Synapse Engine: A research oriented modern AAA game engine

Created By: Péter Tamás

Github: https://github.com/TamasPetii/SynapseEngine

2026

Intoduction I.

About me:

- 23 years old
- M.Sc. Student in Computer Engineering at Budapest University of Tehcnology and Economics (BME)
- B.Sc. In Computer Science from Eötvös Loránd University (ELTE), graduated in 2025.
- Planning to continue with a Ph.D. in the field of computer graphics.
- Started exporing computer graphics and game engine development during university.
- Currently focused on modern rendering techniques, GPU-driven rendering, high-performance engine architecture and data-oriented design.
- **Developer of the Synapse Engine**, a modern experimental AAA-style game engine.

Intoduction II.

Vision & Goals:

- Originally started as my M.Sc thesis project, but it quickly evolved into something much larger.
- To provide a modern AAA-style research oriented platform for universities and computer graphics researchers.
- This engine enables rapid prototyping, benchmarking, and evolution of new rendering techniques without the complexity of multi-million-line production engines.
- To offer studios and engine developers a clean, modular environment for experimenting with modern graphics technologies.
- To bridge the gap between academic research and production-quality engine architecture.
- To create an educational resource, demonstrating modern engine design, data-oriented programming, GPU-driven rendering and software architecture.
- To help students and ethusiasts understand how modern game engine can be built from scratch.
- To establish a long-term research platform for future Ph.D. work and the development of novel rendering techniques.

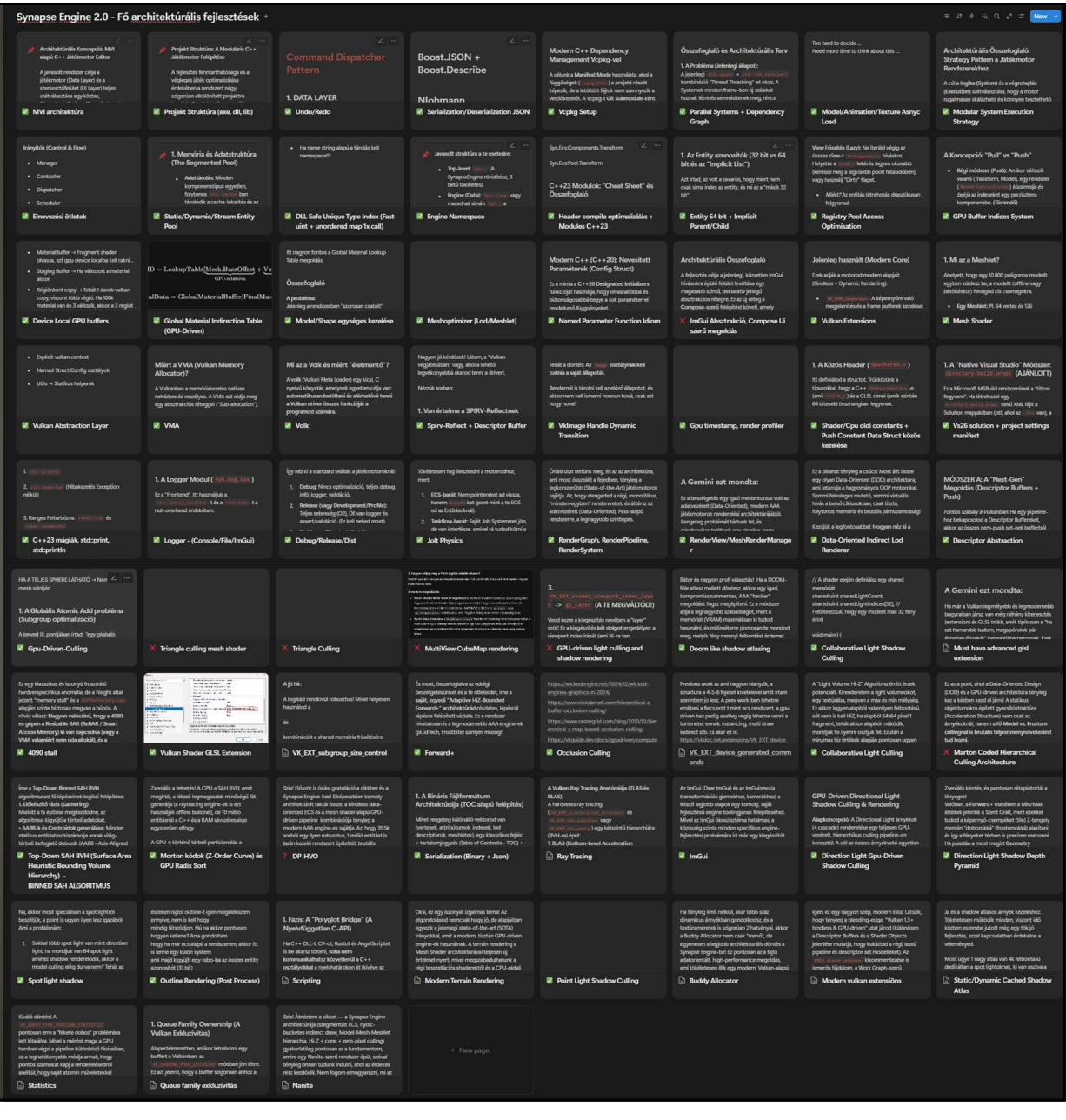
Mission:

- **Build an open, modern, and educational game engine that serves as both a research platform and a reference implementation for contemporary AAA engine architecture.**

Intoduction III.

The Journey of Synapse Engine:

- Discovered OpenGL and real-time graphics during my B.Sc. Studies.
- Became deeply interested in graphics programming and spent countless hours studying LearnOpenGL, also created 2-3 engine implementation.
- Developed my first data-oriented OpenGL engine as my B.Sc. Thesis, which is publicly available on GitHub and received a special award.
- Started exporing Vulkan during my M.Sc. Studies and build the first version of Synapse Engine while learning modern techniques.
  (The origin implementation still available on v1-engine-old branch.)
- In early 2026, I decided to rewrite the engine completely from scratch, applying everything I had learned about engine architecture, data-oriented design, and modern rendering.
- Before writing the new version, I spent nearly two months designing and refining the architecture to ensure a solid long-term foundation.
- Approximately 6-7 months later (10-12 hrs daily), the current version of Synapse Engine was completed.
- Development was significatly accelerated using AI (Gemini Pro Student), **but only after establishing a deep understandidng of the underlying concepts and a clear architectural vision**.



Intoduction IV.

Acknowledgements:

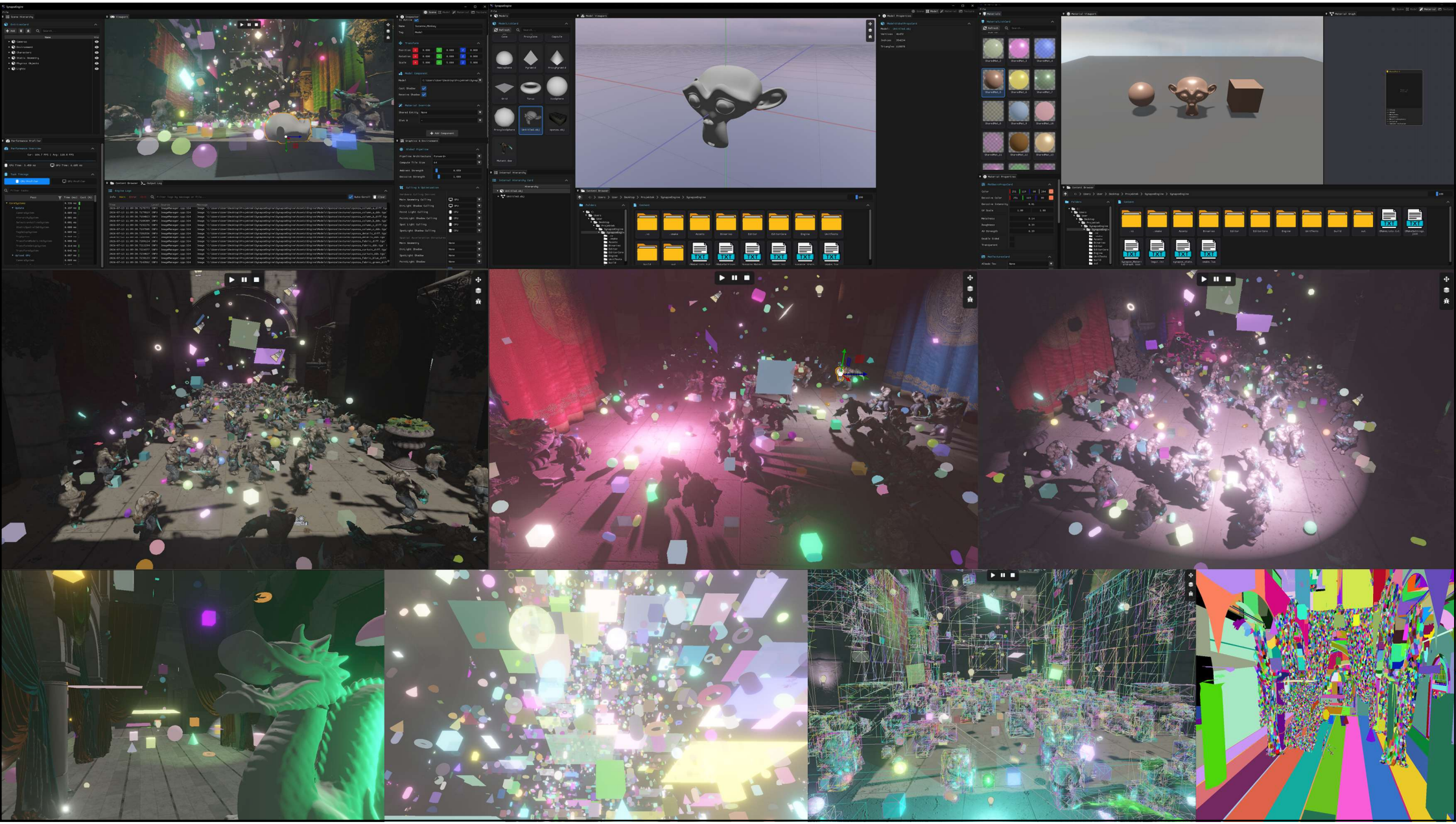
- **LearnOpenGL** (https://learnopengl.com/): An outstanding introduction to OpenGL and real-time graphics.
- **Vulkan Tutorial** (https://vulkan-tutorial.com/): A fantastic resource for learning Vulkan from the ground up.
- **Vulkan Guide** (https://vkguide.dev/): An invaluable reference for modern Vulkan development and best practices.
- **Alan Wake 2 – Gpu Driven Rendering** (https://youtu.be/EtX7WnFhxtQ): The single most influential resource behind the Synapse Engine. It fundamentaly shaped my understanding of modern GPU-driven rendering and large scale engine architecture. Without this presentation, the engine would likely never have reached its current form.

Academic Support:

- Special thanks to my supervisor, **András Máté Fridvalszky**, for his guidance and support throughout this project.

Related Projects:

- **Dendrite Engine** (B.Sc. Thesis – OpenGL engine): https://github.com/TamasPetii/DendriteEngine
- **Synapse Engine** (Original implementation) : https://github.com/TamasPetii/SynapseEngine/tree/v1-engine-old
- **Synapse Engine** (Current version) : https://github.com/TamasPetii/SynapseEngine



Synapse Engine:
Entity-Component-System

Created By: Péter Tamás

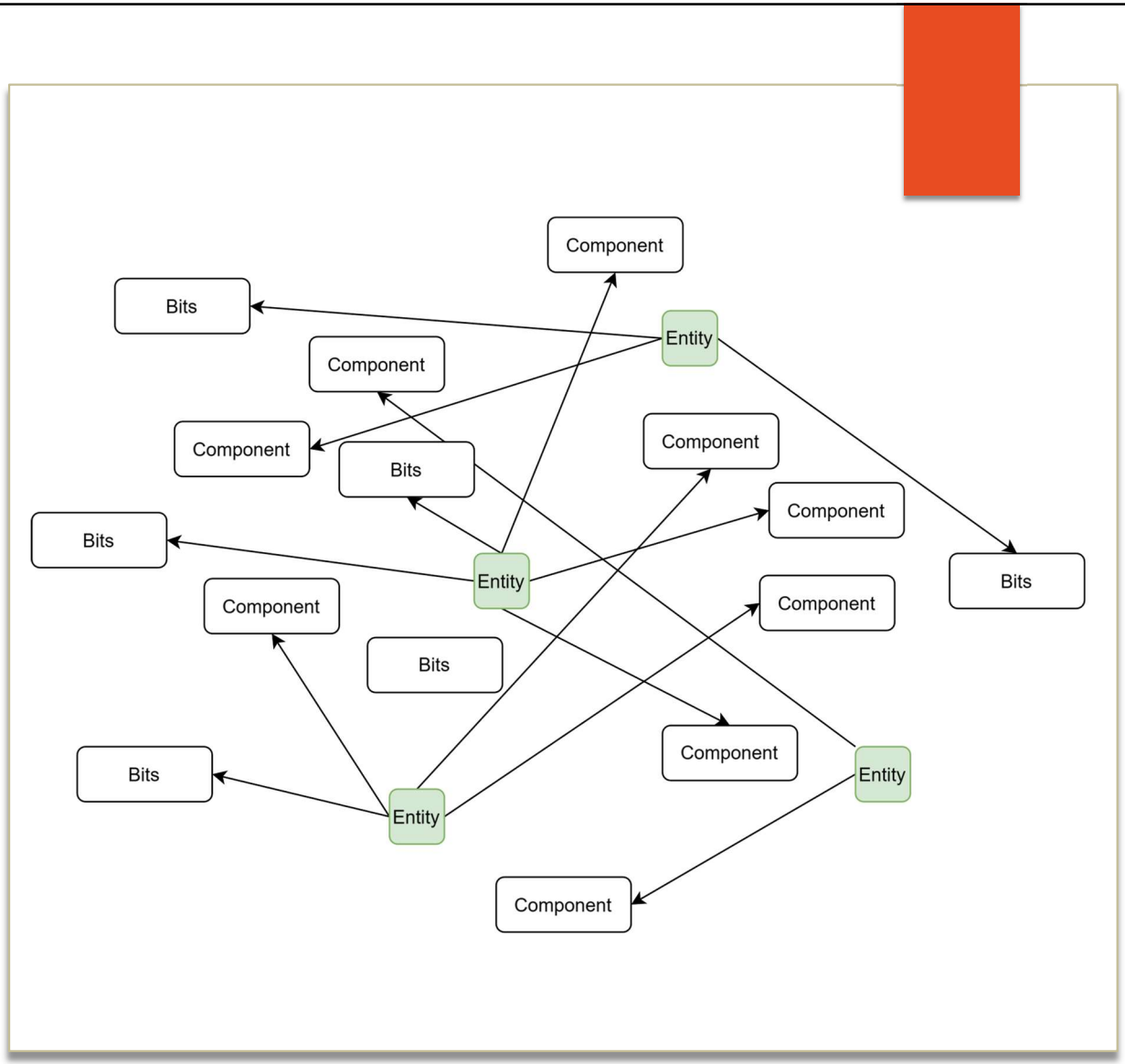
Github: https://github.com/TamasPetii/SynapseEngine

2026

Object-Oriented Programming Memory Model

Problems I:

- Scattered Memory Layout
- Poor cache locality and ineffective prefetching
- Virtual function calls and deep inheritance introduce overhead
- Difficult to parallelize effective across multiple CPU cores



Object-Oriented Entity-Component-System

Problems II:

- Matches how developers think, not how modern CPUs executes code
- Significantly slower data access
- Complex CPU/GPU synchronization
- Object-by-Object update model is difficulty to optimize
- Poor scalability with increasing entity counts

```
class Component {
public:
    virtual ~Component() = default;
};

class TransformComponent : public Component {
public:
    glm::vec3 position{0.0f, 0.0f, 0.0f};
    glm::vec3 rotation{0.0f, 0.0f, 0.0f};
    glm::vec3 scale{1.0f, 1.0f, 1.0f};
    glm::mat4 transform{1.0f};
};

class Entity {
public:
    template <typename T, typename... Args>
    T* AddComponent(Args&&... args) {
        //...
    }

    template <typename T>
    T* GetComponent() {
       //...
    }

    template <typename T>
    bool HasComponent() {
        //...
    }
private:
    std::unordered_map<std::type_index, std::unique_ptr<Component>> components;
};
```

Data-Oriented Entity-Component-System I.

Data-Oriented Model

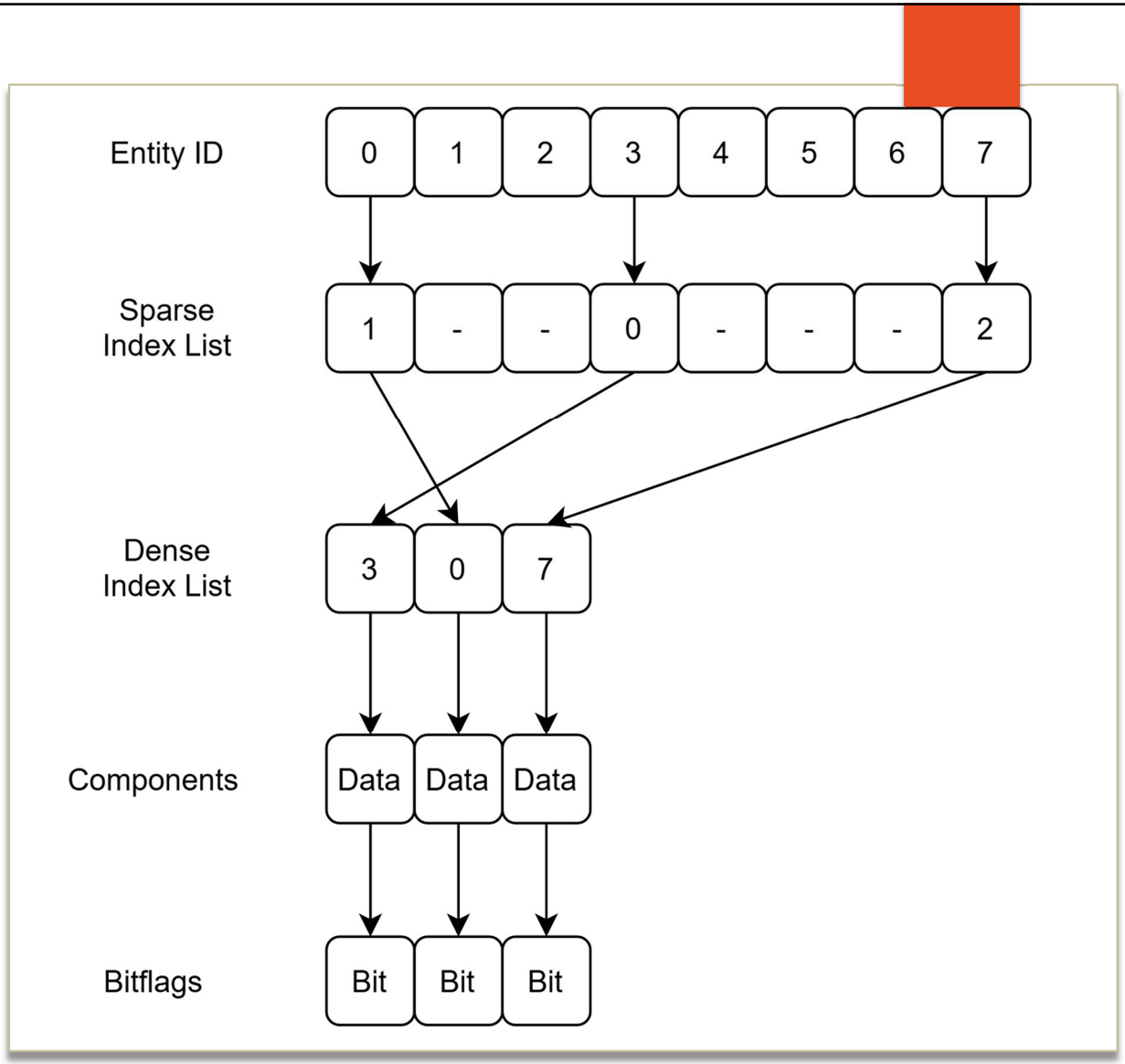
- Entities: Unsigned integer identifiers (UINT32_MAX = invalid entity)
- Components: Plain data structures, inheritance is allowed, but **no virtual functions**!!
- Systems: Operate on specific component types, updating them and performing logics
- Memory Model: Components of the same type are stored contiguously in memory!!

```cpp
using EntityID = uint32_t;

struct TransformComponent {
    glm::vec3 translation{0.0f, 0.0f, 0.0f};
    glm::vec3 rotation{0.0f, 0.0f, 0.0f};
    glm::vec3 scale{1.0f, 1.0f, 1.0f};
    glm::mat4 transform{1.0f};
};

std::for_each(std::execution::seq, denseEntities.begin(), denseEntities.end(),
    [&](EntityID entity) {
        size_t denseIndex = sparse[entity];
        TransformComponent& transformComponent = denseComponents[denseIndex];

        glm::mat4 localMat = glm::translate(glm::mat4(1.0f), transformComponent.translation);
        localMat = glm::rotate(localMat, glm::radians(transformComponent.rotation.z), glm::vec3(0, 0, 1));
        localMat = glm::rotate(localMat, glm::radians(transformComponent.rotation.y), glm::vec3(0, 1, 0));
        localMat = glm::rotate(localMat, glm::radians(transformComponent.rotation.x), glm::vec3(1, 0, 0));
        localMat = glm::scale(localMat, transformComponent.scale);

        transformComponent.transform = localMat;
    });
```

Data-Oriented Entity-Component-System II.

Sparse-Set architecture

- Sparse Index Array with gaps, and a tightly packed Dense index array
- Bidirectional mapping between Sparse and Dense array
- Entity Ids index directly into the Sparse array
- The dense array can be paired 1:1 with any arbitrary data array
- In my case: Component Storage + Bitmask

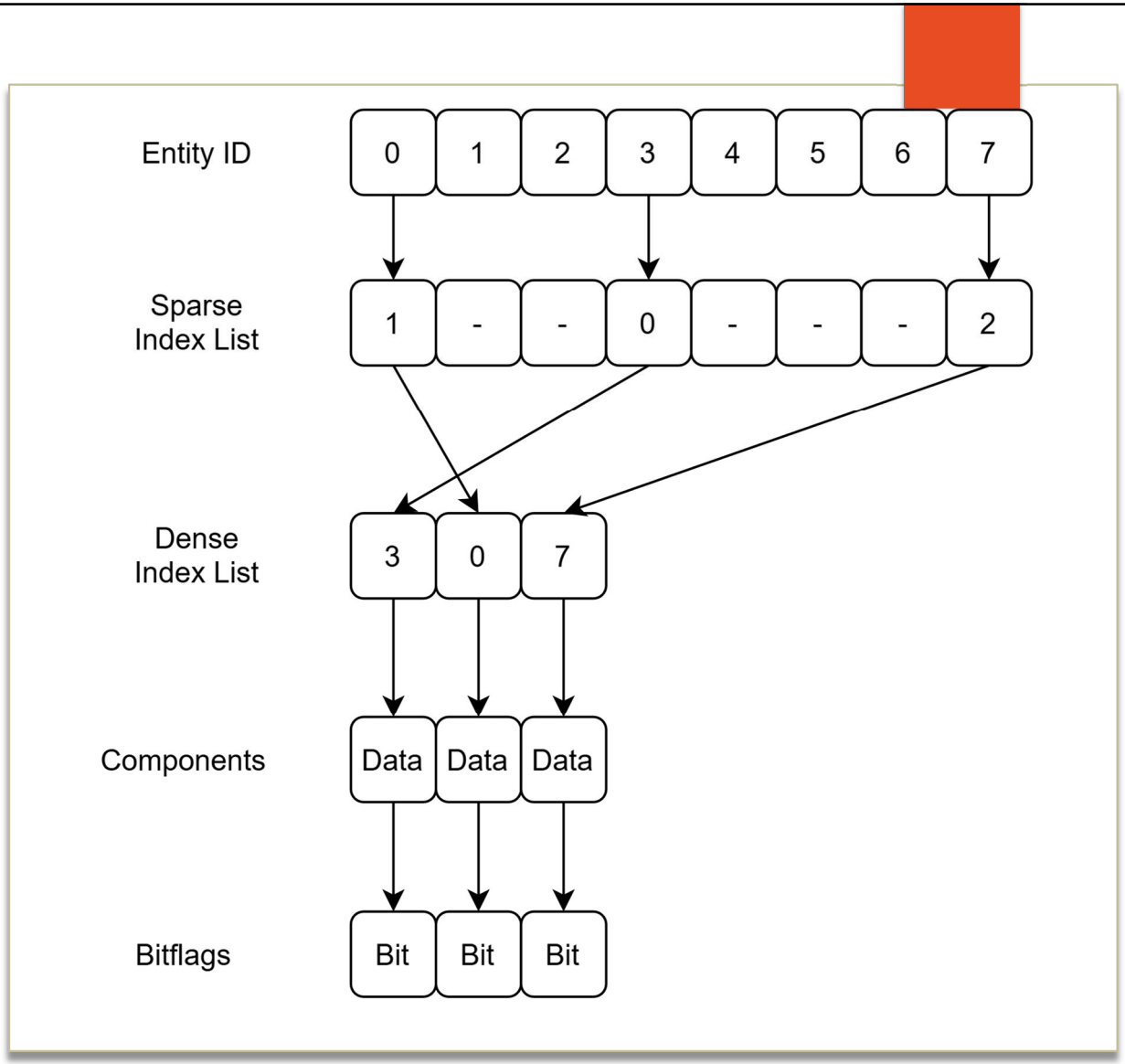


https://skypjack.github.io/

Data-Oriented Entity-Component-System III.

