the context of game development, the modularity and clear differentiation between data and system logic makes writing optimization passes easier and enhances the serviceability of the system [6].

For the goal of speeding up code, the ECS pattern shows advantages. The *WONDER* software suite is a network controlled, real-time 3D audio rendering Software. In *Re-implementation of a Real-Time 3D Audio Rendering Software Using an Entity Component System Architecture*, this software suite was re-implemented using a data oriented paradigm. The resulting software *CoRGII* implements the features using a sparse set ECS [37].

Another possible use of the ECS is presented in a modular and scalable approach for an IoT broker based on the ECS pattern. An IoT broker application using EnTT as the ECS Framework was created and benchmarked to explore the usability of the pattern in IoT-Brokers. In the paper, this implementation is evaluated and shows potential to improve bandwidth usage, power efficiency, topic management and more [29].

Even in microarchitecture research, the ECS can be implemented to achieve improvement and speed ups. Instruction pipelines of existing processor simulators are not designed to be easily modified. One approach to change this is to adopt the ECS pattern into the pipeline and memory subsystem design. In the cycle-accurate processor simulator *Phalanx*, the ECS pattern was used to improve the simulation speed with success [22].

Beyond the classical use cases of the Entity Component System, the pattern also has been investigated in more specialized technical areas.

The Entity Component System pattern also has its possible uses in Underwater Remotely Operated Vehicles (ROVs). Here, a modular and generalizable Entity Component System with automatic state synchronization was used for the controlling of an underwater ROV. The System has enough flexibility, is adaptable, and shows good usability for robotic underwater applications [23].

Traditional simulation tools, especially if they are intended to be used for cyber-physical power systems (CPPS), are at a disadvantage when using an object-oriented approach. For CPPS, a more flexible and scalable data oriented approach is needed. In a paper, the ECS-Grid was created, the first data-oriented real-time electromagnetic transient simulation platform for CPPS. This ECS Grid makes use of the advantages of the ECS, namely the high flexibility, high extensibility and high scalability

[3].

The usability of the ECS pattern can also be applied to city infrastructure, especially in the energy sector. One proposed approach is to implement a high performance Energy Services Interface with database architectures for real life application. This proposed replacement of databases with ECS systems is hoped to deliver performance advantages. The results of the paper are a bit limited, however, since prior to the writing of the paper no close comparison of performance between databases and Entity Component Systems exists in research [34].

3 Foundations of Entity Component Systems

Entity Component Systems were introduced to help the dynamic requirements in game development, where new game objects, e.g. enemies or item types, can be introduced at runtime, and in Object Oriented programming can cause problems in defining the properties of the classes. In classic Object Oriented Programming, rigid hierarchies and inflexible designs can sometimes lead to the "diamond of death". This means that to have all functions and variables needed in an Object, a lot of inheritance and bubble up functions have to be used. The Diamond Problem happens when a class inherits from multiple base classes that share the same ancestor, creating a diamond-shaped inheritance structure. This causes ambiguity because the derived class ends up with more than one path to the same member or method from the common ancestor. As a result, the program can not always tell which version to use, leading to confusion and potential errors in method calls or data access [13].

Underlying to ECS frameworks is Data-Oriented Design. In data oriented design, the efficient transformation and organization of data is given priority over abstract object models. We try to minimize cache misses and the goal is to maximize CPU cache efficiency. In contrast to Object Oriented Programming, in DOD we use the cache of the processors which are performing transformations on the data. These caches deliver data access at a very high speed. Like this, the data is also stored more closely together which speeds up execution. An Entity Component System uses these principles to store data more efficient and makes the creation of more individual objects possible. ECS break up rigid data structure of Object Oriented Programming. The goal of an Entity Component System is to separate the data (Components), the behavior (Systems) and the identity (Entities) to make the code more flexible and less rigid. The Entity Component System is an architectural pattern used primarily in video game design. It allows for dynamic and flexible data composition, prioritizing data oriented design over inheritance.

An entity component system is made of:

- Entities : objects in the program, for example a player character or a light source. An Entity is defined by its Components. It is usually referenced by a simple ID, also called handle. This unique handle makes it easier to find.
- Components: a component is pure data. It can be an integer, string, or any other data type. These are pure data structures and hold no logic. If needed, you can also define a `struct` to hold more values, making it very customizable and adaptable.
- Systems: the logic using the ECS. It iterates over all entities containing specific components or a specific combination.

Using this pattern, a program becomes way more flexible, an ECS also enhances the scalability and performance. When using an ECS in a program, we save our (game) objects as entities in the archetypes. With these specific sets and combinations of data, we can efficiently iterate over the data while keeping the objects flexible to change in the data.

An ECS represents data as entities with components. These entities are divided into archetypes depending on which components they have. "With ECS, an entity just becomes an index that lets you

look up components assigned to that entity.” [4] This simplifies data representation for large projects and dynamic data type changes in entities are made possible. With an ECS, we can create flexible data based on composition instead of inheritance. To reach this goal effectively, there are many approaches. All are similar at their core but use different angles to address the same problem. One of these approaches in ECS programming is archetype-based entity component systems. ECS frameworks like Flecs and VECS use these archetypes. Many ECSs and also the VECS which is used and expanded use slotmaps to keep track of its handles and components:

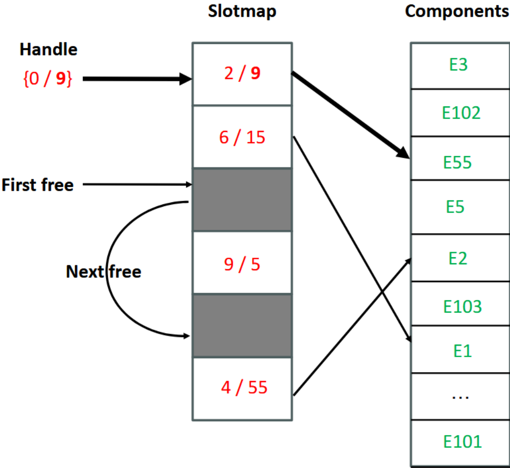

Figure 1: The Slotmap [20]

As shown in 1, the handles stored in this slotmap have two parts. It stores the handle and the version of it, so that if a component gets added or deleted, no reference will be left pointing into nothing. The slot map provides stable handles and is efficient in allocation, lookup and deallocation. Furthermore this slotmap should only grow and not shrink. If an Entity gets deleted, its place in the slotmap becomes free. The next new entity will be saved in this new free place.

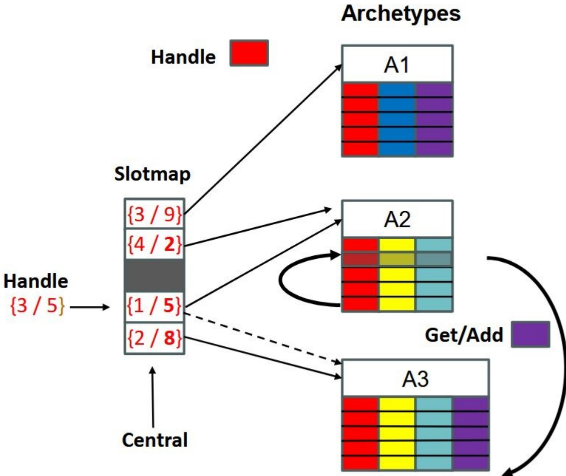

Figure 2: Diagram showing Archetypes [20]

An Archetype is a collection of a certain combination of components. All the Entities in one Archetype have the same component types and tags. If an Entity gains a component, or if a component gets deleted, the entity no longer belongs to this specific archetype and gets moved to the archetype with matching component types, as shown in 2. If such an archetype does not exist yet, it gets created. This ensures fast iteration over entities with the same components and orders the Data. The Slotmap maps handles to archetypes and an index in the archetype for the components. In this way, all entities with identical components are automatically grouped, making it easy to iterate over them when needed. This approach ensures cache optimal iteration and fast insertion and deletion of components. There is an overhead due to archetype management.

