

# Entity Component framework of Fools Engine

What are we even talking about?

Entity-Component-System

!=

Entity-Component system

Since a departure from the “Actor” model with inheritance oriented implementation for scene representation in commercial game engines, online tutorials, blog posts, and conference presentations towards the “Entity-Component” framework with a composition oriented approach, implemented in form of generic, runtime dynamic, game agnostic system, term “Entity-Component system” has gained popularity.

This creates a great deal of confusion, as “Entity-Component system” and “Entity-Component-System” (data oriented design pattern sometimes used in those frameworks and a trendy buzzword) can only be differentiated from context. Not to mention improper use of those terms by people not knowing and understanding the difference.

Additionally, scene representation frameworks in different game engines use incompatible fundamental nomenclatures (e.g. GameObject, Pawn, Actor, Entity, Node).

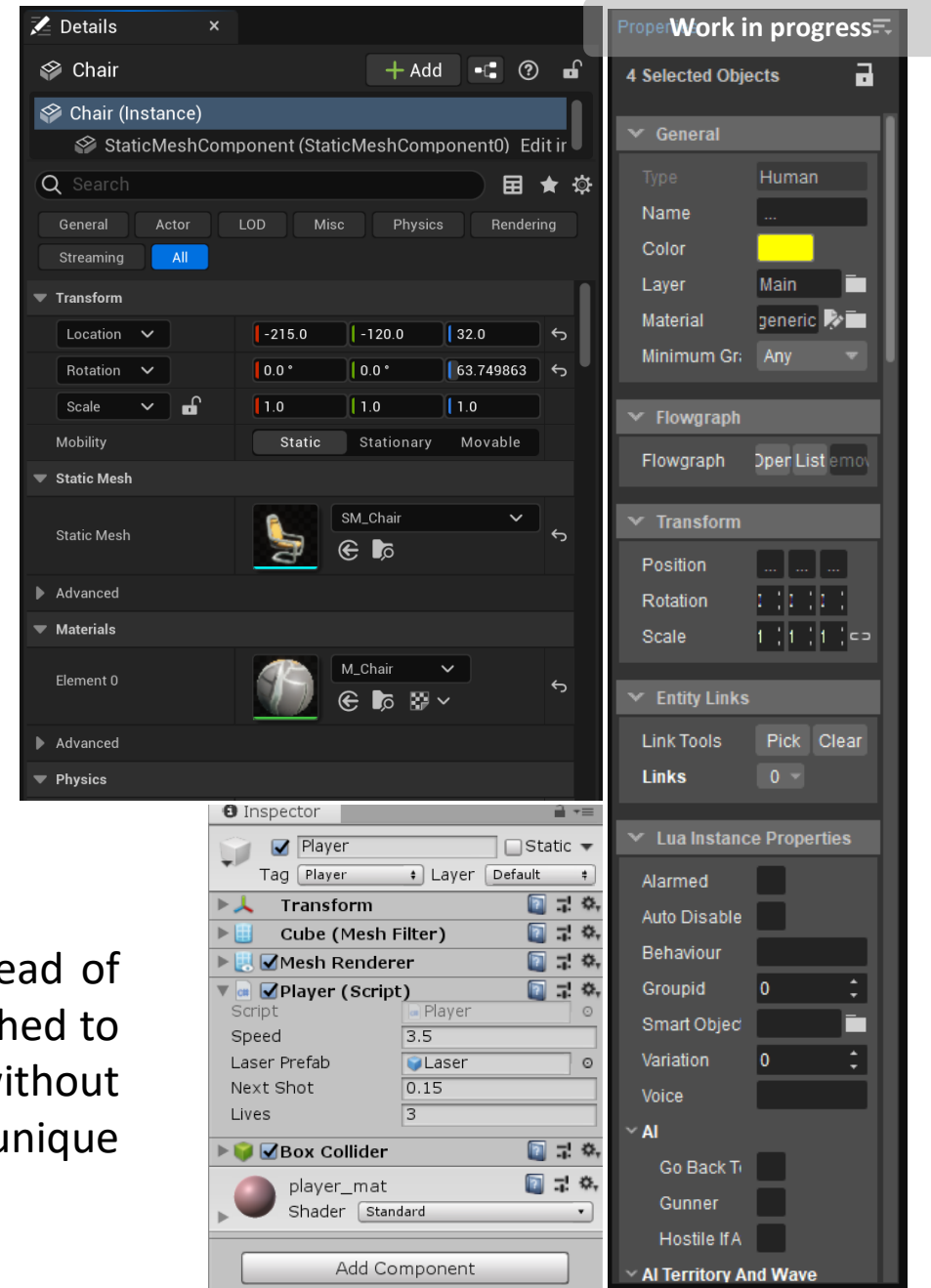
# Entity Component framework of Fools Engine

Framework – what do we need it for?

What a framework provides:

- The game is built out of "Entities", which themselves are composed of "Components".
- Components represent the game.
- Components are meant to be reusable.
- An engine provides a set of components well integrated with big engine systems.
- Entities fulfill the service locator pattern - they can be used to retrieve a child component by type.
- Components can be dynamically added to an entity.

The main purpose here is to enable dynamic, runtime composition. Instead of entity types being hard-coded, they can be constructed in the editor, attached to the engine systems (rendering, physics, etc.), serialized, and deserialized without writing a single line of code. This allows designers to create their own, unique kinds of objects.



# Entity Component framework of Fools Engine

State of the rot – architecture issue

```
// get a component of type T, or null if it does not exist on this game object
template<typename T>
T* GetComponent()
{
    for (auto i : m_Components) { T* c = dynamic_cast<T*>(i); if (c != nullptr) return c; }
    return nullptr;
}
```

The drawback of this solution is that the core code of the framework is in some sense constructing a whole new meta-language, and a VM to run that meta-language on. The problem is so abstract and generic, that implementing a solution is extremely difficult. A fully elegant one is perhaps impossible, as it requires breaking numerous principles of a good software architecture design.