Sparse-Set architecture pros:

- Dense component storage excellent cache locality and hardware prefetching
- True O(1) lookup and access!!
- Easy to parallelize by iterating over the dense array
- Independent systems (component pools) can run fully in parallel
- Compact data layout enables efficient CPU/GPU synchronization
- Both Sparse and Dense arrays can be uploaded directly to the GPU!!



Data-Oriented Entity-Component-System IV.

Parallelization:

- Replacinbg std::execution::seq with std::execution::par parallelizes a system with minimal code changes
- Good for quick wins, but not ideal for production game engines.
- No control over scheduling, thread affinity, or worker count!!
- Use a custom thread pool (~ one worker per CPU core) for maximum performance
- Split works into chunks and schedule them as jobs.
- **Taskflow:** provides excellent task scheduling and dependency management

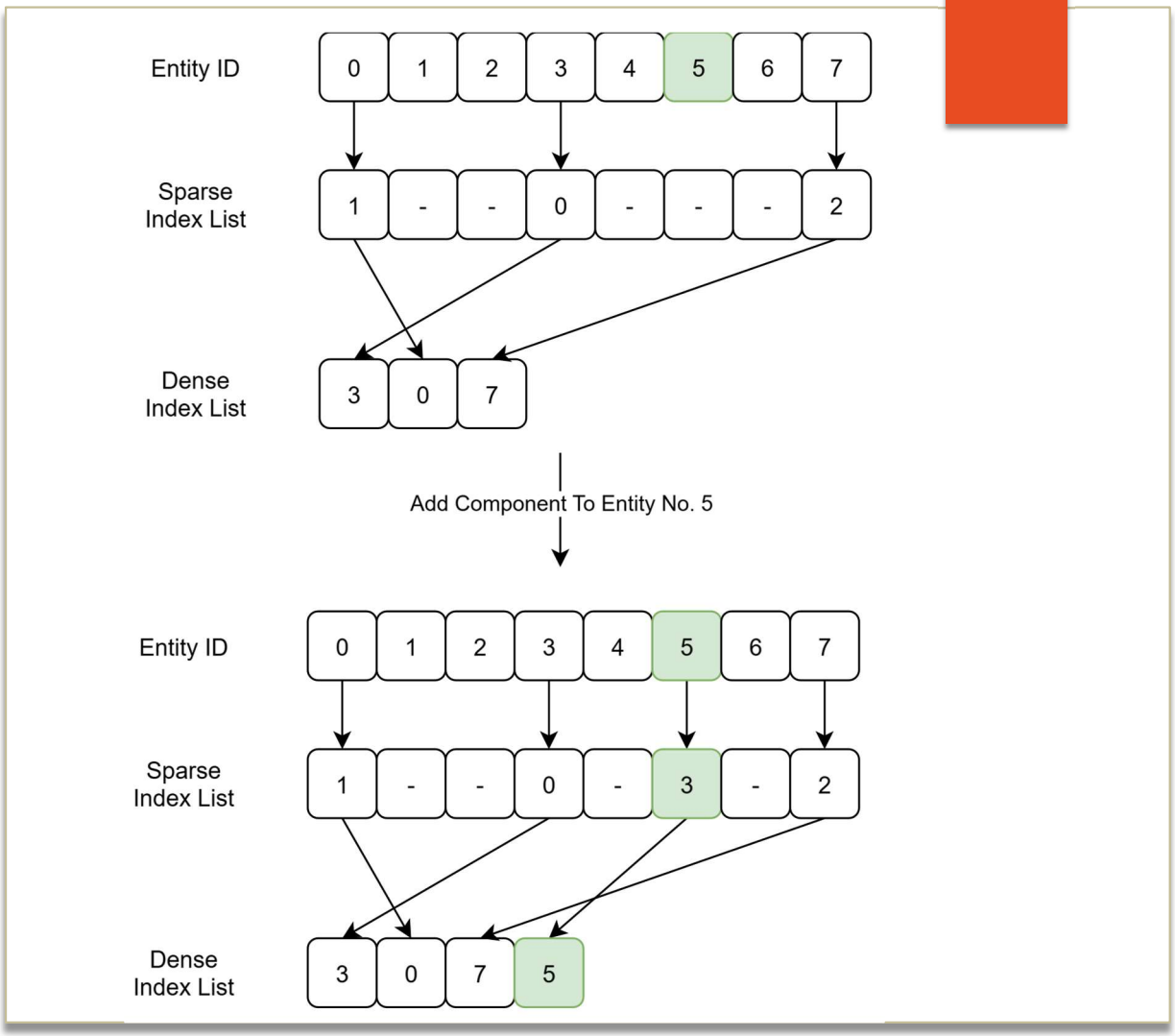
```
std::for_each(std::execution::par, denseEntities.begin(), denseEntities.end(),
    [&](EntityID entity) {
        size_t denseIndex = sparse[entity];
        TransformComponent& transformComponent = denseComponents[denseIndex];

        glm::mat4 localMat = glm::translate(glm::mat4(1.0f), transformComponent.translation);
        localMat = glm::rotate(localMat, glm::radians(transformComponent.rotation.z), glm::vec3(0, 0, 1));
        localMat = glm::rotate(localMat, glm::radians(transformComponent.rotation.y), glm::vec3(0, 1, 0));
        localMat = glm::rotate(localMat, glm::radians(transformComponent.rotation.x), glm::vec3(1, 0, 0));
        localMat = glm::scale(localMat, transformComponent.scale);

        transformComponent.transform = localMat;
    });
```

https://github.com/taskflow/taskflow

Data-Oriented Entity-Component-System V.

Sparse Set – Add operation:

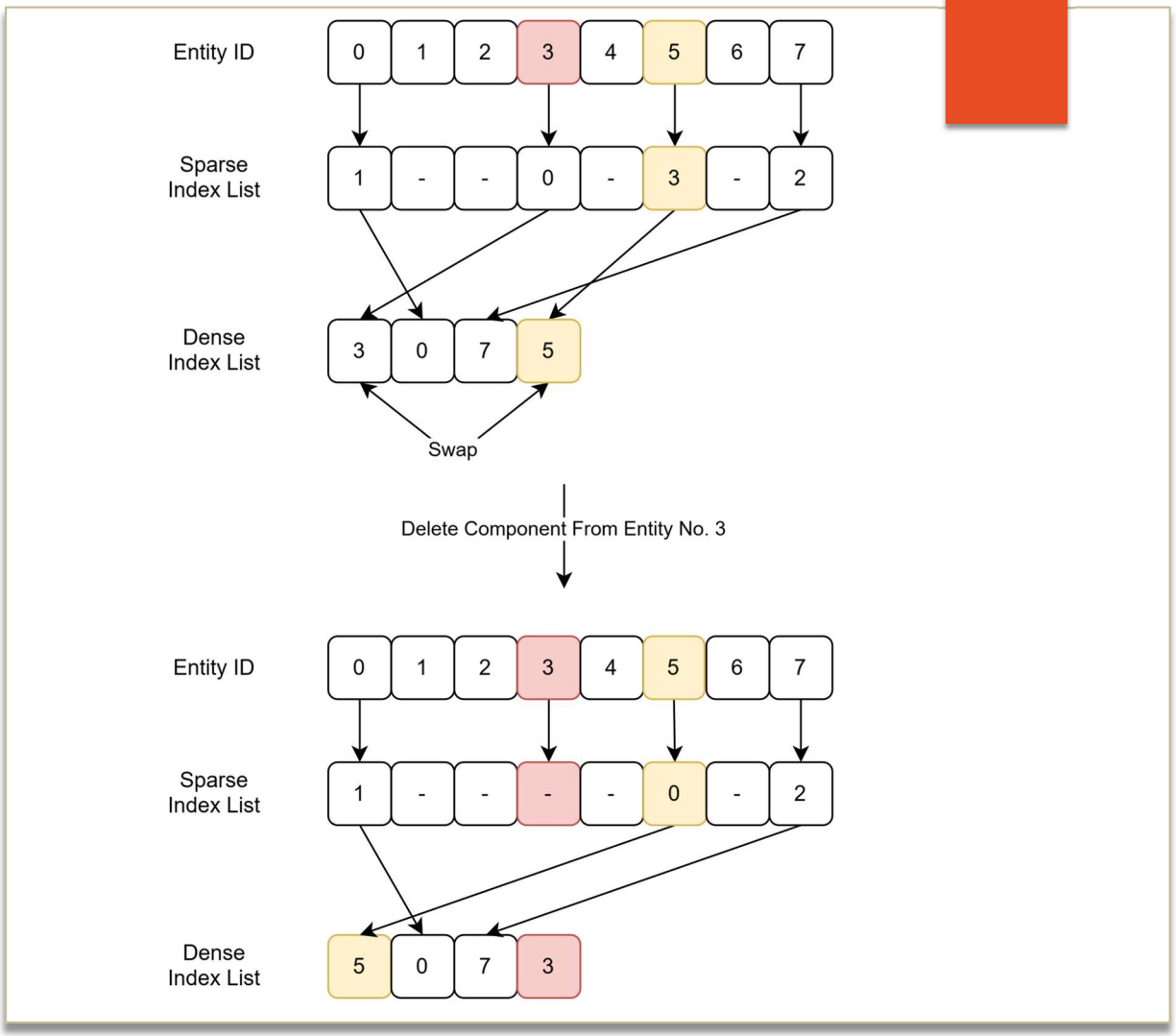
- Append the new component and entity to the end of the dense arrays.
- Update the correspondidng entry in the sparse array
- Insertion is true O(1) (excluding reallocation)
- The dense array remains compact and cache-friendly
- No shifting or reordering of existing elements is required!!



Data-Oriented Entity-Component-System VI.

Sparse Set – Remove operation:

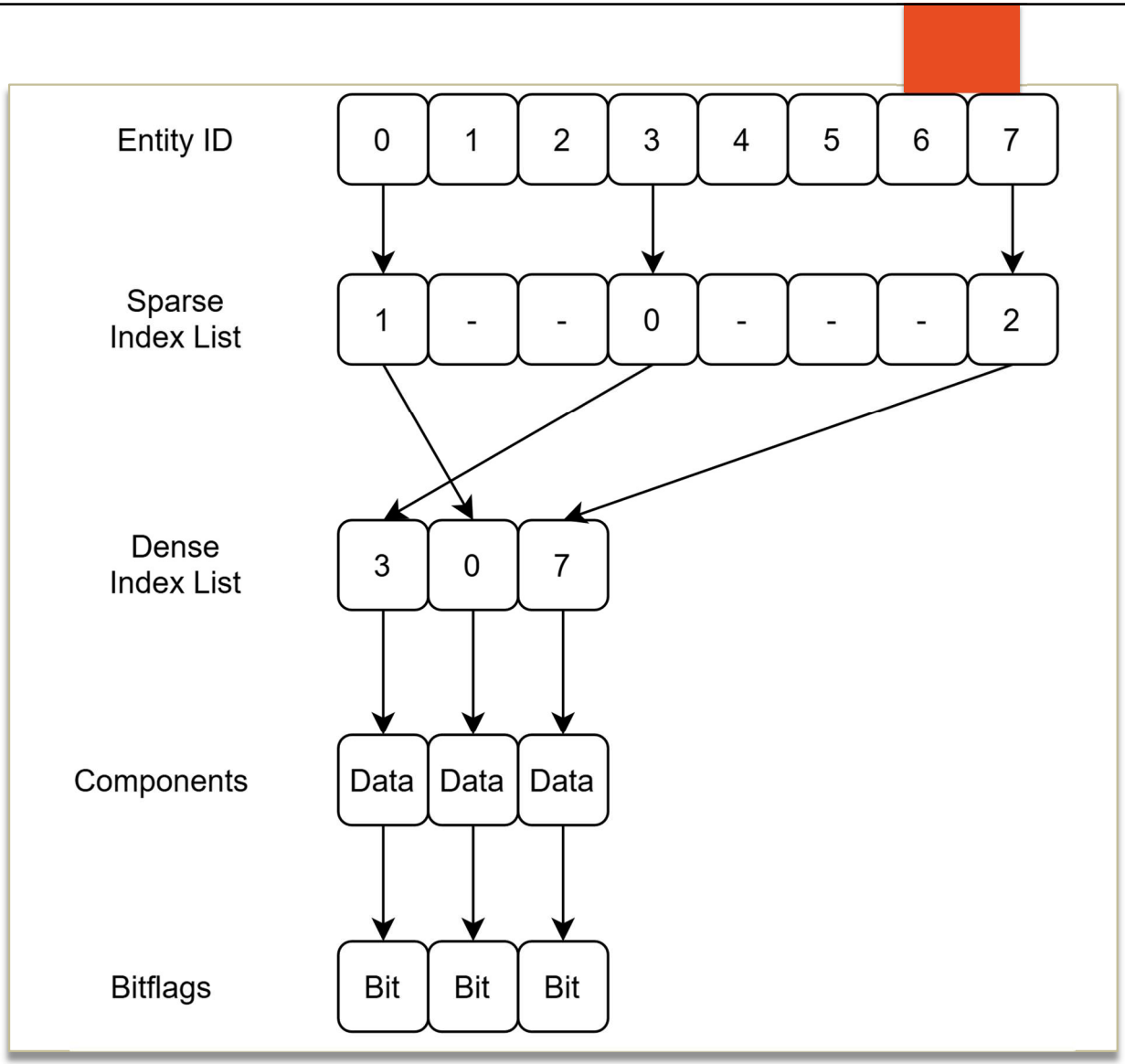
- Swap the element to remove with the last element in the dense arrays.
- The removed element is now at the end and can be erased with a single pop_back
- Update the the sparse array: Mark the removed entity as invalid, and remap the swapped entity to its new dense index.
- Removal is also true O(1)
- Only one element is relocated, keeping reordering to a minimum!!



Data-Oriented Entity-Component-System VII.

Sparse Set – Remaining Challanges:

- The sparse array grows with the maximum entity ID, leaving unused memory for gaps
- This is more noticable on the GPU, but even 1 million entities only require a few MB
- Dense iteration is highly efficient, but not every component needs to be updated every frame
- Ideally, only modified or „dirty" components should be processed.



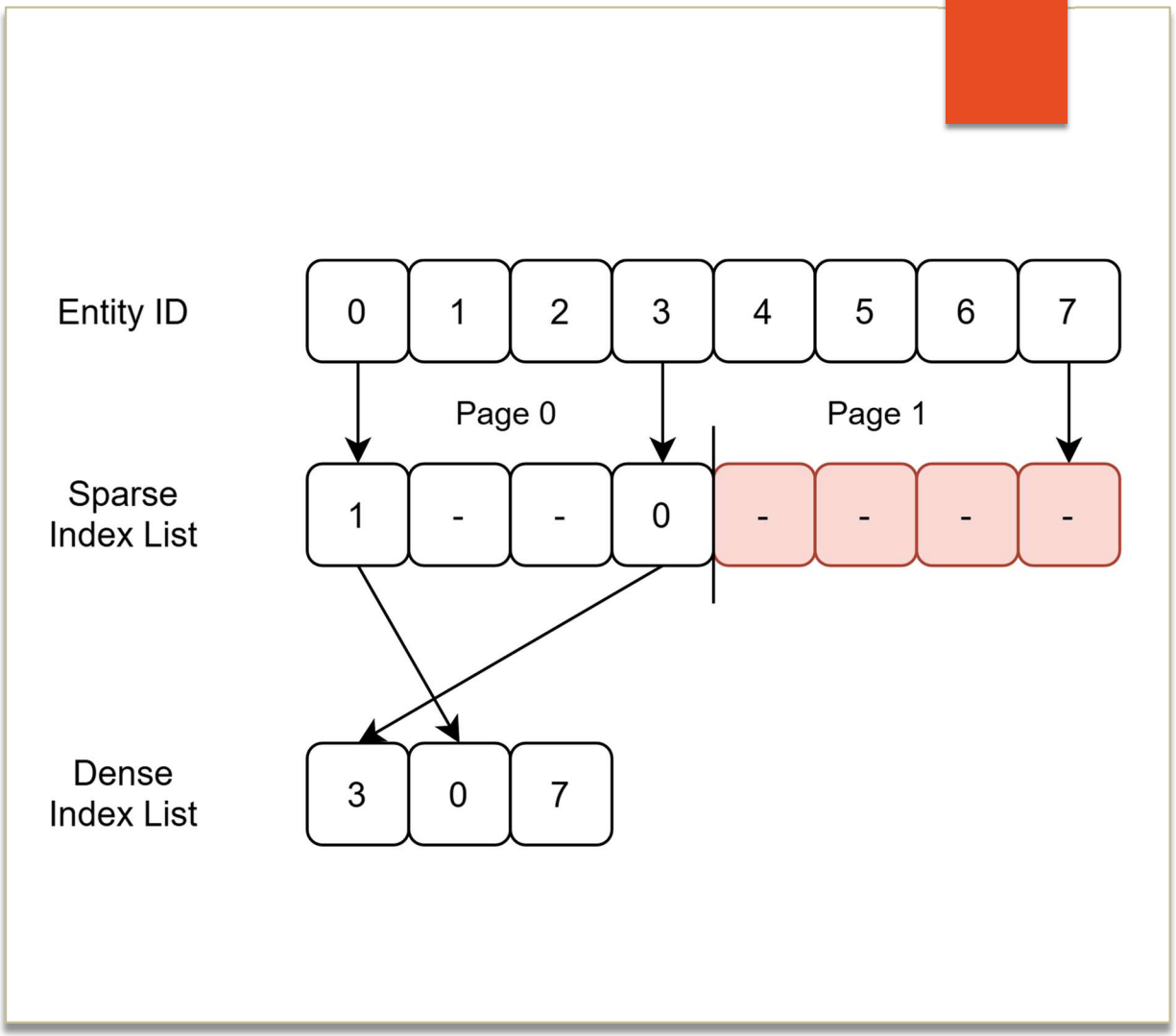
Data-Oriented Entity-Component-System VIII.

Solution – Paged Sparse Map

- Split the sparse array into fixed-size pages (1024 entities per page)
- Allocate a page only when it contains at least one component.
- Completely empty pages consume no memory, a good trade-off when memory usage is the primary concern!

Cons

- Extra level of indirection and page-to-dense index calculation.
- Overhead on both CPU and GPU
- **In this engine, the standard flat sparse set is used!!**



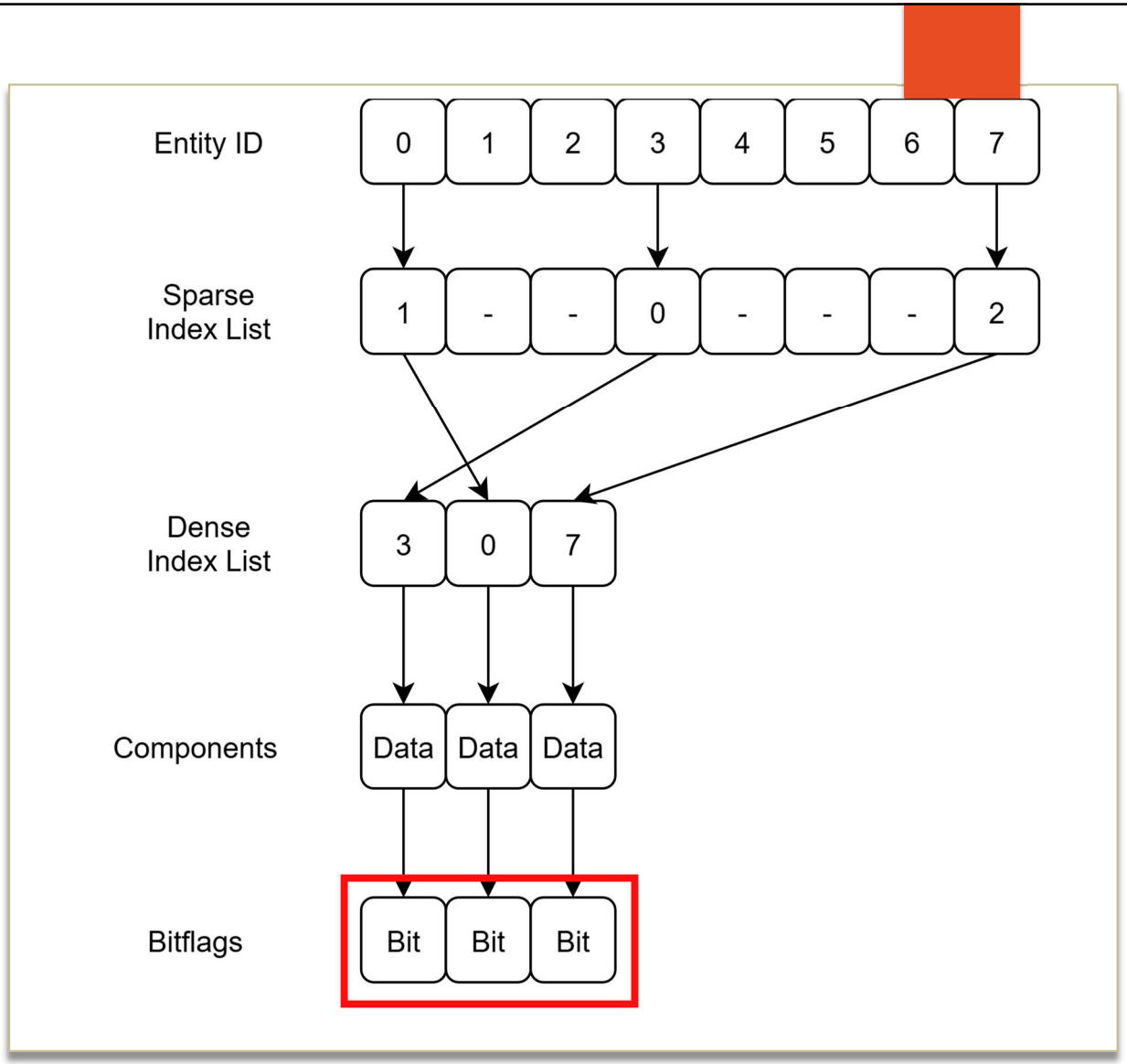
Data-Oriented Entity-Component-System IX.

Solution – Bitflag Array

- State flags provide a lightweight communication channel between systems.
- **Update Bit:** Marks a component that needs to be processed.
- **Changed Bit:** Indicates the component was modified, enabling hierarchical/dependent updates.
- Additional flags: Queued, custom sates or engine-specific flags.

Optimization – Pool-level dirty flag

- If no entity has changed, the entire pool can be skipped without iterations.