Flecs is a lightweight and open-source Entity Component System using the archetype approach, with a zero-dependency C99 API. It does not use STL containers and is quick to install and use. Flecs was the first open-source ECS to fully support entity relationships and has native support for hierarchies and prefabs. It comes with an integrated reflection framework, a JSON serializer, runtime components, unit annotations and a statistics add-on for profiling ECS performance. On top of that, Flecs provides a web-based UI to inspect and control the entities in use [11]. A different approach in entity component systems is using sparse sets. One ECS using this technology is EnTT. A sparse set uses two main arrays: a sparse array, where the entity ID serves as the index, and a packed array, which stores the actual component data.

EnTT is a header-only ECS, which makes it easy to install and use in a program. It is battle-tested in Minecraft. A main difference from Flecs is that it uses sparse sets. These sparse sets make the program faster when inserting only a few components [7], faster when inserting few entities, and faster when changing components. Flecs and VECS are slower when changing components but faster when inserting many entities because of archetypes [11].

Another Entity Component System is pico. Pico is a minimalist ECS, it is header only in a single file. Instead of using archetypes like Flecs or sparse sets like EnTT, pico uses arrays and other simple minimalist data structures to store the data. This ECS shows that also simple mechanisms can be used in Data Oriented Design. The level of complexity depends on the use case of the ECS. In many cases, these simple data structure will be sufficient [28].

The Vienna Entity Component System, abbreviated as VECS, is header-only, written in C++20, and supports multithreading. [19] It is used in the new Version of the Vienna Vulkan Engine used by the students for educational game programming. The VECS ensures that the data structures will not get too rigid and avoids problems like the diamond of death. Instead of using inheritance, it uses Entities and Components to composite the data needed for a game. Like this, it is possible to more dynamically get the data you need with less cache misses. By using the VECS in the Vienna Vulkan Engine, the students also learn to use Entity Component Systems, which are used more and more in the professional world. The Vienna Entity Component System is using the same archetype based approach like Flecs. It compiles all entities using the same unique component combinations into archetypes to iterate over them.

In these three different examples of ECS we see three different versions of data storage. Flecs, like the VECS, uses archetypes to order the data, while EnTT uses sparse sets and pico uses arrays and minimalist logic. Among these examples, only Flecs comes with a UI to visualize data and assist with debugging. This UI is web based and can also handle queries of user input. When dealing with large amounts of data, proper visualization becomes essential; simply printing all entities to a console is not enough. Instead of creating a web-based UI, in this thesis I built a console that runs locally on the computer, using TCP to transfer the data directly from the running programm to the console for visualization and monitoring.

4 Implementing a Debug Console and UI for VECS

The goal of this thesis is to implement a Debug Console for VECS Applications. It should connect to a running VECS Application, request data and display it for debugging purposes, using snapshots and a Live View. For the snapshot I tried to display the data in the clearest and most structured way while not distracting the user. For this, a tabular approach seemed the most reasonable, especially since the subject matter in this bachelor thesis is the data of an ECS. The design went through a few iterations, and then was finalized with the concrete decision of the technologies to be used. ImGui gave a lot of structure in the design process and shaped the final look. In 3 the first design of the snapshot window is shown.

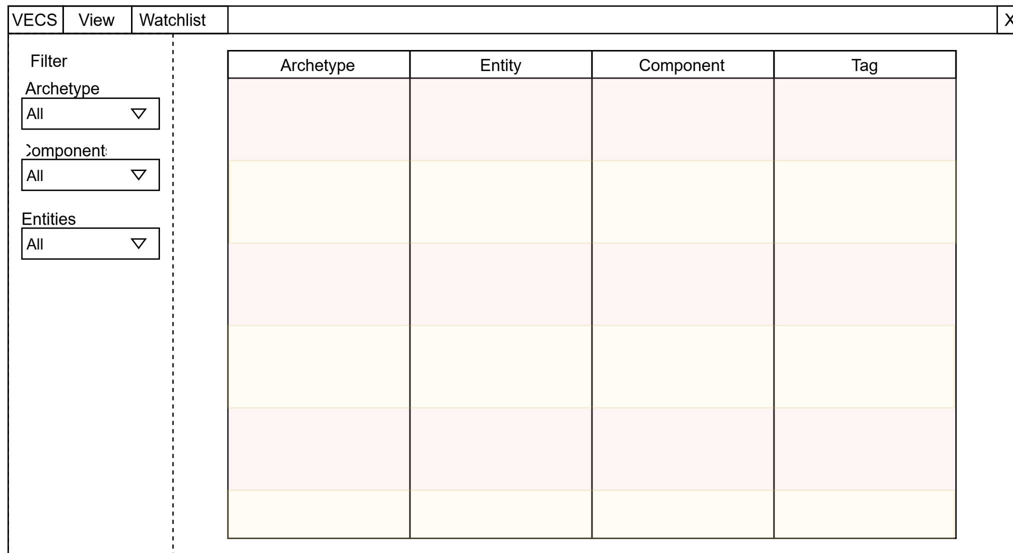

Figure 3: Mockup of the Snaphot window

To make the data visible and still comprehensible, a tabular approach was chosen. In the middle of the window, the plan is to show all entities and their component data ordered by their archetypes. On the left, some space is reserved for filters to be able to look for concrete data. The displayed data columns also went through a few changes after gaining a deeper understanding of the VECS and its inner workings. In the implemented version, the column "Component" is "Typename" and displays the data type of the Component, the value is then displayed in an additional column "Value".

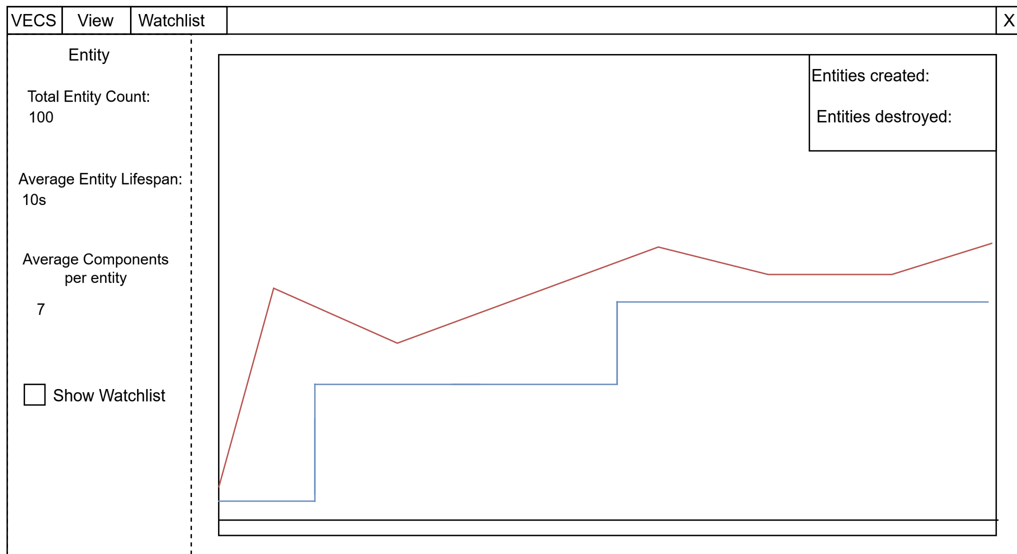

Figure 4: Mockup of the Live View window

The Graph in the mockup of the Live View as shown in 4 was also redesigned after insights into the VECS and ImPlot. In the finished version, instead of a line graph, a bar graph is used, to display the number of entities in a more concise way. After all, the number of components only rises in full numbers, and it is not possible to have 1.5 entities. The second line in the mockup was planned to show current memory usage of the VECS application. For usability and technical reasons, this idea was transformed to show the estimated memory usage with the statistics below the graph. Also the statistics displayed under the live view were changed to statistics that match the technical possibilities without slowing down the program too much, and moved below the graph, so that the graph has more room to display.