To name a few examples:

- Rampant virtual methods obfuscate both the execution and data flows by breaking the single responsibility principle
- The entity's implementation of the service locator pattern for retrieving components breaks Liskov's substitute principle
- Allowing to retrieve components anywhere in the code creates implicit, unmanageable dependencies
- Components rely on inheritance of at least an interface, but also often of implementation what violates composition over inheritance principle

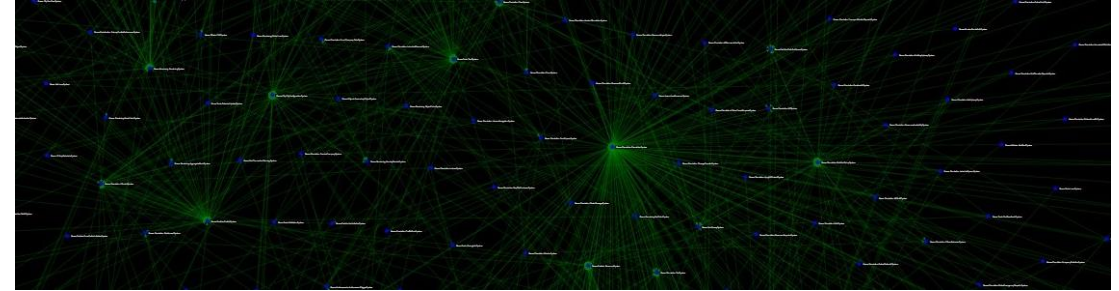
# Entity Component framework of Fools Engine

Work in progress

State of the rot – framework usability

Game engines typically include 2 interfaces for their scene representation framework:

- for engine build-in systems (rendering, physics, etc.) and systemic game features (crowds, traffic, etc.)
- for unique, high-level gameplay features (player's character movement, camera control, etc.).



Systems in City Skyline 2

The systemic side often creates implicit dependencies from the get-go and requires specific predefined setups to work, sometimes even as an undocumented technological debt. The gameplay side usually comes in form of scripting (C#, Lua, BluePrints, etc.) with full access to all data of the scene, intended to be used by various types of designers without knowledge of good programming principles and practices, who inevitably end up abusing that access.

Games are complex. That complexity is unavoidable and has to be resolved somewhere. Most entity-component frameworks settle down on a model that supports only a narrow set of design patterns for authoring a game's logic. That puts all the responsibility and difficulty of managing that complexity on game developers. Their lack of intuitive grasp of the model and its design philosophy leads to misuse and overcomplication, even with a logical understanding of its implementation architecture!

# Entity Component framework of Fools Engine

State of the rot – framework utilization

Single, from a player's perspective conceptually indivisible, element of the game (including its behavior) is implemented using multiple components and systems (and potentially entities) with various tricky dependencies between them, but also with their own internal complexities. Games usually have unintuitive execution flows, as a result of this coupling, combined with interactions between systemic and gameplay interfaces. That makes reasoning about game code difficult (sometimes even literally impossible without actually running the game and testing it). Familiarizing yourself with existing game code or simply keeping up with it becomes a very big cognitive load. At the end of the development process, the game is too complex for anyone to know how it works anymore. Pieces of spaghetti code are also sometimes carried over to the next project as black boxes with legacy dragons and get stacked over time like a rotting onion.

But even with proper usage, a simple design of a scene representation framework's execution model causes problems, because it cannot be suitable for all types of features. If it is closed and restrictive, developers will fight it with workarounds and dirty hacks. If it is open, it will cause inconsistencies and will result in a spaghetti of dependencies.

# Entity Component framework of Fools Engine

## Alternatives – hot loading

Programming languages already come with support for composition as a feature – there is no need for a bloated framework on top to use it. The tricky part is to make it runtime dynamic, but even that is achievable. We could dynamically generate code, compile it into a dll, and hot load it. This approach is impractical for the game's runtime, but a fully dynamic composition is needed for very few games. The main goal is to enable designers to author objects in the editor during the development process. In this case, we can even simplify things by defining objects using scripting languages instead of the editor's GUI with complex declarative native code behind it.

That however does not solve the problem at its root. The engine's core will never be able to use specific object types without an abstract interface over them unless we are willing to auto-generate its code and recompile it too whenever a new type of object appears. And that idea does not sound promising at all. So all the issues we had with the entity-component relation architectural design are now shifted upon a relation between an entity and a specific entity type. Perhaps this is a more elegant approach that would enable greater control over the game's runtime execution. However, it still does not address the issue of helping with managing the complexity of our entities' behaviors.

# Entity Component framework of Fools Engine

Alternatives - embracing complexity

Complex  $\neq$  **Difficult**

Simple  $\neq$  **Easy**

Instead of fighting the complexity of a game by denying its existence, we could come to terms with it and accept its inevitability. Acceptance does not mean indifference towards the problem. Instead of pushing the responsibility down to game developers, we could provide them with tools for the proper expression of the game's logic.

Organization and structurization of both data and logic should be the fundamental roles of the scene representation framework. That requires a model that includes a comprehensive set of features for the representation of not only data but also execution flow. The complexity and diversity of its subject are an argument **for** the complexity of the model! The generality and abstractness of the model are an argument **for** the complexity of the model!

Putting data and logic in order properly and comprehensively using a framework is an occasion to better conceptualize and optimize game code while keeping it manageable, extendable, and reusable. The diversity of tools available in a framework for that purpose is not a sign of a bad design.

As long as we keep our framework's execution model composed in an intuitive way of standardized, corresponding concepts we can make it as complex as we need.



