Entity Component framework of Fools Engine

Work in progress

What do we actually need?

The engine needs to “run” the game – simulate and visualize it (render). This means that the game needs to be expressed in a way that is understandable by the engine. In other words, we need a model that the game will follow and the engine will use to interpret the game.

Not only we want to “run” a game, but also we need to make it. So we also need a framework that is going to ensure that developers working on a game can understand each other’s work and the results are compatible and coherent. This framework will also need to be runtime dynamic (both data and logic) so that iteration times would be small. We don’t want to recompile the whole game and the whole engine whenever a game designer wants to make a new type of game object or give it some new functionality.

The runtime model and development framework have slightly different requirements, so we could design and implement them separately to exploit unique opportunities and make tradeoffs that are not acceptable for the other. That however not only doubles the work but also requires a translation (necessarily automated for bigger projects) from one model to the other and creates an additional requirement of compatibility between them (it can be done however, e.g. Unity’s DOTS and GameObject).

Entity Component framework of Fools Engine

Work in progress

Design challenges

Game's Data and Logic	Game development process	Usability	Run-Time	Integration
Complex	Complex	Multiple users in one project	Performant	Engine functionalities
Divers	Random	Both technical and non-technical users	Hardware scalable (up and down)	Systemic game features
Dynamic	Volatile	Users of all levels of familiarity and experience	Stable and bug free	Unique game features
Densely interdependent				

Entity Component framework of Fools Engine

Work in progress

Entity-Component architecture

Entity-Component-System	!=	Entity-Component system

There is observable departure from inheritance-oriented scene representation and execution models in commercial game engines, online tutorials, blog posts, and conference presentations towards models with a composition-oriented approach where we have a featureless “entity” fulfilling service-locator pattern for reusable components. Those new models lay foundation for game development frameworks that are implemented in form of generic, runtime dynamic, game agnostic systems, so the term “Entity-Component system” has gained popularity.

There is also “Entity-Component-System” - data-oriented design pattern and aspiring paradigm sometimes used in implementing those models and a trendy buzzword.

This creates a great deal of confusion, as “Entity-Component system” and “Entity-Component-System” can only be differentiated from context. Not to mention improper use of those terms by people not knowing and understanding the difference. Additionally, game development frameworks in different game engines use incompatible fundamental nomenclatures (e.g. entity can be called Game Object, Pawn, Actor, Node…).

Entity Component framework of Fools Engine

Overview of existing design patterns

Work in progress

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 with 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 - DOD			

Update

Sequential	Per entity	
	Per component	Individually
		Through entity
Concurent	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			?
Dynamic allocations			?
Pointer stability			

Components storage

Allocation strategy	Individually	malloc()
		Custom
	As container	Array
		Vactor
		Sparse set
Grouping	None (individual types)	
	Archetypes	
	Hybrid	
Allocation coverage	Exact	
	Sparse	Empty tail optimized
		Empty tail not optimized
Pagination	Yes	Virtual
		Manual
	No	

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)		
	Cashed	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	

Work in progress

Entity Component framework of Fools Engine

State of the rot – usability

Game engines typically include 2 interfaces for their scene representation:

- 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.).

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.) intended to be used by various types of designers and with full access to all data of the scene. Their lack of good programming principles and practices inevitably leads to abuse of 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 architecture!

Entity Component framework of Fools Engine

Work in progress

State of the rot – 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. Due to this coupling, but also interactions between game objects through both engine and scripting interfaces, games have extremely unintuitive and complicated execution flows. That makes reasoning about the game difficult (sometimes even literally impossible without actually running 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'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

Alternative - embracing complexity

Work in progress

Complex ≠ Difficult Simple ≠ Easy

Instead of denying the existence of games complexity, 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. Putting data and logic in order properly and comprehensively using a framework is an occasion to conceptualize and optimize game code while keeping it manageable, extendable, and reusable. That requires a model that includes a comprehensive set of features for the representation of not only data but also execution flow. The complexity, diversity, generality and abstractness of its subject are an argument for the diversity of tools within the model! As long as we keep our framework's model composed in an intuitive way of standardized, corresponding concepts we can make as many tools for expressing the game as we need.