The console makes it possible to establish connections to any number of VECS programs and thus examine the data in various ways. The console is implemented platform-independently and has been tested under Windows and Linux. Communication is based on a TCP/IP connection. Both the port and IP address are configurable for this connection — on the VECS side, the port and IP address, and on the console side, the port only. This makes it possible to establish communication across different devices, which also enables debugging from a secondary device. The main use case of the console is to communicate with programs that use the Vienna Vulkan Engine; therefore, the console is built on Vulkan and SDL3. For the user interface, Dear ImGUI (docking branch) is used. This allows dynamic arrangement of windows to adapt the interface to individual needs. Using the TCP connection, the data is sent from VECS to the console as a JSON object, which is then translated by the console into its own data structure. JSON was chosen for this to maintain human readability. In the console, it is possible to save the snapshot to a file, containing a JSON version of the snapshot.

4.1 Adaptations for VECS

The VECS header files had to be adapted to make the connecting and sending data to the console possible. While writing the adaptions, the goal was to keep the changes to the existing code to a minimum and to keep interference with the execution of the VECS application as low as possible. In the main header file of the VECS project, `VECS.h` the following helper functions were added to connect to the console and send data.

- `std::string ToJSONString(std::string s)`

– Added as a general method to convert a string into JSON format.

- `template<typename T> std::string ToJSON(const T& value)`
  – For the retrieval of the Components of an Entity

In addition to that, the two concepts HasToString and Streamable have been added to convert the contents of any value into JSON format. This is suitable for primitive values, such as integers and floating point values. They are converted into numbers, whereas most other types are converted into JSON strings, wherever this is possible. If a value has a type that evaluates to a class which either has a to_string() method or, if that is not available, can be streamed to an std::ostream, these methods are used. Also, a forward declaration of the VECSConsoleComm class and the function GetConsoleComm() had to be added, since there's an inter-dependency between the additions in VECSRegistry.h and the newly added VECSConsoleComm.h.

To get the needed data and also the statistics for the live view, some changes to `VECSArchetype.h` were also implemented:

- `std::string ToJSON(size_t aindex)`
  – Added as a way to convert the encapsulated contents of an entity stored in the archetype to JSON.
- `size_t GetComponents()`
  – Added as a way to get easier statistics about the total number of components currently handled by an archetype.
- `size_t GetEstSize()`
  – Added to get the estimated size of all entities in this archetype.
- `std::string ToJSON()`
  – Added to convert the complete archetype into a JSON string

The size calculated in `size_t GetEstSize()` is an estimate only; if the archetype holds components of complex types, structures or objects that manage additional data, such as an std::string, for example, the size of these additional data is not taken into consideration. Only the size of the stored components themselves is calculated. Also the JSON string created in `std::string ToJSON()` includes all managed entities by the archetype, so it can potentially be very large, depending on the amount of components and their complexity. This JSON string is then used when transferring a snapshot of the registry to the Console. The following method has been added to the VECS HandleT structure in `VECSHandle.h`:

- `std::string ToJSON()`
  – Added to convert a VECS::Handle into a JSON string.

In the `Registry()` constructor, in `VECSRegistry.h`, a call to `GetConsoleComm()` has been added for debug builds, where the Console is expected to be used most often. This connection uses the default values (connect to localhost at 127.0.0.1, port 2000). This call is the reason for the forward declaration of `GetConsoleComm()` in `VECS.h`. When the application using VECS is built in Release Mode, it does not connect to the Console, since the Debug Console is intended exclusively for debugging purposes. Also some new Methods have been added to the `VECSRegistry.h` to support Console communication and functionality:

- `std::string GetLiveView()`
  - Added to return basic information in LiveView mode as a JSON string.
- `float GetAvgComp()`
  - Added to return the average number of components for all entities stored in the registry.
- `size_t GetEstSize()`
  - Added to get the estimated size of all entities in this registry.
- `std::string ToJSON(Handle h)`
  - Added to convert an entity's data to JSON.
- `std::string GetSnapshot()`
  - Added to assemble a complete snapshot of the registry's contents as a JSON object.

The Estimate generated by `size_t GetEstSize()` is calculated by accumulating the estimated size over all archetypes. This is an estimate only; if the archetypes hold components of complex types, structures or objects that manage additional data (such as an std::string, for example), the size of these additional data is not taken into consideration. Only the size of the stored components themselves is calculated. The JSON string created in `std::string ToJSON(Handle h)` is needed in LiveView communication only. For snapshots, all JSON data are directly aggregated through calls to Archetype::ToJSON() and handled there.

The following added methods in `VECSVector.h` provide an interface to be used in the template class Vector. This is necessary as it is the only possibility to access the required information when composing snapshots. In the Archetype class, only the VectorBase class is known. The same methods have also been implemented in the template class Vector, where they demonstrate their functionality by providing access to detailed information about the data stored in an archetype.

- `virtual auto ToJSON() -> std::string`
  - Added to generate a JSON representation of the vector.
- `virtual auto ToJSON(size_t index) -> std::string`
  - Added to generate a JSON representation of an element in the vector.
- `virtual size_t GetType()`
  - Added to get access to the type handled in this vector.
- `virtual size_t ElemSize()`
  - Added to get the size of the elements stored in this vector.

The new header file `VECSConsoleComm.h` handles all the communication with the console. It manages the TCP connection, which is transparently done in a background thread. The communication between the console and the running VECS application is exclusively done using JSON messages. All JSON objects incoming from the console contain commands. Based on the command the VECS receives, it then either gathers all data for a snapshot or sends liveview data with simple statistics. From the user's perspective, there is exactly one function of interest here:

- `inline VECSConsoleComm* GetConsoleComm(Registry* reg, std::string host, int port)`

– Initiates a Console connection.

The `inline VECSConsoleComm* GetConsoleComm(...)` function can be called from the program using VECS to initiate a console connection if this is also wanted in a release build, or if the connection to Console is done with other parameters than the default ones for example to connect to another port, or if wanted to debug from a different machine.