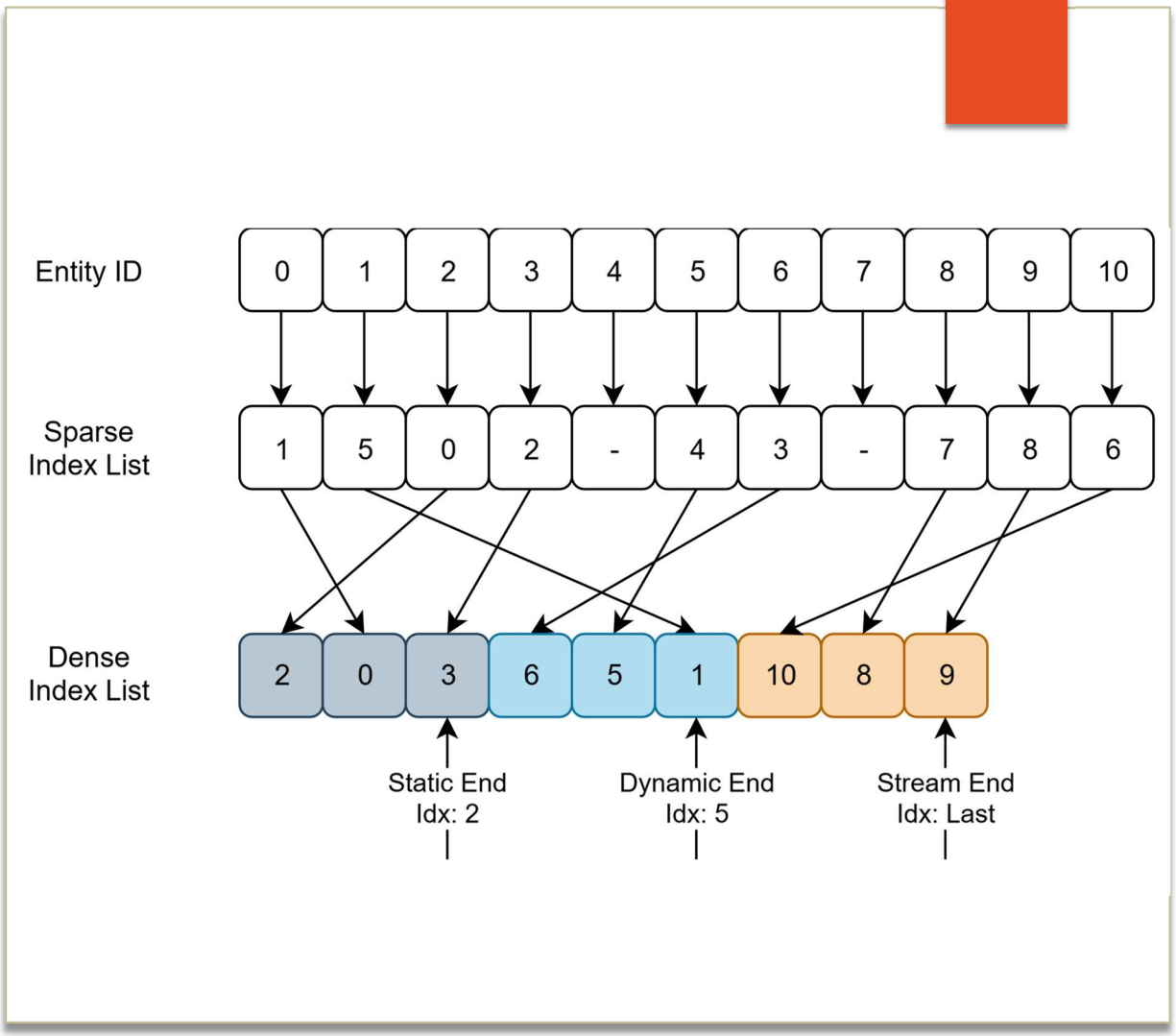
Data-Oriented Entity-Component-System X.

Problem – Sparse Set Scalability

- Imagine we have 1 million entities, even if most iterations exit immediately due to the bitflag optimization, scanning the entire array is way too expensive…
- Iteration cost should scale with the amount of changed data, not total entity/component count.

Solution – Segmented Sparse Set

- Partition the dense array into **update-frequency segments**.



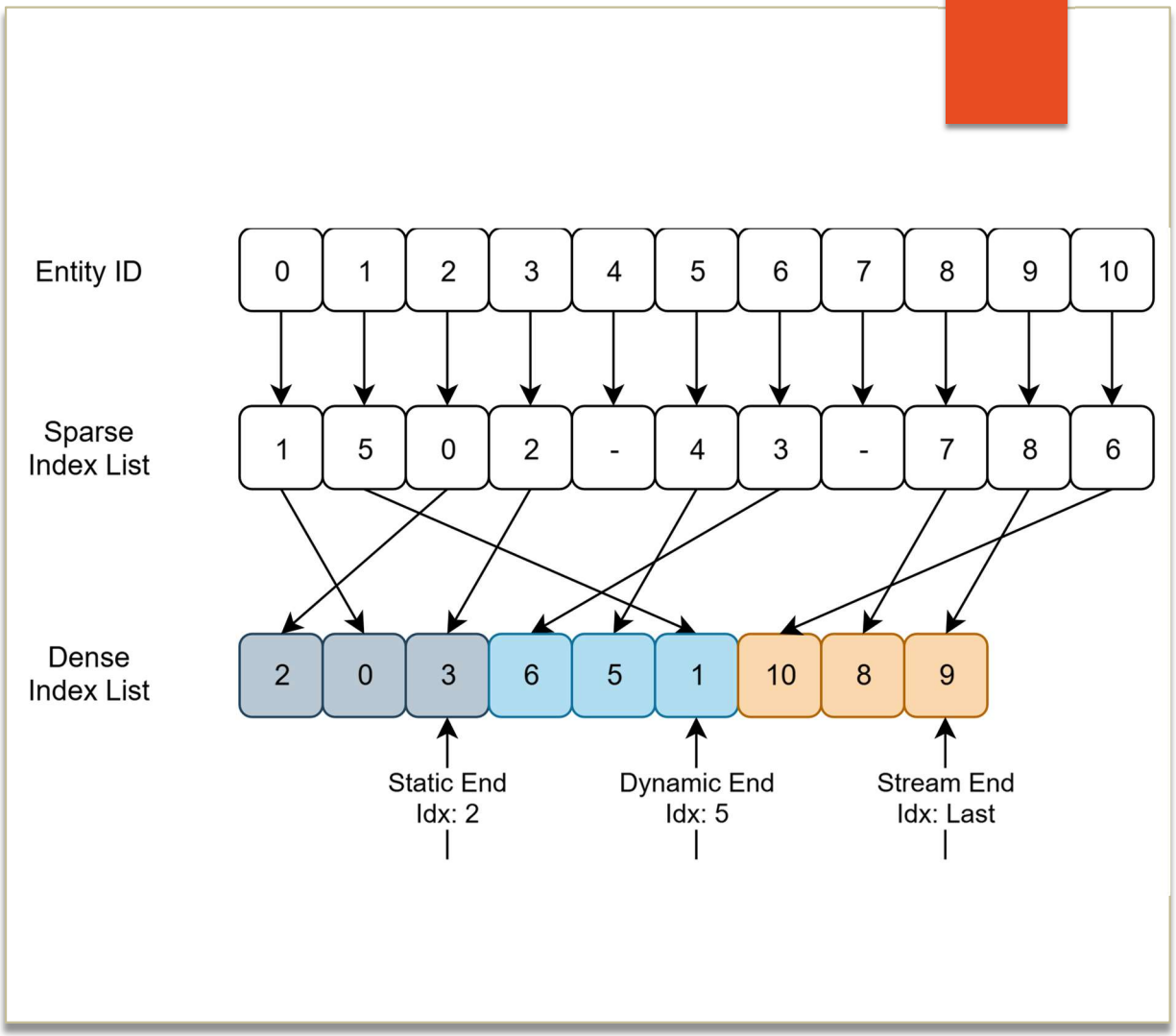
Data-Oriented Entity-Component-System XI.

Static Segment

- Components that rarely change.
- Changed entities are appended to a dedicated dirty index array.
- An atomic counter tracks the number of dirty entries.
- Only the dirty array is iterated during updated!!

Dynamic Segment

- Components that change occasionally.
- Uses per-component bitflags (Update/Changed…)
- Uses a pool-level dirty flag to skip the entire segment when unchanged.
- Iterates the segment and updated only flagged components



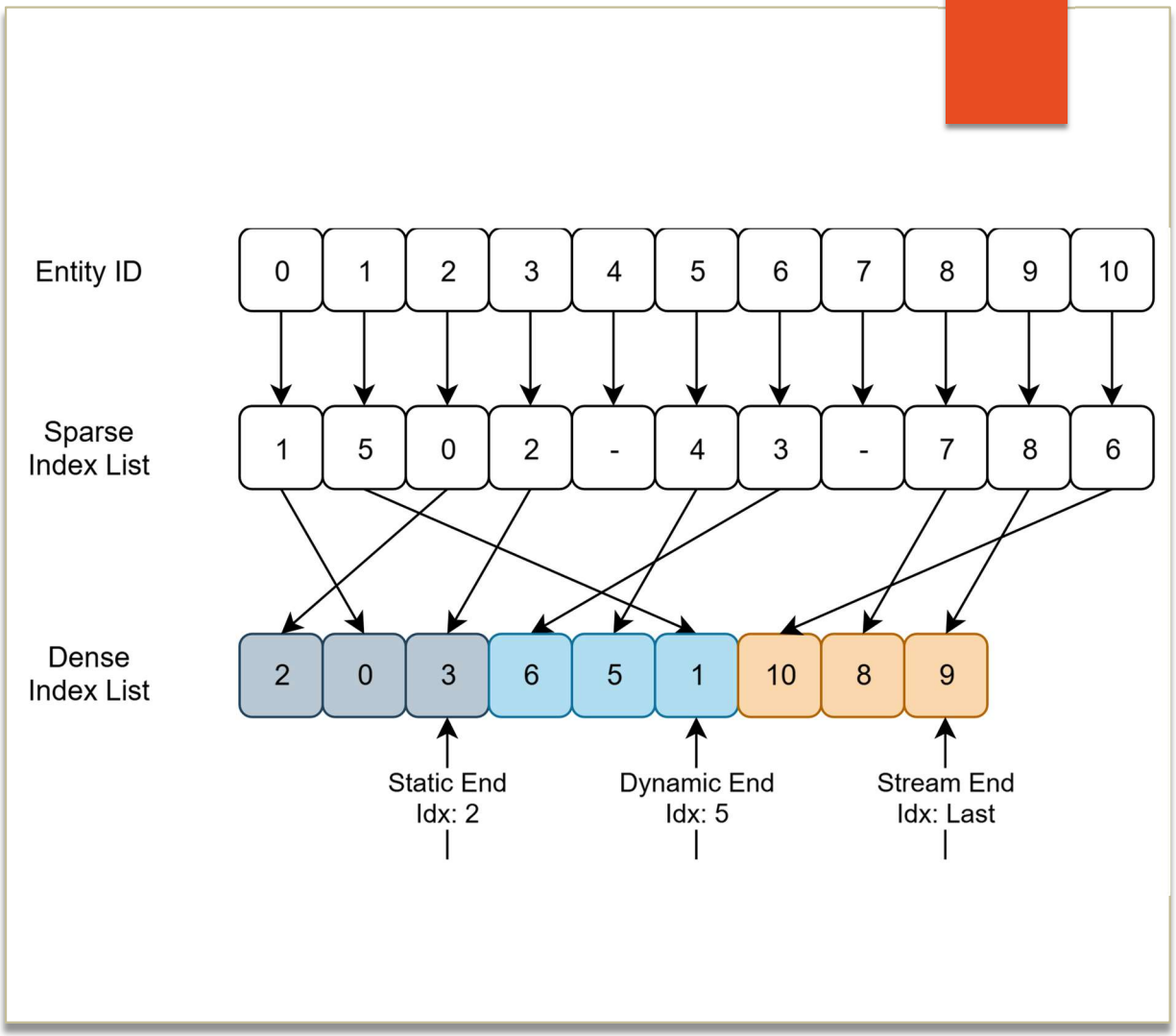
Data-Oriented Entity-Component-System XII.

Stream Segment

- Components updated every frame.
- No dirty tracking or filtering.
- The entire segment is processed every frame. (Physics, Animation, Scripting…)

Segmented Sparse Set Layout

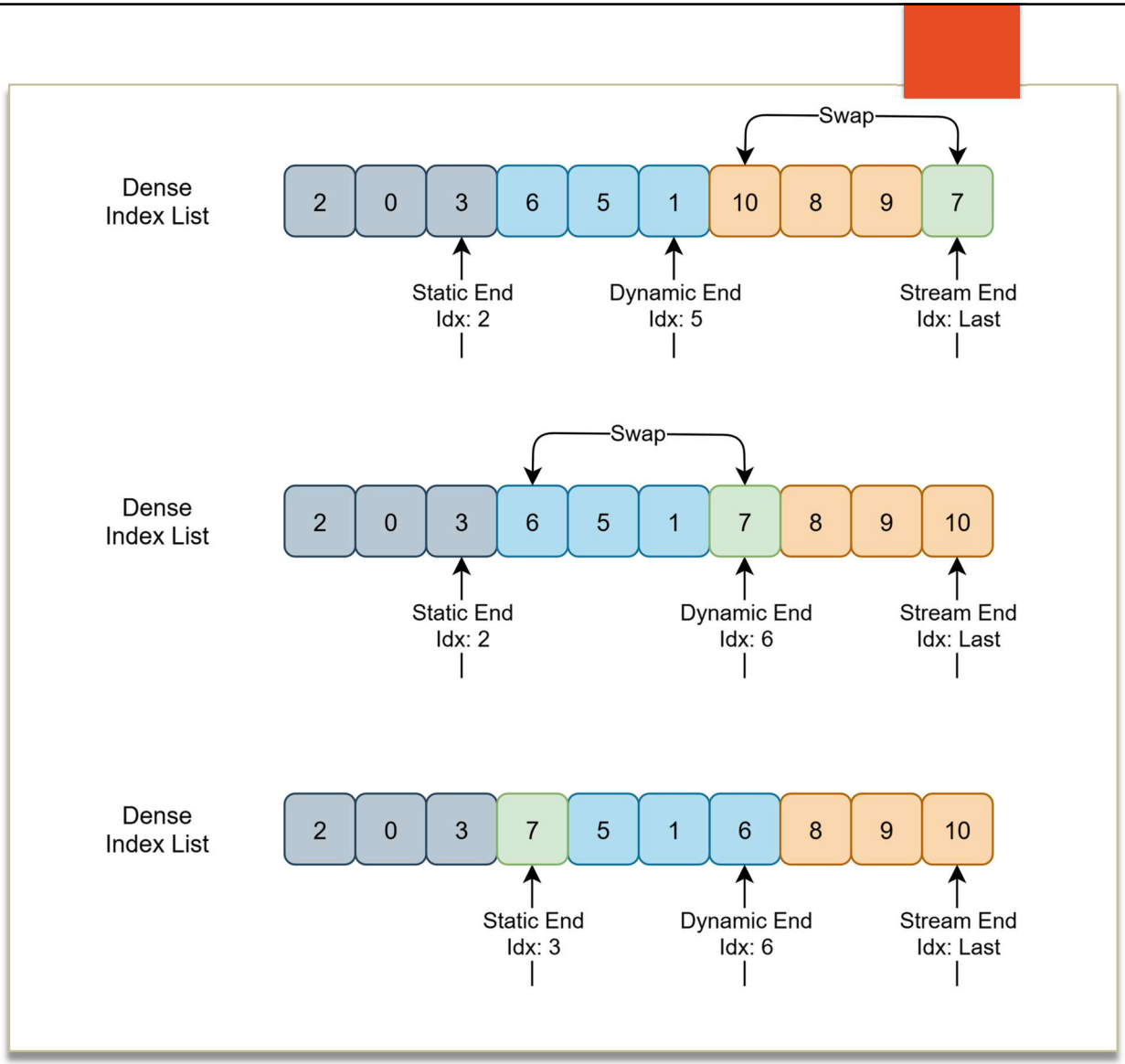
- Two boundary indices, static-end and dynamic-end define the segment ranges
- These boundaries are updated when components are added, removed or change update frequency.



Data-Oriented Entity-Component-System XIII.

Segmented Sparse Set – Add OP

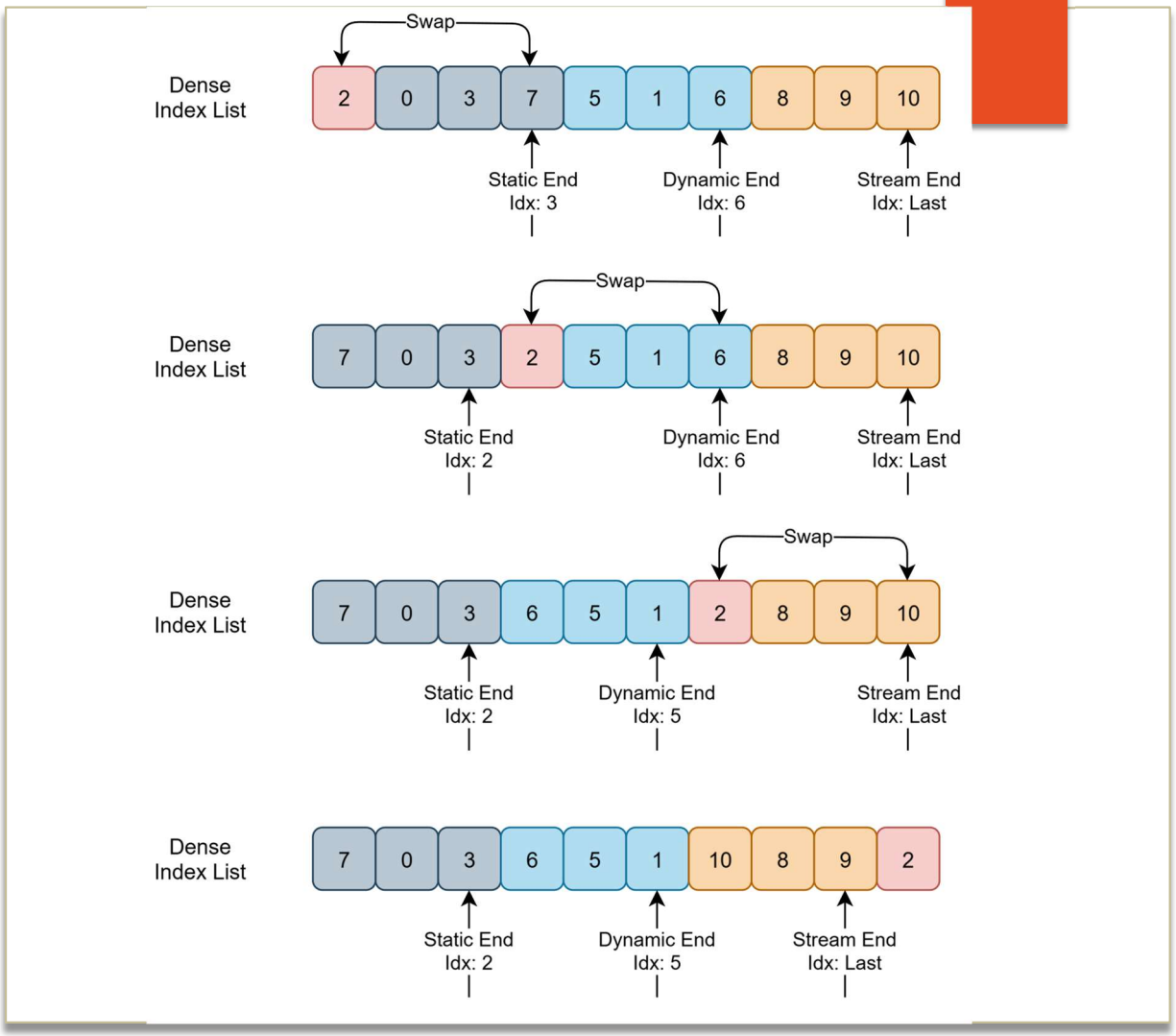
- Worst case: only 2 swaps are required
- Insert the new component at the end of the dense array (end of stream region)
- Swap it with the first element of the stream region (dynamic-end + 1) and update the boundary
- Swap it with the first element of the dynamic region (static-end + 1) and update the boundary
- The new component is finally at the end of the static region, preserving the segmented layout.



Data-Oriented Entity-Component-System XIV.

Segmented Sparse Set – Remove

- Worst case: only 3 swaps are required
- Swap the element to remove with the last element of the static region and update the boundary
- Swap it with the last element of the dynamic region and update the boundary.
- Swap it with the last element of the stream region, moving it to the end of the dense array.
- Remove the element with a single pop_back() operation.



Data-Oriented Entity-Component-System XV.

Performance Benchmark

- Compared agains EnTT, the benchmark shows similar overall performance.
- In the engine update loop, this ECS performs better due to native support of static/dynamic/stream segmentation.
- **Test Setup**: 1 million entity, 70/20/10 distribution.
- EnTT overhead was minimized as well, using flags to disable checks and runtime overhead.
- **This ECS is puprose-built for this engine**, allowing more aggressive optimizations and better update performance!!

Table 1: *Performance comparison between the Custom Segmented ECS (Synapse) and EnTT. CPU time in ms (lower is better).*

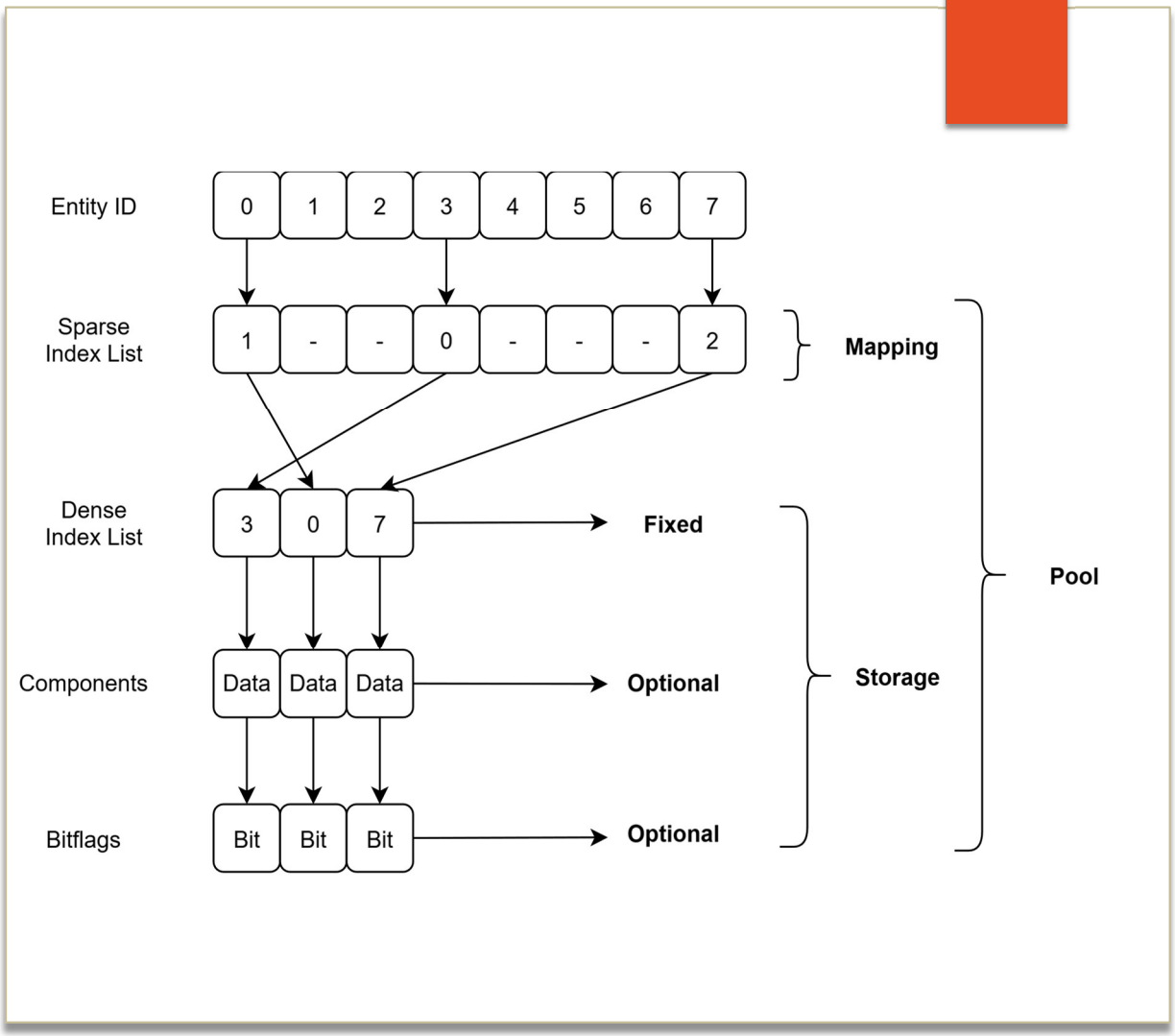
Operation	Entities	Synapse (ms)	EnTT (ms)
Add	100k	6.08	2.99
	1M	59.66	35.59
Remove	100k	2.31	3.29
	1M	26.18	37.01
Iterate (Seq)	100k	0.24	0.20
	1M	2.46	2.25
Iterate (Par)	100k	0.25	0.25
	1M	2.15	2.26
Engine Update	100k	0.11	0.14
	1M	1.15	1.76

Data-Oriented Entity-Component-System XVI.