# Entity Component framework of Fools Engine

Work in progress

Design challenges

## Encompassed Data and Logic

Complex

Divers

Dynamic

Densely interdependent

## Game development process

Complex

Random

Volatile

## Usability

Multiple users in one project

Both technical and non-technical users

Users of all levels of familiarity and experience

## Run-Time

Performant

Hardware scalable

Stable and bug free

## Integration

Engine functionalities

Systemic game features

Unique game features



# Entity Component framework of Fools Engine

Work in progress

## Goals

### Performant

- Build-in extensive parallelization (including engine-game concurrency)
- Data oriented
- Service oriented (pay for what you need)

### Flexible

- Allows for multiple ways of implementing features
- Organizes game code
- Favors extendable and modifiable game code
- Customizable

### Safe

- Isolates game features
- Has clear rules for data access and dependencies
- Resolves execution order dependencies
- Prevents initialization order issues
- Does not allow for data races

### Intuitive

- Provides multiple levels of conceptualization
- Has corresponding rules on them
- Has predictable execution flow
- Favors simple standardized designs patterns
- Fully symmetric life-cycle API
- Unified (engine build-in features and game code)

# Entity Component framework of Fools Engine

Work in progress

## Overview of existing design patterns

The concept of a scene representation architecture is not new. In fact, it's as old as the concept of a video game itself. Various excellent programmers in the past tackled it and approached it from different perspectives and still do to this day. There are great lessons in their successes and failures.

Entity design			
	inheritance	composition	
		unique	universal
Object - OOP			
ID only - DOP	N/A		

Update calls		
Sequential	Per entity	
	Per component	Individual Through entity
Concurrent	Actor model	
	Relational model	ECS "Inverse" ECS

Execution order control	
Multi-staged	Initialization
	Simulation

Component design			
	Banned	Allowed	Enforced
Logic			
Inheritance			
Hierarchical components			
Spatial components			?
External storage			?
Pointer stability			

Components storage		
Internal		
External	Allocated individually	
	Allocated sequentially	Arrays
		Paged vectors
		(Paged) "sparse" vectors
		(Paged) sparse sets
	(Paged) archetypes	

Spatial hierarchy			
What is a node	Component (entity should have a single root component)		
	Entity	Hardcoded	
		As component	Enforced Optional
Definition	Parent		
	List of children		
	Siblings as linked list + first child		
Reference type	Pointer	To entity	
		To component	
	ID		
Application of changes	Once per frame (delta accumulation)		
	Instant		
Global transform recalculation	Not cached (recalculation upon access)		
	Cached	Recalculated upon change	
		Dirty flag set upon change	Recalculated once per frame Recalculated upon access if needed
Skeleton's space	Local - translated to global for rendering		
	Global	Fixed	
		Recalculated when entity's transform changes	

# Entity Component framework of Fools Engine

## Corner Stones

### ECS Storage

- Entity-Component-System compliant components storage architecture
- Customizable optimizations
- Full continuity of a component's storage (raw pointer iteration)
- In frame pointer stability – no reallocation during component / entity creation and deferred destruction

### Build-in spatial hierarchy

- Fully managed by the engine, hidden from the user
- Automated propagation of transform changes
- Global transform is always accurate on access
- Dictates update order (parent => child)
- Children are defined as a linked list to avoid additional heap allocations and indirections during hierarchy traversal
- Storage structure reflects hierarchy structure (cache-friendly traversal)

### Comprehensive feature set

- Provides methods to represent every aspect of a scene, including fully independent purely visual features (foliage, particles, etc.)
- Different composition/abstraction layers to choose from to implement any game feature in a suitable way
- Full isolation within the same conceptual layer
- Trivial parallelization of updates and initialization

### 2 levels of logic: Behavior and System

- Behavior deals with object's internal state
  - Component update
  - Component–component data transfer
- System deals with world's state
  - Mass update of components (ECS style)
  - Object–object data transfer

### Explicit execution flow control

- No default update calls from the engine – enrollment always required
- A simple way of controlling execution order with 2 levels of control
  - 1 - Simulation stages (big sync-points)
  - 2 - Numerical priorities for update enrolls

# Entity Component framework of Fools Engine

Decisions => hybrid everywhere

## Entity

- ID only
- Some components universally guaranteed and protected (e.g. transform)
- Other dynamically added

## Component storage

- Off-the-shelf solution – EnTT library
  - Paged sparse sets
  - Archetype-like customizable optimizations

## Execution order control

- Multi-staged execution
- Explicit per instance update enrolls
- Explicit numerical priorities per update enroll for finer control

## Component

- Logic allowed and not enforced, banned referencing other components/entities – black box with no dependencies
- Inheritance enforced (basic types)
- Further inheritance allowed, but considered as independent types
- Hierarchical components not available
- Spatial components allowed
- External storage allowed, but discouraged
- Pointer stability in frame, but not across frames – deferred destructions

## Spatial hierarchy

- Entity is a node stored as a component
- Definition: siblings as linked list + first child + parent
- Reference as ID
- Application of changes: instant
- Global transform: cached, recalculated upon access if needed
- Skeleton space: local
- No caching of global transforms of special components