4.2 Console

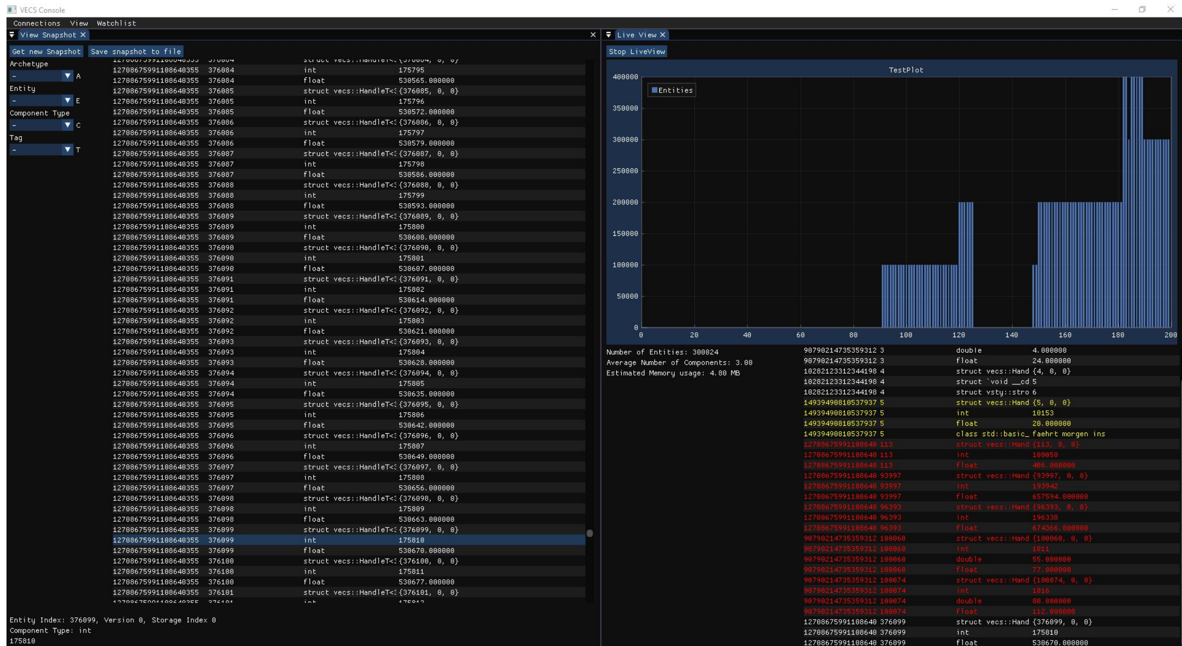

Figure 5: Overview GUI

In 5 an example of a Console being used for debugging a VECS Application is shown. Due to the chosen technology DearImGui it is also possible to use "docking", meaning to display multiple different windows next to each other. In the example picture the user is currently inspecting a snapshot while also monitoring the Live View of the currently active entities. Under the Live View, the watchlist is displayed to show data changes as they happen. In this chapter all the different windows and functionalities will be shown. The Console is structured as a server, while the VECS programs connect as clients. This creates a 1:n relationship between the Console and the programs. The console places a listener on a freely selectable TCP port. If no specific port is defined, port 2000 is selected as the default. A separate handler thread is created for each incoming connection, which manages communication between the console and the VECS program.

This listener consists of a generic base class that handles tasks such as establishing connections, managing communication threads, and basic data transfer functionality. A communication class is derived from this base class, which handles communication between VECS and Console. The chosen data protocol is JSON, as it offers a good balance between data minimization and readability. Communication between the console and VECS is bidirectional, using JSON objects. However, these data volumes are asymmetrical, as the console usually only sends short commands with additional information, while the VECS program can respond with megabytes of data in the case of a snapshot. Requested JSON

objects with data are received via the listener. These are then parsed by the nlohmann JSON library. This library internally builds a tree to make this data usable [25]. This data is then divided into the internal data structure. The structure is based on the data structure also used in VECS to ensure easy traceability. Each connected VECS Application sends its JSON commands and data to its own Listener thread in the Console. This thread handles the Communication and also stores Data. The JSON communication handled by the thread is a quite straightforward operation. In the Console UI, the User can call methods, request snapshots or start the Live View, which trigger the sending of the corresponding commands to the VECS application. In the VECS application, these commands are processed and handled in a background thread. On the Console side, incoming JSON objects, sent from the VECS application over the TCP connection, are parsed into an internal tree, which is then processed according to the command passed in the JSON object.

Another functionality of the Listener thread is the storage of received and parsed snapshot data, which is then used to display the snapshot table in the GUI. The Console uses a internal storage format which matches the the format used by the VECS - up to a point. The various components that make up an entity have an arbitrary format; the only requirement is that they each have a unique type. This, however, is not something which can easily transported to Console, which has no insight into the makeup of the objects or basic types used inside the VECS program. Therefore, the data stored in the entities are sent, and stored in the Console, as strings, as far as possible - since it is not mandatory that only objects that contain a to_string() method or can be sent to an output stream are stored in VECS entities, some data might not be displayable, and their content is sent as `"<unknown>"`. To avoid any concurrency problems when the UI is displaying a snapshot while it is being replaced, the listener thread stores up to two snapshots and switches the active to the incoming one, once it has been fully processed. It is possible to switch between different VECS, and use one Console to debug multiple applications at the same time. For every connection a new listener thread is created. Since the thread stores the data, it is easy to switch to another connected VECS application. In the main loop, all windows are displayed depending on whether the boolean is set to true. The menu is built using DearImGUI and uses booleans to open and close the selected windows. Also, since ImGUI is used in the docking branch, the user can pull the different windows out of the main window to order them for their individual needs.

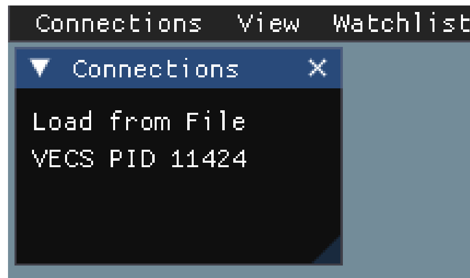

Figure 6: Connection Window

When started, Console opens a window with a menu. This Menu 6 can also be opened from the Menu item "Connections" above, if it was closed. "Manage Connections" opens or hides the "Connections" window. "Connections" lists all current connections to VECS programs, plus a "Load from File" entry which allows to select a previously saved snapshot file for inspection. Since this is an essential prerequisite to operate Console, this window is opened by default when Console is started. In the menu band are the options to show the other windows with the snapshot and Live View functionality, and an option to show the watchlist. However, before connecting to a Console all these windows will show up empty. When the Console is connected then it is possible to request a snapshot and start a Live View.

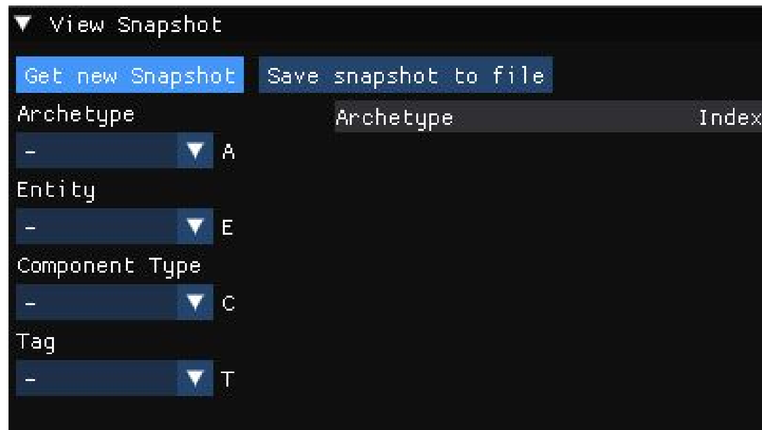

Figure 7: Empty Snapshot Window

The next entry in the Menu Bar is "View" which gives the options "Snapshot" and "Live View". When Snapshot is selected, the corresponding window opens. This window allows the inspection of snapshots from a VECS program, and the (de-)selection of items for LiveView. If opened without a selected VECS connection or snapshot file, the Snapshot window is empty. Once a selection has been made, however, this changes to the layout shown in 7. "Get new snapshot" allows to fetch a snapshot from the selected VECS program; if a snapshot file has been selected, this button has no effect. Once the Data from the VECS Application has been fetched, the Data table in the middle displays all Entities from all archetypes per default. An example of a complete snapshot is shown in 8. It is also posible to save a complete snapshot to a json file. This "Save snapshot to file" button creates a new json file in the folder where the Console runs. This can also later be loaded in the Connection Menu instead of connecting to a Console. This saved file can be used to log possible errors and save a state to inspect it later, or even on a different device. While implementing the Console, one difference to the mock up is the addition of the "Tag" filter, and to filter the components by their type.

