universität wien

Insights into Entity Component Systems

Univ.-Prof. Dr. Helmut Hlavacs, Isabella Hölzl, Marlene Lucia Kasper
helmut.hlavacs@univie.ac.at

Research Group Education, Didactics and Entertainment Computing (EDEN)
Faculty of Computer Science, University of Vienna

universität wien

Who are We

- Professor/students for Computer Science at University of Vienna, Faculty of CS
- Into C++ since early 1990s
- Research: video game technologies and game applications (engines, serious games, …)
- Teaching: game AI, physics, parall., data, Vulkan API + CG, streaming, ray tracing, …

https://github.com/hlavacs

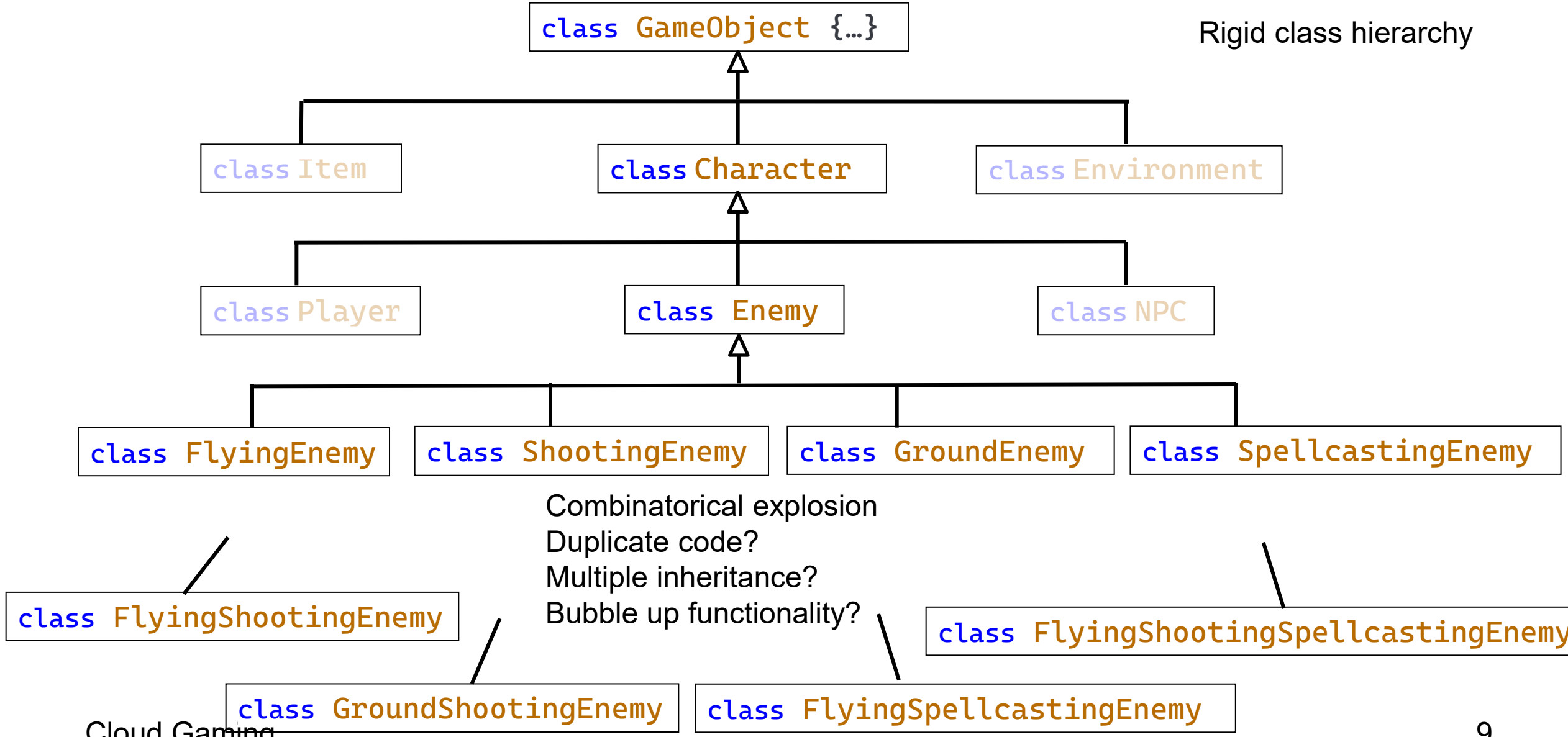
- **Vienna Entity Component System (VECS), Vulkan** based game engine, Physics engine, …
- Learn, teaching resource, basis for courses, theses, …
- No production code

universität wien

Modern Video Game Requirements

- **Thousands** of entities (bullets, particles, items, NPCs, …)
- Dynamic runtime behavior, AI, physics, rendering
- Memory efficiency, cache-friendly – data-driven design
- Parallel processing
- Complex interactions between systems

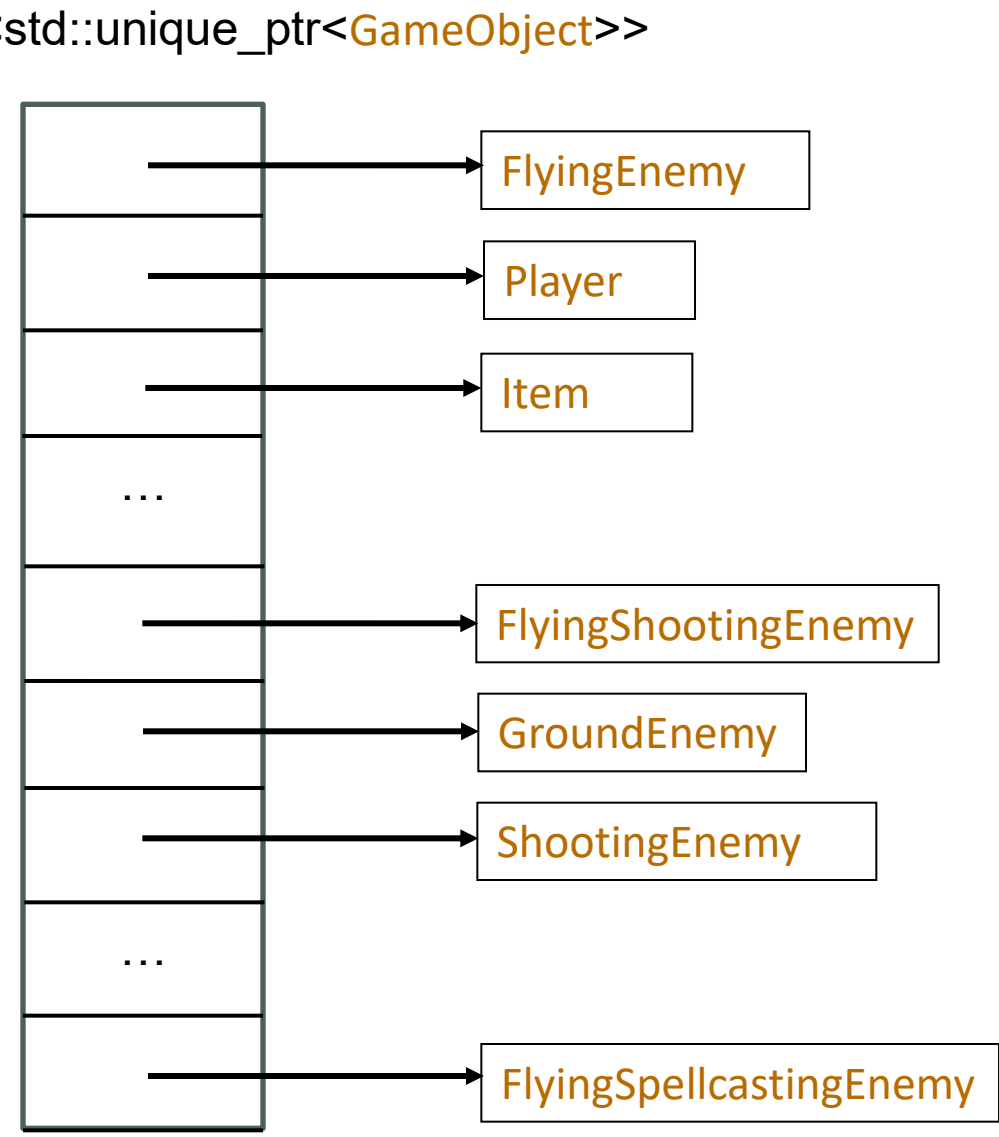
Let‘s have some **Video Game Entities** with OOP



universität wien

Game Objects and OOP

- Rigid hierarchies, inflexible designs, diamond of death
- Combinatorical explosion, separation of concern
- New behavior requires a new class
- Virtual functions used in a **bad way**
- Jumping around in memory, cache misses



Solution – **Entities** are Combinations of **Components**

Entity 1:	GameObject	Character	Enemy	Flying	
Entity 2:	GameObject	Character	Player		
Entity 3:	GameObject	Item			
…					
Entity 101:	GameObject	Character	Enemy	Flying	Shooting
Entity 102:	GameObject	Character	Enemy	Ground	
Entity 103:	GameObject	Character	Enemy	Shooting	
…					
Entity 999:	GameObject	Character	Enemy	Flying	Spellcasting

universität wien

Three Pillars of **Entity Component Systems** (ECS)

1. **Entity** - Unique ID (handle), like a `std::set<std::any>`

2. **Component** - Pure data (no behavior),
   Examples: **position**, **velocity, …**

3. **System** - Logic operating on entities with specific components

Key Principle: **Composition** over Inheritance

Typical ECS Operations

- **Create** a new entity and get handle / ID
- **Add** or **erase** components
- **Read** or **change** component values
- **Erase** whole entities
- **Iterate** over entity **groups**
  - **Tags** (**labels**) allow grouping
  - **Inclusion/exclusion** of **sets** of **types** and **tags**

- Example:
  AI system iterates over **all entities** that **have** a Position **and** Velocity, and **have tag** „Intelligent“, **but not** the tag „Heavy“