Implementation – Design Goals

- No virtual functions, zero runtime polymorphism overhead
- Highly modular architecture, mapping and storage layers must be replacable independetly
- Support multiple implementations:
  - Mapping: Flat/Paged
  - Storage: Flat/Segmented
- Support different use cases:
  - Sparse Set indexing layer only
  - Data stroage without bitflags
  - Bitflags without component data
  - Full data and bitflag storage

Solution: Heavy template metaprogramming magic ☺



https://github.com/TamasPetii/SynapseEngine/tree/main/SynapseEngine/Engine/Registry

Data-Oriented Entity-Component-System XVII.

CRTP – Static Polymorphism

- Provides compile-time polymorphism without virtual functions.
- The derived class is passed as the template parameter.
- Base class forwards calls to the derived implementation using static_cast.
- Function calls are resolved at compilte time, zero runtime overhead.
- Concepts enforce the required interface at compilte time!!

Mapping Layer

- SparseVectorMapping and SparsePagedMapping both implement the same CRTP interface, any mapping type can be used as long as it satisfies MappingConstraint.

```
template<typename T>
concept MappingConstraint = std::is_base_of_v<MappingCRTP<T>, T>;

template<typename Derived>
struct MappingCRTP
{
    SYN_INLINE void Set(EntityID entity, DenseIndex index) { static_cast<Derived*>(this)->Set(entity, index); }
    SYN_INLINE DenseIndex Get(EntityID entity) const       { return static_cast<const Derived*>(this)->Get(entity); }
    SYN_INLINE void Remove(EntityID entity)                { static_cast<Derived*>(this)->Remove(entity); }
    SYN_INLINE bool Contains(EntityID entity) const        { return static_cast<const Derived*>(this)->Contains(entity); }
};

class SYN_API SparseVectorMapping : public MappingCRTP<SparseVectorMapping>
{
public:
    SYN_INLINE void Set(EntityID entity, DenseIndex index);
    SYN_INLINE DenseIndex Get(EntityID entity) const;
    SYN_INLINE void Remove(EntityID entity);
    SYN_INLINE bool Contains(EntityID entity) const;
private:
    std::vector<DenseIndex> _indices;
};

class SYN_API SparsePagedMapping : public MappingCRTP<SparsePagedMapping>
{
    SYN_INLINE void Set(EntityID entity, DenseIndex index);
    SYN_INLINE DenseIndex Get(EntityID entity) const;
    SYN_INLINE void Remove(EntityID entity);
    SYN_INLINE bool Contains(EntityID entity) const;
private:
    std::vector<std::unique_ptr<DenseIndex[]>> _pages;
};
```

https://github.com/TamasPetii/SynapseEngine/tree/main/SynapseEngine/Engine/Registry/Pool/Mapping

Data-Oriented Entity-Component-System XVIII.

Mixin-Based Design

- Mixins allow optional features to be composed at compile time.
- Storage can be configures with any combinations of features:
  Data-Only / Flags-Only
  Data + Flags / No extra storage
- Unused functionality is replacced with empty implementation, compiler eliminates these calls during optimization!!

Storage Backend

- DataMixin<T>: provides component storage
- FlagMixinPoliy: provides flag storage (atomic, bitset, uint flags!!)
- **Features are composed through inheritance at compile time!!**

```
template<typename T>
struct DataMixin
{
public:
    SYN_INLINE void PushData(T&& value);
    SYN_INLINE void PopData();
    SYN_INLINE void SwapData(DenseIndex a, DenseIndex b);
protected:
    std::vector<T> _data;
};

template<>
struct DataMixin<void>
{
    template<typename U>
    SYN_INLINE void PushData(U&&) {}
    SYN_INLINE void PopData() {}
    SYN_INLINE void SwapData(DenseIndex, DenseIndex) {}
};

template<typename T, typename FlagMixinPolicy = NoFlagMixin> requires FlagMixinConstraint<FlagMixinPolicy>
class StorageBackend : public DataMixin<T>, public FlagMixinPolicy
{
public:
    template<typename U = T> requires (!std::is_void_v<U>)
    SYN_INLINE void PushBackend(EntityID entity, U&& value);
protected:
    std::vector<EntityID> _entities;
};

template<typename T, typename FlagMixinPolicy> requires FlagMixinConstraint<FlagMixinPolicy>
template<typename U> requires (!std::is_void_v<U>)
SYN_INLINE void StorageBackend<T, FlagMixinPolicy>::PushBackend(EntityID entity, U&& value)
{
    _entities.push_back(entity);
    this->PushData(std::forward<U>(value));
    this->PushFlag();
}
```

https://github.com/TamasPetii/SynapseEngine/tree/main/SynapseEngine/Engine/Registry/Pool/Storage/Mixin

Data-Oriented Entity-Component-System XIX.

Storage Layer

- StorageBackend provides the common storage functionality
- CRTP defines a common interface with zero runtime ovearhead
- FlatStorageImpl is a thin wrapper around the backend.
- SegmentedStorageImpl extends the backend with additional logic, adding the static/dynamic pointers, and the dirty list and atomic counter for the static region.
- New storage implementations (Only Static/Dynamic no stream, or making more regions…) can be added without modifying the ECS core!!

```cpp
template<typename T>
concept StorageConstraint = std::is_base_of_v<StorageCRTP<T, typename T::ValueType>, T>;

template<typename Derived, typename DataType>
struct StorageCRTP
{
    template<typename U = DataType> requires (!std::is_void_v<U>)
    SYN_INLINE U& Get(DenseIndex index) { return static_cast<Derived*>(this)->template Get<U>(index); }

    template<typename U = DataType> requires (!std::is_void_v<U>)
    SYN_INLINE void Push(EntityID entity, U&& value) { static_cast<Derived*>(this)->Push(entity, std::forward<U>(value)); }
};

template<typename T, typename FlagMixinPolicy = NoFlagMixin>
class FlatStorageImpl :
    public StorageBackend<T, FlagMixinPolicy>,
    public StorageCRTP<FlatStorageImpl<T, FlagMixinPolicy>, T>
{
    using Base = StorageBackend<T, FlagMixinPolicy>;
public:

    template<typename U = T> requires (!std::is_void_v<U>)
    SYN_INLINE U& Get(DenseIndex index);

    template<typename U = T> requires (!std::is_void_v<U>)
    SYN_INLINE void Push(EntityID entity, U&& value);
};

template<typename T, typename FlagMixinPolicy = SimpleFlagMixin>
class SegmentedStorageImpl :
    public StorageBackend<T, FlagMixinPolicy>,
    public StorageCRTP<SegmentedStorageImpl<T, FlagMixinPolicy>, T>
{
    using Base = StorageBackend<T, FlagMixinPolicy>;
public:
    template<typename U = T> requires (!std::is_void_v<U>)
    SYN_INLINE U& Get(DenseIndex index);

    template<typename U = T> requires (!std::is_void_v<U>)
    SYN_INLINE void Push(EntityID entity, U&& value);
protected:
    size_t _staticEnd = 0;
    size_t _dynamicEnd = 0;
    std::vector<EntityID> _dirtyStaticList;
    std::atomic<size_t>   _dirtyStaticCount{ 0 };
};
```

https://github.com/TamasPetii/SynapseEngine/tree/main/SynapseEngine/Engine/Registry/Pool/Storage/Core

Data-Oriented Entity-Component-System XX.

Pool – Combining Mapping and Storage

- The Pool synchronizes the Mapping and Storage layers.
- **External code interacts only with the Pool**, not the individual Mapping or Storage implementation!!
- The Pool provides a single, unifued interface while remaining fully modular.

Remaining Challange

- Different Mapping and Storage implementations expose their own specialized functionality.
- Ideally, these functions should also be accessible thrgough the Pool without exposing its internal implementation!!

```
template<typename T, typename StoragePolicy, typename MappingPolicy>
    requires StorageConstraint<StoragePolicy>&& MappingConstraint<MappingPolicy>
class Pool :
    public StorageTraits<StoragePolicy>::template Extension<Pool<T, StoragePolicy, MappingPolicy>>,
    public MappingTraits<MappingPolicy>::template Extension<Pool<T, StoragePolicy, MappingPolicy>>
{
    template<typename U = T> requires (!std::is_void_v<U>)
    SYN_INLINE void Add(EntityID entity, U&& value);

    template<typename U = T> requires (!std::is_void_v<U>)
    SYN_INLINE U& Get(EntityID entity);
private:
    StoragePolicy _storage;
    MappingPolicy _mapping;
};

template<typename T, typename StoragePolicy, typename MappingPolicy>
requires StorageConstraint<StoragePolicy> && MappingConstraint<MappingPolicy>
template<typename U> requires (!std::is_void_v<U>)
SYN_INLINE void Pool<T, StoragePolicy, MappingPolicy>::Add(EntityID entity, U&& value)
{
    SYN_ASSERT(!_mapping.Contains(entity), "Entity already has this component");
    _storage.Push(entity, std::forward<U>(value));
    _mapping.Set(entity, static_cast<DenseIndex>(_storage.Size() - 1));
}

template<typename T, typename StoragePolicy, typename MappingPolicy>
requires StorageConstraint<StoragePolicy> && MappingConstraint<MappingPolicy>
template<typename U> requires (!std::is_void_v<U>)
    SYN_INLINE U& Pool<T, StoragePolicy, MappingPolicy>::Get(EntityID entity)
{
    return _storage.Get(_mapping.Get(entity));
}
```

https://github.com/TamasPetii/SynapseEngine/tree/main/SynapseEngine/Engine/Registry/Pool

Data-Oriented Entity-Component-System XXI.

Template Extension Pattern

- The Pool exposes only the functionality supported by its selected policies.
- Each Mapping/Storage implementation can provide its own Pool extension.
- StorageTraits selects the correct extension at compile time.
- **No virtual functions, no runtime dispatch, and no overhead. Everything is resolved at compile time.**

Example

- The SegmentedStorageImpl injects SetCategory() directly into Pool.

```
template<typename PoolType>
struct DefaultStorageExtension {};

template<typename StoragePolicy>
struct StorageTraits
{
    template<typename PoolType>
    using Extension = DefaultStorageExtension<PoolType>;
};

template<typename DerivedPool>
struct SegmentedStorageImplExtension
{
private:
    SYN_INLINE DerivedPool& AsDerived() { return static_cast<DerivedPool&>(*this); }
    SYN_INLINE const DerivedPool& AsDerived() const { return static_cast<const DerivedPool&>(*this); }
public:

    SYN_INLINE void SetCategory(EntityID entity, StorageCategory newCat)
    {
        auto& pool = AsDerived();
        SYN_ASSERT(pool.GetMapping().Contains(entity), "Entity not in pool");

        const DenseIndex index = pool.GetMapping().Get(entity);

        pool.GetStorage().SetCategory(index, newCat, [&pool](EntityID movedEntity, DenseIndex newIndex) {
            pool.GetMapping().Set(movedEntity, newIndex);
            });
    }
};

template<typename T>
struct StorageTraits<SegmentedStorageImpl<T>>
{
    template<typename PoolType>
    using Extension = SegmentedStorageImplExtension<PoolType>;
};

template<typename T, typename StoragePolicy, typename MappingPolicy>
requires StorageConstraint<StoragePolicy>&& MappingConstraint<MappingPolicy>
class Pool :
public StorageTraits<StoragePolicy>::template Extension<Pool<T, StoragePolicy, MappingPolicy>>,
public MappingTraits<MappingPolicy>::template Extension<Pool<T, StoragePolicy, MappingPolicy>>
{
}
```

https://github.com/TamasPetii/SynapseEngine/tree/main/SynapseEngine/Engine/Registry/Pool/Storage/Extension

Data-Oriented Entity-Component-System XXII.

Registry

- The Registry owns and manages all component Pools.
- Each component type is associated with its own Pool.
- Active entities are stored in a Sparse Set, providing true O(1) lookup!!
- The Pools themselves are stored in a Data-Only Pool. (No flags are required)

Next challange

- How do we map a component type to its Pool safely across DLL boundaries?

```
class SYN_API Registry
    {
    public:
        template<typename T>
        using DefaultPoolType = ComponentPool<T>;

        template<typename T>
        using WrapperType = PoolWrapper<DefaultPoolType<T>>;
    public:
        template<typename T>
        void AddComponent(EntityID entity, T&& component);

        template<typename T>
        void RemoveComponent(EntityID entity);

        template<typename T>
        T& GetComponent(EntityID entity);

    private:
        EntityID _entityCounter = 0;
        std::vector<EntityID> _freeEntities;
        SparseSet _activeEntities;
        DataPool<IPool*> _pools;
    };
```

https://github.com/TamasPetii/SynapseEngine/tree/main/SynapseEngine/Engine/Registry

Data-Oriented Entity-Component-System XXIII.

DLL-Safe Type IDs

- The first access registers the component type in the global TypeManager.
- Registration is synchronized and relatively expensive, but happens only once per type.
- The assigned TypeID is cached in a template static variable (TypeInfo<T>::ID)
- All subsequent access are true O(1), with no map lookup or string hashing!

```cpp
template<typename T>
class TypeInfo {
public:
    static const TypeID ID;
private:
    static const char* GetName() {
        #if defined(_MSC_VER)
        return __FUNCSIG__;
        #elif defined(__clang__) || defined(__GNUC__)
                return __PRETTY_FUNCTION__;
        #else
                #error "Unsupported compiler for TypeInfo generation."
        #endif
    }
};

template<typename T>
const TypeID TypeInfo<T>::ID = TypeManager::Get().GetOrRegisterID(TypeInfo<T>::GetName());

class SYN_API TypeManager {
public:
    static TypeManager& Get();
    TypeID GetOrRegisterID(const std::string& typeName);
private:
    TypeManager() = default;
    TypeID _counter = 0;
    std::mutex _mtx;
    std::unordered_map<std::string, TypeID> _registry;
};

TypeID TypeManager::GetOrRegisterID(const std::string& typeName)
{
    std::scoped_lock lock(_mtx);

    auto it = _registry.find(typeName);
    if (it != _registry.end())
        return it->second;

    TypeID newID = _counter++;
    _registry[typeName] = newID;

    return newID;
}
```

https://github.com/TamasPetii/SynapseEngine/tree/main/SynapseEngine/Engine/Registry/Type

Data-Oriented Entity-Component-System XXIV.

Hierarchy – Parent/Child

- Hierarchy is implemented as a regular ECS component.
- Represents an intrusive double linked tree using entity ids (no pointers!!)
- Parent: Previoues hierarchy level
- FirstChild: Next hierarchy level
- NextSibling/PrevSibling: O(1) sibling insertion and removal
- DepthLevel and Topological Index locate the entity within the topological level arrays managed by the Hierarchy Manager

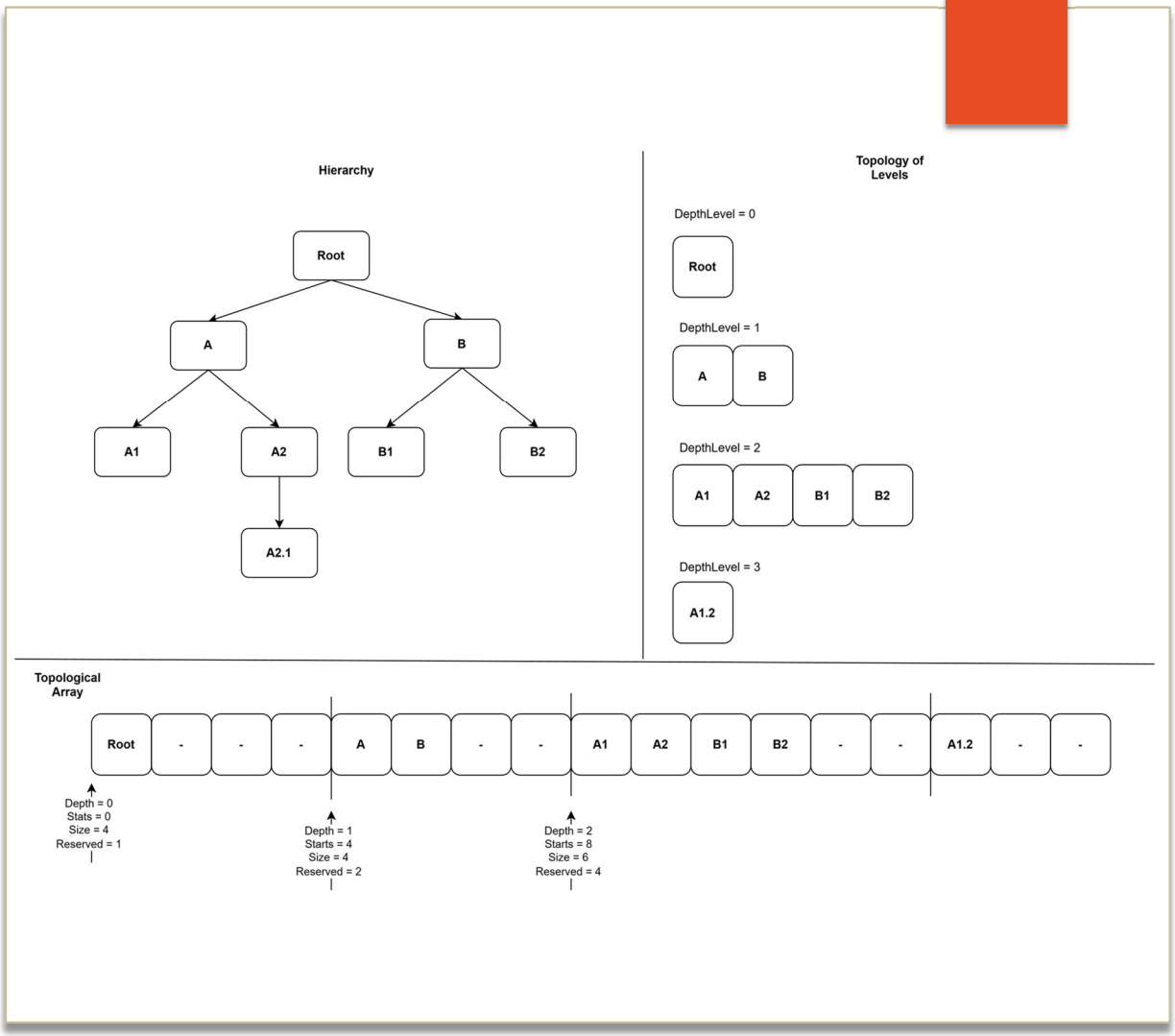
```
struct SYN_API HierarchyComponent
{
    EntityID parent = NULL_ENTITY;
    EntityID firstChild = NULL_ENTITY;
    EntityID nextSibling = NULL_ENTITY;
    EntityID prevSibling = NULL_ENTITY;

    uint32_t depthLevel = 0;
    uint32_t topoIndex = NULL_INDEX;
};
```

https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/Component/Core/HierarchyComponent.h

Data-Oriented Entity-Component-System XXV.

Hierarchy Manager

- The entire hierarchy is stored in a single flat topological array.
- Each hierarchy level owns a preallocated contiguous memory region for efficient insertions.
- Entities within the same level can be processed fully in parallel (parent doesn't matter!!)
- Excellent cache locality, thanks to packed memory.
- Rich adjacency information is available in O(1) like parent, children, siblings, level, topological index, topological siblings.



https://github.com/TamasPetii/SynapseEngine/tree/main/SynapseEngine/Engine/Scene/Hierarchy

Data-Oriented Entity-Component-System XXVI.

Hierarchy Updates

- Without hierarchy, the segmented ECS scales with the number of changed entities.
- Static/Dynamic/Stream segments can be updated independently and in parallel.
- Hierarchy introduces parent-child dependencies that prevent fully parallel execution. (Child cannot be updated before its parent)

The Challange

- The segmented pool has no knowledge of the hierarchy or update order.
- Changed parents imply changed descendants, but this information is not available in the pool.
- The challange is to merge hierarchy traversal with segmented updates while still iterating only over **affected entities**.

Hierarchy Updates Requirements

- Process hierarchy levels sequentially.
- Process all entities within each level fully in parallel.
- Iteralte only over entities that actually require an update.
- Maintain cache-efficient, data-oriented traversal.
- Keep complexity proportional to the number of affected entities, not the total entity count.

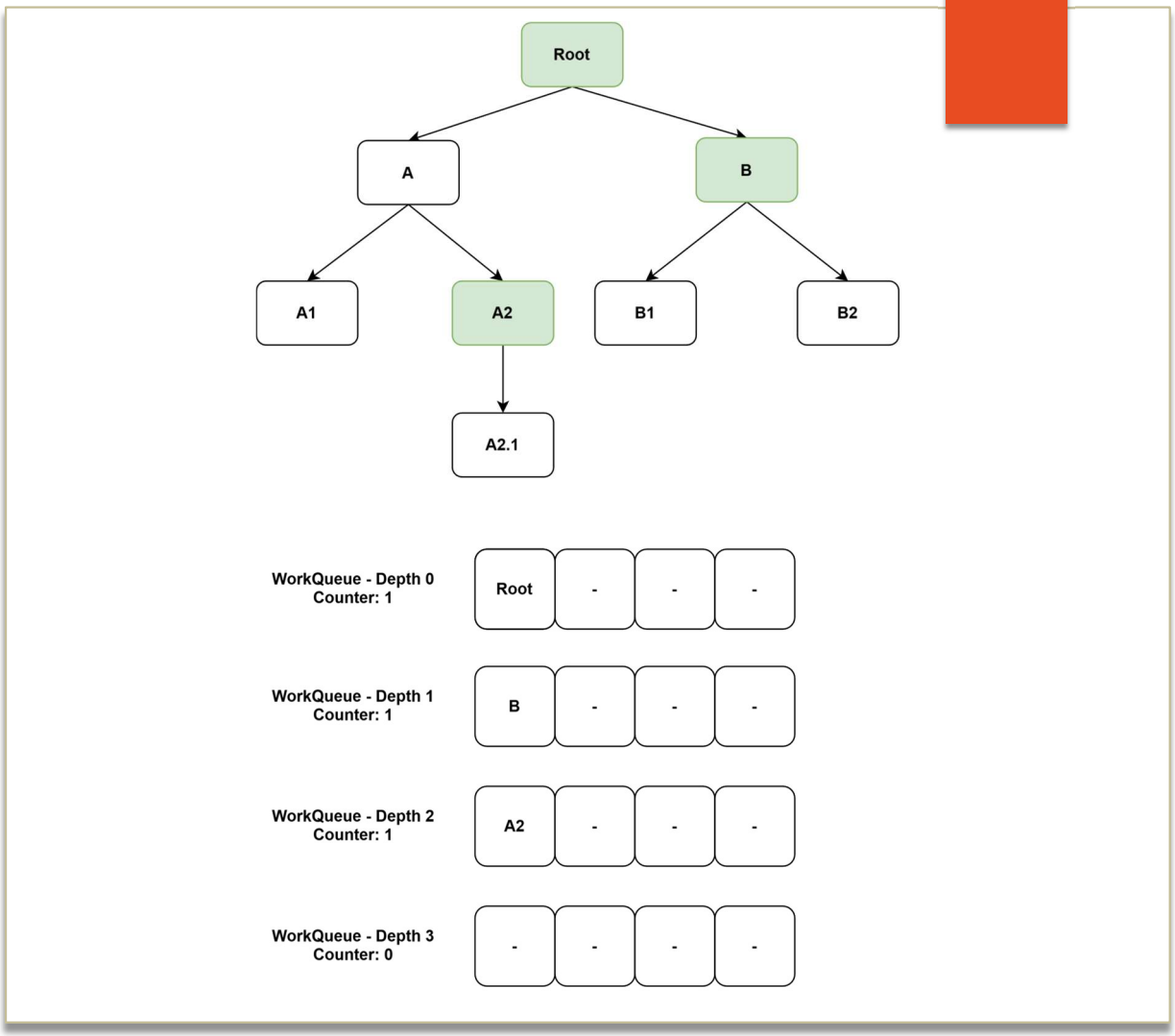
Update Category Rules

- Static parent: can have static/dynamic/stream child
- Dyanmic parent: can have dynamic/stream child.
- Stream parent: can have only stream child.

