Scene Representation Model of Fools Engine

What are we even talking about?

Work in progress

Entity-Component-System != **Entity-Component system**

Since a complete departure from using pure object oriented “Actor” model for scene representation in commercial game engines, online tutorials and conference presentations towards composition oriented (“Game Object” a.k.a. “Component-Object” a.k.a. “Entity-Component” model), data oriented (“Entity-Component-System” model) and hybrid approaches, implemented in form of a generic, game agnostic system, term “scene representation model” is going out of use, replaced by term “Entity-Component system”.

This creates a great deal of confusion, as “Entity-Component system” and “Entity-Component-System” can only be differentiated from context or in writing. Not to even mention improper use of those terms by people not knowing / understanding the difference.

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

Scene Representation Model of Fools Engine

Work in progress

State of the rot

Game engines typically include 2 interfaces for their model, one for engine build-in functionalities (rendering, physics, etc.) and systemic game features (crowds, traffic, etc.), and another one for unique, complex, high-level game features (player's character movement, AI, etc.). Engine side often creates implicit dependencies from the get go and requires specific predefined setups to work, sometimes even as an undocumented technological debt. Game 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 good programming principles and practices, who inevitably end up abusing it.

Single, from players perspective conceptually indivisible, element of the game (including its behavior) is implemented using multiple components and systems (and potentially game objects) 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 engine and scripting 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 and of development process the game is literally too complex for anyone to know how it works any more. 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.

Scene Representation Model of Fools Engine

Work in progress

Facing the truth

Lack of intuitive grasp of a model and it's design philosophy leads to misuse and overcomplication, even with logical understanding of it's implementation architecture!

Fighting complexity

- Games are complex. That complexity is unavoidable and has to be resolved somewhere.
- Settling down on a simple scene representation model that supports only a narrow set of design patterns puts all responsibility and difficulty of managing that complexity on game developers.
- Simple architecture always 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 spaghetti of dependencies.

Embracing complexity

- Organization and structurization of both data and logic should be the fundamental roles of a scene representation model:
  - Complexity and diversity of its subject is an argument **for** complexity of the model!
  - Genericness and abstractness of the model is an argument **for** complexity of the model!
- Putting data and logic in order properly and comprehensively is an occasion to help users create, conceptualize and optimize game code, while keeping it manageable.
- Complex != Difficult; Simple != Easy;