Example: VECS Operations

universität wien

```
vecs::Registry world;
vecs::Handle handle;


assert(!handle.IsValid());


std::string s = "AAA";
auto handle = world.Insert(s);
assert(world.Exists(handle));


handle = world.Insert(5, 5.5f);
assert(world.Has<int>(handle));
```

```
auto v1 = world.Get<int>(handle);
assert(v1 == 5);

world.Erase(handle);
assert(!world.Exists(handle));

//vecs::Handle hx = world.Insert(5, 6);
//compile error
struct height_t { int i; };
struct weight_t { int i; };

handle = world.Insert(
                5,
                height_t{ 6 },
                weight_t{ 6 } ); //works
```

Specifying Components with **Types**

universität wien

```
using strong_size_t  = vsty::strong_type_t<size_t, vsty::counter<>>;
using strong_size2_t = vsty::strong_type_t<size_t, vsty::counter<>>;

auto handle = world.Insert(5, 5.5f); //int and float
auto v1 = world.Get<int>(handle); //int is unique
```

- Components **unordered** (**sets** rather than tuples), **types unique** (enforce!)
- Use **wrappers**
- VECS uses **Vienna Strong Type** git repo
- **Wrapper + convenience**
- **Type counter** to be used as **phantom** type
  https://mc-deltat.github.io/articles/stateful-metaprogramming-cpp20

universität wien

Iterating with Views – Example: Symplectic Euler Step

```
Registry world;

void update(float dt) {
   auto view = world.template
        GetView< Handle, Position&, Velocity&, Acceleration > ();

   for (auto [handle, pos, vel, acc] : view) {
      vel += acc * dt;
      pos += vel * dt;
   }
}
```

Using an ECS in your Project

universität wien

Choose in this order:

1. Use **existing** battle tested (entt, Flecs, pico, …)
2. Create with an LLM
3. Create a **simple** one yourself
   - Your API, focus on functionality, performance comes later

4. Only if you **really really**
   - need high performance and a special API and
   - you have a lot of time and nerves
   - then create the perfect one -> and be called architecture astronaut

Claude, create a simple C++ ECS with templates

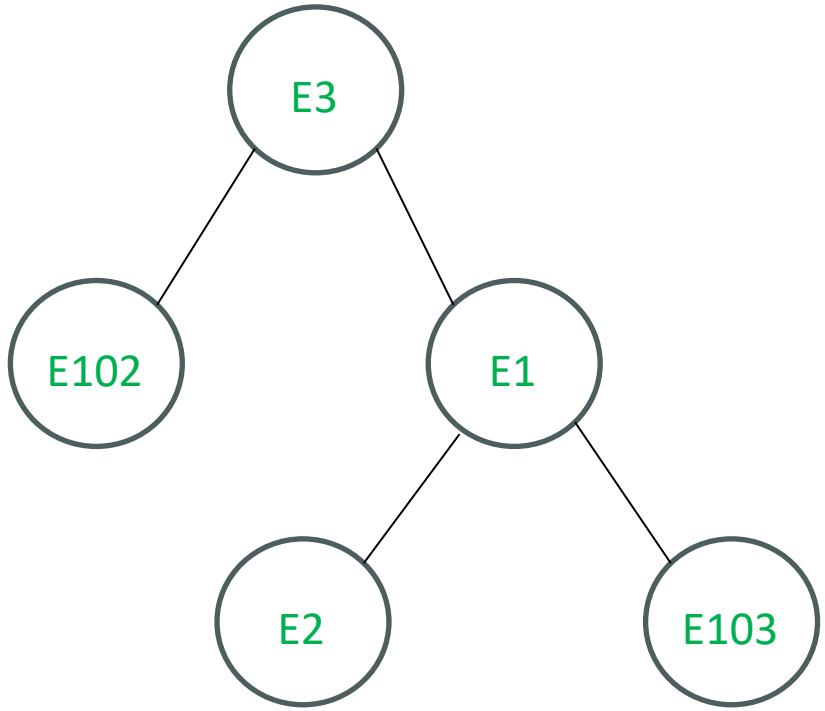
```cpp
#include <unordered_map>
#include <typeindex>

using Entity = size_t;

class ECS {
    Entity next_id = 1;

    template<typename T>
    static std::unordered_map<Entity, T>& get_storage()
        static std::unordered_map<Entity, T> storage;
        return storage;
    }

public:
    Entity create() { return next_id++; }

    template<typename T>
    void add(Entity e, T&& component) {
        get_storage<T>()[e] =
            std::forward<T>(component);
    }

    template<typename T>
    T* get(Entity e) {
        auto& storage = get_storage<T>();
        auto it = storage.find(e);
        return it != storage.end() ?
                        &it->second : nullptr;
    }

    template<typename T>
    void remove(Entity e) {
        get_storage<T>().erase(e);
    }

    template<typename T>
    auto& view() { return get_storage<T>();  }
};
```

Data Layout and Performance

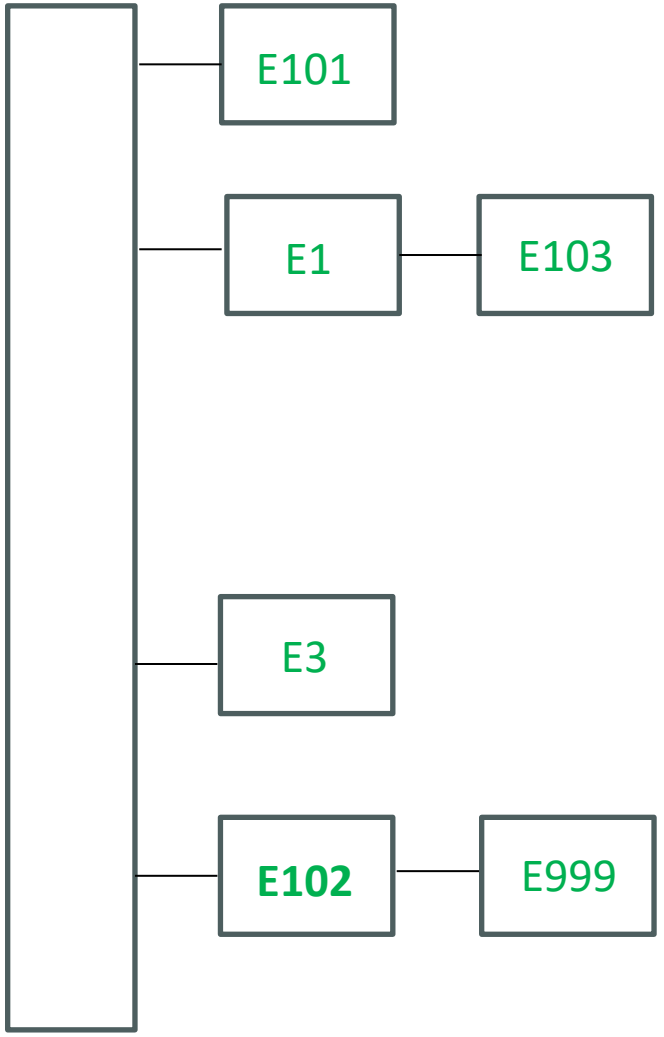
- Each choice represents some kind of compromise
- Is performance important?
  - No -> choose simple maps or ask LLM to make you a simple ECS
  - Yes
    - Which **operations** should be optimized?
    - Is **latency** or **throughput** important?
    - Data layout? Cache misses?
    - Number of levels of indirection?
    - O(.) of creation, lookups, change of components, deletion