Data-Oriented Entity-Component-System XXVII.

Solution – Waterfall Propagation

- The segmented transform pool provides the initial set of changed entities.
- The HierarchyComponent provides the hierarchy depth for every entity.
- Each depth level owns a dirty array and an atomic counter.
- The initial dirty entities are distributed into their corresponding level queues in parallel.
- This produces the starting point for a hierarchical propagation

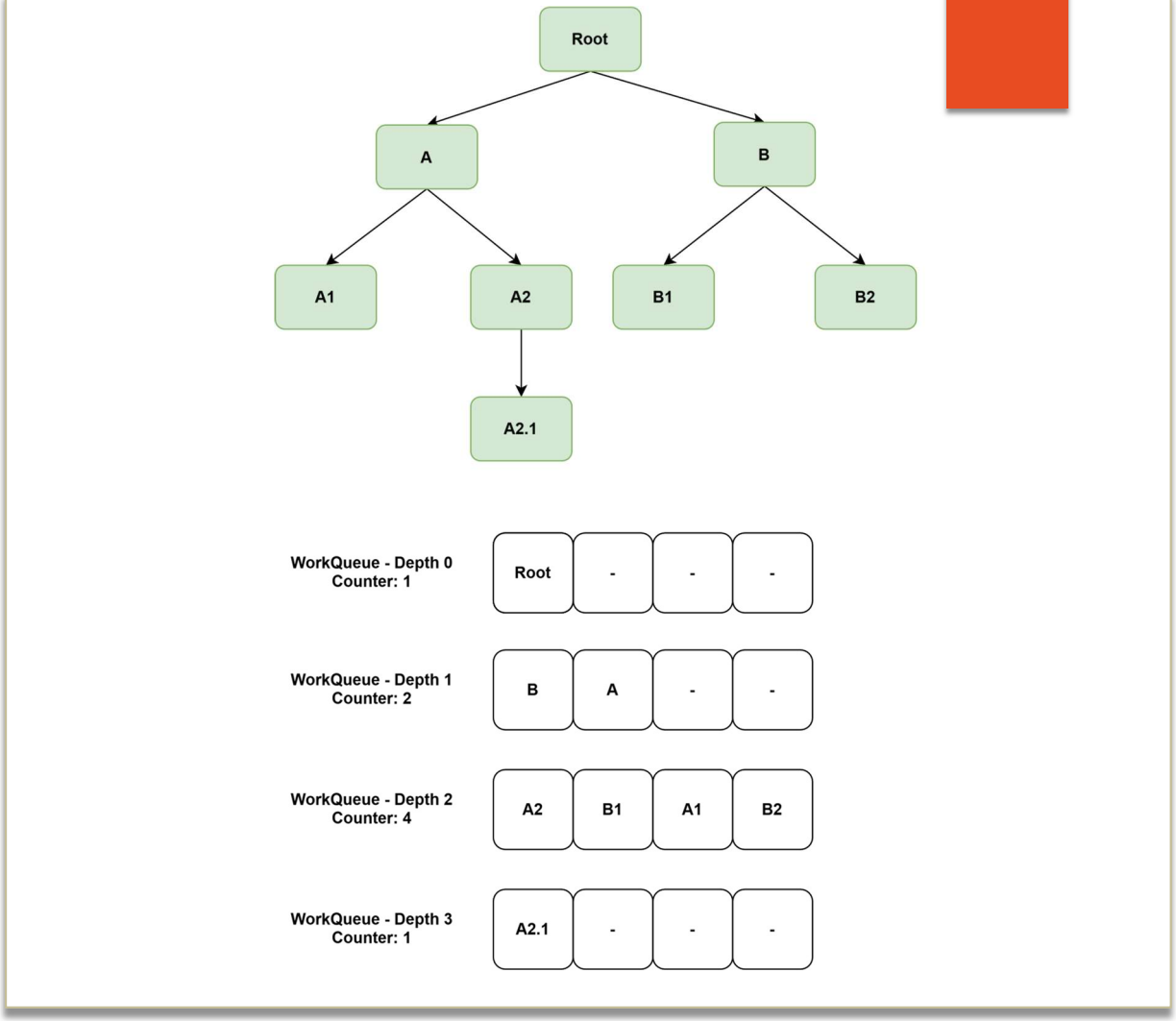


https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/Scene/Hierarchy/HierarchyWorkQueue.h

Data-Oriented Entity-Component-System XXVIII.

Waterfall Update

- Levels are processed sequentially from root to leaves.
- Entities within each level are processed fully in parallel.
- Each updated entitiy propagates update intent only to its direct children.
- Children are inserted into the next level's dirty queue.
- The process repeats level-by-level until the entire affected hierarchy is covered.
- After propagation, each level contains exactly the entities that must be updated.

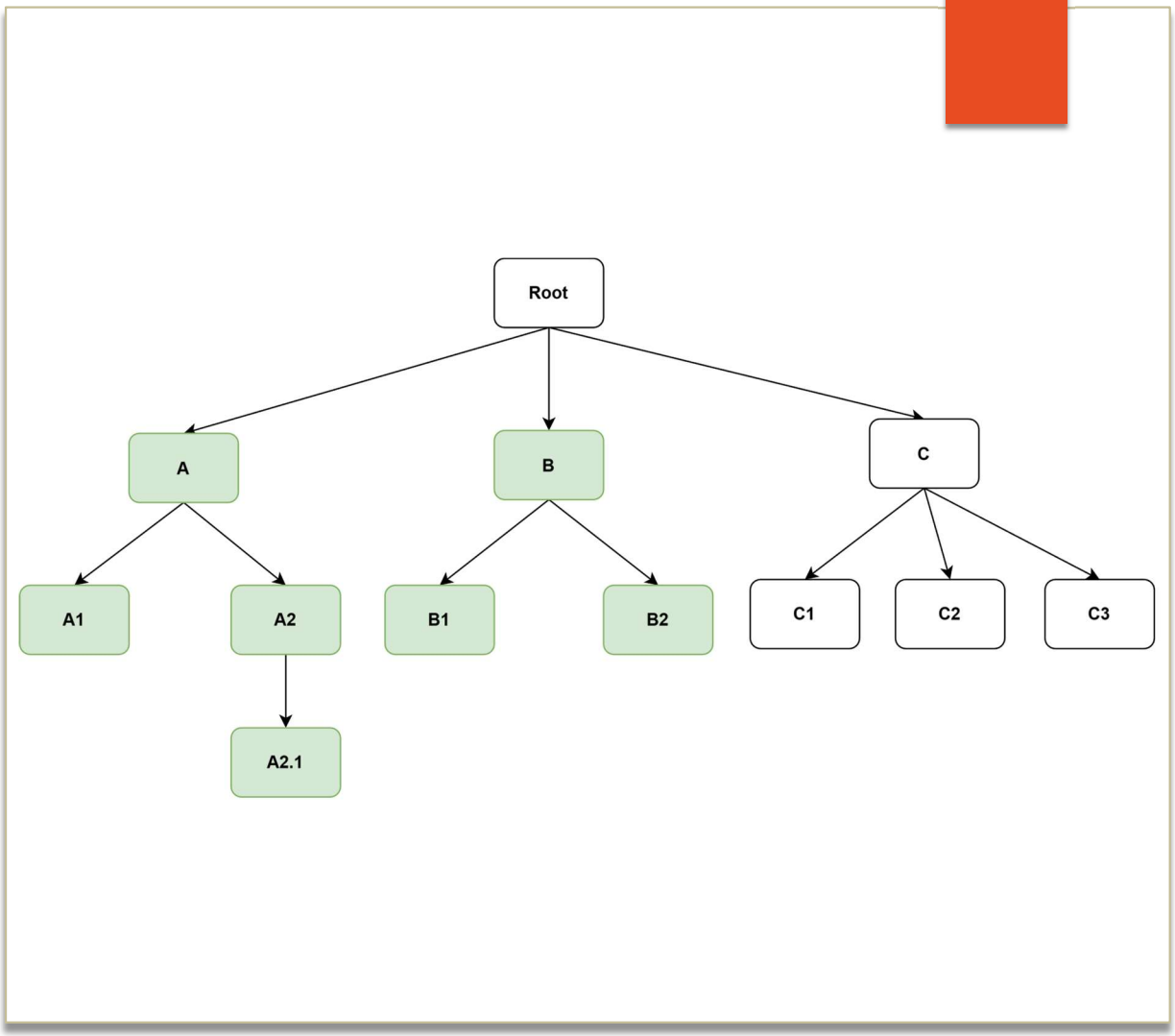


https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/System/Core/TransformSetupSystem.cpp

Data-Oriented Entity-Component-System XXIX.

Duplicate Prevention & Scalability

- Each entity can be inserted into work queue only once!!
- The **QUEUED_BIT** prevents duplicate insertions during propagation. (Remember dense bitset part of the pool!!)
- Unaffected subtrees are never traversed.
- Large static branches incur zero iteration cost unless an ancestor changes.
- **The algorithm scales with the size of the affected hierarchy, not the total scene!!**



Data-Oriented Entity-Component-System XXX.

GPU-Oriented Design

- Data-oriented layout maps naturally to GPU memory.
- Each component has a GPU-ready GLSL-compatible representation.
- Only modified components are written to a persistent mapped GPU memory in parallel.
- GPU component buffers mirror the CPU dense arrays.
- GPU sparse maps provide O(1) lookup from Entity ID to component data.
- **EntityIDs become the universal handle on both the CPU and GPU!**

```cpp
auto transformDataBuffer = componentBufferManager->GetComponentBuffer(BufferNames::TransformData, frameIndex);
if (!transformDataBuffer.buffer) return;
auto transformDataBufferHandler = static_cast<TransformComponentGPU*>(transformDataBuffer.buffer->Map());

auto processUpload = [transformPool, transformDataBuffer, transformDataBufferHandler](EntityID entity) {
    auto& transformComponent = transformPool->Get(entity);
    auto transformIndex = transformPool->GetMapping().Get(entity);

    if (transformDataBuffer.versions[transformIndex] != transformComponent.version)
    {
        transformDataBuffer.versions[transformIndex] = transformComponent.version;
        transformDataBufferHandler[transformIndex] = TransformComponentGPU(transformComponent);
    }
};
```

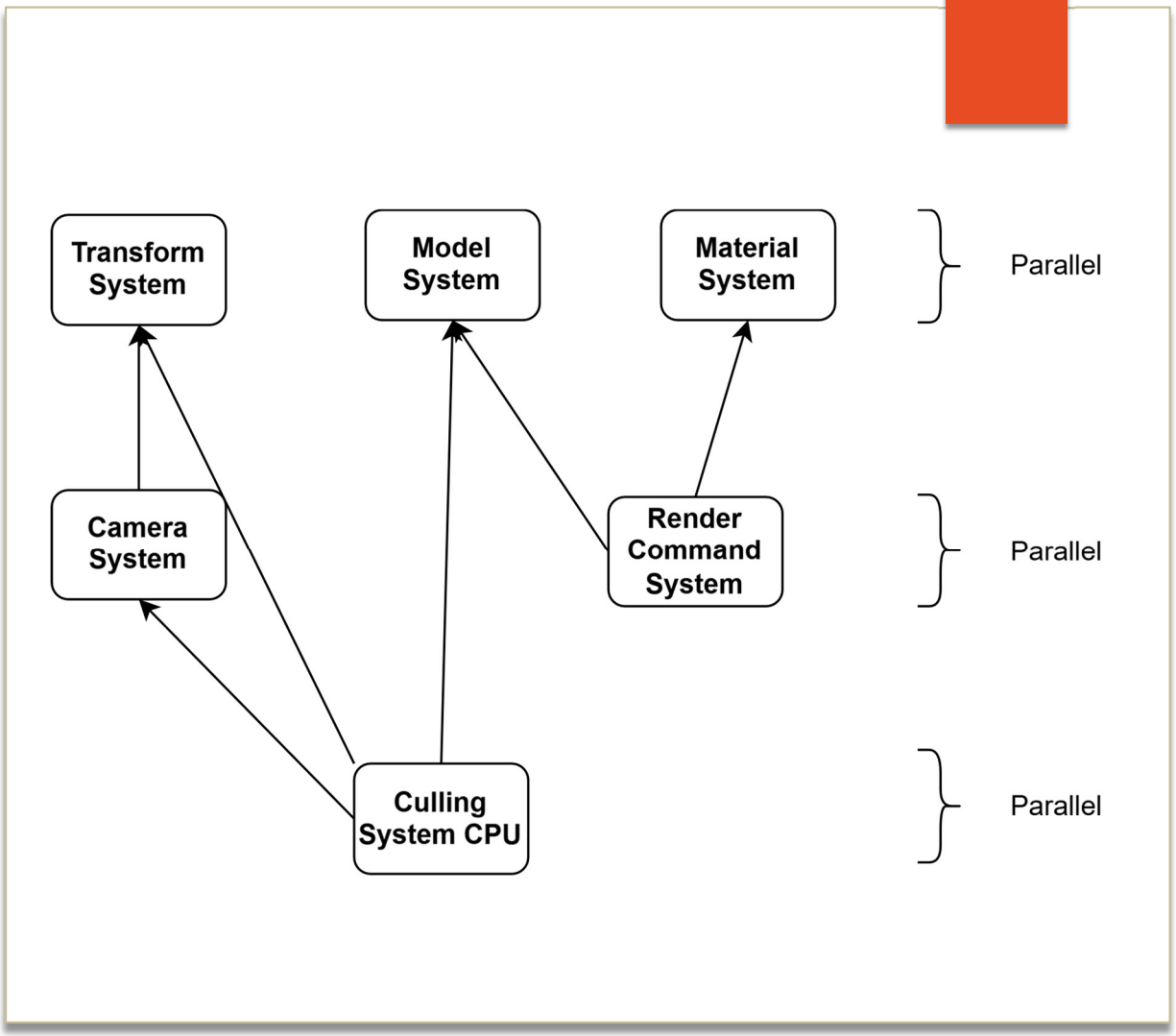
```glsl
struct TransformComponent {
    mat4 transform;
    mat4 transformIT;
};

layout(buffer_reference, std430) readonly restrict buffer TransformPool {
    TransformComponent data[];
};

#define GET_TRANSFORM(addr, idx) TransformPool(addr).data[idx]

//Shader Code:
uint transformDenseIndex = GET_SPARSE_INDEX(ctx.transformSparseMapBufferAddr, entityId);
TransformComponent transform = GET_TRANSFORM(ctx.transformBufferAddr, transformDenseIndex);
```

https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/System/Core/TransformSystem.cpp

Data-Oriented Entity-Component-System XXXI.

Task-Based Parallel Execution

- Every system is represented as a Taskflow task.
- Dependencies are generated automatically from component read/write access.
- Independent systems execute concurrently without manual scheduling.
- Each system internally parallelizes its workload across CPU cores.
- Taskflow provides efficient task scheduling, synchronization and work strealing!
- **I recommend using Taskflow, I really liked it personally ☺**



https://github.com/taskflow/taskflow

Synapse Engine:
Software Architecture

Created By: Péter Tamás

Github: https://github.com/TamasPetii/SynapseEngine

2026

Software Architecture I.

Main design Goals

- Writing code is easy, but writing maintainable code is really hard, especially in the age of AI.
- Synapse Engine consists of ~75.000 lines of code developed by a single engineer (me ☺).
- Without a well-designed architecture, maintaining and extending a project of this size becomes increasingly difficult.
- Every subsystem should answer one important question: **Can its implementation be replaced tomorrow without breaking the rest of the engine?**
- Modularity and scalability are essential for long-term maintainability: we already explored these principles in the ECS implementation through compile-time composition and interchangable Pool policies.
- Clear architecture reduces complexity and accelarates development, easier to test code and also developers can understand code much easier.
- The SOLID principles, especially Single Responsibility and Dependency Inversion play a key role in clean code.
- Layered architecture should exist at every scale, from the entire engine down to individual subsystems.
- Data-oriented design and object-oriented architecture are complementary, the goal is to combine the strengths of both.

Software Architecture II.

God Object Anti-Pattern

- The Model class is responsible for loading the assets (Assimp)
- The asset loading implementation is hard-coded, violation Dependency Inversion
- The class owns and manages all Cpu-side data.
- It also creates and manages GPU resources.
- Additional responsibilities include AABB generation, LOD management, instancing, and animation support.
- The class violates Single Responsibility principle and becomes difficult to extend, test and maintain.

Design Problems

- Replacing assimp with another importer required modifily the model class.
- Procedural meshes and imported meshes cannot share a common pipeline.
- **One class becomes the dependency hub of the entire rendering pipeline.**

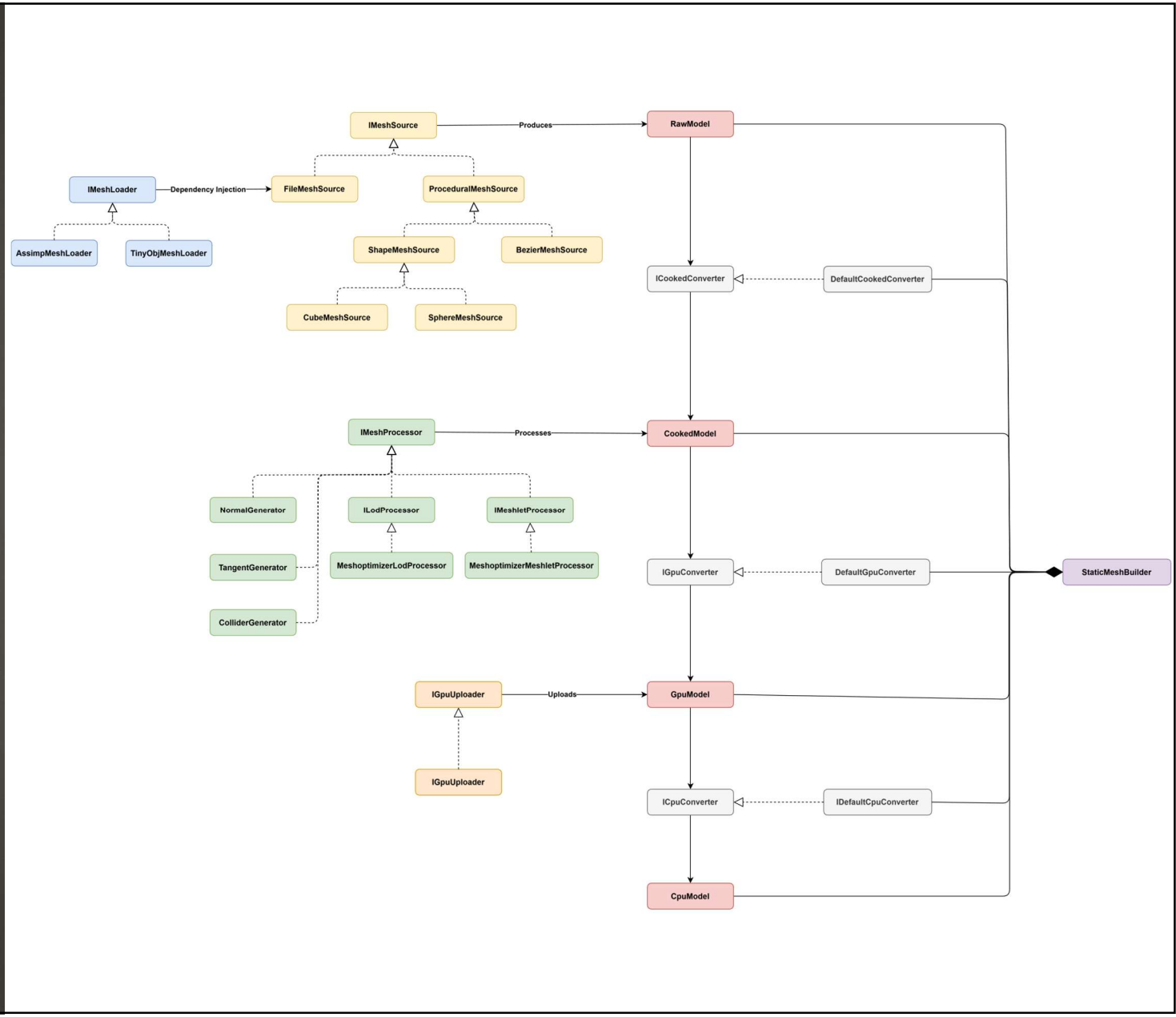
```
class ENGINE_API Model
{
public:
    Model();
    virtual ~Model();
    bool Load(const std::string& path);

    void Bind();
    void UnBind();

    void UpdateInstanceSsbo();
    void UpdateShadowInstanceSsbo();
private:
    void GenerateBuffers();
    void Process(aiNode* node, const aiScene* scene);
    void ProcessGeometry(aiMesh* mesh, const aiScene* scene, unsigned int& count);
private:
    std::vector<Vertex> m_Vertices;
    std::vector<glm::uvec4> m_Instances;
private: //GPU RESOURCES
    std::unique_ptr<VertexArrayGL> m_Vao;
    std::unique_ptr<BufferGL> m_Ibo;
    std::unique_ptr<BufferGL> m_Vbo;
    std::unique_ptr<ShaderStorageBufferGL> m_MaterialSsbo;
    std::unique_ptr<ShaderStorageBufferGL> m_InstanceSsbo;
};

bool Model::Load(const std::string& path)
{
    Assimp::Importer importer;
    const aiScene* scene = importer.ReadFile(path, aiProcess_Triangulate |
                                                   aiProcess_JoinIdenticalVertices |
                                                   aiProcess_CalcTangentSpace);

    if (!scene || scene->mFlags & AI_SCENE_FLAGS_INCOMPLETE || !scene->mRootNode)
        return false;

    hasAnimation = scene->HasAnimations();
    m_Path = path;
    m_Directory = path.substr(0, path.find_last_of('/'));

    Process(scene->mRootNode, scene);
    GenerateBuffers();
    GenerateObb();

    return true;
}
```

https://github.com/TamasPetii/DendriteEngine/blob/main/GameEngine/Engine/Model/Model.h

Software Architecture III.

Mesh Pipeline – Desing Goals

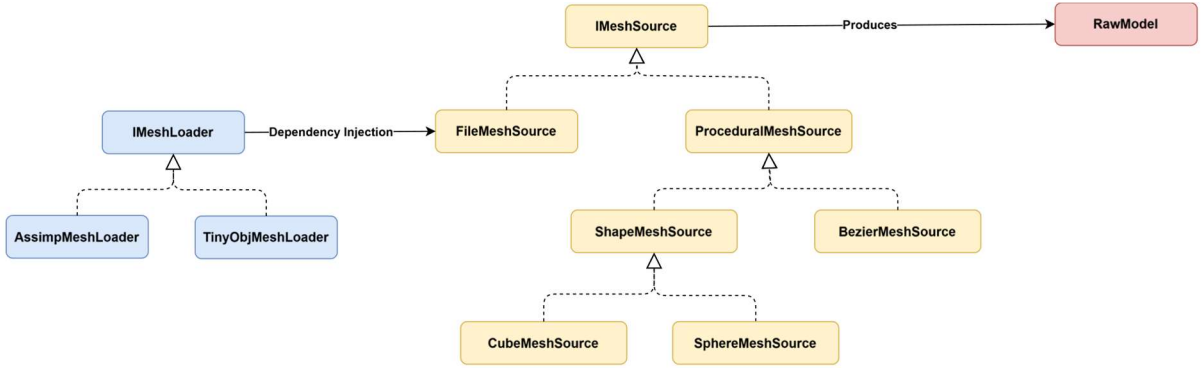
- Layered architecture with clear separation of responsibilities.
- Every stage follows the Single Responsibility Principle.
- Use Dependency Injection to eliminate hard-coded dependencies.
- Keep all data representations data-oriented and cache-friendly.
- Each representation stores only the data required at its stage.
- Procedural and imported assets follow the exact same pipeline.
- Compose the pipeline from small, reusable processing stages.
- **Keep the architecture modular, scalable, and easy to extend.**



Software Architecture IV.