View Snapshot

Get new Snapshot | Save snapshot to file

Archetype: - A
Entity: - E
Component Type: - C
Tag: - T

Archetype	Index	Typename	Value	Tag
9012002715490895206	0	struct vecs::HandleT<32,24,	{0, 0, 0}	47
9012002715490895206	0	double	4.000000	47
9012002715490895206	0	float	3.000000	47
9012002715490895206	0	int	5	47
9012010567730595535	1	struct vecs::HandleT<32,24,	{1, 0, 0}	666
9012010567730595535	1	double	3.000000	666
9012010567730595535	1	float	23.000000	666
9012010567730595535	1	int	1	666
9079021473535931274	2	struct vecs::HandleT<32,24,	{2, 0, 0}	
9079021473535931274	2	int	6	
9079021473535931274	2	double	8.000000	
9079021473535931274	2	float	7.000000	
9079021473535931274	3	struct vecs::HandleT<32,24,	{3, 0, 0}	
9079021473535931274	3	int	2	
9079021473535931274	3	double	4.000000	
9079021473535931274	3	float	24.000000	

Figure 8: Snapshot Window

Entities are cached for displaying the snapshot to avoid lag, since in a running VECS Application we can use up to thousands of entities at once. The same applies to the lists in the filters to ensure smooth operation. In the snapshot window, you can also add individual entities to the Watchlist. This list is used to save specific entities for later inspection in the live view. "Save snapshot to file" allows

to save the currently processed snapshot to a JSON file for later inspection. This could be used for recording a specific error and check later for changes, or inspecting it later without the running VECS application. On the left side, there is a set of filters which allow selecting parts of the potentially very large list of entities and their components on the right side. The following filter possibilities have been implemented:

- Archetype: display only entities that are stored in a specific archetype. Unfortunately, the archetypes don't have symbolic names, so it is only possible to select them by their hash value as it is defined in VECS.
- Entity: display only the components of a specific entity
- Component type: display only components of the selected type
- Tag: display only entities that have a specific tag associated with them

On the right side, the components matching the filter settings are displayed. Right-clicking on an entry allows to add it to or remove it from the watch list.

Watchlist

▼ Watchlist

Archetype	Index	Typename	Value	Tag
9079021473535931274	2	struct vecs::HandleT<32,24,8>	{2, 0, 0}	
9079021473535931274	2	int	6	
9079021473535931274	2	double	8.000000	
9079021473535931274	2	float	7.000000	
12708675991108640355	270	struct vecs::HandleT<32,24,8>	{270, 0, 0}	
12708675991108640355	270	int	99	
12708675991108640355	270	float	252.000000	
12708675991108640355	279	struct vecs::HandleT<32,24,8>	{279, 0, 0}	

Figure 9: Watchlist Window

Here, entities that have been added to the watch list using the functionality provided in the snapshot window, are displayed 9. A right-click on an entry opens a popup menu where the entity can be removed from the watch list. This watchlist is then used in the Live View for monitoring and data validation. In this list the data is stored as is, if some of the data changes it will be displayed in the table under the live view graph. If opened without a selected VECS connection or snapshot file, the Live View window is empty. Once a selection has been made, however, this changes to the layout as shown in 10. This shows an example of a running Live View being used to monitor a running Entity Component System.

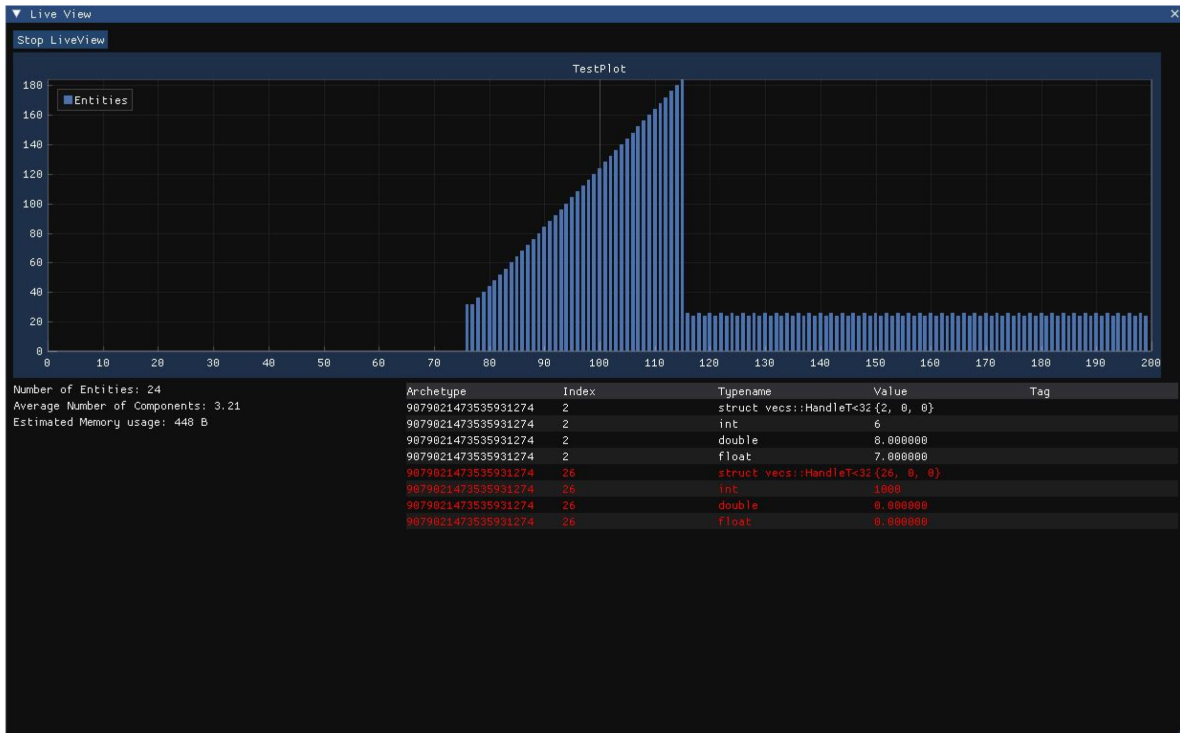

Figure 10: Live View Window

Since Live View slightly interferes with the VECS program's operation by periodically inspecting the VECS registry, the functionality has to be deliberately turned on; a click on "Start LiveView" does that. Once started, the button text changes to "Stop LiveView" to turn it off again. Like this, unwanted interference with VECS can be kept to a minimum and data is only gathered when explicitly desired. Once LiveView has been started, the graph below shows the number of entities currently held in the VECS registry in a sliding window, the current value is the rightmost. The number of entities, their average number of components and the estimated memory usage are updated to the current value each time a new LiveView object is sent to the Console.

Looking at the Graph in more detail as shown in 11, it displays the number of entities currently existing in the connected VECS Application using a bar graph. The y-axis shows the number of entities.

Figure 11: Plot over Entity Count in Live View

Below the graph are three statistics:

- Number of entities
- Average number of components per entity
- Estimated memory usage

The statistics are calculated by the VECS Application and transmitted when changes occur. They are intended to be useful tools while debugging, for example aiding in finding data leaks or an early alert if an error in the VECS application continually creates entities, causing the used memory to rise fast. By checking the estimated memory usage it is also possible to get indication if the components of the entities are more complex than intended.