Scene Representation Model 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

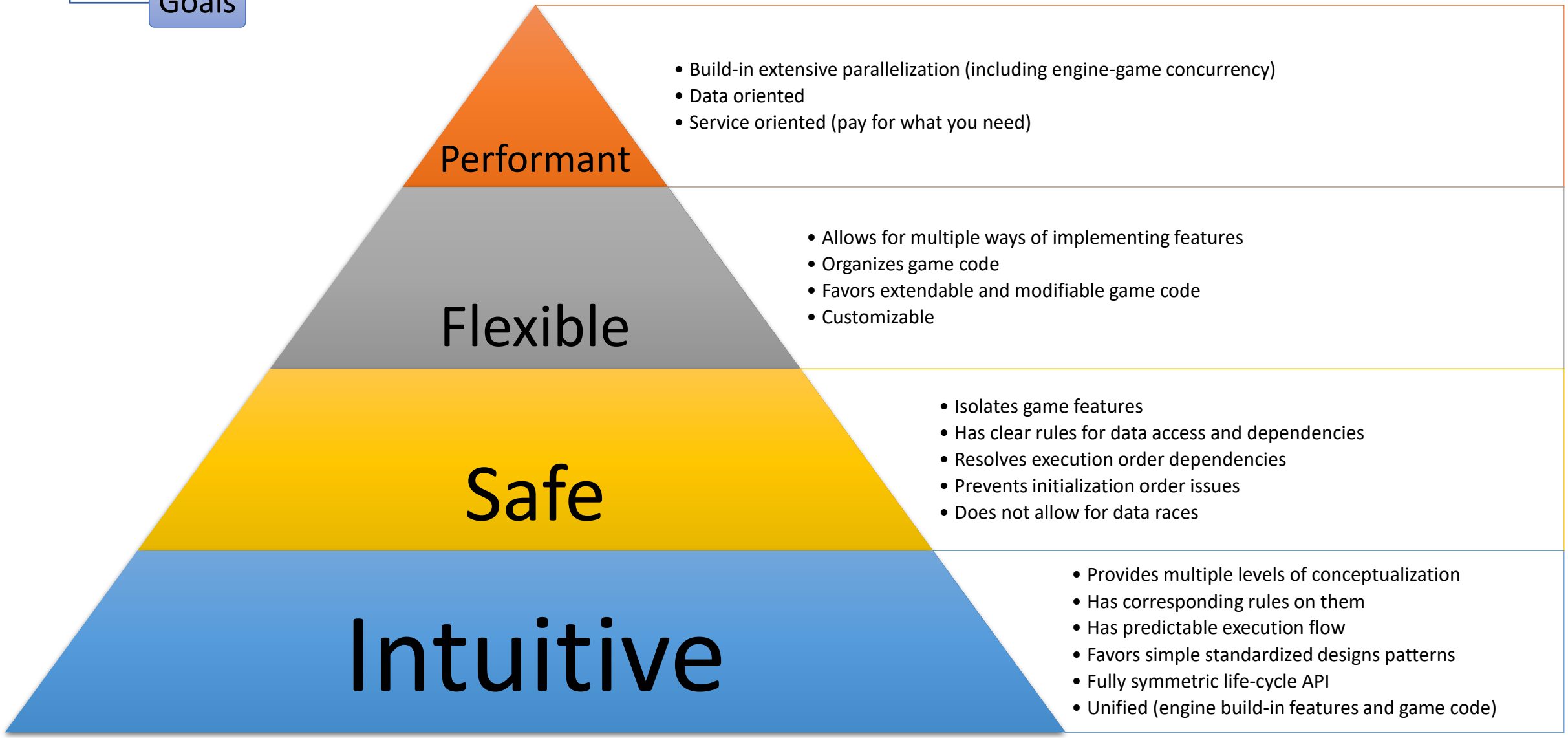
Integration

- Engine functionalities
- Systemic game features
- Unique game features

Scene Representation Model of Fools Engine

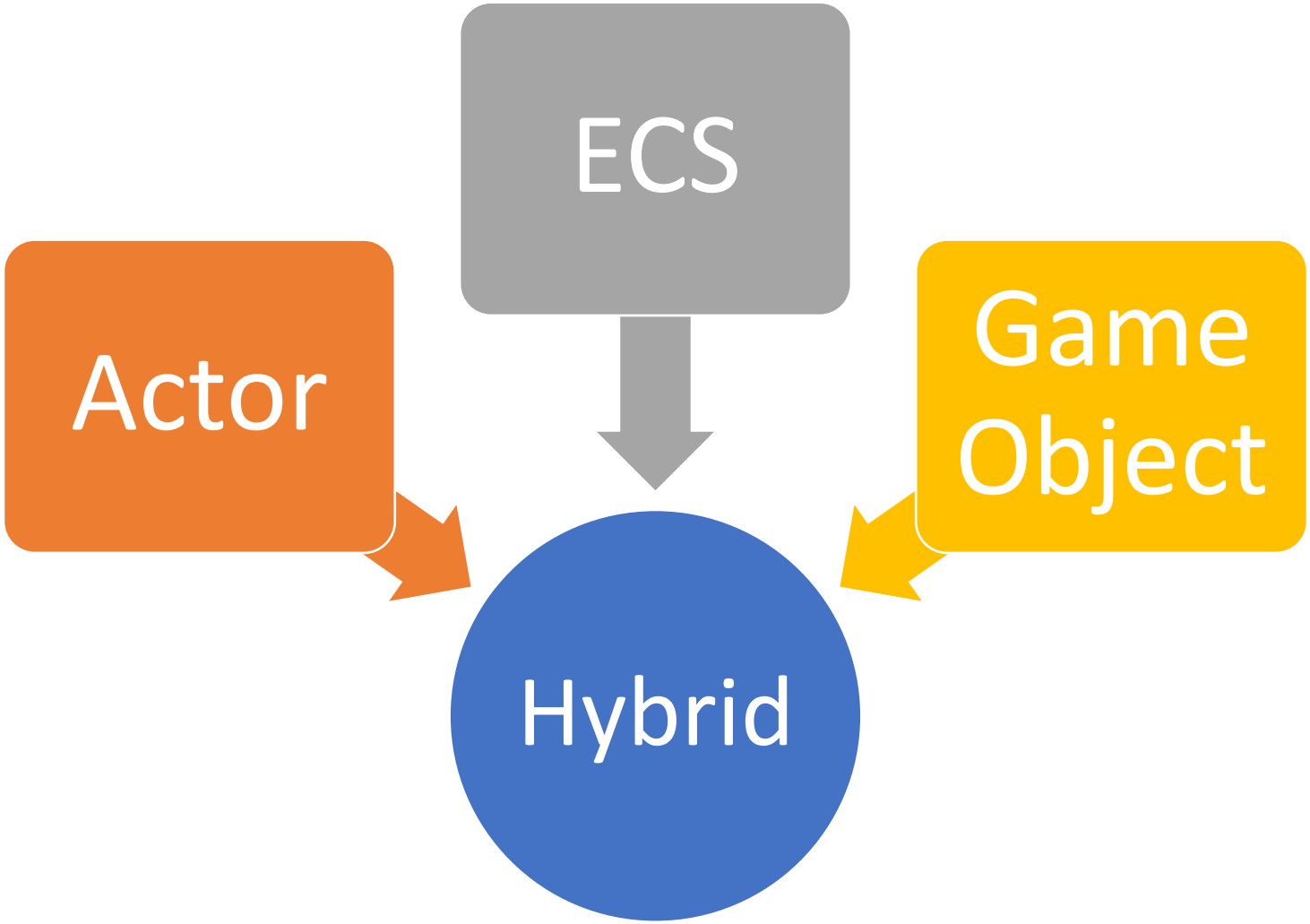
Work in progress

Goals



Scene Representation Model of Fools Engine

Solution => Hybrid



Work in progress

Scene Representation Model of Fools Engine

Work in progress

Corner Stones

ECS Storage

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

Build-in spatial hierarchy

- Fully managed by engine, hidden from user
- Automated propagation of transform changes
- Transform always accurate on read, but not recalculated on every change
- Dictates actors update order (parent => child)
- Children 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 actors update and initialization

2 levels of logic: System and Behavior

- Behavior deals with actor's internal state
  - Component update
  - Component–component data transfer
- System deals with world's state
  - Mass update of components
  - Actor–actor data transfer