Complex does not mean difficult, and simple does not mean easy.

Entity Component framework of Fools Engine

Work in progress

Goals

Performant

- Build-in extensive parallelism (including engine-game parallelism)
- 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 spatial and 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 API
- Unified (engine build-in features and game code)

Entity Component framework of Fools Engine

Work in progress

Corner Stones

ECS Storage

- Entity-Component-System compliant components storage architecture
- Customizable optimizations
- Cache-friendly 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 (cash-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

Work in progress

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
  - Sparse sets with manual pagination
  - 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 – a black box with no dependencies
- Inheritance enforced (basic types)
- Further inheritance allowed (enforced), 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

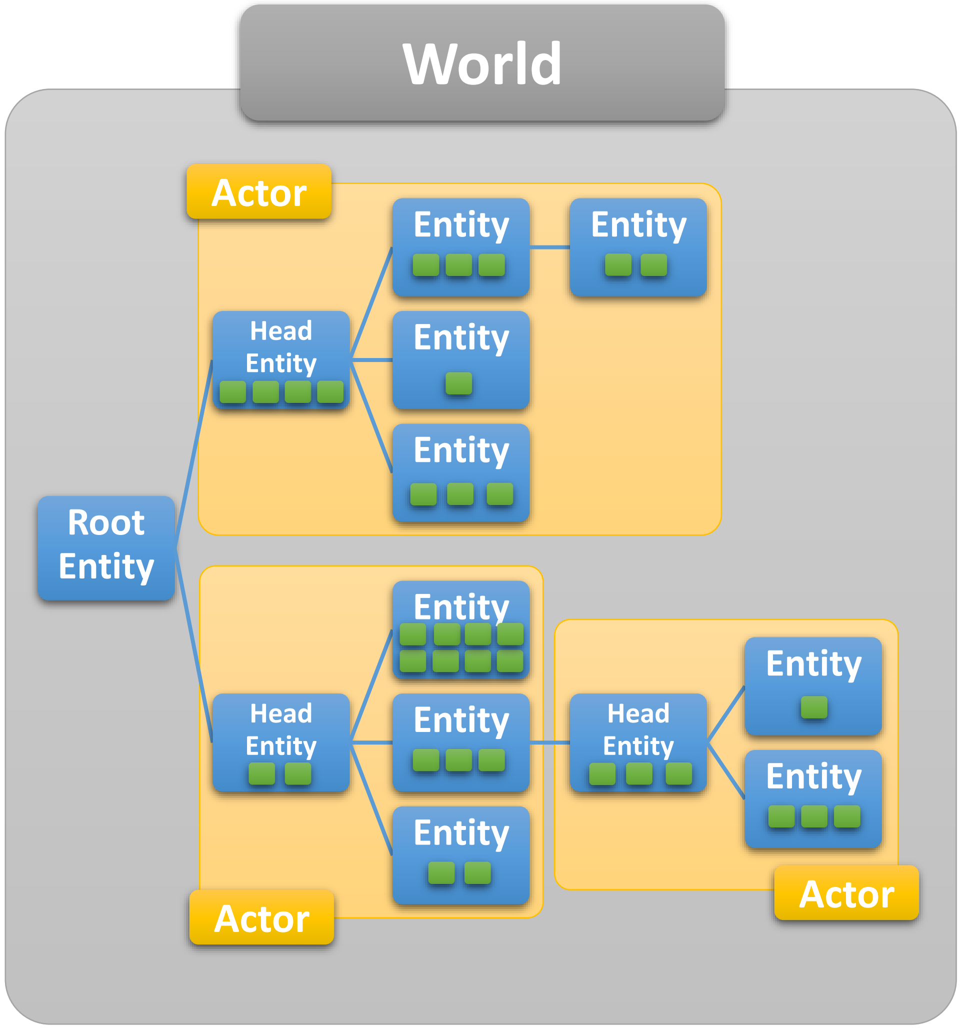
- Entity is a node stored as a component
- Definition: siblings as linked list + first child + parent
- Reference as entity ID
- Application of transform changes: instant
- Global transform: cashed, recalculated upon access if needed (dirty flag)
- Skeleton space: local
- No caching of global transforms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