Raw Model & Mesh Sources

- **RawModel**
  - The first representation in the mesh pipeline.
  - **Contains only raw geometry data** (vertices and indices)
  - No LODs, meshlets, colliders, GPU resources…
  - A RawModel consists of one or more RawMesh objects
- **Mesh Sources**
  - IMeshSource has a single responsibility: **produce a RawModel**
  - FileMeshSource generates a RawModel using and injected mesh loader (Dependency Injection)
  - Procedural mesh sources generate the same RawModel directly from code.
  - Every asset source produces and **identical representation, independent of its origin.**




```cpp
struct SYN_API RawMesh
{
    std::vector<uint32_t> indices;
    std::vector<Vertex> vertices;
    //...
};

struct SYN_API RawModel
{
    std::vector<RawMesh> meshes;
    //...
};

class SYN_API IMeshSource
{
public:
    virtual std::optional<RawModel> Produce() = 0;
};

class SYN_API IMeshLoader
{
public:
    virtual std::optional<RawModel> LoadFile(const std::filesystem::path& path) = 0;
    virtual std::vector<std::string> GetSupportedExtensions() const = 0;
};

class SYN_API FileMeshSource : public IMeshSource
{
public:
    FileMeshSource(const std::filesystem::path& path, IMeshLoader* loader);
    virtual std::optional<RawModel> Produce() override;
private:
    IMeshLoader* _loader;
    std::filesystem::path _path;
};
```

https://github.com/TamasPetii/SynapseEngine/tree/main/SynapseEngine/Engine/Mesh/Data/Raw

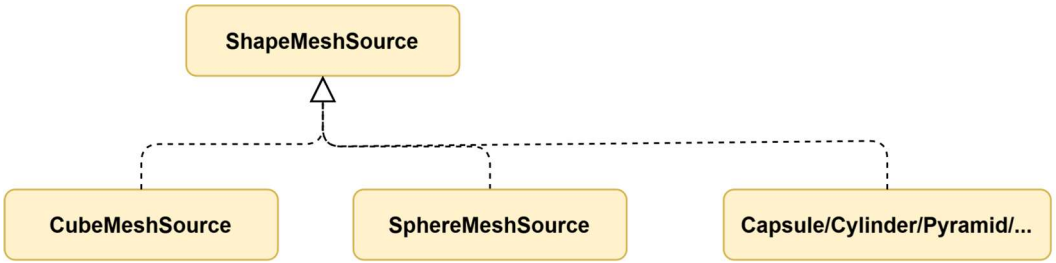
Software Architecture V.

Procedural Mesh Generation

- Implements IMeshSource to produce a standard RawModel.
- Uses the **Template Method Pattern** to define the common generation workflow.
- The overall algorithm is implemented once in the base class.
- Derived shapes override only geometry-specific steps.
- GeneratePositions, GenerateIndices, GenerateUVs, GenerateNormals are virtual functaions which are going to be overriden.

Benefits

- Eliminates duplicating mesh generation logics.
- Each shape implements what is unique.
- **Code is super clean**!



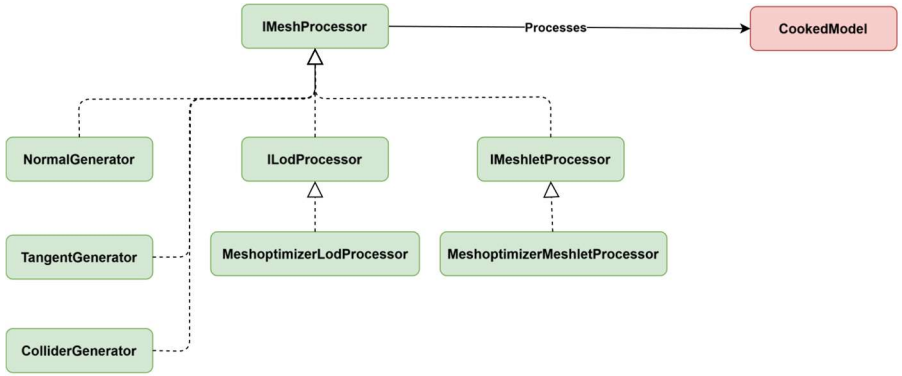

```cpp
std::optional<RawModel> ShapeMeshSource::Produce()
{
    std::vector<glm::vec3> positions;
    std::vector<glm::vec3> normals;
    std::vector<glm::vec2> uvs;

    std::vector<uint32_t> indices;

    GeneratePositions(positions);
    GenerateIndices(indices);

    uvs.resize(positions.size(), glm::vec2(0.0f));
    normals.resize(positions.size(), glm::vec3(0.0f, 1.0f, 0.0f));

    GenerateUVs(uvs);
    GenerateNormals(positions, indices, normals);

    RawModel model{};
    RawMesh mesh{};
    mesh.name = _name;
    mesh.materialIndex = 0;
    mesh.indices = std::move(indices);
    mesh.hasNormals = true;
    mesh.hasTangents = false;

    mesh.vertices.reserve(positions.size());
    for (size_t i = 0; i < positions.size(); ++i)
    {
        Vertex v{ v.position = positions[i], v.normal = normals[i], v.tangent = glm::vec3(0.0f), v.uv = uvs[i], };
        mesh.vertices.push_back(v);
    }

    model.meshes.push_back(std::move(mesh));

    //...
}
```

https://github.com/TamasPetii/SynapseEngine/tree/main/SynapseEngine/Engine/Mesh/Source/Procedural/Shape

Software Architecture VI.

Cooked Representation

- The second stage of the mesh pipline is the cooked stage.
- A dedicated converter transforms RawModel into a CookedModel.
- Extends the RawModel with engine-specific runtime data.
- Adds storage for colliders, LODs, meshlets, and other derived data.
- Initially, these structures are empty, only the data layout is defined.
- The mesh-based origanization from the RawModel is preserved, this representation becomes the foundation for all asset processing.
- Specialized processors populate the empty data (LODs, meshlets, colliders)




```
struct SYN_API CookedMeshLod {
    //Normal pipeline
    std::vector<uint32_t> indices;

    //Meshlet pipeline
    std::vector<CookedMeshlet> meshlets;
    std::vector<uint32_t> meshletVertexIndices;
    std::vector<uint8_t> meshletTriangleIndices;
};

struct SYN_API CookedMesh
{
    //...
    std::vector<Vertex> vertices;
    std::vector<CookedMeshLod> lods;
    CookedMeshCollisionData collider;
};

struct SYN_API CookedModel
{
    CookedMeshCollisionData globalCollider;
    std::vector<CookedMesh> meshes;
    //...
};
```

https://github.com/TamasPetii/SynapseEngine/tree/main/SynapseEngine/Engine/Mesh/Data/Cooked

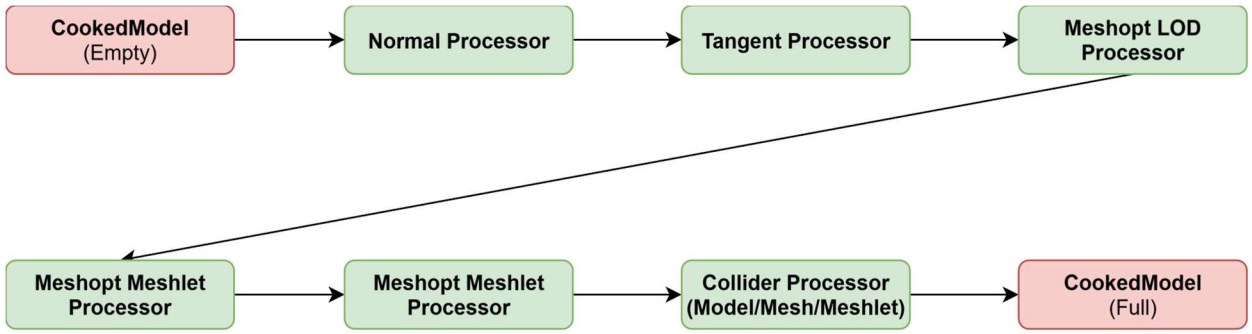
Software Architecture VII.

Mesh Processors

- Each processor has a single responsibility.
- Processors operate on the CookedModel representation.
- Each stage enriches the model with additional derived data.
- Processing is performed independenty for each mesh.
- The processing order is defined by the pipeline.
- Processors can be added, removed or replaced without affecting the rest of the system.

Example Processors

- **Lod Processor**: Generated simplified lod index buffers using Meshoptimizer.
- **Meshlet Processor**: Generates meshlet vertex indices and triangle indices using Meshoptimizer.
- **Collider Processor**: Generates collision geometry.




```
class SYN_API IMeshProcessor
{
public:
    virtual ~IMeshProcessor() = default;
    virtual void Process(CookedModel& cookedModel) = 0;
};
```

https://github.com/TamasPetii/SynapseEngine/tree/main/SynapseEngine/Engine/Mesh/Processor/MeshProcessor

Software Architecture VIII.

Parallel Mesh Processing

- The CookedModel stores geometry on per-mesh basis.
- Each mesh can be processed completely independently.
- This enabled maximum CPU parallelism during asset processing.
- Every processor follows the same parallel execution model.
- Taskflow distributes mesh-processing jobs across the shared engine thread pool.

Collider Processing Example

- Every mesh is transform into its local node space.
- Bounding volumes (**AABB and Sphere**) are generated for each mesh.
- Meshlet bounds are computed in parallel for every meshlet.
- The final global bounds are produced with a parallel **Map-Reduce step**.

```
void ColliderProcessor::Process(CookedModel& cookedModel)
{
    tf::Taskflow taskflow;
    tf::GuidedPartitioner partitioner(1);

    taskflow.for_each(cookedModel.meshes.begin(), cookedModel.meshes.end(),
        [this](CookedMesh& mesh) {
        if (!mesh.vertices.empty()) {
            ComputeMeshLocalBounds(mesh);
        }
        },
        partitioner
    );

    ServiceLocator::GetTaskExecutor()->run(taskflow).wait();
    taskflow.clear();

    std::vector<MeshletJob> meshletJobs;
    for (auto& mesh : cookedModel.meshes) {
        for (auto& lod : mesh.lods) {
            for (auto& meshlet : lod.meshlets) {
                meshletJobs.push_back({ &mesh, &lod, &meshlet });
            }
        }
    }

    taskflow.for_each(meshletJobs.begin(), meshletJobs.end(),
        [this](MeshletJob& job) {
        ComputeMeshletBoundsJob(*job.mesh, *job.lod, *job.meshlet);
        },
        partitioner
    );

    ServiceLocator::GetTaskExecutor()->run(taskflow).wait();
    taskflow.clear();

    // (Map-Reduce)
    ComputeGlobalBounds(cookedModel, taskflow);
}
```

https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/Mesh/Processor/MeshProcessor/Geometry/ColliderProcessor.cpp

Software Architecture IX.

GPU Model Represetion

- The final representation is optimized for GPU memory and rendering.
- All mesh data is flattened into contiguous batched arrays.
- Each resource type has its own dedicated buffer (position, attributes, indices, meshlets, colliders)
- **Everything is pure data matching with GLSL shader structure layout.**

GPU Model Converter and Uploader

- The converter converts the cooked representation into a batched flat GPU representation.
- Generates descriptor structures describing the location of each mesh within the batched arrays.
- A staging buffer is created and all data copied once into that and uploaded as a large contiguous transfer.
- **The GPU representation will be explained later, since all of the shaders, culling and dynamic LOD builds on it.**

```
struct SYN_API GpuVertexPosition
{
    glm::vec3 position;
    uint32_t packedIndex; // nodeIndex, meshIndex

    void SetMeshAndNodeIndex(uint16_t meshIndex, uint16_t nodeIndex);
};

struct SYN_API GpuVertexAttributes {
    glm::vec3 normal;
    float uv_x;
    glm::vec3 tangent;
    float uv_y;
};

struct SYN_API GpuVertexData
{
    std::vector<GpuVertexPosition> vertexPositions;
    std::vector<GpuVertexAttributes> vertexAttributes;
};

struct SYN_API GpuIndexedDrawData
{
    std::vector<uint32_t> indices;

    //New: [(Mesh0_Lod0, Mesh0_Lod1, Mesh0_Lod2, Mesh0_Lod3), (Mesh1_Lod0, ...)]
    std::vector<GpuMeshDescriptor> meshDescriptors;

    std::vector<GpuMeshLodDescriptor> lodDescriptors;

    std::vector<GpuMeshCollider> meshColliders;
};

struct SYN_API GpuBatchedModel
{
    GpuVertexData vertexData;
    GpuIndexedDrawData indexedData;
    GpuMeshletDrawData meshletData;
    //...
};
```

https://github.com/TamasPetii/SynapseEngine/tree/main/SynapseEngine/Engine/Mesh/Data/Gpu

Software Architecture X.

CPU Runtime Representation

- This is the last representational layer, which contains only the data required by CPU-side systems.
- Retains descriptors, colliders, draw descriptors and runtime metadata.
- Vertex and Index buffers are preserved for physics simulation triangle collider.

StaticMeshBuilder – Asset Pipeline Orchestration

- Coordinates the complete asset processing pipeline.
- Every stage transforms the model into a more specialized representation.
- The builder itself contains no processing logic, it only orchestrates the pipeline. Everyting is dependency injected.
- GPU upload is managed by the ModelManager, not the builder since models are loaded async and GPU upload require syncronization, fences and graphics API commands!

```
struct SYN_API CpuModelData
{
    uint32_t globalVertexCount = 0;
    uint32_t globalIndexCount = 0;
    uint32_t globalMeshCount = 0;
    uint32_t globalAverageLodIndexCount = 0;

    GpuMeshCollider globalCollider;
    std::vector<GpuMeshCollider> meshColliders;
    std::vector<GpuMeshDescriptor> meshDescriptors;
    std::vector<GpuMeshletDrawDescriptor> meshletDrawDescriptors;
    std::vector<GpuMeshLodDescriptor> lodDescriptors;
    std::vector<MeshInstanceDescriptor> meshNodeDescriptors;

    std::vector<glm::vec3> vertices;
    std::vector<uint32_t> indices;
};

std::shared_ptr<StaticMesh> StaticMeshBuilder::BuildFromSource(IMeshSource& source)
{
    auto rawModelOpt = source.Produce();

    if (!rawModelOpt)
        return nullptr;

    auto staticMesh = std::make_shared<StaticMesh>();
    staticMesh->transientCpuData = std::make_unique<CookedModel>();
    staticMesh->transientGpuData = std::make_unique<GpuBatchedModel>();

    *(staticMesh->transientCpuData) = _cooker->Cook(std::move(rawModelOpt).value());
    _meshPipeline->Run(*(staticMesh->transientCpuData));
    *(staticMesh->transientGpuData) = _converter->Convert(*(staticMesh->transientCpuData));

    _extractor->Extract(*(staticMesh->transientGpuData), staticMesh->cpuData);
    _cpuModelPipeline->Run(staticMesh->cpuData);

    return staticMesh;
}
```

https://github.com/TamasPetii/SynapseEngine/tree/main/SynapseEngine/Engine/Mesh/Builder

Software Architecture XI.

Model Manager

- Central entry point for all model loading requests.
- All assets are loading asynchronously using Taskflow futures and worker threads.
- Schedules GPU uploads while handling synchronization.
- All loaded models are stored in a compact contiguous array.
- Data-oriented storage forms a foundation for GPU-driven rendering, since all GPU resources (vertex, index buffers…) are accessable via centralized buffer addresses.
- All model resosurces become globally accessable to every shaders.

```cpp
template <typename TResource>
void BaseResourceManager<TResource>::Load() {
    EntryType newEntry{};
    newEntry.path = key;
    newEntry.state = ResourceState::LoadingCPU;
    newEntry.isSyncLoad = !isAsync;

    if (isAsync) {
        auto executor = ServiceLocator::Get<tf::Executor>();
        newEntry.cpuFuture = executor->async(std::move(task));
        _entries.push_back(std::move(newEntry));

        OnEntryCreated(newId);
    }

    //...
}

template <typename TResource>
void BaseResourceManager<TResource>::Update() {
    std::lock_guard lock(_mutex);

    for (auto& entry : _entries) {
        if (entry.state == ResourceState::LoadingCPU) {
            if (entry.cpuFuture.valid() && entry.cpuFuture.wait_for(std::chrono::seconds(0)) == std::future_status::ready) {
                auto result = entry.cpuFuture.get();
                if (result != nullptr) {
                    entry.resource = result;
                    entry.state = ResourceState::UploadingGPU;
                    StartGpuUpload(entry);
                }
                else {
                    entry.state = ResourceState::Failed;
                }
            }
        }
    }
}
```

https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/Mesh/ModelManager.cpp

Software Architecture XII.

Mesh Pipeline Summary

- Layered architecture with dedicated Raw, Cooked, Gpu, Cpu representations, each stores only the data required for its specific stage.
- Processing is performed through a modular pipeline of independent processors.
- The entire asset pipeline is highly parallel and scales naturally using Taskflow.
- Dependency Injection enables interchangeable loaders and asset sources.
- Single Responsibility is appliead throughout the pipeline, from loaders to processors and converters.
- Procedural and imported assets follow the exact same processing pipeline.
- The architecture is modular, scalable and straightforward to extend with new processors or asset formats.
- The same architectural principles are reused across the engine for animation, images.
- **Animation Pipeline**: The architecture is 100% the same, supporting both imported animations and procedurally generated data.
- **Image Pipeline**: The architecture is 100% the same, supporting multiple image loaders (GLI, SVG, STB) alongside procedural image generations (noise, lookup table, generated textures).

Software Architecture XIII.

Service Locator vs Singleton

- **Why not to use Singleton?**
  - Singletons hide object lifetime and ownership.
  - Implementation becomes hard-coded and difficult to replace.
  - Strong coupling reduces modularity and testability.
  - Mock implementations cannot be injected easily.
- **Service Locator Pros**
  - The Engine owns and manages the lifetime of all services.
  - Services are created, registered and destroyed explicitly.
  - Stores only interface pointers, concrete implementations can be replaced transparently.
  - Supports polymorphism, dependency inversion and uint testing.

```
class SYN_API ServiceLocator {
public:
    ServiceLocator() = delete;
    ServiceLocator(const ServiceLocator&) = delete;
    ServiceLocator& operator=(const ServiceLocator&) = delete;

    template <typename T>
    static void Provide(typename ServiceTrait<T>::ProvideType service);

    template <typename T>
    static typename ServiceTrait<T>::ReturnType Get();
private:
    static IGpuProfiler* _gpuProfiler;
    static ICpuProfiler* _cpuProfiler;
};

template <> SYN_API void ServiceLocator::Provide<IGpuProfiler>(IGpuProfiler*);
template <> SYN_API void ServiceLocator::Provide<ICpuProfiler>(ICpuProfiler*);

template <> SYN_API IGpuProfiler* ServiceLocator::Get<IGpuProfiler>();
template <> SYN_API ICpuProfiler* ServiceLocator::Get<ICpuProfiler>();

//Engine
void Engine::InitProfilers()
{
    _gpuProfiler = std::make_unique<DefaultGpuProfiler>(_frameContext.framesInFlight, timestampPeriod);
    ServiceLocator::Provide<IGpuProfiler>(_gpuProfiler.get());

    _cpuProfiler = std::make_unique<DefaultCpuProfiler>(_frameContext.framesInFlight);
    ServiceLocator::Provide<ICpuProfiler>(_cpuProfiler.get());
}
```

https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/ServiceLocator.h

Software Architecture XIV.

Callback-Based Communication

- Managers never store references to other managers.
- Cross-system operations are injected as callbacks or lambdas.
- The caller defines what should happend, the manager only decides when to invoke it.
- Eliminates unnecessary dependencies between subsystems.
- Support Dependency Injection at a function level.

Examples

- ModelManager requests material loading through and injected callback instead of directly accessing the MaterialManager.
- The Vulkan renderer invokes the ImGui rendering callback without depending on the editor or UI framework.
- The Engine remains completely independent from the Presentation Layers.

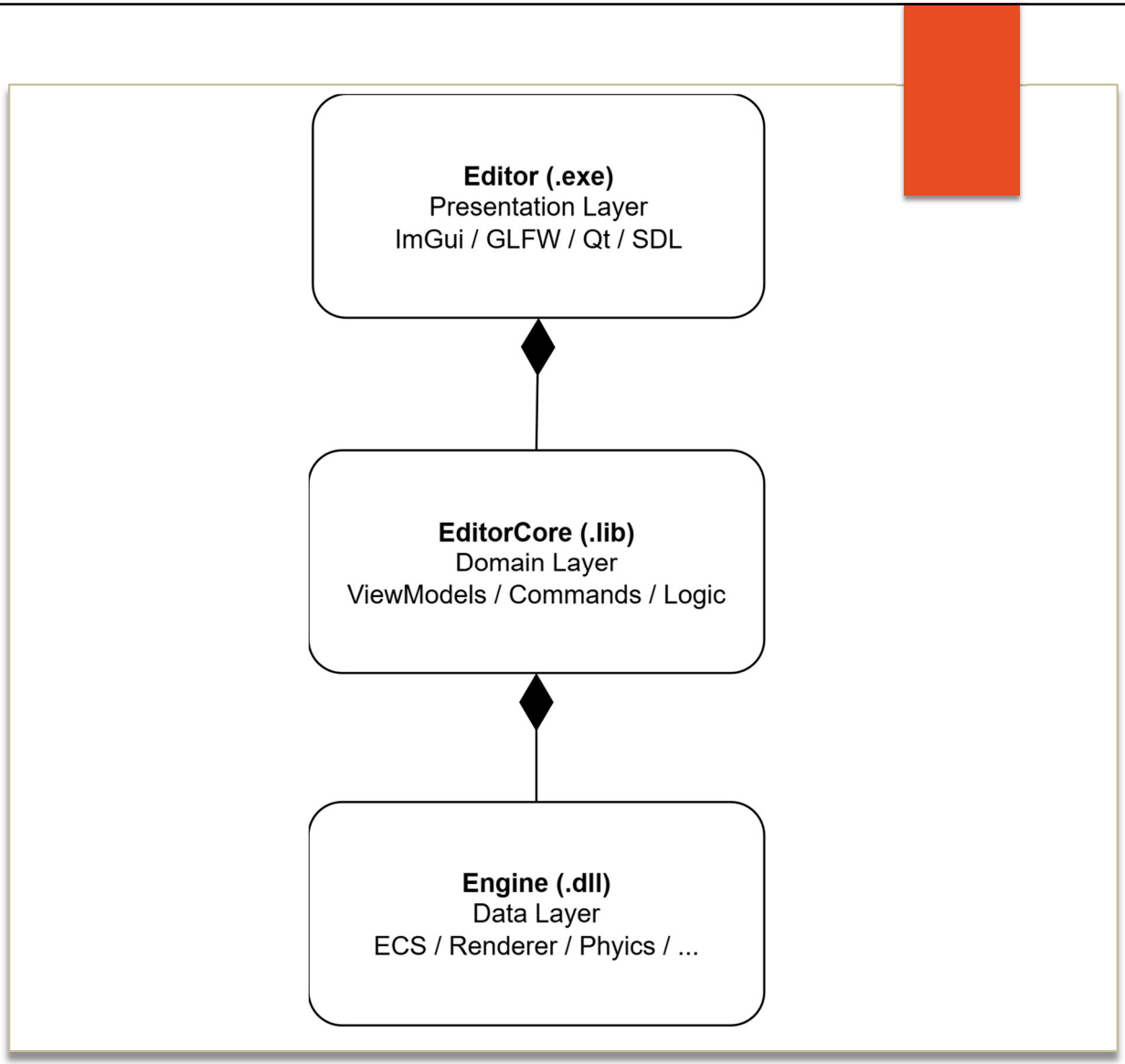
```cpp
struct SYN_API EngineInitParams {
    std::function<void(VkInstance, VkSurfaceKHR*)> createSurfaceCallback;
    std::function<std::pair<uint32_t, uint32_t>()> getWindowExtentCallback;
    std::function<std::vector<const char*>()> getSurfaceExtensionsCallback;
    std::function<void(VkCommandBuffer)> onRenderGuiCallback;
    std::function<void(uint32_t)> onGuiFlushCallback;
};

void Synapse::OnInit() {
    Syn::EngineInitParams params;

    params.createSurfaceCallback = [&](VkInstance instance, VkSurfaceKHR* surface) {
        GetWindow().CreateSurface(instance, surface);
    };

    params.getSurfaceExtensionsCallback = [&]() {
        return GetWindow().GetRequiredExtensions();
    };

    params.getWindowExtentCallback = [&]() {
        return GetWindow().GetSize();
    };

    params.onRenderGuiCallback = [&](VkCommandBuffer cmd) {
        if (_guiManager) {
            _guiManager->Render(cmd);
        }
    };

    params.onGuiFlushCallback = [&](uint32_t frameIndex) {
        _guiManager->GetTextureManager()->FlushQueue(frameIndex);
    };

    _engine = std::make_unique<Syn::Engine>(params);
}
```

https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Editor/Synapse.cpp

Software Architecture XV.