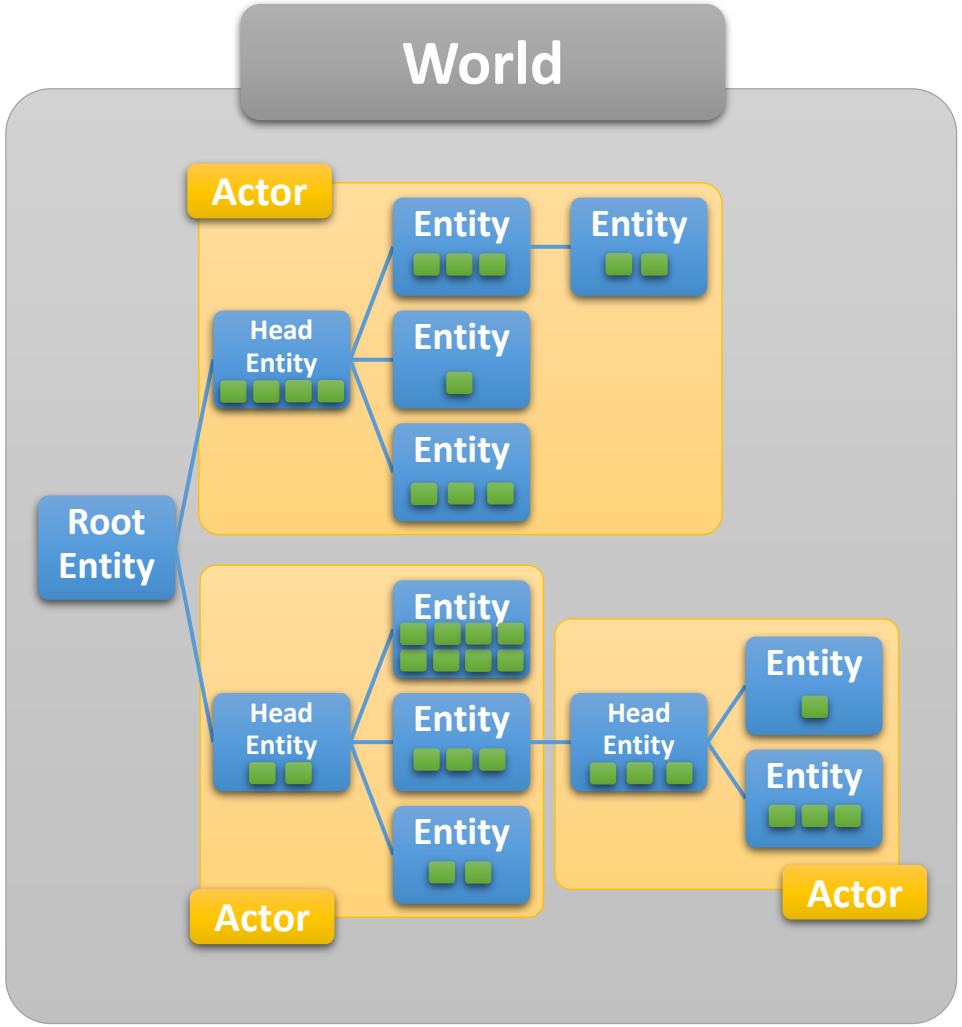
Explicit execution flow control

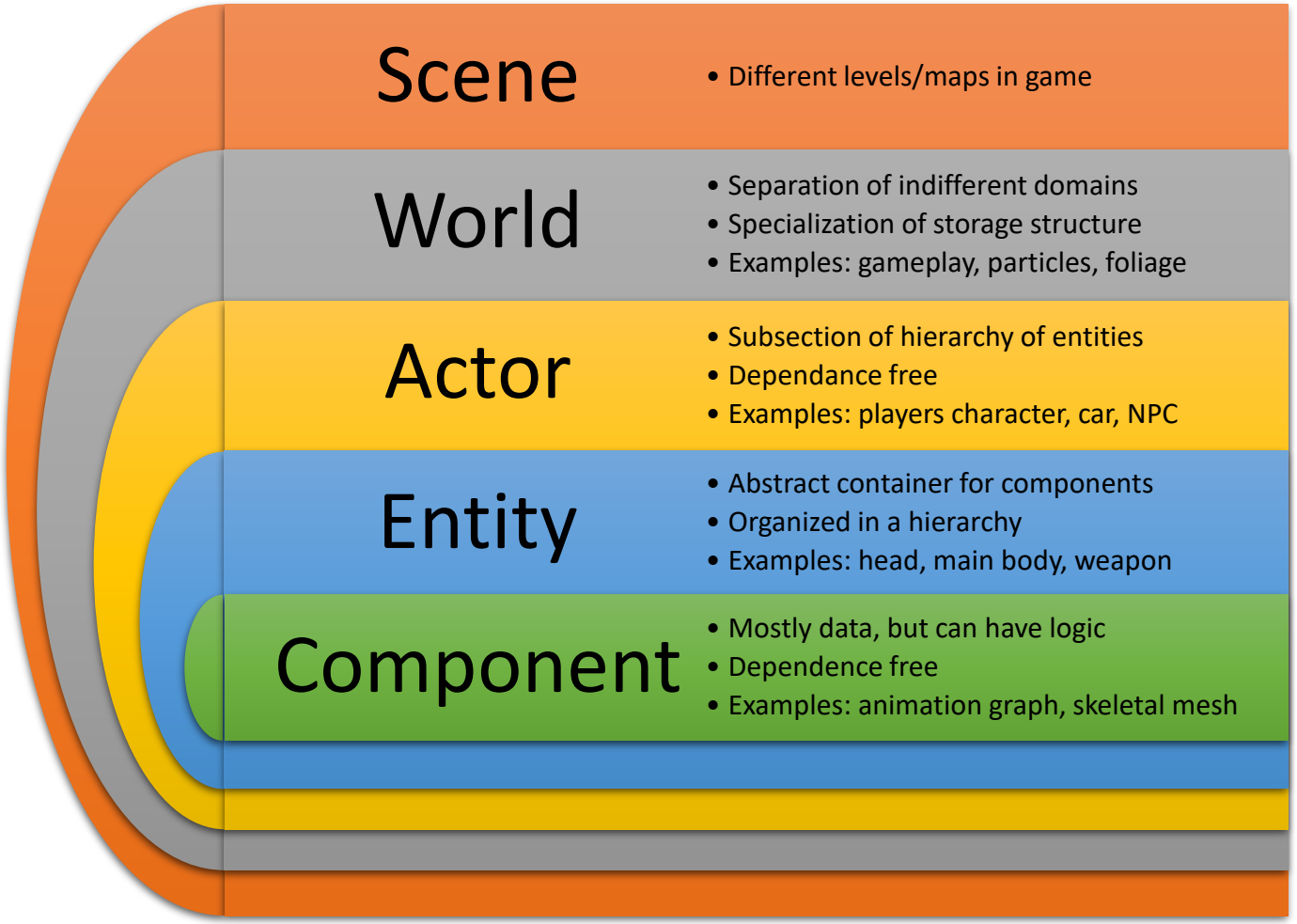
- No default update calls from the engine – manual enrollment always required
- Simple way of controlling execution order with 2 levels of control
  - 1 - Simulation stages (big sync-points)
  - 2 - Numerical priorities for Behaviors