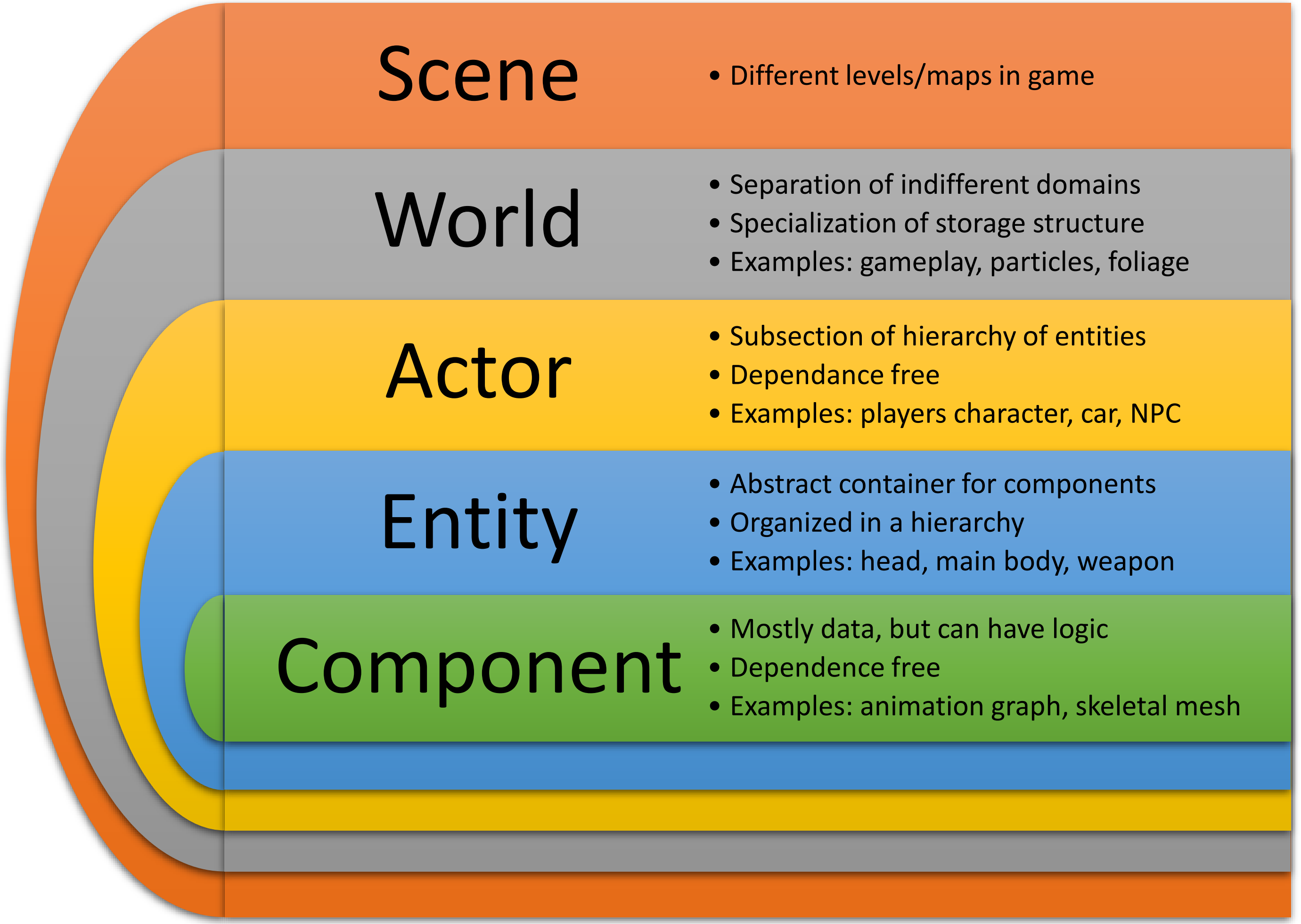
Update calls

- 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

Structure overview





Entity Component framework of Fools Engine

Execution and data routing

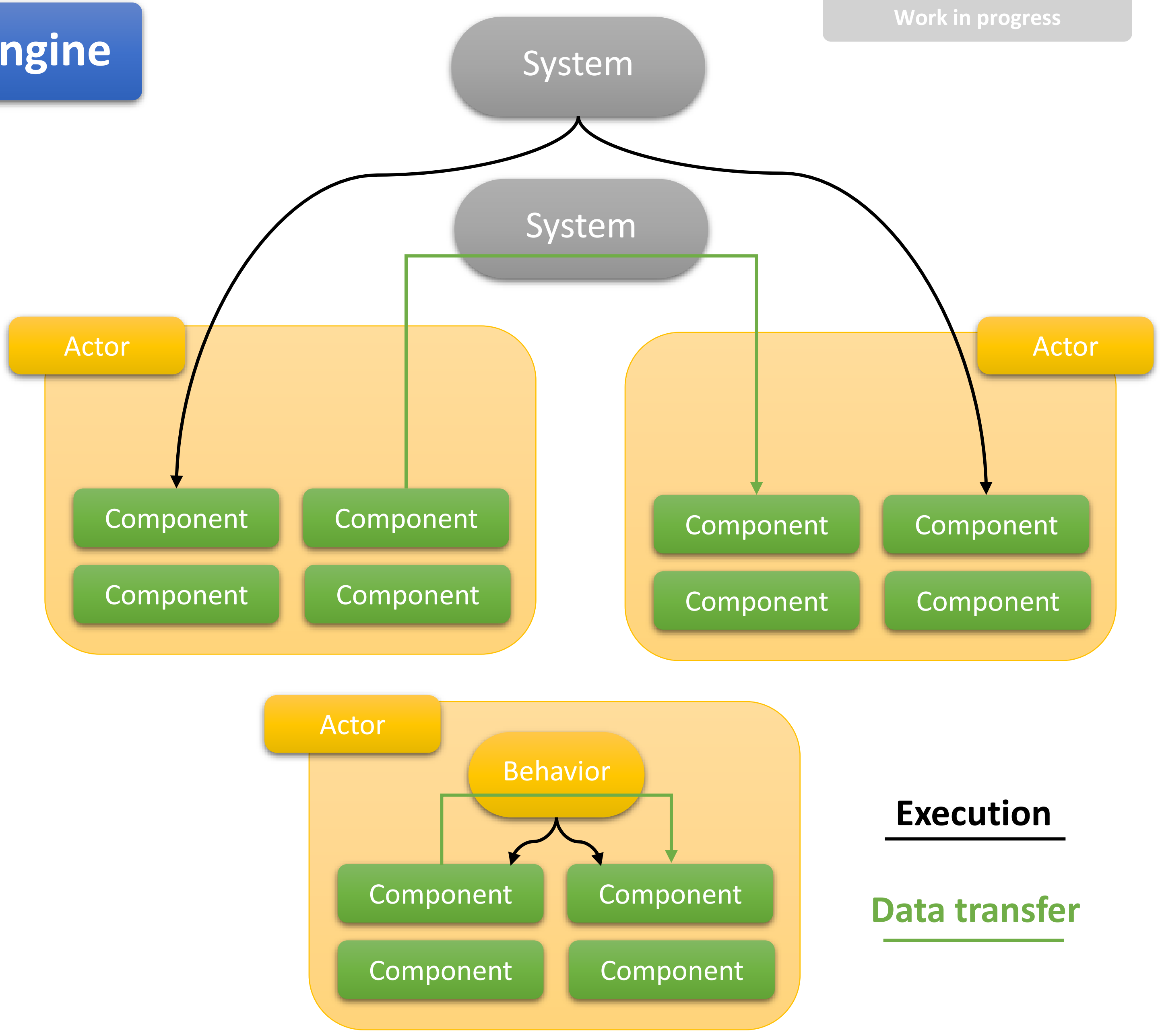