Storing Components: Map vs. Unordered Map vs. Dense Arrays

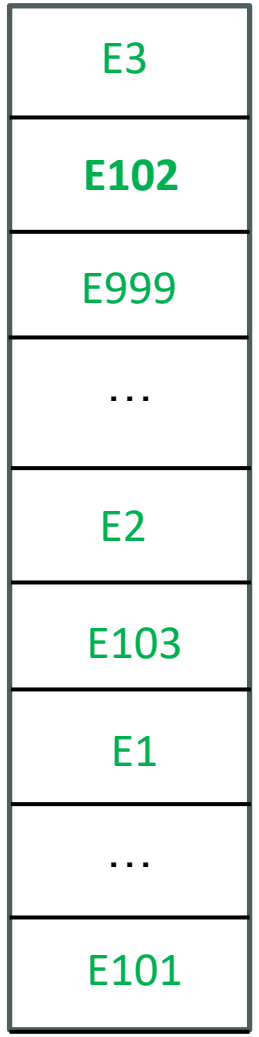
`std::map<uint64_t,GameObject>`



`std::unordered_map<uint64_t,Character>`



`std::vector<Enemy>`



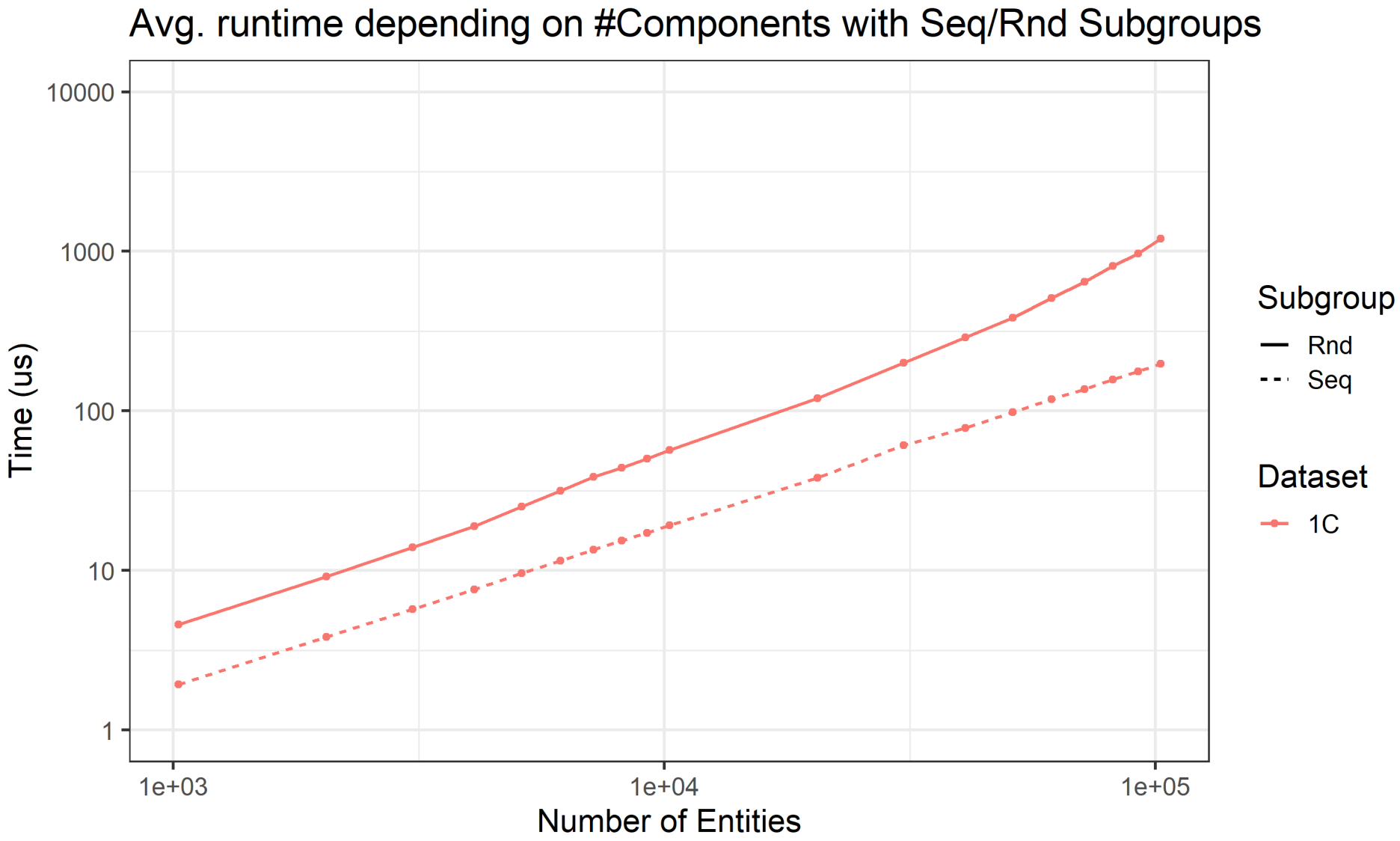
std::unordered_multimap<Handle, std::any>

Cache Efficient Data Layout Benefits

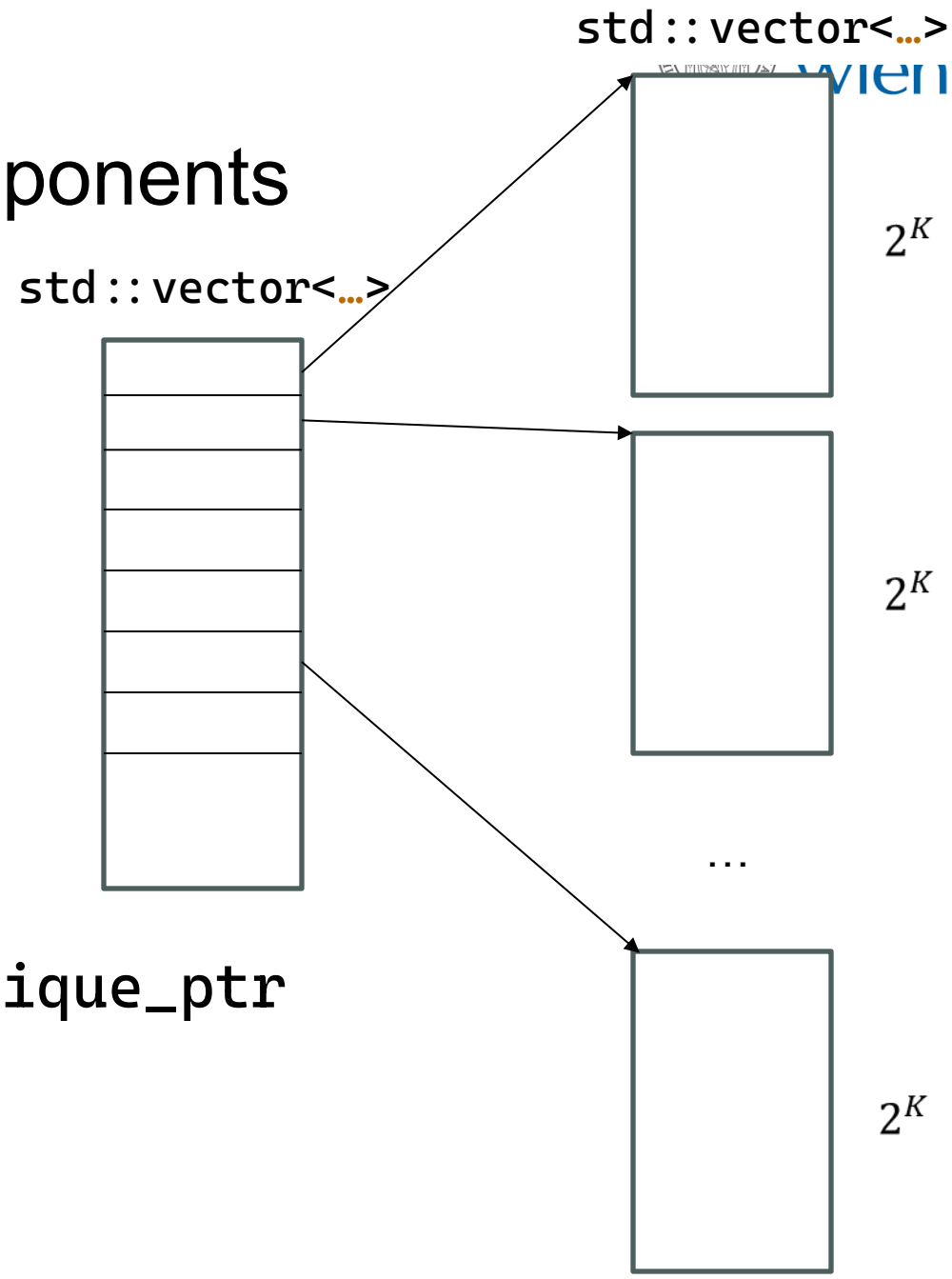
universität wien

- Systems process similar data together
- Data stored in **contiguous** (and dense) arrays
- Predictable memory access patterns -> cache prefetch
- Better CPU cache utilization and reduced memory fragmentation
  - 10-100x faster iteration over large entity sets
- SIMD optimization opportunities

Sequential vs. Random Access – Reading Data from 1 Column



Storing large Amounts of Components



- `std::vector` is not ideal
- Use more specialized container
- Better to use a **segmented** Vector
- Keeps references **static**
- Not limited by **max contiguous** size
- All allocations have **same size**
- `std::aligned_alloc` and `std::unique_ptr`
- **Two memory** reads per lookup

Using Separated Arrays

```
Vector<std::pair<Handle,GameObject>>
```

3	E3
102	**E102**
999	E999
…	…
2	E2
103	E103

…

```
Vector<std::pair<Handle,Character>>
```

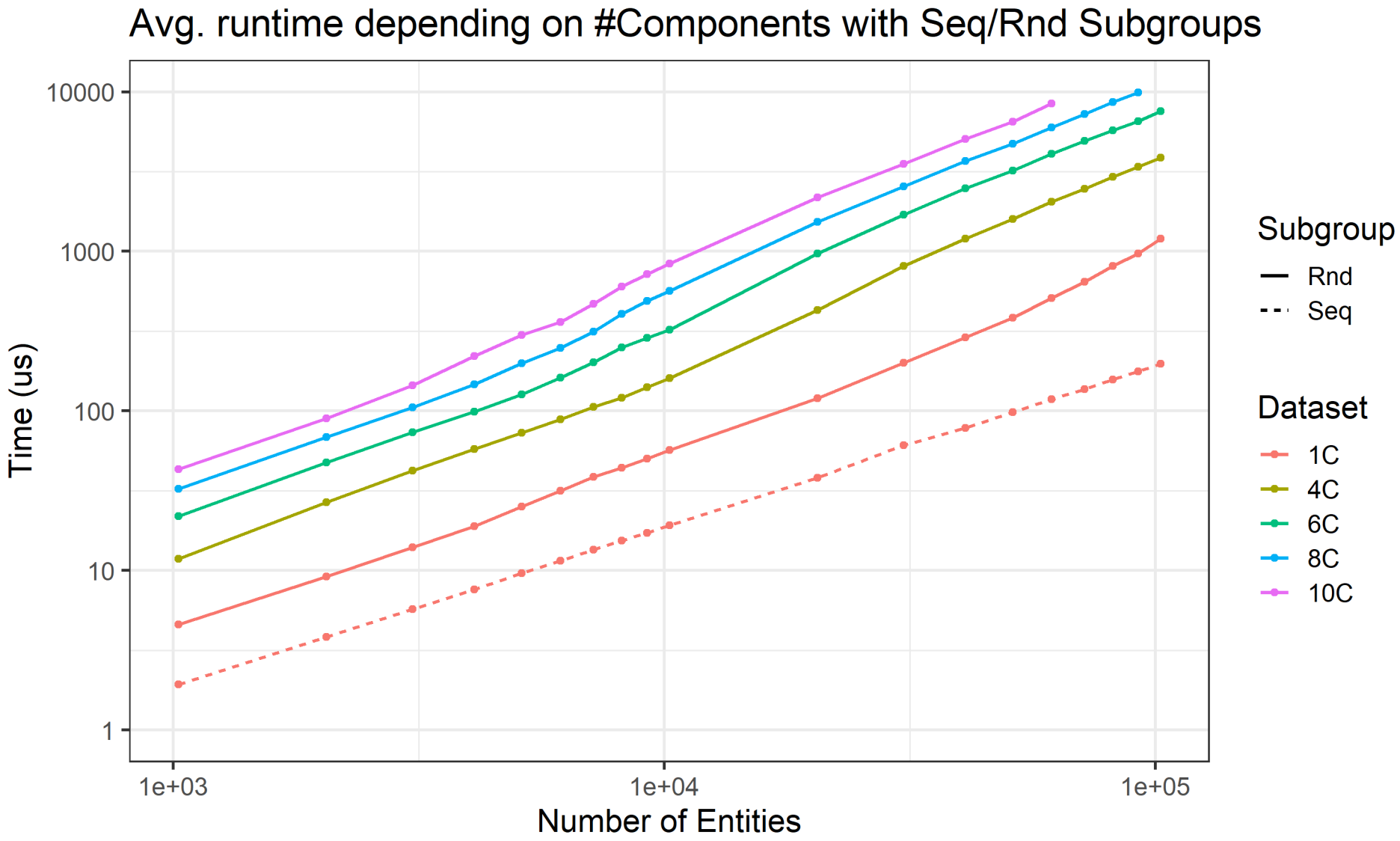
999	E999
1	E1
2	E2
…	…
103	E103
102	**E102**