Archetype	Index	Typename	Value	Tag
9079021473535931274	2	struct vecs::HandleT<	{2, 0, 0}	
9079021473535931274	2	int	6	
9079021473535931274	2	double	8.000000	
9079021473535931274	2	float	7.000000	
14939490810537937315	5	struct vecs::HandleT<	{5, 0, 0}	
14939490810537937315	5	int	10046	
14939490810537937315	5	float	20.000000	
14939490810537937315	5	class std::basic_stri	string 2	
9079021473535931274	32	struct vecs::HandleT<	{32, 0, 0}	
9079021473535931274	32	int	1002	
9079021473535931274	32	double	10.000000	
9079021473535931274	32	float	14.000000	

Figure 12: Colored Watchlist in the Live View

Entities which have been added to the watch list on the Snapshot Window are listed below the graph. If their content changes or the entity is removed from VECS, their color changes. This is shown in 12. If a row is colored yellow, then a value has been changed. Here in the watchlist under the graph, the changed value will be displayed. Like this it is also possible to open the Watchlist window next to it and compare the original value to the changed value while the application is running. When an Entity is colored red, then it was deleted and is no longer active in VECS. This allows for data validation without reducing the list size.

5 Evaluation and Discussion

To test the Console for the sending of snapshots and the live View monitoring, VECS additions and the communication, it was necessary to create a test program that creates entities and is able to connect to a Console and have adaptable output to the Console. For this use, a test program called testvecscons has been created. This test program can be run in two different modes; simple entities and complex entities. The test program can be called with following arguments:

```
testvecscons [-h<hostname>[:<port>]] [-p<port>] [-c]
```

- -h ... directs VECS to initiate a TCP connection to Console running on the machine with the given hostname. The hostname can be an IPv4 address or a name that is resolved through DNS. For convenience, a port number or service name can be added after a colon; this can also be done through -p.
- -p ... directs VECS to initiate a TCP connection to Console listening on this port number.
- -c ... enables ComplexMode. The test programm will now add more complex entities, designed to stress test the Console.

If none of these parameters are given, testvecscons uses the default values to initiate a connection to Console running on the same machine (i.e., localhost), on the default port (2000), using simpler entities only.

5.1 Simple Entities

In simple mode, testvecscons first populates a VECS registry with a set of entities. These simple entities have components with the types `int, double, float, string`. These Components are then given to the entities to create different archetypes. After that, it tries to establish the connection to Console; this can be terminated by pressing Escape. Once connected, it loops for up to 600 seconds, continuously adding or removing some entities from the VECS registry each second. After 80 seconds, the registry is trimmed down to 20 entities. Two additional features allow the addition of 100,000 entities to the registry by pressing 'a', and the removal of up to 100,000 entities from the registry by pressing 'd'. After 600 seconds, of if 'x' is pressed, testvecscons terminates. All interaction with the Console is handled in a background thread. Using this test program and based on the metrics built into the program, we arrive at the following data:

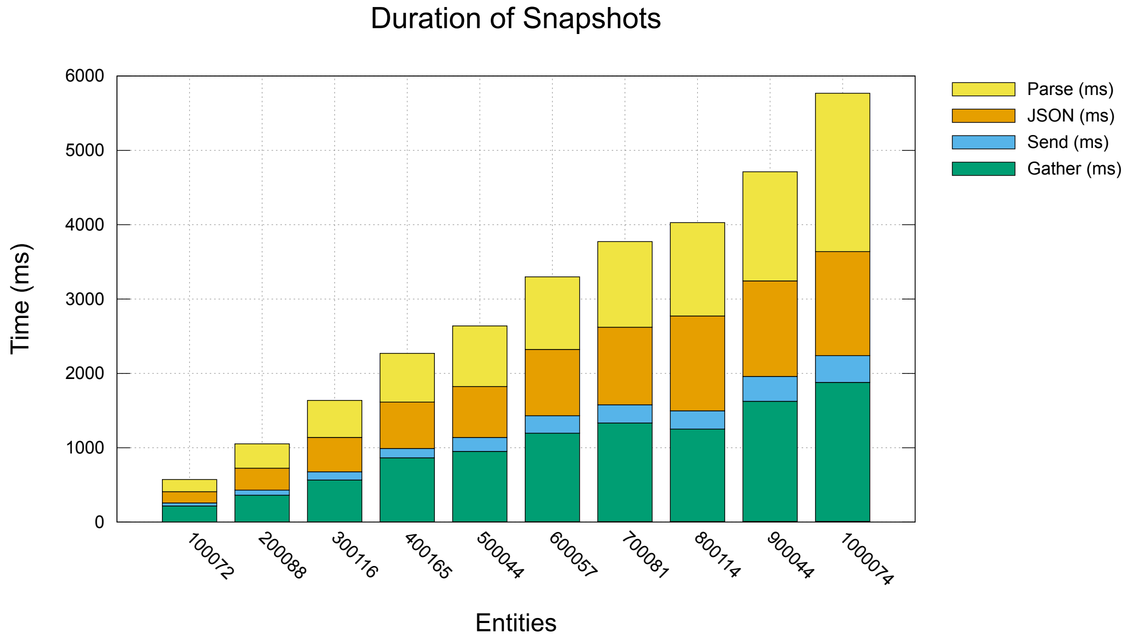

Figure 13: Parse Time

Figure 13 displays the time used to build a snapshot, send it over the TCP connection, and parse the resulting JSON object on the other side. The work steps are as follows:

- **Gather:** Time spent gathering the component data of all currently existing entities in the VECS Application and transforming them to strings in JSON form.
- **Send:** Time spent sending the created JSON string over the established TCP connection to the Console.
- **JSON:** Time spent parsing the JSON string into a JSON file and building the tree for internal representation.
- **Parse** Time spent parsing the created JSON tree into the internal data storage.

This graph shows a natural increase in the time used to make and send a snapshot. All the times increase with higher entity numbers. I tested with up to one million entities and arrived at a maximum time of under 6 seconds. For one million entities about 2 seconds are used for gathering the entity and component data of the application. This is most likely due to the encapsulation of the VECS which causes the workflow to go through several for loops in the program to access all the entities. The parse and JSON time is increasing slowly per 100,000 Entities up until about 800,000 entities when the JSON parsing time does not increase at the same rate anymore. Due to the test program and the console being run on the same device for this test case, the send time only increases slightly, even for higher numbers of entities being sent. Furthermore, TCP is a fast connection and a suitable choice for this use case. Most times, if used, both the console and the application using VECS will be running on the same device, so the TCP connection is done through a local loopback interface which serves as a pure memory transfer, not including any network devices, and thus is very fast.

All in all, the Console is efficient enough to be used for debugging purposes. While being almost in Real Time with smaller amount of data, like up to 1000 Entities, it then increases in processing time with about 0.5 to 0.6 seconds per 100,000 Entities. For most use cases, this will suffice to show an accurate snapshot of the VECS System and can be used in debugging.

5.2 Complex Entities

To stress test the Console and evaluate if the time used for snapshots is useable for more complex cases, the Complex Mode was implemented. If the test is started in Complex Mode, it uses the following structs to create complex entities:

```
struct1{ integer,double }
struct2{integer, integer, string}
struct3{float, double, char, integer}
struct4{ string, string }
```

These structs were defined to, on one hand, show the use case of a programmer using the structs to add more values of the same type per Entity, and on the other hand to be quite complex and show what happens when the Console is faced with high number of components for the snapshot.