Scene Representation Model of Fools Engine

Work in progress

Structure overview





Scene Representation Model of Fools Engine

Execution and data routing

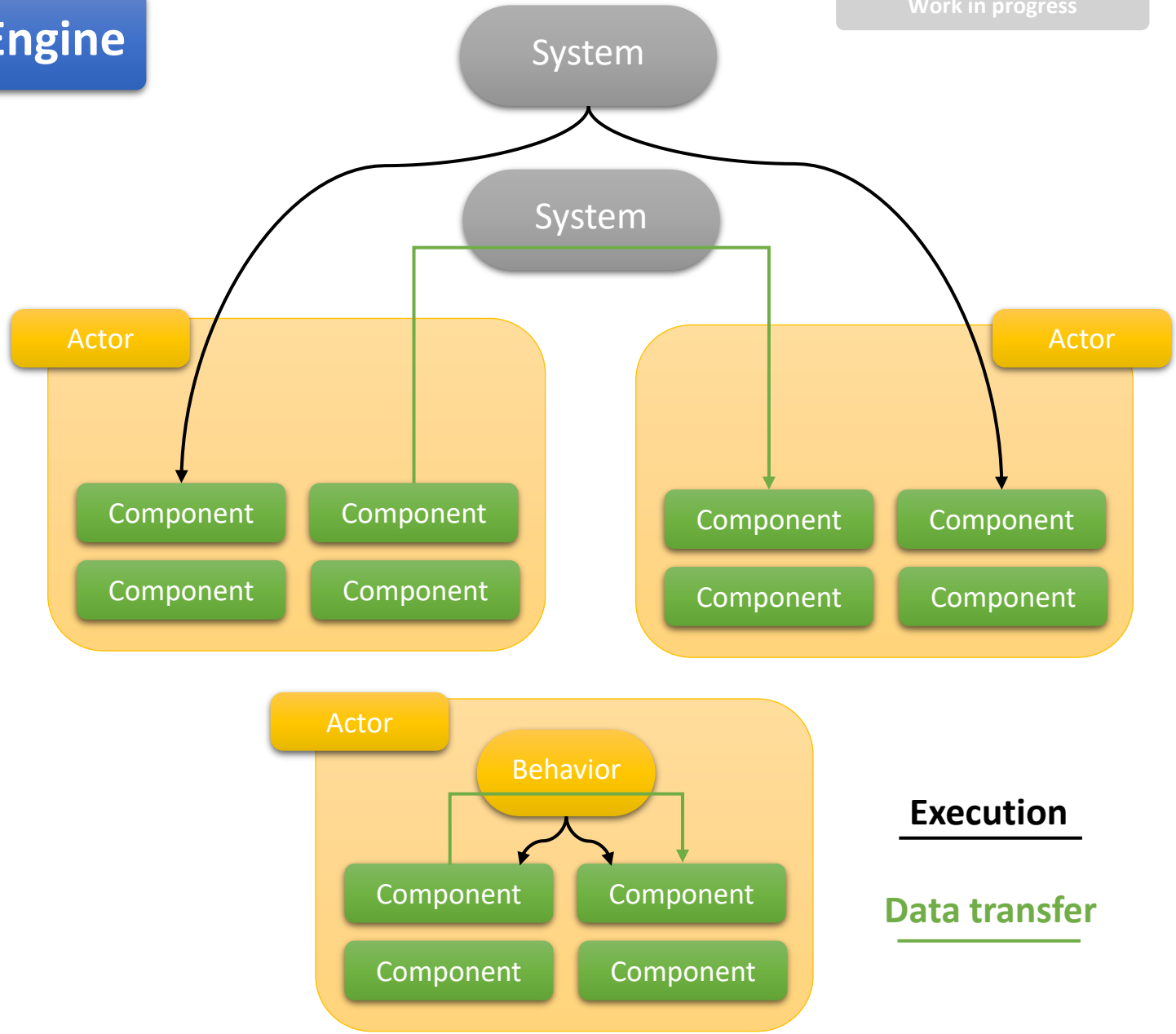
Work in progress

System

- Global – owned by World
- 2 roles:
  - Actor-actor data transfer by state injection (similarly to messages in actor model)
  - Components update (ECS style)
- Can be an opt-in or opt-out service
- Enrollment 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

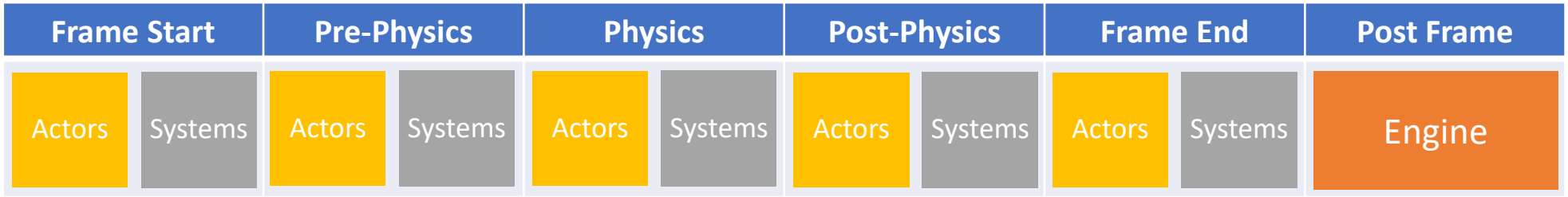
- Actor's "internal system" – owned by Actor
- 2 roles:
  - Execution flow control of dependent Components
  - Component–component data transfer
- Uses direct component references (entity IDs)
- Actors update is an update of its behaviors
- Behaviors register for update from Actor
- Examples: animation,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