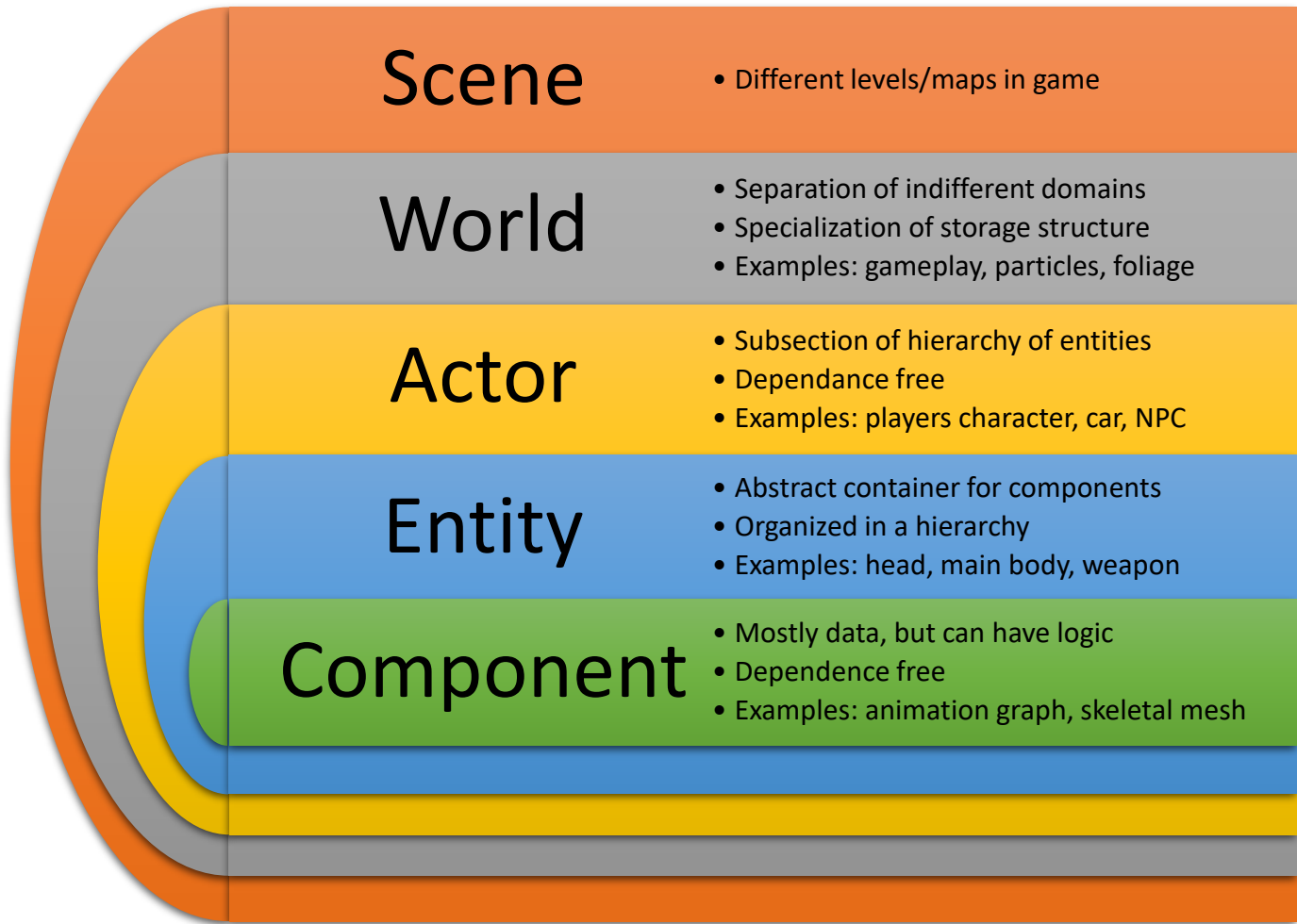
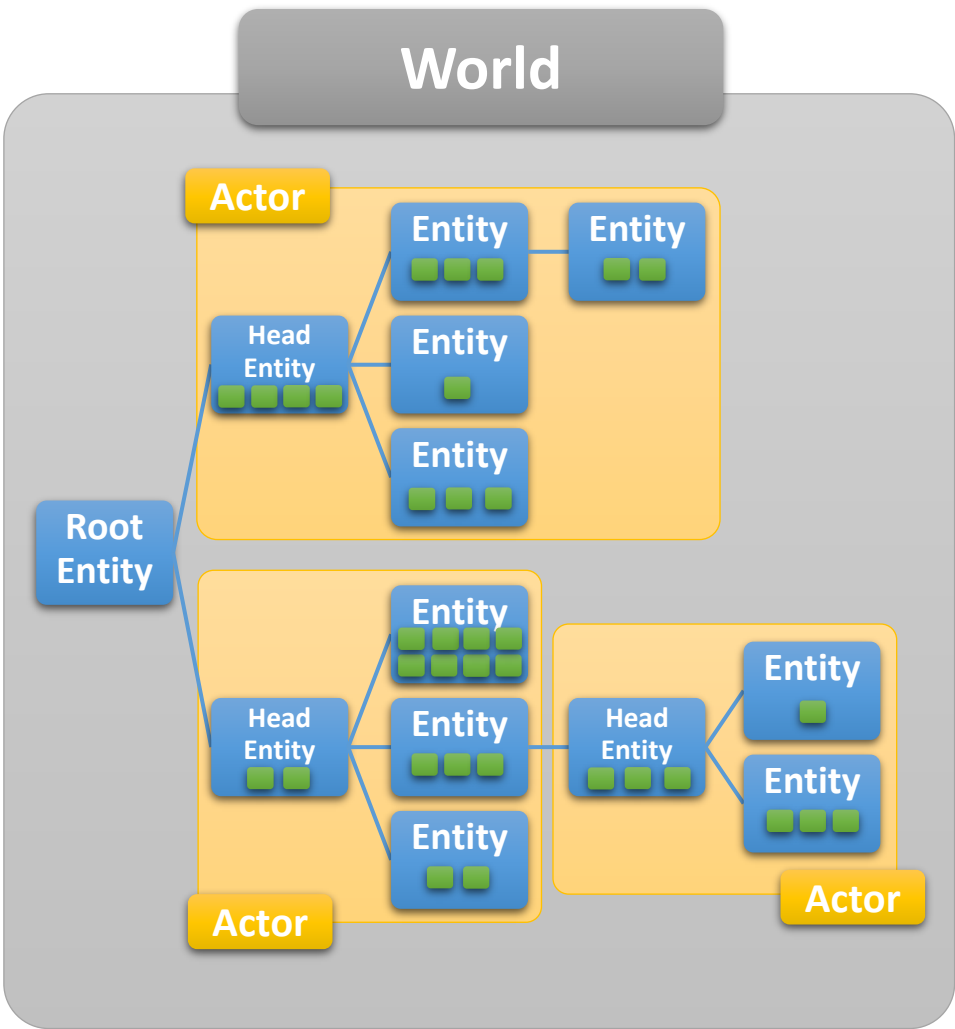
## Update calls

- Concurrent – hybrid
  - Relational model (ECS)
  - Actor model:
    - Actor is a subsection of the entities' hierarchy representing a single object (car, NPC, etc.)
    - Actors are updated in the order defined by the hierarchy
    - Internal execution of an actor is sequential using Behaviors that operate on components