…

```
std::unordered_map<Handle, std::pair<Handle,GameObject>*>
```

```
std::unordered_map<Handle, std::pair<Handle,Character>*>
```

Sequential vs. Random Access – Reading Data from N Columns



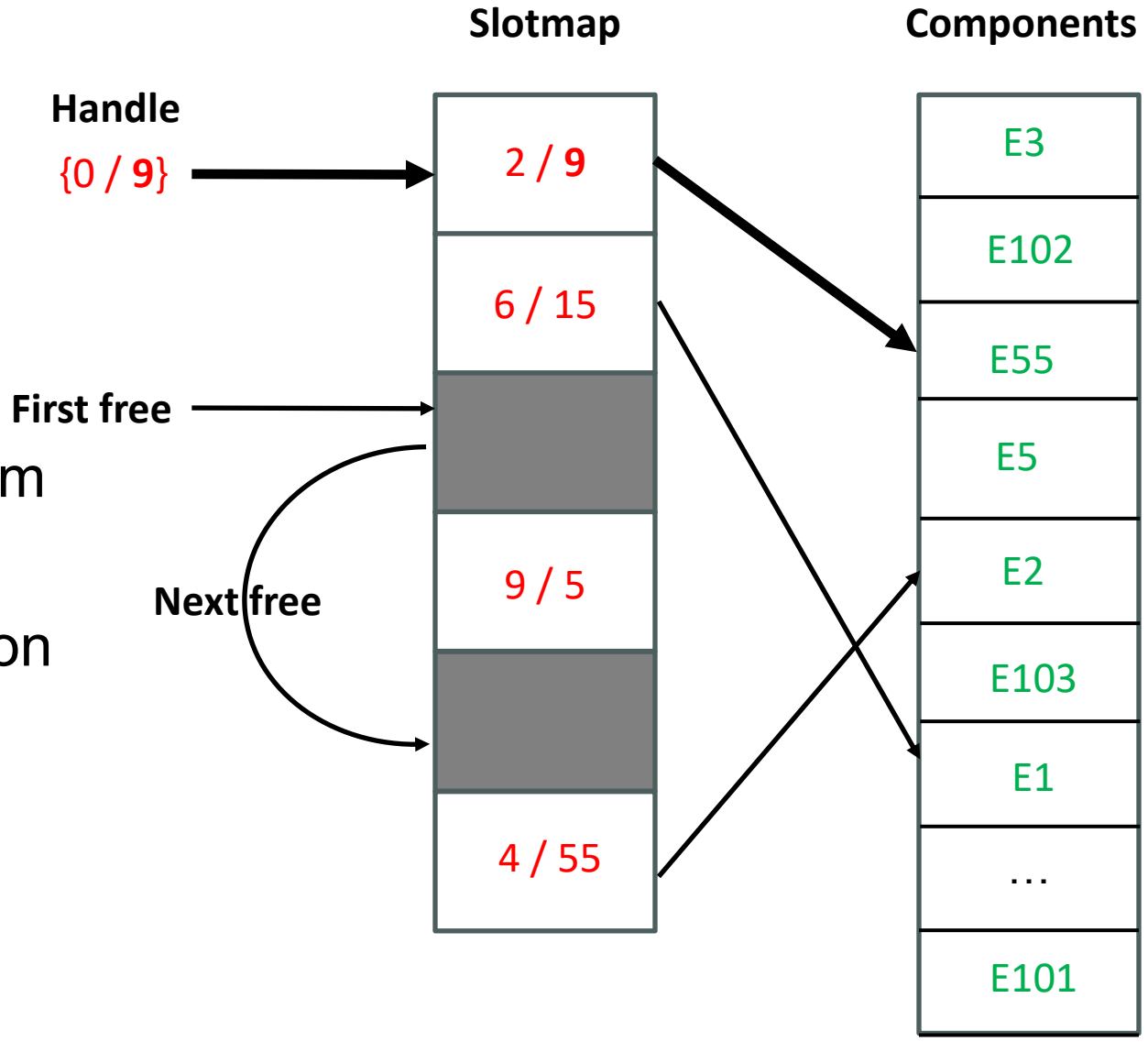
universität wien

Performance

	Flat Map	Map	UO Map	UO Map + Dense	Slotmap + Dense	Slotmap + Archetypes
Insert	O(N)	O(Log N)	O(1) / O(N)	O(1) / O(N)	O(1)	O(1)
Lookup	O(Log N)	O(Log N)	O(1) / O(N)	O(1) / O(N)	O(1)	O(1)
Erase	O(N)	O(Log N)	O(1) / O(N)	O(1) / O(N)	O(1)	O(1)
Iterate 1 Com	Good Cache	Bad Cache	Medium Cache	Good Cache	Good Cache	Good Cache
Iterate *N* Com	Bad Cache	Bad Cache	Bad Cache	Bad Cache	Bad Cache	Good Cache

Slotmaps („Sparse“)

- Handles have **two** parts (8 bytes?): slotmap **index** and **version**
- Provide stable handles
-
- Solve the "dangling reference" problem
- Efficient allocation, lookup, deallocation
- Should never shrink
- **Free list**