Work in progress

System

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

Behavior

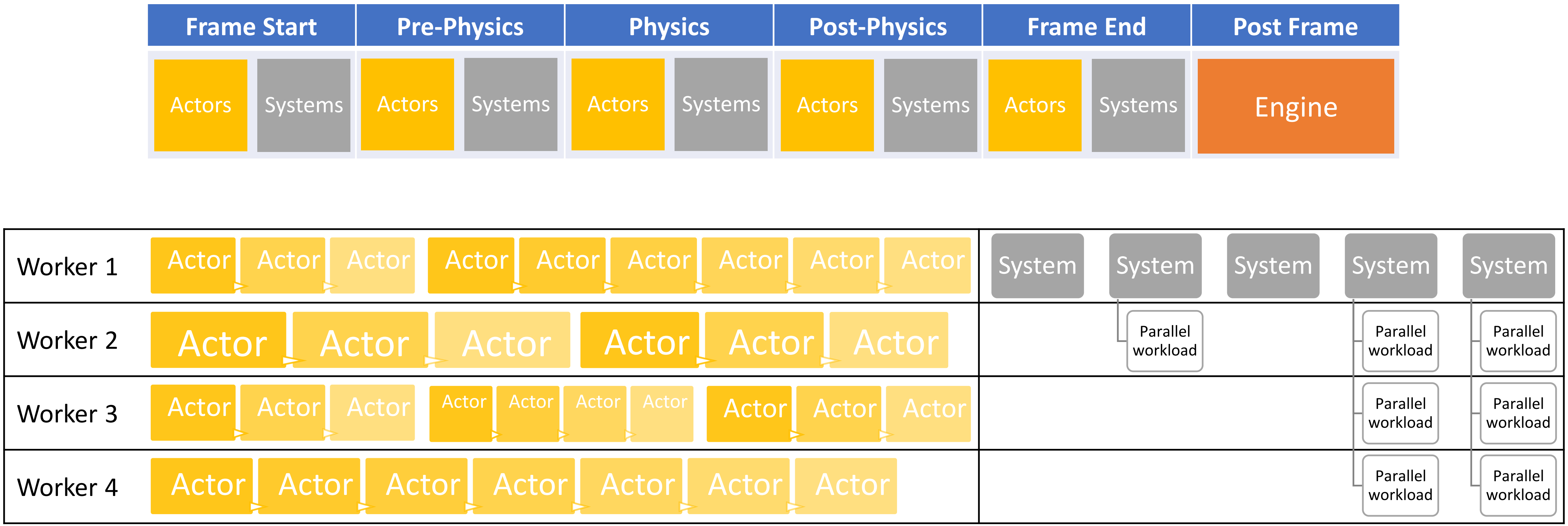
- Actor's "internal system" – owned by Actor
- 2 roles:
  - Execution flow control between Components
  - Component–component data transfer
- Uses direct component references (entity IDs)
- Actors update is an update of its behaviors
- Behaviors register for update from Actor
- Examples: animation, characterController