# Entity Component framework of Fools Engine

Work in progress

## Structure overview



# Entity Component framework of Fools Engine

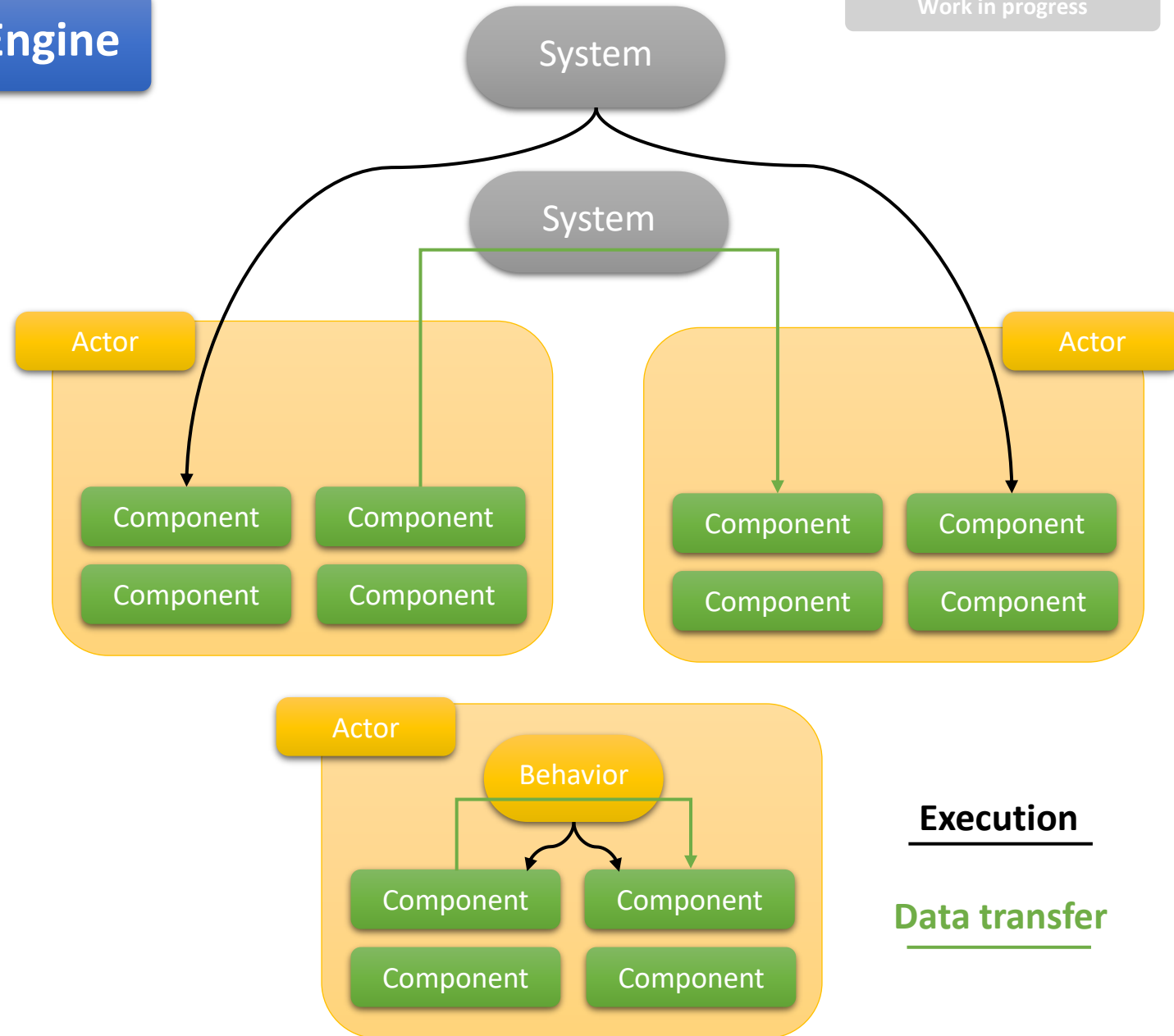
## Execution and data routing

### System

- Global – owned by World
- 2 roles:
  - Actor-actor data transfer by state injection (similarly to messages in actor model)
  - Components update (ECS style)
- Can be an opt-in or opt-out service
- Opt-in can be direct or using empty flag component and permanent or one-time
- Can use direct component references (entity IDs)
- Systems register for update from world
- Examples: AI stim system, crowd sim system

### Behavior

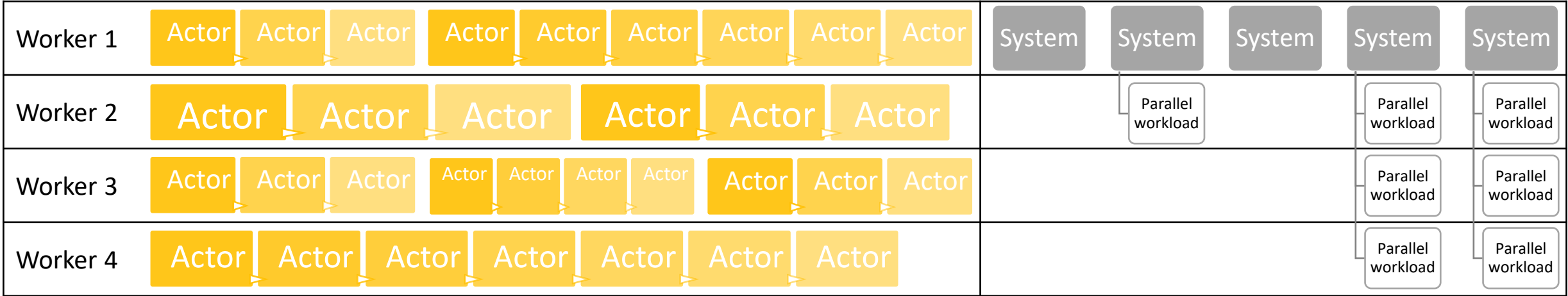
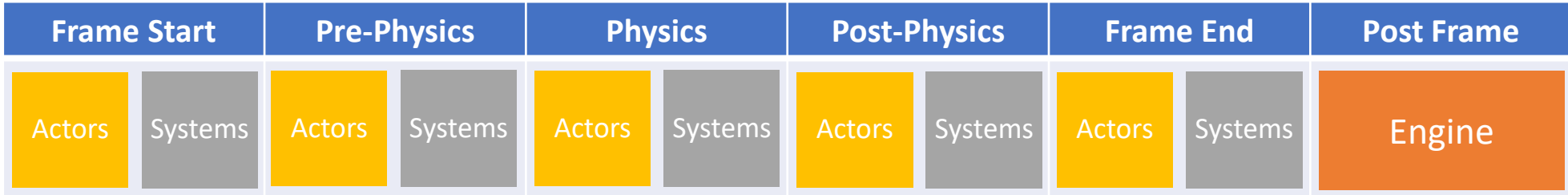
- Actor's "internal system" – owned by Actor
- 2 roles:
  - Execution flow control of dependent Components
  - Component–component data transfer
- Uses direct component references (entity IDs)
- Actors update is an update of its behaviors
- Behaviors register for update from Actor
- Examples: animation, characterController