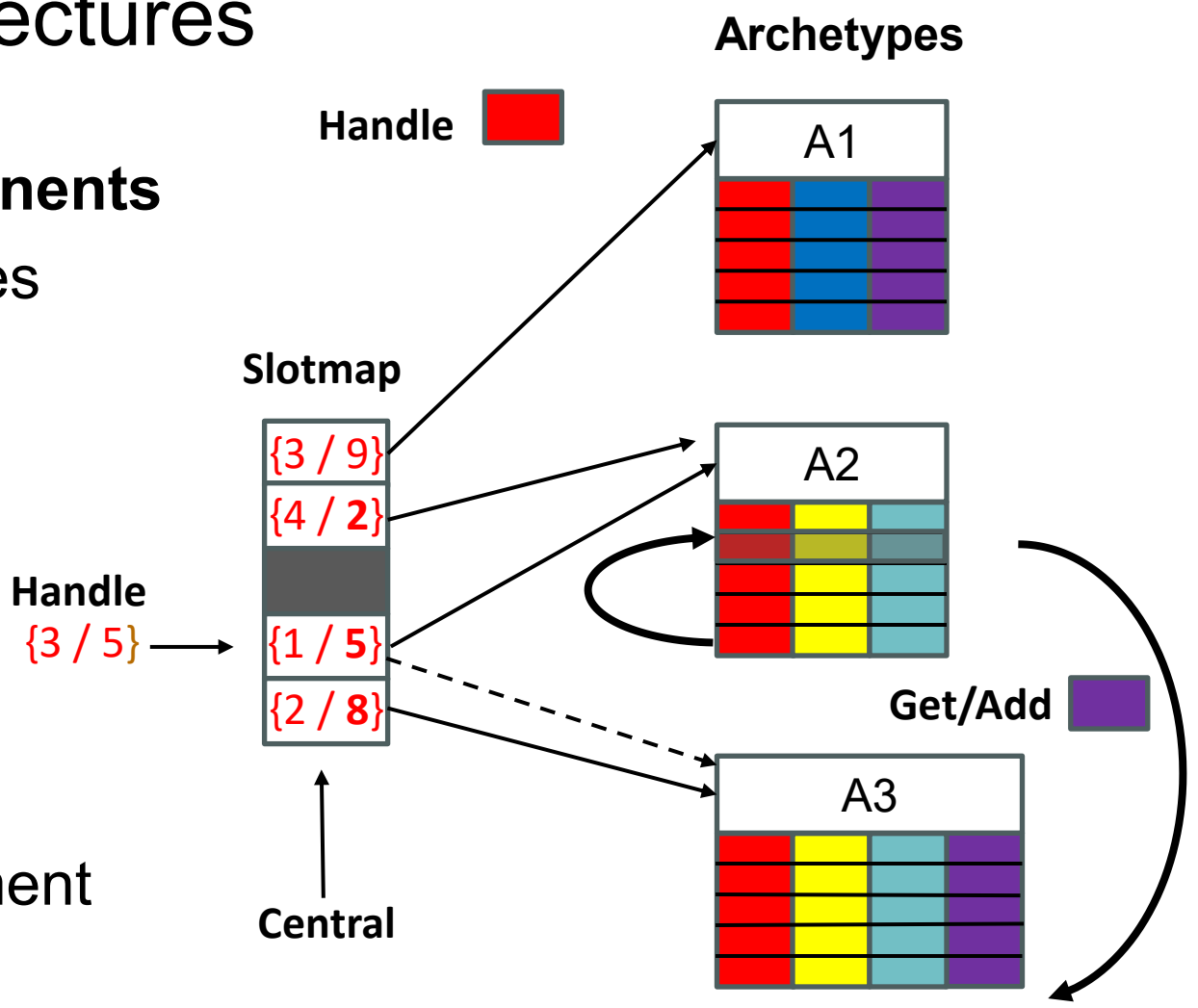


universität wien

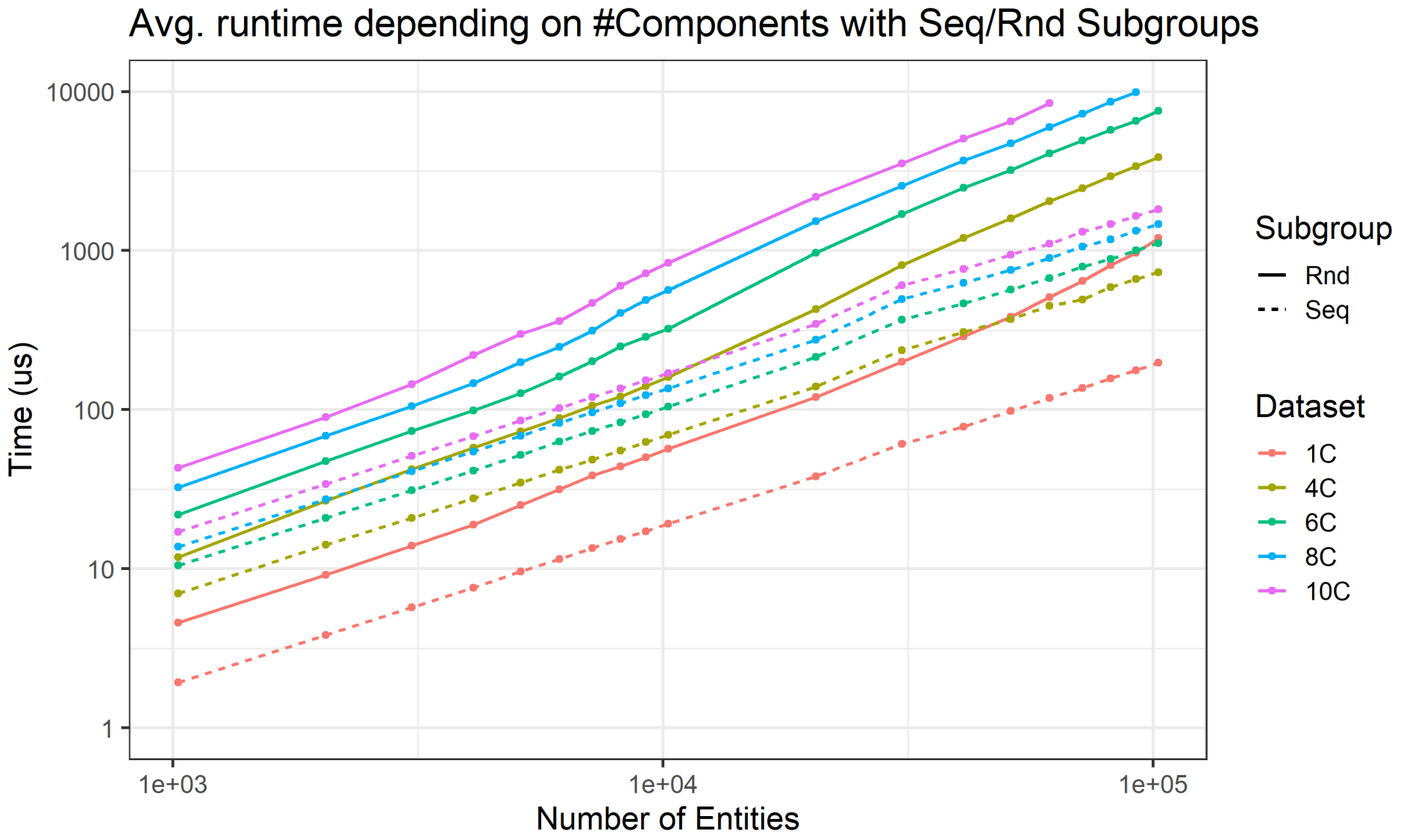
Archetype/Group Based Architectures

- An archetype is a **collection** of **components**
- Entities have the same component types (and tags)
- **Slotmap points** to archetypes
- And **index** to components

- Cache optimal iteration
- Fast O(1) insertion/deletion
- Overhead through archetype management



Sequential vs. Random Access – Reading Data from *N* Columns



universität wien

Consequences

- **Adding/erasing** components or tags **moves** components to another archetype!!!
  - **Do not use C++ references!**
  - Either **no references: transactional** read/write
  - Or: **Ref<T> objects** pointing to **slotmap** rather component

- **Moving/deleting** entities while **iterating** over an **archetype**
  - **Last** entity *L* **moved** into **gap**
  - Iterator **increased** to entity **after** gap (gap now holding *L*)
  - Entity *L* is **never seen** in this iteration
  - Solution: **do not fill gaps** until the iterator reaches the **end** of the archetype

Inserting a new Entity into VECS

universität wien

```
template<typename... Ts>
   requires ((sizeof...(Ts) > 0) && (vtll::unique<vtll::tl<Ts...>>::value) &&
                                    !vtll::has_type< vtll::tl<Ts...>, Handle>::value)
[[nodiscard]] auto Insert(Ts&&... component) -> Handle {
   size_t slotMapIndex = GetNewSlotmapIndex();
   auto [handle, slot] = m_slotMaps[slotMapIndex].m_slotMap.Insert({ nullptr, 0 });
   slot.m_value.m_arch = GetArchetype<Ts...>(nullptr, {}, {});
   slot.m_value.m_index = slot.m_value.m_arch->Insert(handle, std::forward<Ts>(component)...);
   ++m_size;
   return handle;
}
```

```
template <typename... Ts> struct type_list { … };

template <> struct type_list<> { … };

template<typename... Ts> using tl = type_list<Ts...>;
```

universität wien

Parallelization Strategy

`vecs::Manager`:

- Proxy class
- Specialized methods: `ForEachView`, `InsertBulk`, `EraseBulk`

Manager
std::shared_ptr<vecs::Registry> m_system std::shared_ptr<IThreadPool> m_threadpool

```
mng.ForEachView<Position&, const Velocity&>(
    [](auto& pos, const auto& vel) {
    pos.x += vel.x * static_cast<float>(0.1);
    pos.y += vel.y * static_cast<float>(0.1);
});
```

universität wien

Locking

- `std::shared_lock` for reading
- `std::scoped_lock` for writing

Reading access				
Registry lock	GetView<>()	Size()		
Slotmap lock	Get<>()	Exists()	Has<component>()	Has(tags)
Archetype lock	HasAll<>()			

Writing access					
Registry lock	Clear()	Insert()			
Slotmap lock	Erase(entity)				
Archetype lock	Put<>()	AddTags()	EraseTags()	Erase<component>()	EraseBulk()

universität wien

Benchmark Components & Systems

PositionComponent
float x {0.0F} float y {0.0F}

VelocityComponent
float x {1.0F} float y {1.0F}

DataComponent
int thingy {0} double dingy {0.0} bool mingy {false}

DamageComponent
int32_t atk {0} int32_t def {0}

HealthComponent
int32_t hp {0} int32_t maxhp {0} StatusEffect status {Spawn}

SpriteComponent
char character {' '}

MovementSystem
updatePosition(pos, vel)

RenderSystem
renderSprite(pos, sprite)

HealthSystem
updateHealth(health)

DamageSystem
updateDamage(health, damage)

SpriteSystem
updateSprite(sprite, player, health)

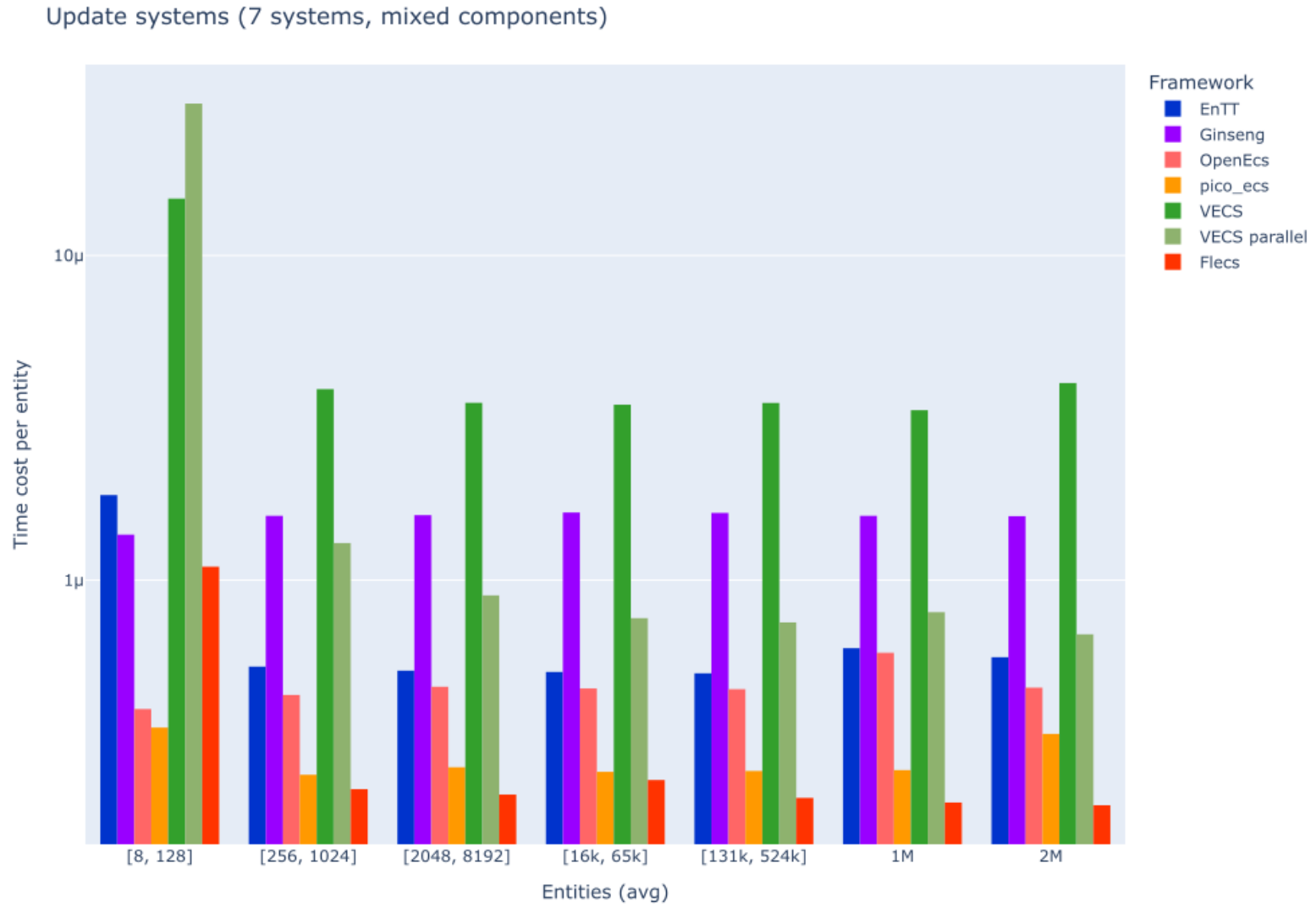
DataSystem
updateData(data)