The resulting four Archetypes in the registry are as follows.

- `integer, struct1, struct2, string`
- `integer, struct2, struct3, float, double`
- `integer, struct3, string, char, double`
- `integer, struct4, struct1, struct3, string, float , double, char`

These data structures were chosen because they show the upper border of possible debugging use. It is not very likely to have a registry populated by these overly complex entities with appearance numbers in the hundred thousands to millions, but by showing this edge case, it is intended to show the robust advantage of using the Console instead of print debugging.

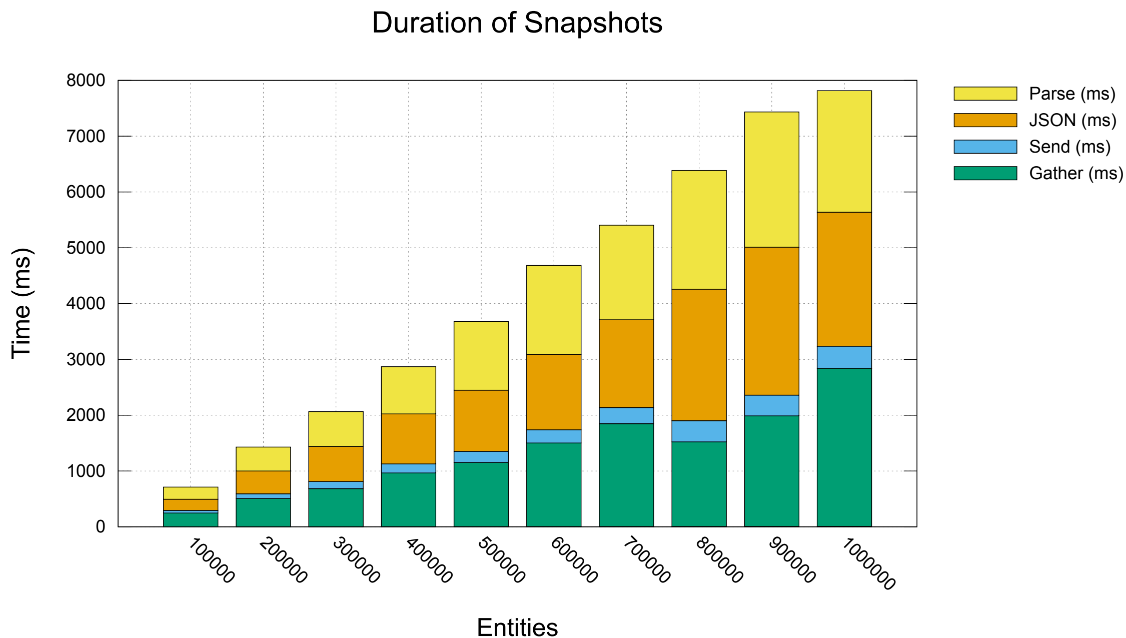

Figure 14: Parse Time of complex entities

For this complex test shown in 14, it was tested again with up to a million Entities and arrived at a maximum time of 8 seconds. Since the test is using more than double of the numbers of components

in comparison to the first test, the resulting time being less than double is satisfactory. In this graph, the Gather time is noticeably higher than in the graph using simple Entities, due to the very high amount of Components. Since a single Entity has more Components in this test and always contains a struct, it is more work to gather these Entities. Once the Entity is complete, the send time is almost the same to before, again, due to both the test and the Console running locally and communicating via TCP. Same as in the test with simple entities, the total time rises in linear fashion with complex entities as well. The difference in this case is that due to the high Component numbers and complex Component types, the JSON and parse time also rises with higher entity numbers. Since the maximum time needed in this test is only about 2 seconds higher than the maximum time for the test with simple Entities only, is should be safe to say that the performance will suffice for most common use cases.

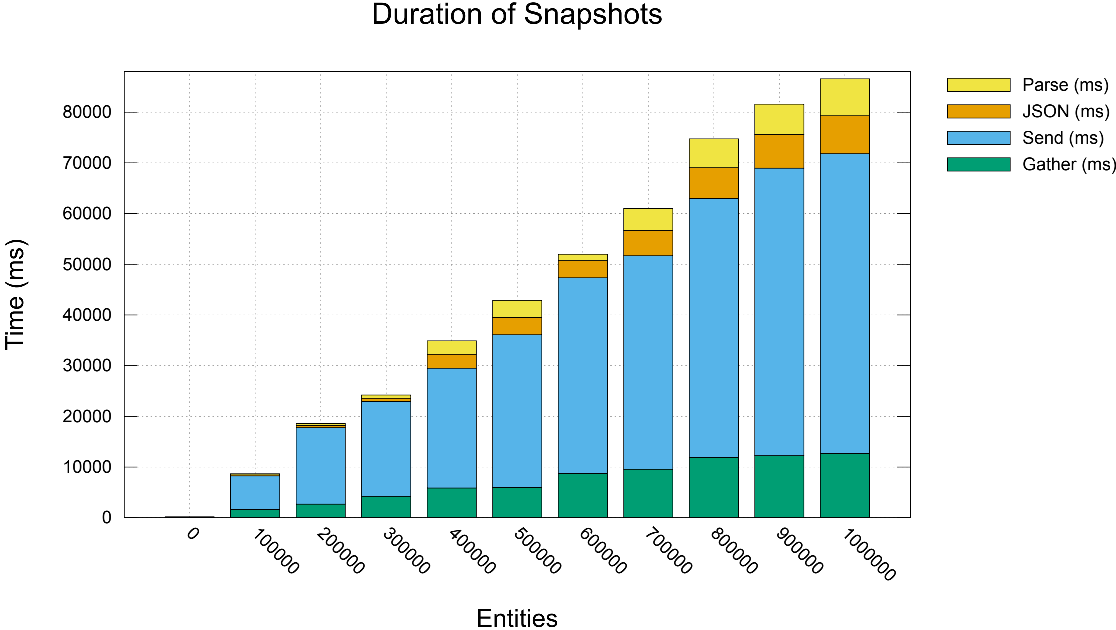

Figure 15: Parse Time of complex entities over WLAN

Since it is also possible to use the Console to debug from another device than the one running the VECS application, in this case, shown in figure 15, it was tested how big the delay over WLAN would be. This test was also designed to show a worst case example to evaluate if debugging remotely is feasible at all. The graph shows that while the gathering and parsing time of the data remains roughly the same, the send time over WLAN rises fast and gets very high. To get a snapshot from another device, the total time for a million Entities took 86 seconds, with most of it being send time only. In most use cases, it is unlikely for the user to use one million overly complex entities, but for regular debugging, the send time is passable. For small projects without fast changing data, the Console could also be used over WLAN and still pose a help.

6 Conclusions and Future Work

Looking at the graphs, it is clear that the console works well for most typical game development use cases. When large amounts of data or overly complex entity types are used, the lag that occurs when a snapshot is requested has potential to influence the data currentness in programs where the data is changing rapidly. This is due to the large amount of data that has to be accumulated from all active

Entities currently in the system, especially when the number of components is high. Depending on the complexity of the Entities, when using the Console for use cases with up to 600,000 Entities the lag is small enough to not get in the way with the program execution and showed to be current. For most applications this will suffice to be an efficient debugging help to give in depth insights into the running application using VECS.