Layered Engine Architecture

- **Engine (.dll) – Data Layer**
  - Containts the core engine implementation.
  - ECS, Renderer, Asset System, Physics, Algorithms, Shaders, Manager
  - **Completely independent form the Editor!!**
- **EditorCore (.lib) – Domain Layer**
  - Containts the Editor's business logic and application state.
  - Implements ViewModels and Editor algorithms, also independent of any UI framework!
- **Editor (.exe) – Presentation Layer**
  - Application entry point and main loop.
  - Window management, user interface.
  - Contains only presentation logic.



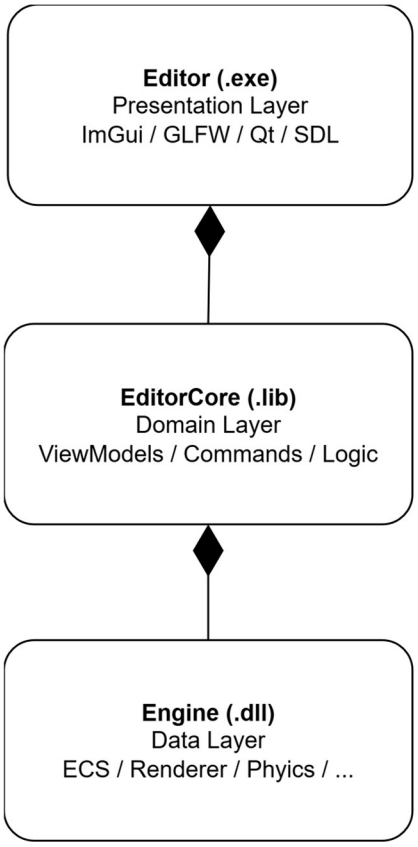
Software Architecture XVI.

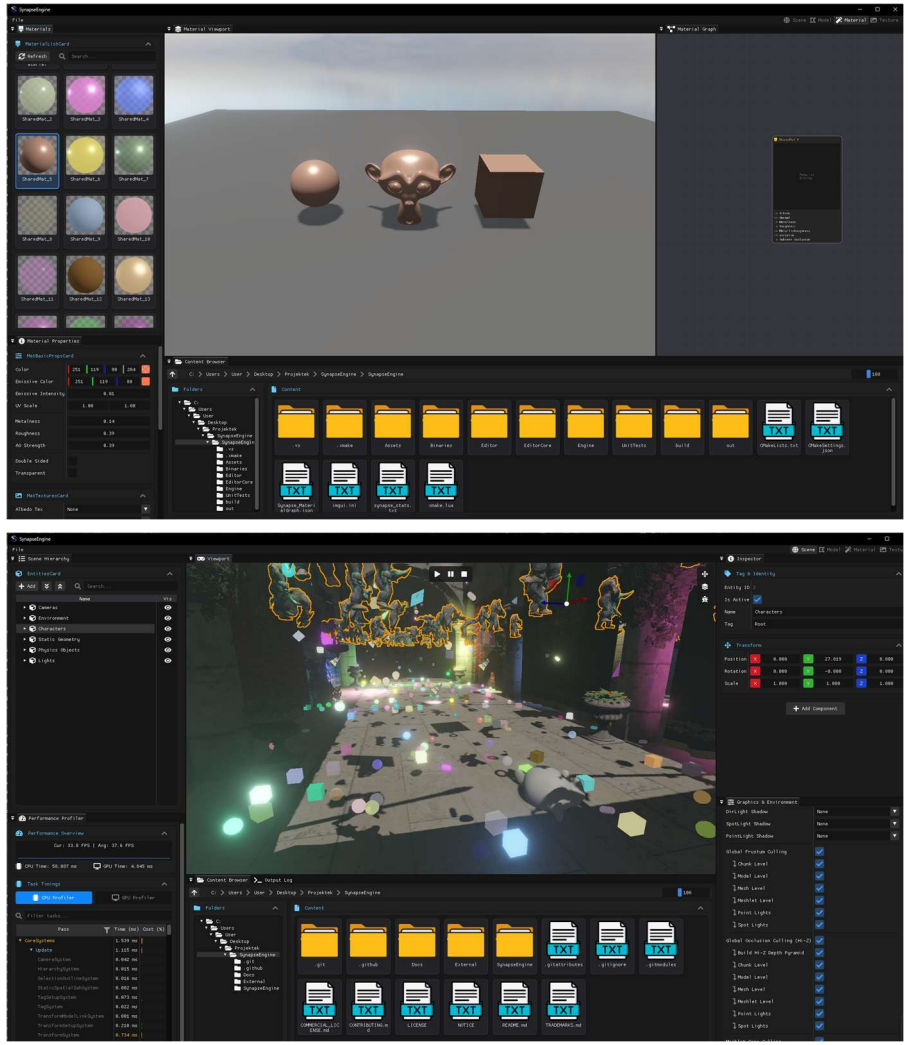
Editor Architecture Goals

- The Engine should remain completely independent of any user interface.
- Rendering, Ecs, Asset management and simulation must work without an editor.
- UI frameworks should be replaceable without modifiying engine logic.
- UI business logic should not depend on ImGui, Qt, Glfw or any presentation framework.
- Editor state and user workflows belong outside the engine.
- Layered architecture improves maintainability, extensibility.

Design Goal

- **The Engine should behave like a standalone application or offline renderer, the editor is a simply one possible user interface built on top of it.**

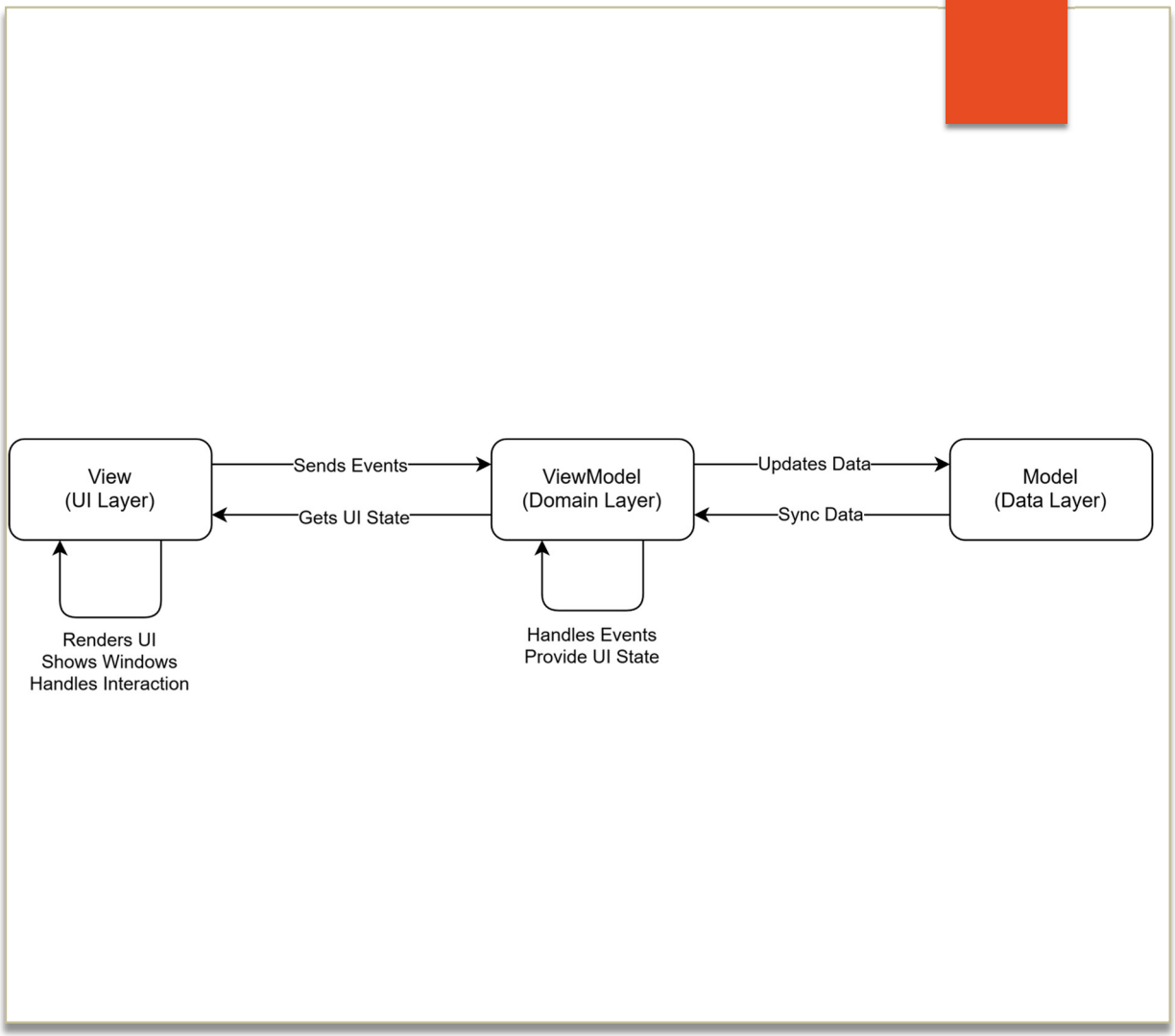




Software Architecture XVII.

Model-View-Intent (MVI) Architecture

- **Three-Layer Architecture**
  - **Data Layer (Engine):** The single source of truth containing all runtime engine data.
  - **Domain Layer (EditorCore):** Business logic, editors tate, viewmodels and intents.
  - **UI Layer (Editor):** ImGui windows responsible onyl for rendering the user interface and sending intents.
- **Core Idea**
  - MVI is similar to MVVM, but communication is intent-driven rather than direct.
  - The View never modifies application state directly.
  - Every user interaction is wrapped into an Intent object describing the requested action.
  - ViewModels receive Intents, execute the required business logic, and update state.
  - The UI simply reflects the current ViewModel state on the next frame.



Software Architecture XVIII.

Data-Layer, Engine components

- Transform Component represents the ground truth of an entity's transform inside the Engine.
- Stores only the runtime data required by engine systems.
- Completely independent form the editor and any UI framework.
- Optimized for runtime execution rather than user interaction.
- All editor modifications eventually propagate to this component.

Key Idea

- **The engine owns the data, the editor never modifies it directly, it communicates through the Domain Layer.**

```
struct SYN_API TransformComponent : public Component
{
    TransformComponent();

    glm::vec3 translation;
    glm::vec3 rotation;
    glm::vec3 scale;
    glm::mat4 transform;
    glm::mat4 transformIT;
};
```

https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/Component/Core/TransformComponent.h

Software Architecture XIX.

Domain Layer – State & Intents

- **State:**
  - Represents the data currently displayed by the View.
  - Contains only the information required by the user inferface.
  - Completely independent form the Engine's internal component representation.
  - Synchronized with the Engine at the beginning of each frame.
- **Intents:**
  - Represents user actions of direct function calls.
  - Each intent describes what the user wants to change, not how it should be changed.
  - Immutable message objects passed from the View to the ViewModel
  - Carry all information required to execute the requested operation.
  - Multiple Intent types are grouped into a single std::variant for type-safe dispatching

Key Idea

- **The UI never modifies the engine directly. It only sends Intents and displays the current State.**

```
struct SetPositionIntent {
        glm::vec3 newPosition;
        bool isDragging;
    };

    struct SetRotationIntent {
        glm::vec3 newRotation;
        bool isDragging;
    };

    struct SetScaleIntent {
        glm::vec3 newScale;
        bool isDragging;
    };

    using TransformIntent = std::variant<
        SetPositionIntent,
        SetRotationIntent,
        SetScaleIntent
    >;

     struct TransformState
    {
        glm::vec3 position{ 0.0f, 0.0f, 0.0f };
        glm::vec3 rotation{ 0.0f, 0.0f, 0.0f };
        glm::vec3 scale{ 1.0f, 1.0f, 1.0f };
    };
```

https://github.com/TamasPetii/SynapseEngine/tree/main/SynapseEngine/EditorCore/ViewModels/SceneWorkspace/Component/Core/Transform

Software Architecture XX.

Domain Layer – ViewModels

- Acts as the coordinator between View and the Engine.
- Owns the current UI state displayed by the View.
- Receives Intents and dispatches them to the appropriate command handlers.
- Communicates with the Engine exlusively through API interfaces.
- Synchronizes its state with the Engine at the beginning of every frame.
- Uses std::visit for compile-time type-safe intent dispatching.
- Avoids virtual inheritance and dynamic casting.
- Adding new user actions only requires defining a new Intent and its handler!

```cpp
class TransformViewModel : public IViewModel<TransformState, TransformIntent> {
public:
    TransformViewModel(ITransformApi* transformApi);
    ~TransformViewModel() override = default;

    const TransformState& GetState() const override;
    void SyncWithEngine() override;
    void Dispatch(const TransformIntent& intent) override;

private:
    void HandleSetPosition(const SetPositionIntent& intent);
    void HandleSetRotation(const SetRotationIntent& intent);
    void HandleSetScale(const SetScaleIntent& intent);
private:
    ITransformApi* _transformApi = nullptr;
    TransformState _state;
};

void TransformViewModel::Dispatch(const TransformIntent& intent) {
    std::visit([this](auto&& arg) {
        using T = std::decay_t<decltype(arg)>;

        if constexpr (std::is_same_v<T, SetPositionIntent>)
            HandleSetPosition(arg);
        else if constexpr (std::is_same_v<T, SetRotationIntent>)
            HandleSetRotation(arg);
        else if constexpr (std::is_same_v<T, SetScaleIntent>)
            HandleSetScale(arg);
        }, intent);
}

void TransformViewModel::SyncWithEngine() {
    if (!_selectionApi || !_transformApi) return;
    EntityID activeEntity = _selectionApi->GetSelectedEntity();

    if (activeEntity != NULL_ENTITY) {
        if (!_positionDrag.IsDragging()) _state.position = _transformApi->GetEntityPosition(activeEntity);
        if (!_rotationDrag.IsDragging()) _state.rotation = _transformApi->GetEntityRotation(activeEntity);
        if (!_scaleDrag.IsDragging())    _state.scale = _transformApi->GetEntityScale(activeEntity);
    }
}
```

https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/EditorCore/ViewModels/SceneWorkspace/Component/Core/Transform/TransformViewModel.cpp

Software Architecture XXI.

UI Layer – View

- Responsible only for rendering the user interface.
- Contains no business logic and no direct Engine access.
- Reads the current State form the ViewModel and converts user interactions into intent objects.
- Forwards every Intent to the ViewModels for processing.
- The view remains completely stateless apart form local UI-only state.
- Different user interactions can produce different Intent types, for example **dragging a value and releasing** it may generate a separate Intents
- This enables features such as Undo/Redo, command history, validation or even deferred execution without modifiying the View.

```cpp
class TransformView : public IView<TransformViewModel> {
public:
    void Draw(TransformViewModel& vm) override;
private:
    bool _isCardOpen = true;
};

void TransformView::Draw(TransformViewModel& vm) {
    constexpr const char* CardTransformTitle = "Transform";

    if (Syn::UI::BeginCard(CardTransformTitle, SYN_ICON_ARROWS_ALT, _isCardOpen)) {

        TransformState tState = vm.GetState();
        bool changed = false;
        bool deactivated = false;

        if (Syn::UI::BeginPropertyGrid("TransformGrid")) {

            Syn::UI::BeginProperty("Position");
            changed = Syn::UI::DrawVec3Control("##Pos", tState.position, 0.0f, deactivated);
            if (changed || deactivated) {
                vm.Dispatch(SetPositionIntent{ tState.position, !deactivated });
            }

            Syn::UI::BeginProperty("Rotation");
            changed = Syn::UI::DrawVec3Control("##Rot", tState.rotation, 0.0f, deactivated);
            if (changed || deactivated) {
                vm.Dispatch(SetRotationIntent{ tState.rotation, !deactivated });
            }

            Syn::UI::BeginProperty("Scale");
            changed = Syn::UI::DrawVec3Control("##Scale", tState.scale, 1.0f, deactivated);
            if (changed || deactivated) {
                vm.Dispatch(SetScaleIntent{ tState.scale, !deactivated });
            }

            Syn::UI::EndPropertyGrid();
        }
    }

    Syn::UI::EndCard();
}
```

https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Editor/Workspace/SceneWorkspace/View/Component/Core/TransformView.cpp

Software Architecture XXII.

Domain Layer – Api interfaces

- Provide the only communication channel between the Domain Layer and the Engine.
- Using Api interfaces in ViewModels allows the Domain Layer to depend only on abstractions not implementations!. (Dependency Inversion Priciple)

UI Layer – Api implementations

- Implement api interface in the Editor layer where both the Engine and the UI are accessible.
- Can coordinate Engine and UI functionality when required (registering Vulkan images as ImGui texture descriptors)
- Encapsulate all engine-specific operations behind a clean API interface.
- Automatically perform engine tasks such as updatying dirty flags, scheduling refreshes.
- **API objects can be stored, passed around and reused by higher-level systems such as Undo/Redo commands!**

```cpp
//Api Interface - Domain Layer
class ITransformApi : public IApi {
public:
    virtual ~ITransformApi() = default;

    virtual glm::vec3 GetEntityPosition(EntityID entity) const = 0;
    virtual glm::vec3 GetEntityRotation(EntityID entity) const = 0;
    virtual glm::vec3 GetEntityScale(EntityID entity) const = 0;

    virtual void SetEntityPosition(EntityID entity, const glm::vec3& position) = 0;
    virtual void SetEntityRotation(EntityID entity, const glm::vec3& rotation) = 0;
    virtual void SetEntityScale(EntityID entity, const glm::vec3& scale) = 0;

    virtual glm::mat4 GetEntityWorldMatrix(EntityID entity) const = 0;
};

//Api implementation - Editor Layer
glm::vec3 TransformApiImpl::GetEntityPosition(EntityID entity) const {
    return EditorApiUtils::ReadComponent<TransformComponent>(_sceneManager, entity, [](const auto& c) { return c.translation; }, glm::vec3(0.0f));
}

void TransformApiImpl::SetEntityPosition(EntityID entity, const glm::vec3& position) {
    EditorApiUtils::ModifyComponent<TransformComponent>(_sceneManager, entity, [&](auto& c, auto pool) {
        c.translation = position;
        pool->template SetBit<TRANSFORM_POS_CHANGED>(entity);
    });
}

template <typename T, typename Getter, typename ReturnType>
    static ReturnType ReadComponent(SceneManager* sm, EntityID entity, Getter&& getter, ReturnType defaultValue) {
        auto scene = sm->GetActiveScene();
        if (!scene) return defaultValue;

        auto registry = scene->GetRegistry();
        if (!registry || !registry->HasComponent<T>(entity)) return defaultValue;

        return getter(registry->GetComponent<T>(entity));
    }

template <typename T, typename Modifier>
static void ModifyComponent(SceneManager* sm, EntityID entity, Modifier&& modifier) {
    auto scene = sm->GetActiveScene();
    if (!scene) return;

    auto registry = scene->GetRegistry();
    if (!registry || !registry->HasComponent<T>(entity)) return;

    auto pool = registry->GetPool<T>();

    modifier(pool->Get(entity), pool);

    if (pool->IsStatic(entity)) {
        pool->MarkStaticDirty(entity);
    }
    else if (pool->IsDynamic(entity)) {
        pool->template SetBit<UPDATE_BIT>(entity);
    }
}
```

https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Editor/EditorApi/Impl/TransformApiImpl.cpp

Software Architecture XXIII.

Automatic Undo/Redo

- Intent-based communication naturally maps to the Command Pattern.
- Commands store the old and new values required for Undo and Redo.
- The same Api used by ViewModels are reused by Commands, eliminationg duplicated editing logic.
- Only finalized user actions create undoable commands, continuoues drag updates modifies the Engine immediately but are not stored in the Undo/Redo command registry!

Command Registry – Domain Layer

- Executed commands are automatically stored in the Undo/Redo history.
- The registry manages the complete editing history independently form the UI!

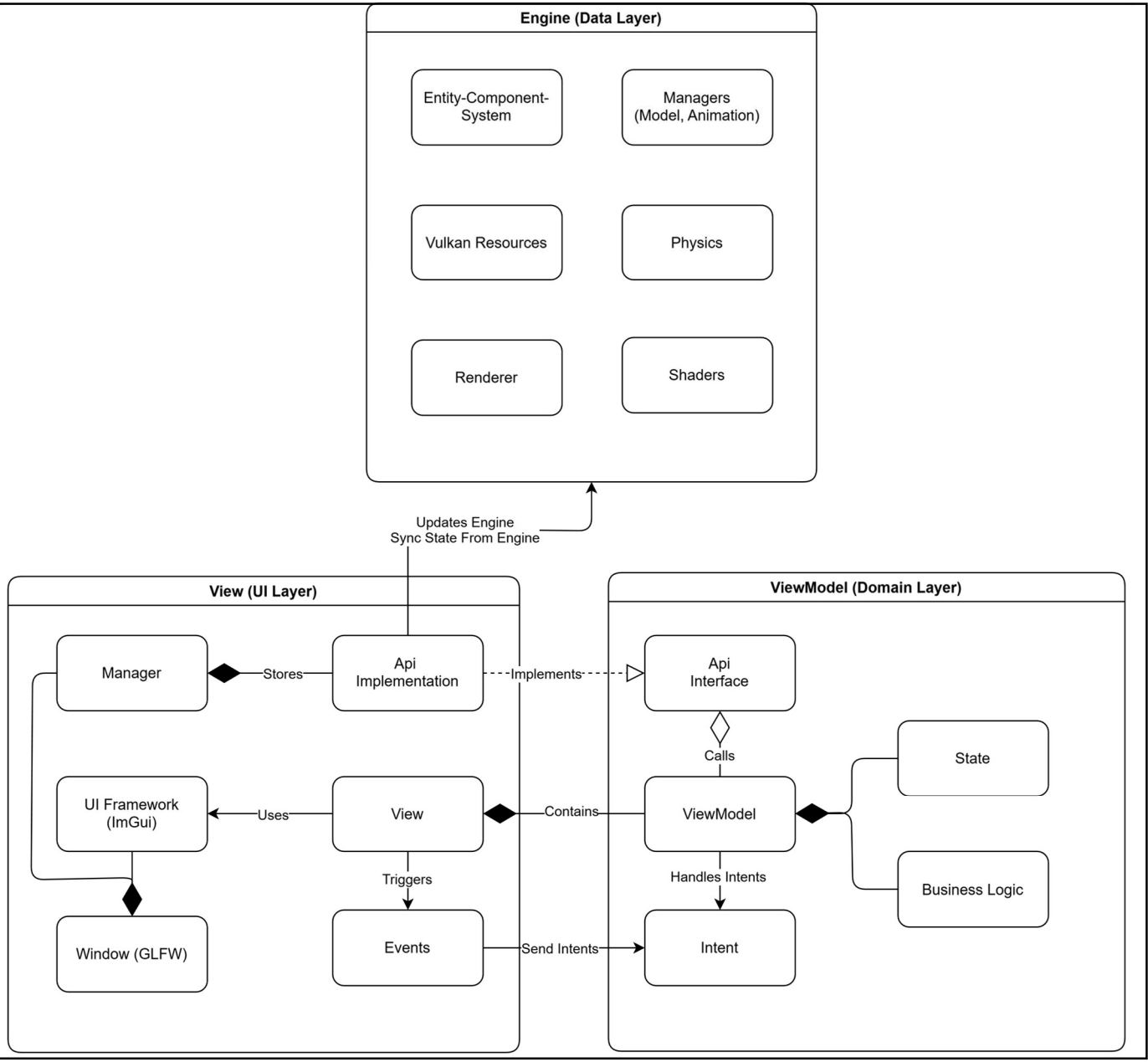
```cpp
template <typename ApiType, typename ValueType, auto SetterFunc>
class ComponentChangeCommand : public ICommand {
public:
    ComponentChangeCommand(ApiType* api, EntityID entity, const ValueType& oldVal, const ValueType& newVal)
        : _api(api), _entity(entity), _oldVal(oldVal), _newVal(newVal) {}

    void Execute() override {
        (_api->*SetterFunc)(_entity, _newVal);
    }

    void Undo() override {
        (_api->*SetterFunc)(_entity, _oldVal);
    }

private:
    ApiType* _api;
    EntityID _entity;
    ValueType _oldVal;
    ValueType _newVal;
};

using ChangePositionCommand = ComponentChangeCommand<ITransformApi, glm::vec3, &ITransformApi::SetEntityPosition>;
using ChangeRotationCommand = ComponentChangeCommand<ITransformApi, glm::vec3, &ITransformApi::SetEntityRotation>;
using ChangeScaleCommand    = ComponentChangeCommand<ITransformApi, glm::vec3, &ITransformApi::SetEntityScale>;

class CommandRegistry {
public:
    void ExecuteCommand(std::shared_ptr<ICommand> command) {
        command->Execute();
        _undoStack.push_back(command);
        _redoStack.clear();
    }

    void Undo() {
        if (_undoStack.empty()) return;
        auto cmd = _undoStack.back();
        cmd->Undo();
        _undoStack.pop_back();
        _redoStack.push_back(cmd);
    }

    void Redo() {
        if (_redoStack.empty()) return;
        auto cmd = _redoStack.back();
        cmd->Execute();
        _redoStack.pop_back();
        _undoStack.push_back(cmd);
    }
private:
    CommandRegistry() = default;
    std::vector<std::shared_ptr<ICommand>> _undoStack;
    std::vector<std::shared_ptr<ICommand>> _redoStack;
};
```

https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/EditorCore/ViewModels/SceneWorkspace/Component/Core/Transform/TransformCommands.h

Software Architecture XXIV.