# Entity Component framework of Fools Engine

Work in progress

Update Order

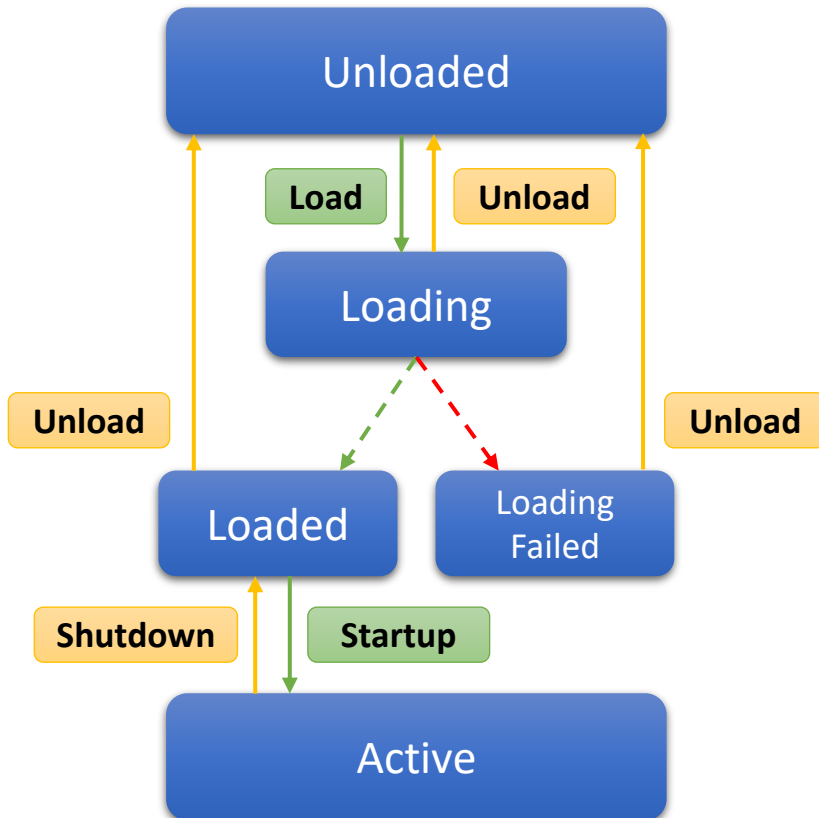




# Entity Component framework of Fools Engine

## Component

### Life Cycle



Work in progress

### Basic building block

- Can have data
- Can have resource references
- Can have logic
- Cannot reference anything in the scene ("black box")

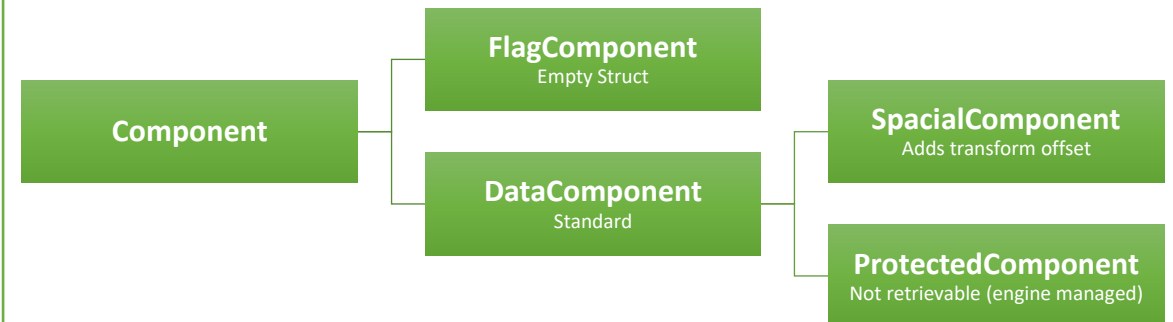
### Can have logic

- Mostly utility methods for operating on that component
- Can restrict access to its internal state and expose getters and setters
- Can perform its own update, but engine does not provide update call
- Can be updated by a Behavior or a System (directly or by a call)

### Stored in global ECS storage

- If inherits from another component they are considered as unrelated types
- Only one instance of a given component type per one EntityID in global storage

### Types



# Entity Component framework of Fools Engine

Work in progress

## Entity

### Abstract container for Components

- Abstract container for conceptually/spatially connected components
- Is represented in memory as `uint32_t` - EntityID
- EntityID acts as a storage access key to components of an Entity
- Can contain only 1 instance of each component type
- Every Entity has LocalTransform, GlobalTransform, EntityName, HierarchyNode and Tags components

### Unique EntityIDs

- `NullEntityID != 0`, but `constexpr`
- `RootEntityID = 0`

### Entity class

- Thin wrapper around `World*` and `EntityID` acting as a handle
- Provides utility API for operating on components of an Entity
- Provides additional handles and API to protected components responsible for carrying out functionalities of the whole model (e.g. Transform, HierarchyNode)

### Scene Hierarchy

- All Entities in a Gameplay World are part of one spatial hierarchy
- Defined by dedicated component – HierarchyNode
  - Children defined as linked list
  - Children are sorted by their EntityID
  - Caches information about its hierarchy depth and children count
- Fully managed by engine
- Used to organize storage of some components (e.g. Tags, HierarchyNode, Transforms) for cash friendly traversals