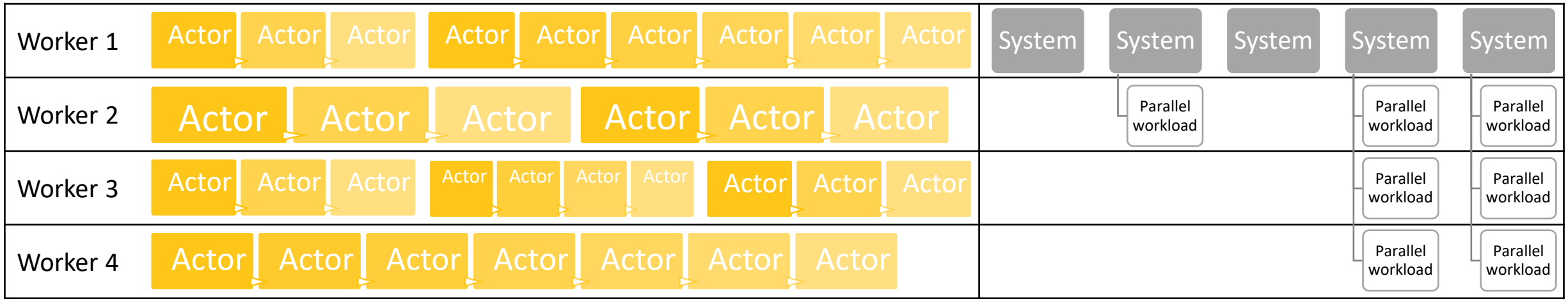


Scene Representation Model of Fools Engine

Work in progress

Update Order



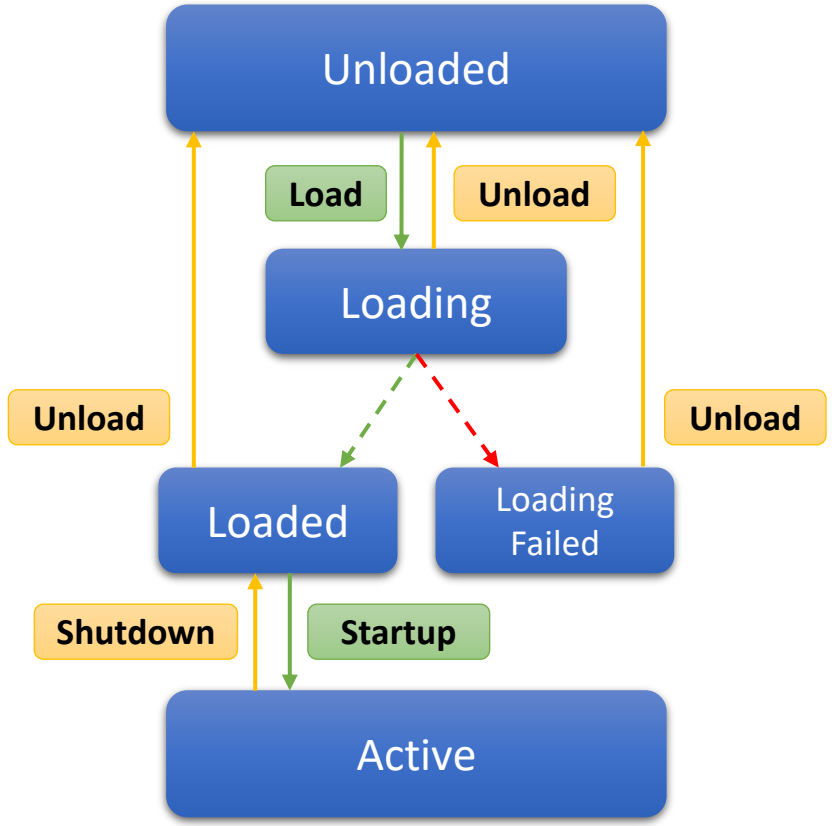


Scene Representation Model of Fools Engine

Work in progress

Component

Life Cycle



Basic building block

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

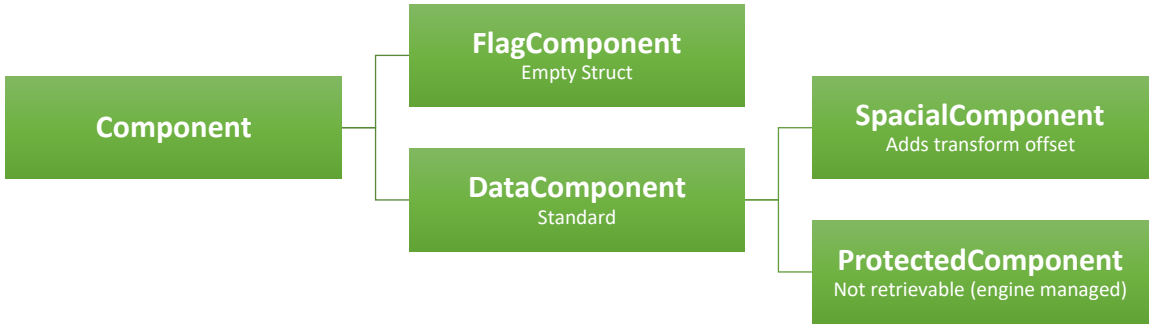
Can have logic

- 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



Scene Representation Model 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
  - 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

Scene Representation Model of Fools Engine

Work in progress

Actor

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

Scene Representation Model of Fools Engine

Work in progress

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

Scene Representation Model 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

Differing

- 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

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();
```