MoreComplexSystem
updateComponents(pos, vel, data)

https://github.com/abeimler/ecs_benchmark

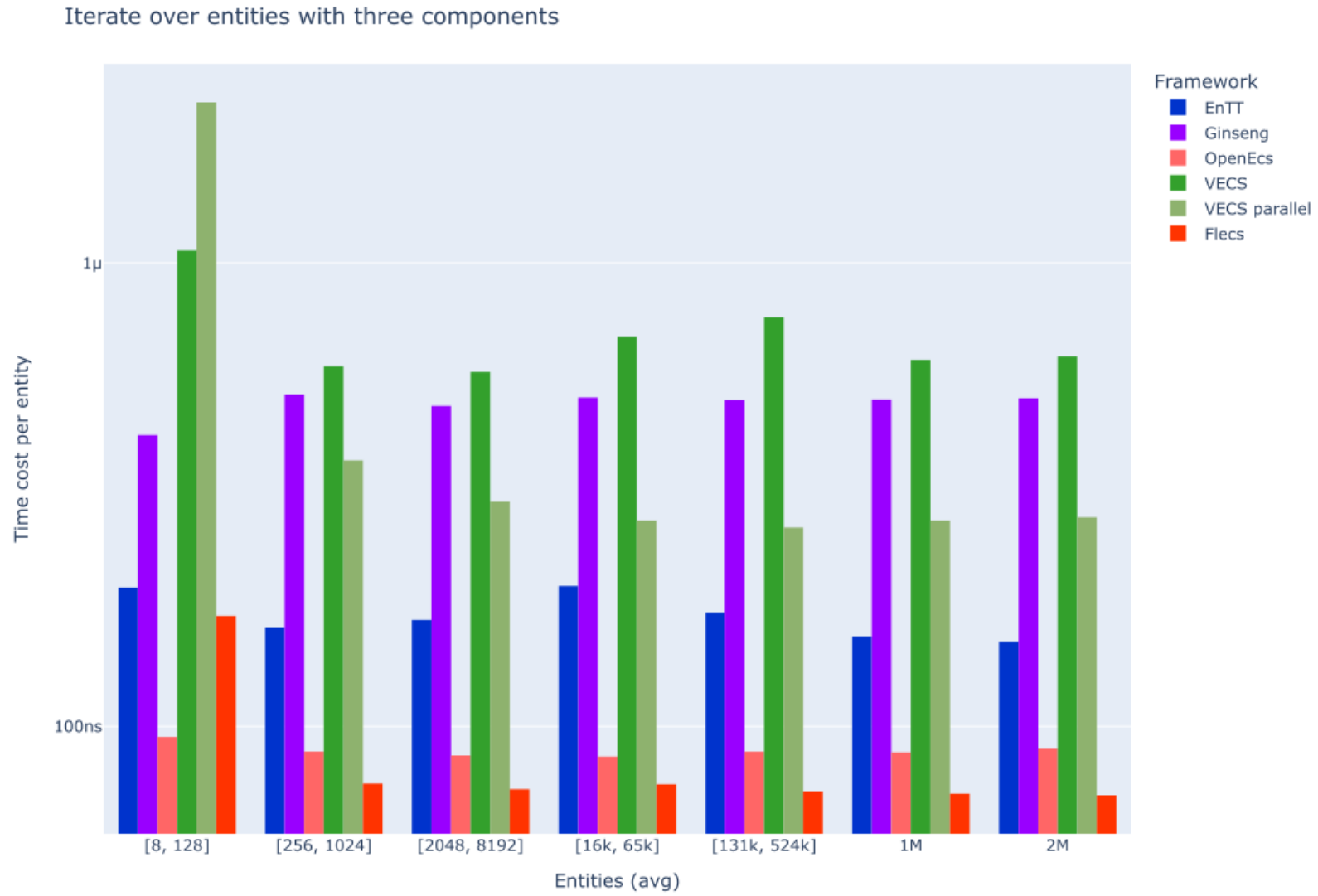
universität wien

Benchmark updating



universität wien

Benchmark iterating



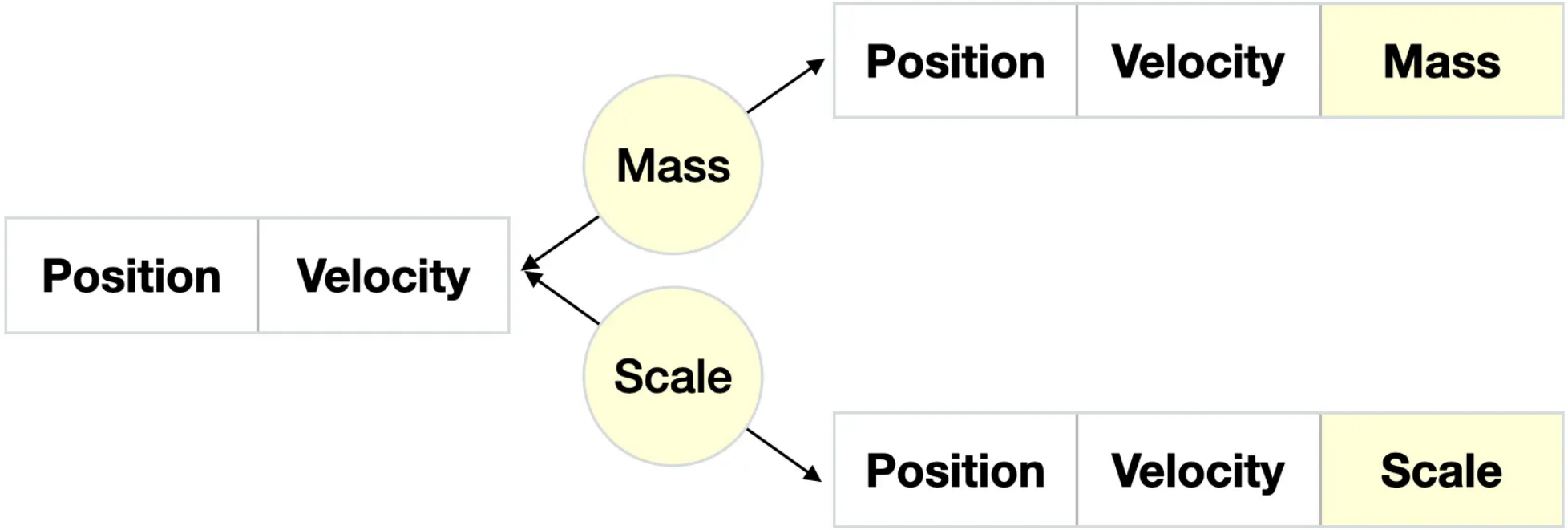
universität wien

Future Optimizations

- Lockless scheduling inspired by Work Contracts (*Michael A. Maniscalco, CppCon 2024*)

- Improve locking granularity – move `ThreadPool` deeper into `Registry`

Future Optimizations: Archetype Graph



Mertens, S. (2024). *Archetype Graph.* Medium. Building an ECS storage in pictures.

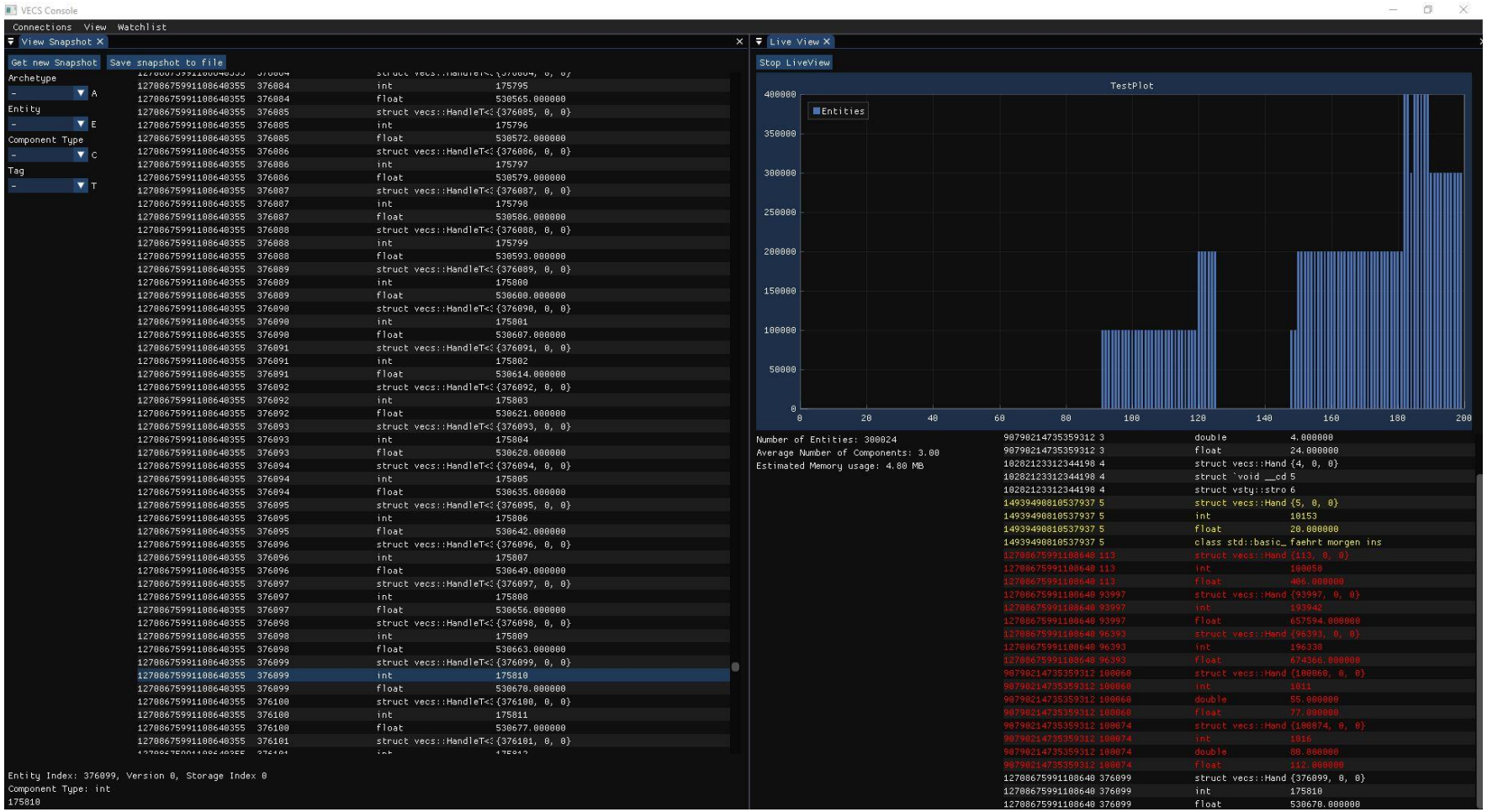
universität wien

Future Optimizations: Bulk operations

- Bulk insertion with parameters
  - `InsertBulk(vector out, componentA{valueA}, componentB{valueB},…)`