Work in progress

Entity Component framework of Fools Engine

Update Order



Work in progress

Entity Component framework of Fools Engine

Component

Basic building block

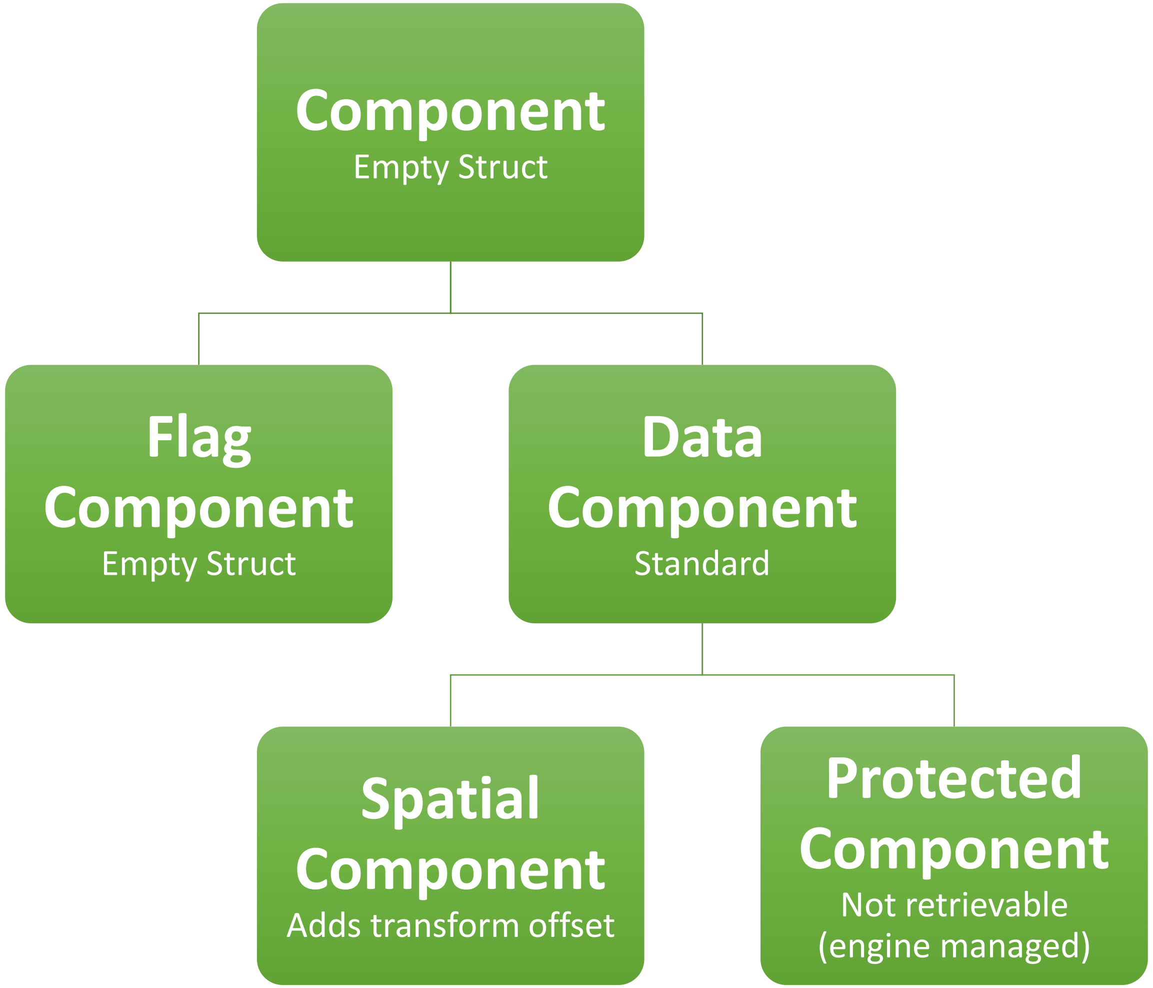
- Can have data
- Can have resource references
- Can have logic
- Cannot reference anything in the scene ("black box")

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 a world



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

- A 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

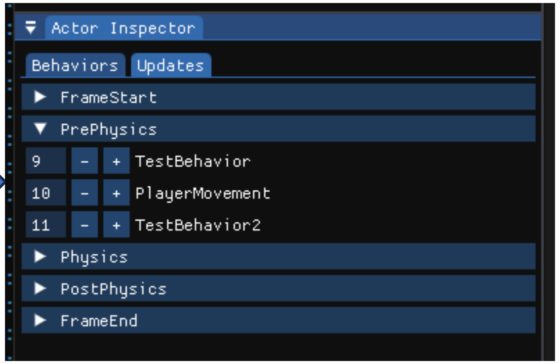
- All Entities in a Gameplay World are part of one spatial hierarchy
- Defined by dedicated component – HierarchyNode
  - Children defined as linked list
  - Children are sorted by their EntityID
  - Cashes information about its hierarchy depth and children count
- Fully managed by engine
- Used to organize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

Entity Component framework of Fools Engine

Work in progress

Actor

Numerical priorities of Behavior updates can be modified in editor's GUI on a per-instance basis



Abstract object

- Abstract object representing single "thing" from players perspective (e.g. player's 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

Update

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

Work in progress

Entity Component framework of Fools Engine

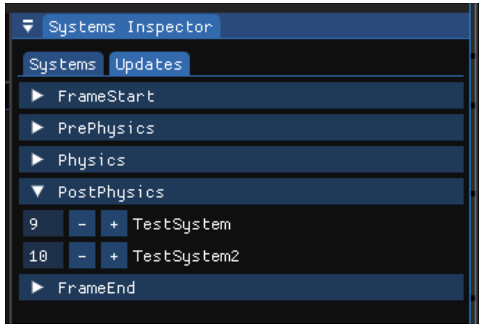
System

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 call 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