# Entity Component framework of Fools Engine

Actor

Work in progress

## Abstract object

- Abstract object representing single “thing” from players perspective (e.g. players character, NPC, vehicle)
- Uses a subsection of hierarchy of Entities for storing it's Components
- Stores it's own data in an ActorData Component in a HeadEntity
- Owns and manages Behaviors
- Provides Update calls to Behaviors
- Caches a dedicated list of Behaviors and their overridden Update methods enrolled for update per each simulation stage
- Resolves execution order dependencies between Behaviors within each simulation stage using numerical priorities

## Actor class

- Thin wrapper around Entity class
- Provides utility API for operating on Behaviors, Entities and Components constituting an Actor

## Behavior

- Updates Component instances in a single Actor
- Facilitate control of data and execution flow between Components
- Can have properties and transient run-time state
- Cannot be referenced by anything – nothing else knows about it's existence

## Update

- Actor update is an update of all it's Behaviors
- Actors are updated by engine in order defined by special hierarchy (parent => child) to ensure correctness of Transform changes propagation
- Update of all Actors is a multithreaded graph search of scene hierarchy
- An Actors hierarchy is maintain in parallel to Entities hierarchy as a simplified, collapsed version

# Entity Component framework of Fools Engine

## System

### Roles

- Global – owned by World
- Operates as a service for actors (behaviors and components):
  - Can be an opt-in or opt-out service
  - Enrollment can be direct or using FlagComponent
  - Enrollment can be permanent or one-time
- 2 roles:
  - Updating multiple instances of the same Component type (ECS style)
  - Actor-Actor communication using state injection (similarly to messages in actor model)
- Systems have internal locks to resolve concurrent accesses from Actors
- System cannot reference other systems

### Execution

- System can parallelize internally their workload
- Systems are executed sequentially by World
- System registers for update during a particular frame simulation stage (can register for multiple stages)
- World resolves execution order dependencies between Systems within each simulation stage using numerical priorities
- World caches a dedicated list of Systems and their overridden Update methods enrolled for update per each simulation stage

# Entity Component framework of Fools Engine

## Deletions

### Pointer stability issue

- Deletion of a Component (and by so also deletion of Entities and Actors) causes reshuffling of other Components of the same type in memory - invalidation of pointers and references
- All deletions has to be deferred to the end of the frame to achieve in-frame pointer stability
- Behaviors and Systems are not affected by this issue, as they do not reside in ECS storage

### Deffering

- Actors – deletion handled as deletion of underlying Entities
- Entities – marking with DestroyFlag component
- Components – enrolling for destruction in a buffer of EntityIDs and pointers to destruction method applicable for given type of component
- Component deletion is handled first, then Entities, as they may relate to the same components

Work in progress

## Component Destruction

```
struct ErasureEnroll
{
    void (Registry::* EraseFuncPtr)(EntityID);
    EntityID m_EntityID;
};

std::vector<ErasureEnroll> m_Erasures;
```

```
template <typename tnComponent>
void ScheduleErasure(EntityID entityID)
{
    m_Erasures.push_back(
        ErasureEnroll{ &Registry::erase<tnComponent>, entityID });
}
```

```
void DestroyComponents(Registry& registry)
{
    FE_PROFILER_FUNC();

    for (auto& enroll : m_Erasures)
    {
        auto& funcPtr = enroll.EraseFuncPtr;
        auto& entityID = enroll.m_EntityID;
        (registry.*funcPtr)(entityID);
    }

    m_Erasures.clear();
}
```

# Entity Component framework of Fools Engine

## Transform

### State Propagation

- Two components: Global and Local
- $\text{this} \rightarrow \text{Global} = \text{parent} \rightarrow \text{Global} + \text{this} \rightarrow \text{Local}$
- Local recalculated upon change, Global recalculated upon access if needed (has changed)

### Availability

- Not accessible directly
- Special handle provides safe interface to operate on either Local or Global part
- Overloaded operators allowing to treat it as if it was Global aspect of the component itself

### DirtyFlag component

- Marks Global part as „outdated“
- Emplaced in all descendant Entities upon Global modification (Local modification implies Global modification)
- Global is recalculated upon access if necessary (if marked as „dirty“) using hierarchy chain starting from closest „clean“ ancestor
- Local is always accurate, Global appears as always accurate

Work in progress

```
struct TransformComponent : DataComponent
{
public:
    Transform GetTransform() { return Transform; }
    const Transform& Get() { return Transform; }
private:
    friend class TransformHandle;
    friend class EntitiesHierarchy;

    Transform Transform;
};

struct CTransformLocal { ... };

struct CTransformGlobal { ... };
```

```
class TransformHandle
{
public:
    TransformHandle(EntityID ID, Registry* registry);

    const Transform& GetLocal() const { return m_Local.Transform; }
    const Transform& GetGlobal() { ... }

    Transform Local() const { return m_Local.Transform; }
    Transform Global() { return GetGlobal(); }

    operator const Transform& () { return GetGlobal(); }
    operator Transform () { return GetGlobal(); }

    void operator= (const Transform& other) { SetGlobal(other); }

    void SetLocal(const Transform& other);
    void SetGlobal(const Transform& other);

private:
    CTransformLocal& m_Local;
    CTransformGlobal& m_Global;
    CEntityNode& m_Node;
    Registry* m_Registry;
    EntityID m_EntityID;
    bool m_ParentRoot = false;

    bool IsDirty(EntityID entityID) const { return m_Registry->all_of<CDirtyFlag<CTransformGlobal>>(entityID); }
    bool IsDirty() const { return IsDirty(m_EntityID); }

    void SetDirty(EntityID entityID) { m_Registry->emplace<CDirtyFlag<CTransformGlobal>>(entityID); }
    void SetClean(EntityID entityID) { m_Registry->erase <CDirtyFlag<CTransformGlobal>>(entityID); }

    void MarkDescendantsDirty();
    void Inherit(EntityID entityID);
};
```