Scene Representation Model 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 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

```
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 { ... };
```

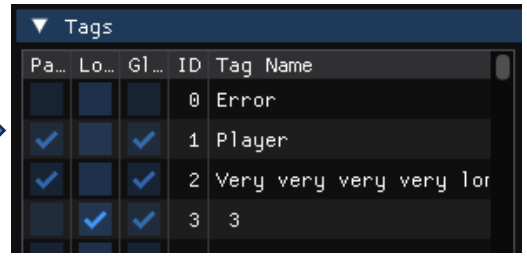
```
class TransformHandle
{
public:
    TransformHandle(EntityID ID, Registry* registry);

    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);
};
```

Scene Representation Model of Fools Engine

Work in progress

Tags

Tags can be inspected end easily edited in level editor




```cpp
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;
};
```

```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); }
```

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

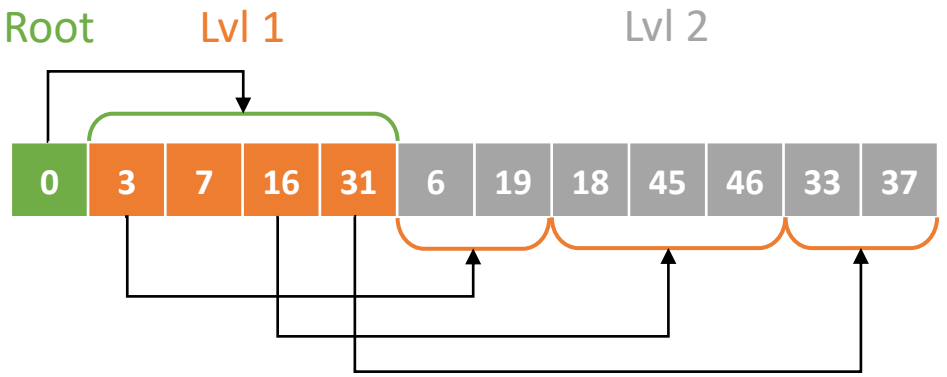
Scene Representation Model of Fools Engine

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

Scene Representation Model 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 model
- 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 cashing of component pointer within one frame) – go though registry/group using raw EntityID each time