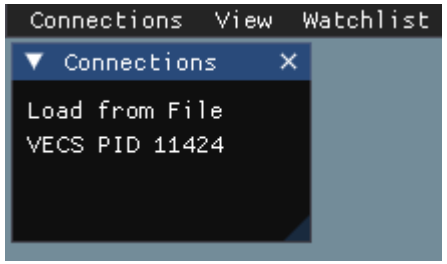
- Bulk erasure for specified components
  - `EraseBulk<componentA, componentB,…>()`

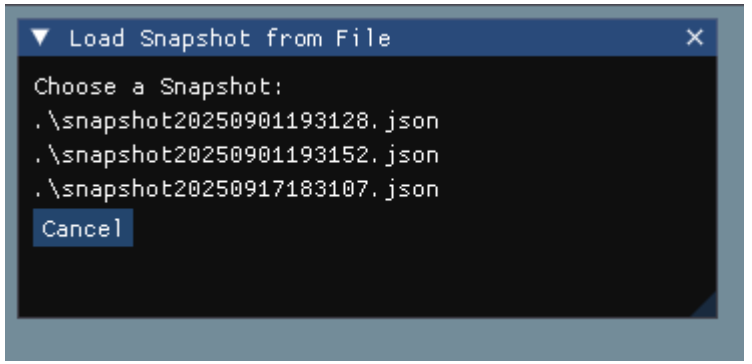
Debugging with the Debug Console



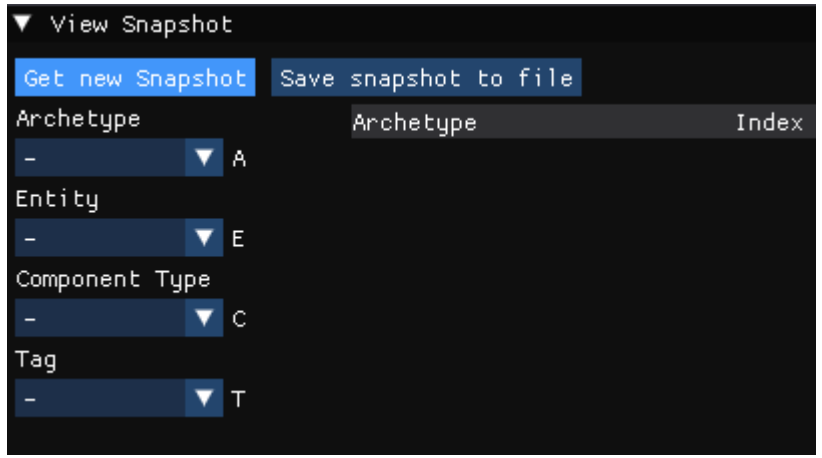
Connect to the Console

- Connect via port and ip
  - If VECS is in **debug** mode, it tries to **connect** to a **console**
  - If not specified otherwise, both console and the VECS application use the same default port
  - Possible to debug from another device in the same network
- Load a saved snapshot

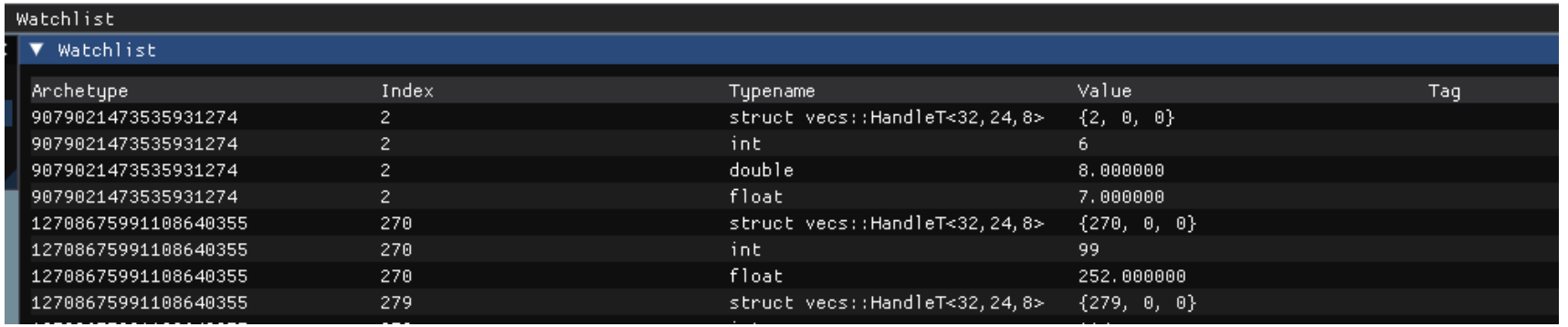




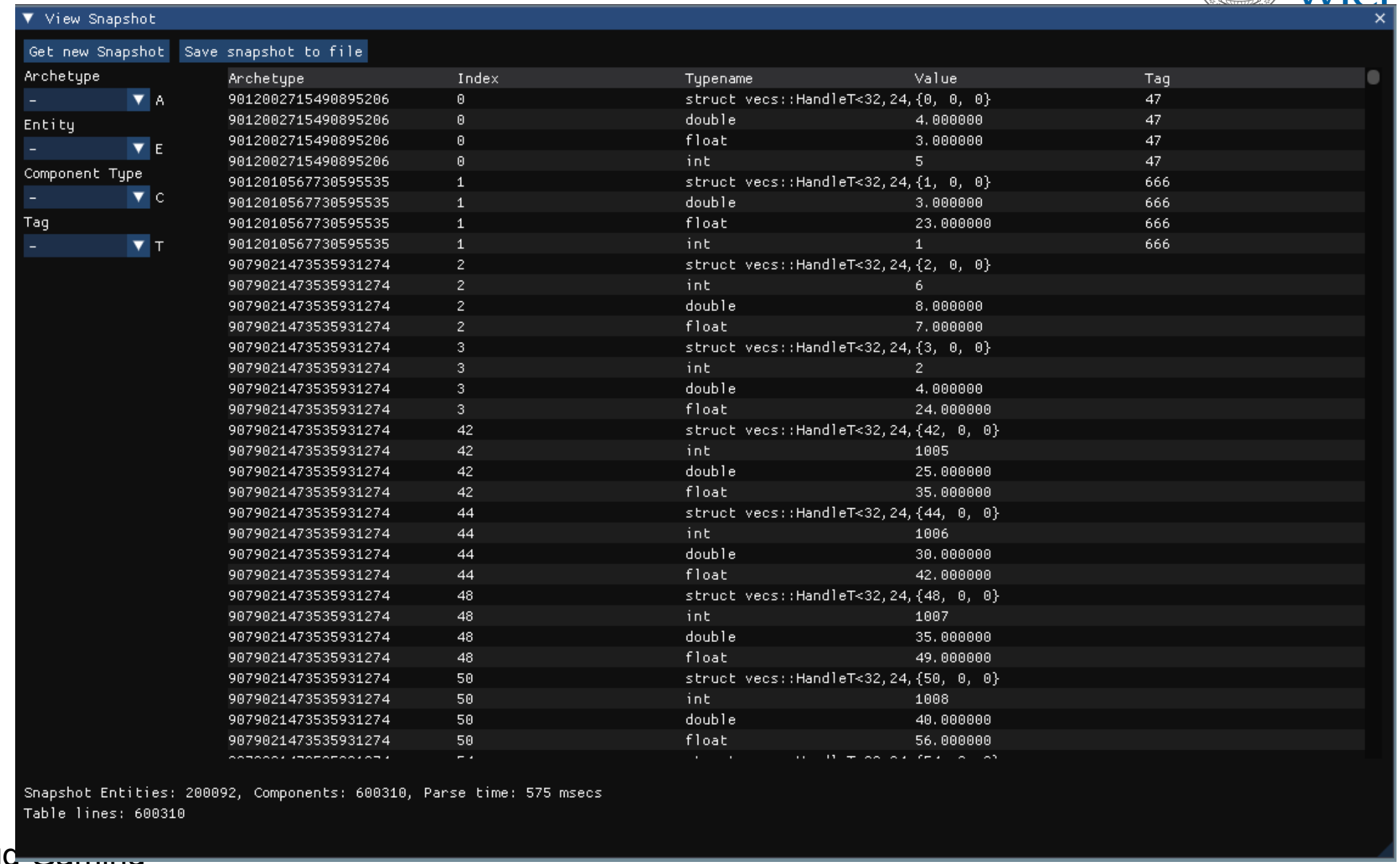
Snapshots



- Get snapshot to view all entities
- Look through all components
- Filtering possible
  - Find specific archetype, entity, component…
- Add entities of interest to the watchlist to see changes and status in the live view
- Save snapshot to file