# Entity Component framework of Fools Engine

## Tags

Tags can be inspected and easily edited in level editor

▼ Tags				
Pa...	Lo...	Gl...	ID	Tag Name
			0	Error
✓		✓	1	Player
✓		✓	2	Very very very very lor
	✓	✓	3	3

```
struct Tags
{
    enum TagList : uint64_t
    {
        Error = WIDE_BIT_FLAG(0),
        Player = WIDE_BIT_FLAG(1)
    };

    Tags() = default;
    Tags(uint64_t tags)
        : TagBitFlags(tags) {};

    operator uint64_t& () { return TagBitFlags; }
    operator const uint64_t& () const { return TagBitFlags; }

    Tags operator+ (Tags other) const { return Tags(this->TagBitFlags | other.TagBitFlags); }
    Tags operator- (Tags other) const { return Tags(this->TagBitFlags & ~other.TagBitFlags); }

    Tags operator+ (TagList other) const { return operator+((Tags)other); }
    Tags operator- (TagList other) const { return operator-((Tags)other); }

    void operator+=(Tags other) { this->TagBitFlags |= other.TagBitFlags; }
    void operator-=(Tags other) { this->TagBitFlags &= ~other.TagBitFlags; }

    void operator+=(TagList other) { operator+((Tags)other); }
    void operator-=(TagList other) { operator-((Tags)other); }

    bool operator==(Tags other) const { return TagBitFlags == other.TagBitFlags; }

private:
    friend class EntityInspector;

    uint64_t TagBitFlags = 0;
};
```

Work in progress

```
class TagsHandle
{
public:
    TagsHandle(EntityID ID, Registry* registry);

    const Tags& GetLocal() const { return m_CTags.Local; }
    const Tags& GetGlobal() { ... }

    Tags Local() const { return m_CTags.Local; }
    Tags Global() { return GetGlobal(); }

    operator const Tags& () { return GetGlobal(); }
    operator Tags () { return GetGlobal(); }

    void SetLocal(const Tags& other);

    bool Contains(Tags tags) const { return m_CTags.Global & tags; }
    bool Contains(Tags::TagList tag) const { return Contains((Tags)tag); }

    void Add(Tags tags) { SetLocal(m_CTags.Local + tags); }
    void Remove(Tags tags) { SetLocal(m_CTags.Local - tags); }

    void Add(Tags::TagList tag) { Add((Tags)tag); }
    void Remove(Tags::TagList tag) { Remove((Tags)tag); }
```

## Tags

- Just uint64\_t with bit flags - each bit flag represents an existence of a given tag
- Meant to be used to easily recognize given Entity's purpose and conceptual identity
- Should not be used to send info down the hierarchy, unless widely used
- Entity class provides utility methods, like FindTagOrigin()

## Propagation

- Two parts: Global, Local (in one component)
- this->Global = parent->Global + this->Local
- Propagates down the hierarchy the same way as Transform and has similar handle, but triggers full recalculation upon each change



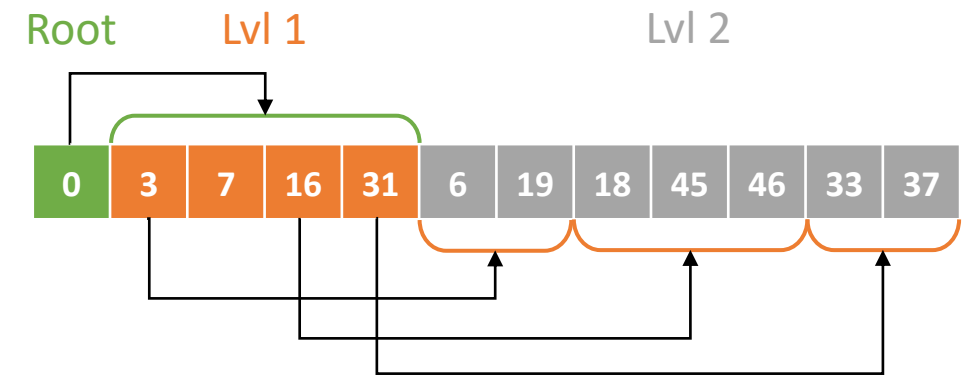
# Entity Component framework of Fools Engine

Work in progress

## Hierarchy Storage

Order of Components in storage reflects logical structure of the hierarchy

- Storages of hierarchy related components are synchronized: EntityNode, LocalTransform, GlobalTransform, Tags
- Siblings are grouped next to each other
- Siblings are sorted based on their EntityID (logically in linked list of Components and in storage)
- Groups of siblings are grouped based on their hierarchy level
- Groups of siblings on the same hierarchy level are sorted based on their parent EntityID



## Maintenance

- This order is not fully guaranteed, as maintaining it at all times would be too expensive
- Instead it is recreated once per frame after the simulation and before rendering
- It's purpose is to make hierarchy traversal more cash friendly

## Sorting algorithm

- Sorting is first performed on an array of just EntityIDs and then resulting permutation applied to component storages in linear time (sorting does not swap components unnecessarily)
- Sorting is performed in 2 stages:
  - 1: QuickSort groups each hierarchy level recursively
  - 2: Each level is then sorted concurrently with `std::sort`
- Continuous, amortized approach is still considered

# Entity Component framework of Fools Engine

Work in progress

## Storage Customization

### ECS implementation

- Current implementation of ECS storage is an external open-source library EnTT
- It provides additional customization features like Groups (synchronizing orders of components) and custom component Storages (accessed through storage ID, not type)
- Direct usage of those customizations can result in breaking of fundamental design corner stones of the framework
- Proper integration of those is not yet designed

### Custom Storage

- Work in progress
- Work in progress

### Limitations

- Custom storage cannot be involved in a Group
- Work in progress
- Work in progress

## Grouping components

- Work in progress
- Work in progress
- EntityHierarchy related Components (e.g. Transforms, Tags, EntityNode) are already grouped
- Actor related Components (ActorData, ActorNode) are already grouped

## Limitations

- Protected Components cannot be included in a Group (e.g. ActorData, Tags)
- Storages of component types managed by a Group lose pointer stability upon component creation
  - They cannot be created by Actors (concurrently running Actor's pointers and references to it's own components will get invalidated) – creation should happen within a global system
- They cannot be accessed using CompPtr (automated caching of component pointer within one frame) – go through registry/group using raw EntityID each time