- Systems can parallelize internally their workload
- Systems are executed sequentially by a World
- Systems register 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



Systems numerical update priorities can be modified in the editor's GUI

Entity Component framework of Fools Engine

Work in progress

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

Deferring

- 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 the destruction method applicable for a given type of component
- Component deletion is handled first, then Entities, as they may relate to the same components

Component Destruction

```
struct ErasureEnroll
{
    void (Registry::* EraseFunkPtr)(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.EraseFunkPtr;
        auto& entityID = enroll.m_EntityID;
        (registry.*funcPtr)(entityID);
    }

    m_Erasures.clear();
```

Entity Component framework of Fools Engine

Work in progress

Transform

State Propagation

- Two components: Global and Local
- this->Global = parent->Global + this->Local
- Local recalculated upon change, Global recalculated upon access if needed (has changed)

Availability

- Not accessible directly
- Special handle provides a safe interface to operate on either Local or Global transform
- Overloaded operators allowing to treat it as if it was Global transform itself

DirtyFlag component

- Marks Global transform 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 a hierarchy chain starting from the closest „clean” ancestor
- Local is always accurate, Global appears as always accurate

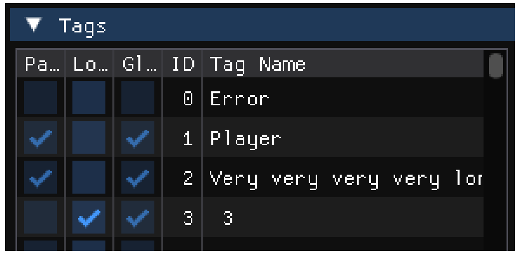
```
struct TransformComponent : DataComponent
{
public:
    Transform GetTransform() { return Transform; }
    const Transform& Get() { return Transform; }
private:
    friend class TransformHandle;
    friend class EntitiesHierarchy;

    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 end easily edited in level editor



Tags

- Just bitset with bit flags - each bit flag represents an existence of a tag
- Meant to be used to easily recognize given Entity's purpose and conceptual identity
- Using it to send info down the hierarchy should be avoided
- 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
- Can only be modified in systems, not in actors

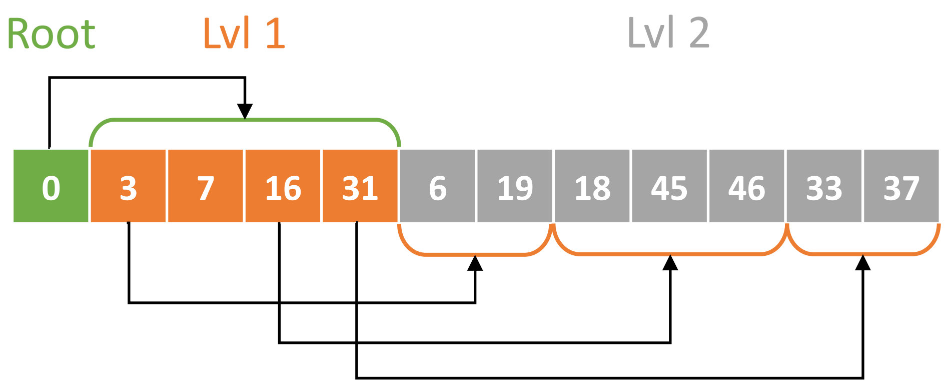
```cpp
class TagsHandle
{
public:
    TagsHandle(EntityID ID, Registry* registry);

    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); }
```

Entity Component framework of Fools Engine

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 a linked list of Node 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
- Its 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 in parallel with std::sort
- Async, amortized approach is still considered

Entity Component framework of Fools Engine

Work in progress

Storage Customization

ECS implementation

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

Custom Storage

- Work in progress

Limitations

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

Grouping components

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

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 cashing of component pointer within one frame) – go through registry/group using raw EntityID each time