It is also possible to use the Console over WLAN/LAN to debug from another device. In the test, the complex Entities were used. The test case shows that even over a slower WLAN connection it is possible to use the Console, even with complex Entities. But since the lag due to the sending rises fast, it is recommended to use the remote debugging for small projects under a hundred thousand Entities maximum. It would need a much more elaborate scheme to send a big number of Entities over WLAN in a reasonable time, which could be done in future work but is out of scope for this project. The Live View has not shown any sign of lag or other issues even when handling large amounts of Entities and also when handling complex Entities. In all tests, it was stable and behaved smoothly, showing the current number of entities and statistics in a timely manner and reacting fast to changes in the system. One possible area of optimization is the data accumulation in the program running VECS. If the data is accumulated faster, then the lag for higher numbers of Entities would also minimize, improving the data currentness. The data transfer could also be sped up further. Currently, JSON is used for data transfer, chosen for its human readability and relatively small overhead. While other transfer protocols might be faster, the human readability would not be there anymore.

References

[1] https://www.vulkan.org/.

[2] Bayliss, J. D. Developing games with data-oriented design. In *Proceedings of the 6th International ICSE Workshop on Games and Software Engineering: Engineering Fun, Inspiration, and Motivation* (New York, NY, USA, 2022), GAS '22, Association for Computing Machinery, p. 30–36.

[3] Cheng, T., Duan, T., and Dinavahi, V. Ecs-grid: Data-oriented real-time simulation platform for cyber-physical power systems. *IEEE Transactions on Industrial Informatics 19*, 11 (2023), 11128–11138.

[4] Colson, D. https://www.david-colson.com/2020/02/09/making-a-simple-ecs.html. *https://www.david-colson.com* (2020).

[5] Cox, L., WILLIAMS, B., VICKERS, J., WARD, D., and HEADLEAND, C. Run-time performance comparison of sparse-set and archetype entity-component systems. In *Computer Graphics & Visual Computing (CGVC) 2025* (2025), The Eurographics Association.

[6] Dahl, J., and Harris, F. C. An argument for the practicality of entity component systems as the primary data structure for an interpreter or compiler. In *Proceedings of the 2025 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (New York, NY, USA, 2025), Onward! '25, Association for Computing Machinery, p. 85–98.

[7] https://skypjack.github.io/entt/.

[8] Fabian, R. Data-oriented design. In *Data-Oriented Design* (2018).

[9] Fischbach, M., Wiebusch, D., and Latoschik, M. E. Semantic entity-component state management techniques to enhance software quality for multimodal vr-systems. *IEEE Transactions on Visualization and Computer Graphics 23*, 4 (2017), 1342–1351.

[10] Fischbach, M. W. *Enhancing Software Quality of Multimodal Interactive Systems.* PhD thesis, Universität Würzburg, 2017.

[11] https://www.flecs.dev/flecs/.

[12] Fontana, T., Netto, R., Livramento, V., Guth, C., Almeida, S., Pilla, L., and Güntzel, J. L. How game engines can inspire eda tools development: A use case for an open-source physical design library. In *Proceedings of the 2017 ACM on International Symposium on Physical Design* (New York, NY, USA, 2017), ISPD '17, Association for Computing Machinery, p. 25–31.

[13] geeksforgeeks. https://www.geeksforgeeks.org/cpp/diamond-problem-in-cpp/.

[14] Götz, D., and von Mammen, S. Modulith: A game engine made for modding. In *Proceedings of the 18th International Conference on the Foundations of Digital Games* (New York, NY, USA, 2023), FDG '23, Association for Computing Machinery.

[15] Hall, D. M. Ecs game engine design.

[16] Hanisch, M. *Konzeption und Evaluation eines Entity-Component-Systems anhand eines rundenbasierten Videospiels.* PhD thesis, Hochschule für angewandte Wissenschaften Hamburg, 2016.

[17] Härkönen, T. Advantages and implementation of entity-component-systems. *Tampere University, Tampere, Finland* (2019).

[18] Hatledal, L. I., Chu, Y., Styve, A., and Zhang, H. Vico: An entity-component-system based co-simulation framework. *Simulation Modelling Practice and Theory 108* (2021), 102243.

[19] Hlavacs, H. https://github.com/hlavacs/ViennaEntityComponentSystem.

[20] Hlavacs, H. Gaming Technologies Lecture 2025.

[21] Lange, P., Weller, R., and Zachmann, G. Wait-free hash maps in the entity-component-system pattern for realtime interactive systems. In *2016 IEEE 9th Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)* (2016), pp. 1–8.

[22] Maekawa, T., Odaki, A., Koizumi, T., Tsumura, T., and Shioya, R. Phalanx: A processor simulator based on the entity component system architecture.

[23] McIvor, E., Sklivanitis, G., and Pados, D. A. A generalizable entity-component-system architecture for underwater rov control. In *2025 Symposium on Maritime Informatics and Robotics (MARIS)* (2025), pp. 1–4.

[24] Muratet, M., and Garbarini, D. Accessibility and serious games: What about entity-component-system software architecture? In *International Conference on Games and Learning Alliance* (2020), Springer, pp. 3–12.

[25] https://json.nlohmann.me/.

[26] Papagiannakis, G., Kamarianakis, M., Protopsaltis, A., Angelis, D., and Zikas, P. Project elements: A computational entity-component-system in a scene-graph pythonic framework, for a neural, geometric computer graphics curriculum. *arXiv preprint arXiv:2302.07691* (2023).

[27] Pax, R., Gomez-Sanz, J. J., Olivenza, I. S., and Bonett, M. C. A cloud based simulation service for 3d crowd simulations. In *2018 IEEE/ACM 22nd International Symposium on Distributed Simulation and Real Time Applications (DS-RT)* (2018), pp. 1–5.

[28] pico. `https://github.com/empyreanx/pico_headers/blob/main/pico_ecs.h/`.

[29] Pouhela, F., Krummacker, D., and Schotten, H. D. Entity component system architecture for scalable, modular, and power-efficient iot-brokers. In *2023 IEEE 21st International Conference on Industrial Informatics (INDIN)* (2023), pp. 1–6.

[30] Raffaillac, T., and Huot, S. Polyphony: Programming interfaces and interactions with the entity-component-system model. *Proc. ACM Hum.-Comput. Interact. 3*, EICS (June 2019).

[31] Redmond, P., Castello, J., Calderón Trilla, J. M., and Kuper, L. Exploring the theory and practice of concurrency in the entity-component-system pattern. *Proc. ACM Program. Lang. 9*, OOPSLA2 (Oct. 2025).

[32] Romeo, V. Analysis of entity encoding techniques, design and implementation of a multithreaded compile-time entity-component-system c++14 library. Master's thesis, Università degli Studi di Messina, 2015/2016.

[33] Sharp, J. A. Data oriented program design. *SIGPLAN Not. 15*, 9 (Sept. 1980), 44–57.

[34] Slay, T., Spitzer, G. B., and Bass, R. B. Proposed application for an entity component system in an energy services interface.

[35] Voisard, L., De Freitas Serra, H., Politowski, C., Petrillo, F., and Guéhéneuc, Y.-G. A mapping study of the entity component system pattern. In *2025 IEEE/ACM 9th International Workshop on Games and Software Engineering (GAS)* (2025), pp. 33–40.

[36] Wiebusch, D., and Latoschik, M. E. Decoupling the entity-component-system pattern using semantic traits for reusable realtime interactive systems. In *2015 IEEE 8th Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)* (2015), pp. 25–32.

[37] Winter, E. Reimplementation of a real-time 3d audio rendering software using an entity component system architecture, 2024.

[38] Zhang, B., Shi, H., and Wang, X. An auxiliary development framework for lightweight rpg games based on unity3d. *Computer Animation and Virtual Worlds 35*, 1 (2024), e2206.