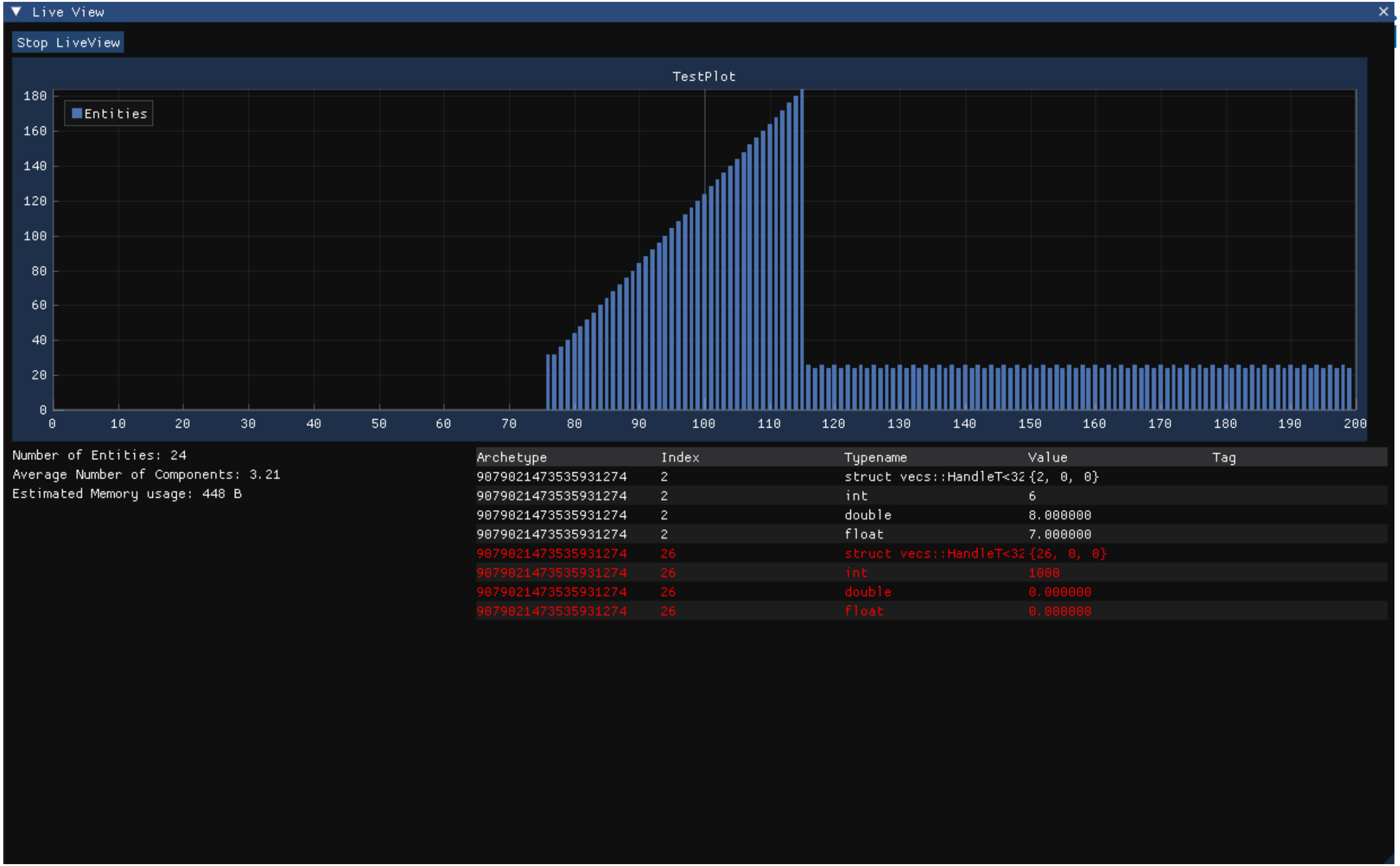
Archetype	Index	Typename	Value	Tag
9079021473535931274	2	struct vecs::HandleT<32,24,8>	{2, 0, 0}	
9079021473535931274	2	int	6	
9079021473535931274	2	double	8.000000	
9079021473535931274	2	float	7.000000	
12708675991108640355	270	struct vecs::HandleT<32,24,8>	{270, 0, 0}	
12708675991108640355	270	int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


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
9079021473535931274 42 struct vecs::HandleT<32,24,{42, 0, 0}>
9079021473535931274 42 int 1005
9079021473535931274 42 double 25.000000
9079021473535931274 42 float 35.000000
9079021473535931274 44 struct vecs::HandleT<32,24,{44, 0, 0}>
9079021473535931274 44 int 1006
9079021473535931274 44 double 30.000000
9079021473535931274 44 float 42.000000
9079021473535931274 48 struct vecs::HandleT<32,24,{48, 0, 0}>
9079021473535931274 48 int 1007
9079021473535931274 48 double 35.000000
9079021473535931274 48 float 49.000000
9079021473535931274 50 struct vecs::HandleT<32,24,{50, 0, 0}>
9079021473535931274 50 int 1008
9079021473535931274 50 double 40.000000
9079021473535931274 50 float 56.000000
Snapshot Entities: 200092, Components: 600310, Parse time: 575 msecs
Table lines: 600310

universität wien

Live View

- Graph showing **number of entities** over time
- Development over time, graph resizes to visible range
- Watchlist is displayed under the graph and marked:
  - White : no changes
  - Yellow: one or more component values changed
  - Red: entitiy was deleted
- Shown statistics
  - Current number of entities
  - Average nuber of components per entity
  - Estimated memory usage (of the entities)

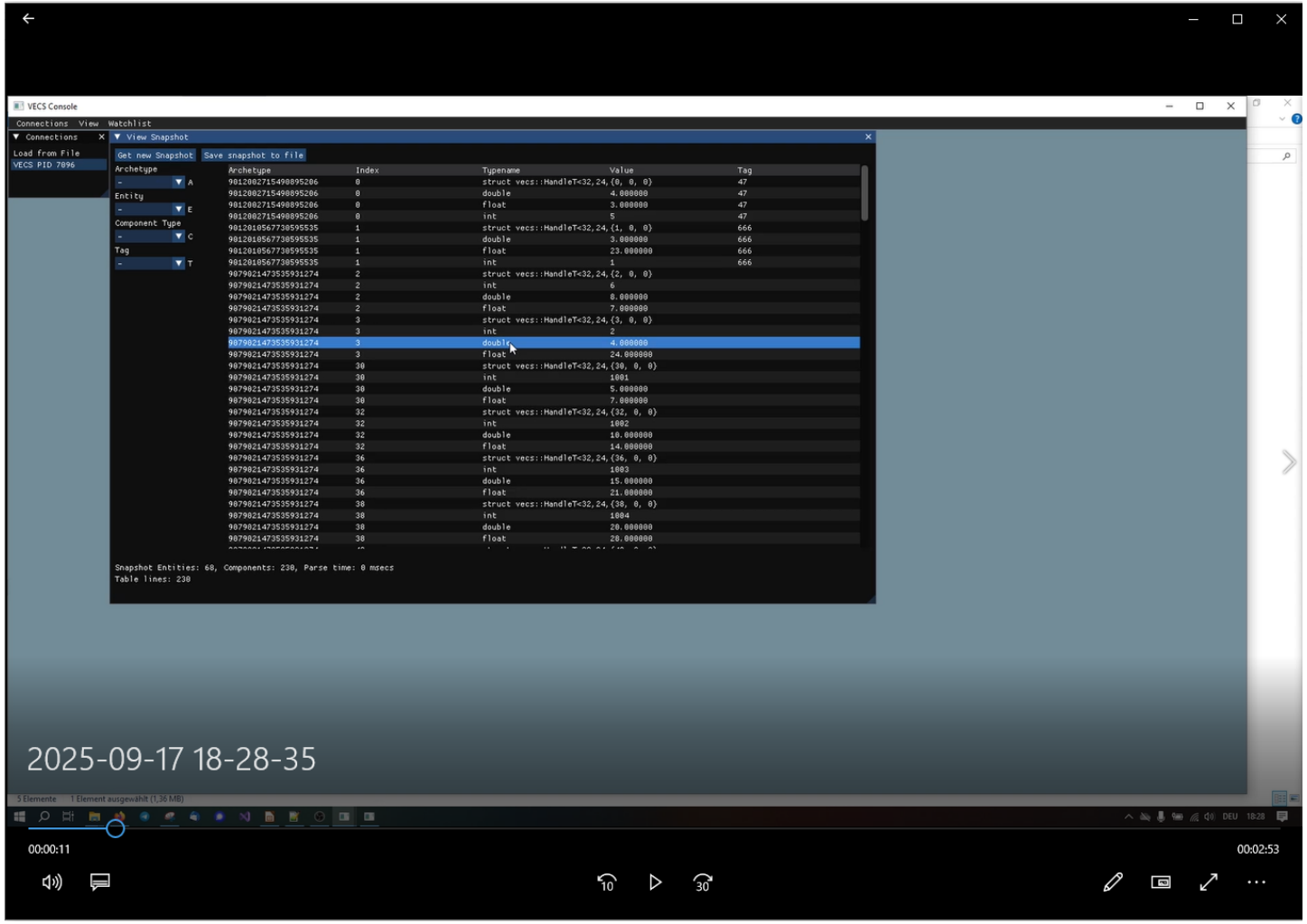


Number of Entities: 24
Average Number of Components: 3.21
Estimated Memory usage: 448 B

Archetype	Index	Typename	Value	Tag
9079021473535931274	2	struct vecs::HandleT<32	{2, 0, 0}	
9079021473535931274	2	int	6	
9079021473535931274	2	double	8.000000	
9079021473535931274	2	float	7.000000	
9079021473535931274	26	struct vecs::HandleT<32	{26, 0, 0}	
9079021473535931274	26	int	1000	
9079021473535931274	26	double	0.000000	
9079021473535931274	26	float	0.000000	

Using the Console

- Console Demo



Conclusions

- ECS are a useful architectural pattern
- Widely used in the games industry
- Many architectural choices with performance impact
- Lots of optimization necessary
- Parallelization is difficult but feasible
- Debugging is hard and requires useful tools
- Use LLMs to create them
- Further optimizations: graphs, bitsets, less STL, …
- Outook: C++26 static reflection?