MVI Architecture Summary

- Clear separation between Engine (Data), EditorCore (Domain), Editor (UI)
- The Engine remains the single sourch of truth for all runtime data.
- The UI is purely declarative and communicates exlusively through Intents.
- APIs isolate engine-specific operations, enabling reuse, testing, and interchangeable implementations.
- Intent-based communication naturally enables features such as Undo/Redo, command history.
- Replacing the UI framework (ImGui to Qt) requires rewriting only the View layer, while the Domain and Engine remain unchanged.
- **Arhcitecture follows Single Responsibility, Dependency Inversion and layered design, resulting in a modular, scalable and maintainable editor architecture.**



Software Architecture XXV.

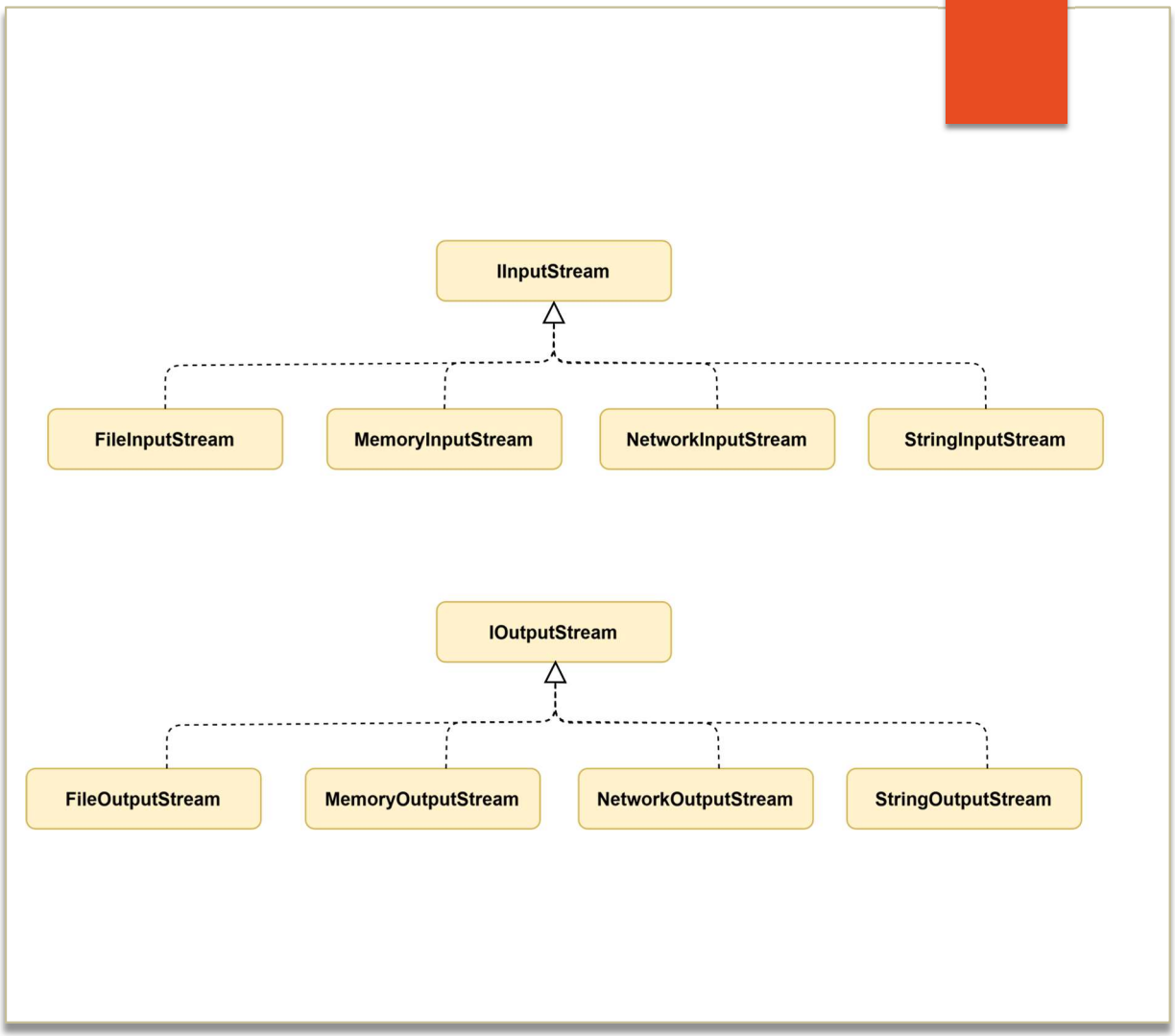
Serialization Architecture – Design Goals

- Layered serialization architecture separating data schemas, archive formats and I/O streams.
- Multiple stream backends supporting file, string, network and custom I/O sources.
- Format-independent data representation allowing the same data to be serialized as JSOn, YAML, XML, BINARY or future formats.
- Schema-driven serialization defining serializable data independenty from the underlying archive implementation
- Compile-time type dispatch using template metaprogramming to automatically handle primitive, enum, and schema-defined types.
- Data-oriented and ECS-friendly serialization designed around compact, contiguous data strcutures and efficient binary persistance.
- Asset-pipeline integration enabling the same serialization architecture to persist and restore engine assets and runtime data.
- Extensible and future-proof allowing new archive formats and stream backends without modifiying existing engine data structures.

Software Architecture XXVI.

Stream Abstraction

- I/O abstraction hides whether data comes from or goes to a file, memory, or network.
- Input/Output are strictly spearated through dedicated stream interfaces.
- **File streams** provide persistent asset and scene serialization
- **Memory streams** enable fast in-memory serialization and temporary data processing.
- **Network streams** allow remote data transfer and multiplayer synchronization
- New stream backends can be introduced independently without modifiying the serialization or schema layers.



https://github.com/TamasPetii/SynapseEngine/tree/main/SynapseEngine/Engine/Serialization/Stream

Software Architecture XXVII.

Schema-Driven Serialization

- Archives define the serialization mechanism while remailing independent form the actual data types.
- Schemas define the serializable structure of complex engine types and explicitly select which properties are persistend
- Template metaprogramming recursively resolves types.
- Complex types are decomposed recursively through their Schema<T> specialization.
- Data structures remail serialization-agnostic, components do no contain JSON, XML, YAML or binary-specific logic.
- Serialization logic is reusable across archive formats.

```cpp
class SYN_API IOutputArchive : public IArchive {
public:
    explicit IOutputArchive(IOutputStream& stream) : _stream(stream) {}
    virtual ~IOutputArchive() = default;

    virtual void PropertyBool(const char* name, bool value) = 0;
    virtual void PropertyInt32(const char* name, int32_t value) = 0;
    virtual void PropertyUint32(const char* name, uint32_t value) = 0;
    virtual void PropertyInt64(const char* name, int64_t value) = 0;
    virtual void PropertyUint64(const char* name, uint64_t value) = 0;
    virtual void PropertyFloat(const char* name, float value) = 0;
    virtual void PropertyString(const char* name, const std::string& value) = 0;
    virtual void PropertyBytes(const char* name, const void* data, size_t size) = 0;

    template <typename T>
    void Property(const char* name, const T& value);
protected:
    IOutputStream& _stream;
};

template <typename T>
void IOutputArchive::Property(const char* name, const T& value) {
    if constexpr (std::is_same_v<T, bool>) PropertyBool(name, value);
    else if constexpr (std::is_same_v<T, int32_t>) PropertyInt32(name, value);
    else if constexpr (std::is_same_v<T, uint32_t>) PropertyUint32(name, value);
    else if constexpr (std::is_same_v<T, int64_t>) PropertyInt64(name, value);
    else if constexpr (std::is_same_v<T, uint64_t>) PropertyUint64(name, value);
    else if constexpr (std::is_same_v<T, float>) PropertyFloat(name, value);
    else if constexpr (std::is_convertible_v<T, std::string>) PropertyString(name, static_cast<std::string>(value));
    else if constexpr (std::is_enum_v<T>) {
        Property(name, static_cast<std::underlying_type_t<T>>(value));
    }
    else if constexpr (has_schema<T, std::remove_reference_t<decltype(*this)>>) {
        Schema<std::remove_cvref_t<T>>::Invoke(*this, name, value);
    }
    else static_assert(has_schema<T>, "Serialization Error: Type has no defined Schema specialization!");
}


template <>
struct Schema<TransformComponent> {
    static constexpr bool exists = true;

    template <typename Archive, typename T>
    static void Invoke(Archive& ar, const char* name, T& val) {
        ScopedArchiveObject obj(ar, name);
        auto& comp = const_cast<std::remove_const_t<T>&>(val);

        ar.Property("translation", comp.translation);
        ar.Property("rotation", comp.rotation);
        ar.Property("scale", comp.scale);
    }
};
```

https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/Serialization/Schema/Component/Core/TransformComponentSchema.h

Software Architecture XXVIII.

Multiple Archive implementation

- Each archive implements the same serialization interface while using a dedicated third-party library internally.
- JSON serialization is backed by nlohmann, while YAML uses yaml-cpp.
- Human-readable formats such as YAML are ideal for development and version control, making scene files easy to review and diff in Git.
- JSON and XML are well suited for structured data exchanged and networking scnearios.
- Binary serialization is used for production assets and runtime data, providing compact storage and fast loading.
- **The data-oriented ECS and asset pipeline are naturally suited for efficient binary serialization, as their tightly packed data can be stored with minimal transformation.**

```cpp
class SYN_API NlohmannJsonOutputArchive : public IJsonOutputArchive
{
public:
    static std::vector<std::string> GetSupportedExtensions() { return {".json", ".jsn"}; }

    explicit NlohmannJsonOutputArchive(IOutputStream& stream);
    ~NlohmannJsonOutputArchive() override = default;

    std::string ToString() const override;
    void Serialize() override;

    void EnterObject(const char* name) override;
    void LeaveObject() override;
    void EnterArray(const char* name, uint32_t size) override;
    void LeaveArray() override;

    void PropertyInt32(const char* name, int32_t value) override;
    void PropertyUint32(const char* name, uint32_t value) override;
    void PropertyFloat(const char* name, float value) override;
    void PropertyDouble(const char* name, double value) override;
    //...
private:
    nlohmann::json _root;
    std::vector<nlohmann::json*> _stack;
};
```

```cpp
 class SYN_API YamlCppOutputArchive : public IYamlOutputArchive
{
public:
    static std::vector<std::string> GetSupportedExtensions() { return { ".yaml", ".yml" }; }

    explicit YamlCppOutputArchive(IOutputStream& stream);
    ~YamlCppOutputArchive() override = default;

    std::string ToString() const override;
    void Serialize() override;

    void EnterObject(const char* name) override;
    void LeaveObject() override;
    void EnterArray(const char* name, uint32_t size) override;
    void LeaveArray() override;

    void PropertyInt32(const char* name, int32_t value) override;
    void PropertyUint32(const char* name, uint32_t value) override;
    void PropertyFloat(const char* name, float value) override;
    void PropertyDouble(const char* name, double value) override;
    void PropertyString(const char* name, const std::string& value) override;
    void PropertyBytes(const char* name, const void* data, size_t size) override;
    //...
private:
    YAML::Node _root;
    std::vector<YAML::Node> _stack;
};
```

```json
"TransformComponent": {
    "rotation": {
        "x": 176.0,
        "y": 286.0,
        "z": 320.0
    },
    "scale": {
        "x": 1.0,
        "y": 1.0,
        "z": 1.0
    },
    "translation": {
        "x": -44.0,
        "y": 759.0,
        "z": -316.0
    }
},
```

```yaml
TransformComponent:
    translation:
        x: 107
        y: 154.99231
        z: 166
    rotation:
        x: 0
        y: -0
        z: 0
    scale:
        x: 1
        y: 1
        z: 1
```

https://github.com/TamasPetii/SynapseEngine/tree/main/SynapseEngine/Engine/Serialization/Archive

Software Architecture XXIX.

Software Architecture – Summary, Key Desing Principles, Closing Thoughts

- Layered Architecture separates responsibilities across systems and allows individual implementations to evolve independenmtly.
- Singe Responsibility keeps each component focused on one well-defined task, making the architecture easier to understand and extend.
- Dependency Inversion & Injection prevent concrete implementations from leaking across system boundaries and enable interchangeable implementations.
- Data-Oriented Design is applied beyond the ECS, shaping the asset pipeline, runtime data representations, serialization and GPU interaces.
- Pipeline Architecture transforms data through clearly defined stages, allowing processing to remain modular, parallel, and replaceable.
- MVI Architecture completely separates the UI from engine data and business logic, enabling framework-independent editor functionality and natural Undo/Redo integration.
- Schema-Based Serialization decouples data structures from storage formats, allowing the same engine data to be serialized to human-readable or high efficient binary representations.
- **The goald is not tosimply to build a working engine, but to build an architecture where every major system can evolve without forcing the rest of the engine to change with it!**

Synapse Engine:
Rendering

Created By: Péter Tamás
Github: https://github.com/TamasPetii/SynapseEngine

2026

Rendering Architecture I.

GPU-Driven Rendering

- Move rendering decisions from the CPU to the GPU wherever possible.
- The CPU should primarily provide scene data and orchestrate high-level execution.
- Visibility, LOD selection, and draw generation should be performed on the GPU.
- Minimize CPU-side iteration over scene entities and rendering objects.
- Generate rendering work dynamically through GPU-driven pipelines.
- Rendering workloads should be generated dynamically based on the current scene and visibility state.
- Indirect execution allows GPU-generated visibility results to directly drive the final rendering workload without CPU intervention.

Data-Oriented GPU Architecture

- GPU data must be stored in tightly packed, cache-friendly representations.
- CPU-orineted object structures should not be directly mirrored on the GPU.
- Large batched buffers are preferred over thousands of individual allocations.
- GPU descriptor structures (its not the vulkan descriptor) provide the indirection required to reconstruct logical objects from flat GPU data.
- Data layout is designed around how shaders consume the data.

Rendering Architecture II.

Bindless Resource Access

- Avoid per-object descriptor binding and CPU-side resource management.
- Resources are globally accessible through GPU-visible identifiers and descriptor tables.
- Shaders resolve models, materials, textures, samplers, and other resources dynamically.
- Resource access should scale independently from the number of rendered objects.

GPU-Driven Visibility

- Culling should happen as close as possible to the actual rendering workload.
- Frustum, hierarchical occlusion, meshlet, and other visibility tests can run parallel on the GPU.
- Invisible objects should be eliminated before expensive rendering work is generated.
- Hierarchical representations allow large parts of the scene to be rejected at once.

Indirect Rendering

- The CPU should not generate individual draw calls for every visible objects.
- GPU-generated commands allow visibility results to directly drive rendering.
- Multi-draw and indirect execution reduce CPU submission overhead.
- Rendering becomes a data flow rahter than a squence of CPU-issued commands.

Meshlet-Based Rendering

- Large meshes are decomposed into small, independently processable meshlets.
- Mehslets provide a natural granularity for GPU culling and rendering.
- Task and Mesh shaders allow geometry amplification and precessing to happen entirely on the GPU.
- Meshlet-level visibility enables finer-grained rejection than object-level culling.

Rendering Architecture III.

Dynamic LOD

- LOD selection should be performed dynamicaly based on the current view.
- Multiple LOD representations remain available directly on the GPU.
- The GPU selects the appropriate representation without CPU-side asset manipulation.
- LOD selection becomes part of the GPU-driven rendering pipeline.

Parallelism as a First-Class Design Goal

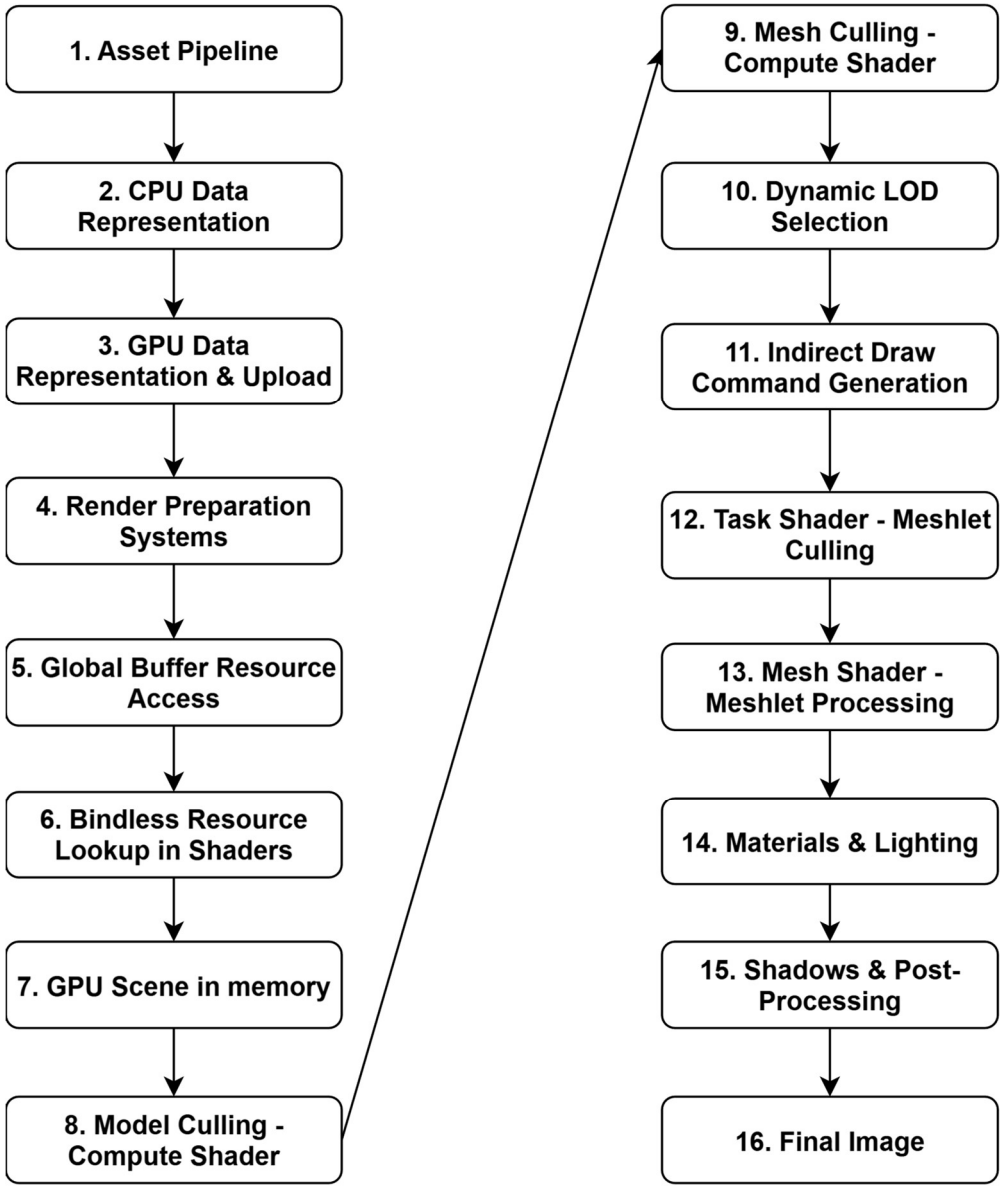
- Rendering workloads should expose as much parallel work as possible.
- GPU workloads are structured around massively parallel execution.
- CPU-side perparation is also parallelized where applicable.
- Task dependencies should be minimized and explicit.

Vulkan as an Enabler

- Vulkan provides explicit control over memory, synchronization, and GPU execution.
- **Modern Vulkan features enable the architecture rather than simply being used as isolated optimizations.**
- Buffer Device Address, Descriptor Indexing, Desciptor Buffers, Mesh Shaders, and Indirect Execution work together as part of the architecture.
- API decisions are driven by rendering requirements rahter than feature availibilty alone.

Scalable Rendering Architecture

- The renderer should scale with scene complexity without linearly increasing CPU overhead.
- Adding more models, materials, or instances should primarily increase GPU workload.
- Resource management, culling, LOD, and rendering should operate on data-oriented representations.
- The architecture should reimain extensible as new rendering techniques are introduced.

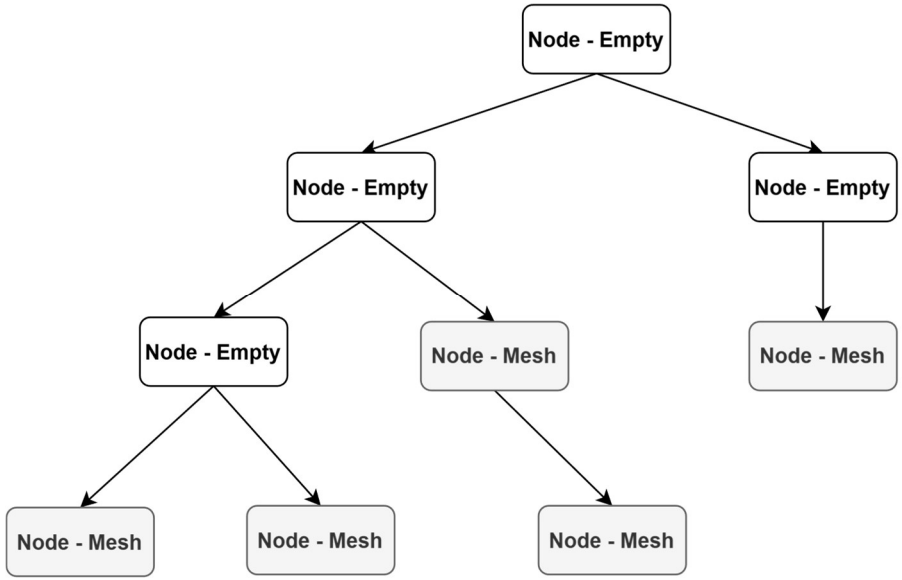


Rendering Architecture IV.

Model Representation I.

Model Structure and Hierarchy

- A model is composed of multiple meshes, each containing its own vertex and index data.
- Mesh vertices are defined in **local mesh space**!
- The hierarchy between meshes is represented by a node transform hierarchy.
- During asset processing, the node hierarchy is traversed and parent transforms are accumulated into each node's global transform!
- Each node stores a transform and may either reference a mesh or exist purely as a transform node.
- To reconstruct the complete model in model space, each mesh's vertices are transformed using the transform of their associated node (in shader).
- The same node hierarchy is reused by the animation system, where bone tracks provide animated node transform instead of static transforms.
- from hierarchical transforms, aThis representation separates geometry allowing **the same structure to support static and animated models**.




```
struct SYN_API GpuNodeTransform
{
    glm::mat4 transform;
    glm::mat4 transformIT;
};
```

https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/Mesh/Loader/AssimpMeshLoader.cpp

Model Representation II.

Vertex representation

- Vertex data is **split into two separate representations**: positions and attributes.
- Vertex positions remain in local mesh space, and do not contain the node transformation, so its not pre-applied on the CPU.
- Separating them allows shaders that only require positions to avoid loading normals, tangets, uvs, so this improves cache locality, memory bandwith and data prefetching.
- Both structure follow the GPU's std430 layout, with explicit packing to match GPU memory alignment requirements.
- Each vertex position stores a packed mesh index and node index (16-16 bit) allowing the shader to reconstruct the corresponding node transforms.
- The mesh index also becomes important later for material batching and GPU-driven rendering.
- Bitangent vectors are not stored explicitly, they are reconstructed dynamically in the shader!

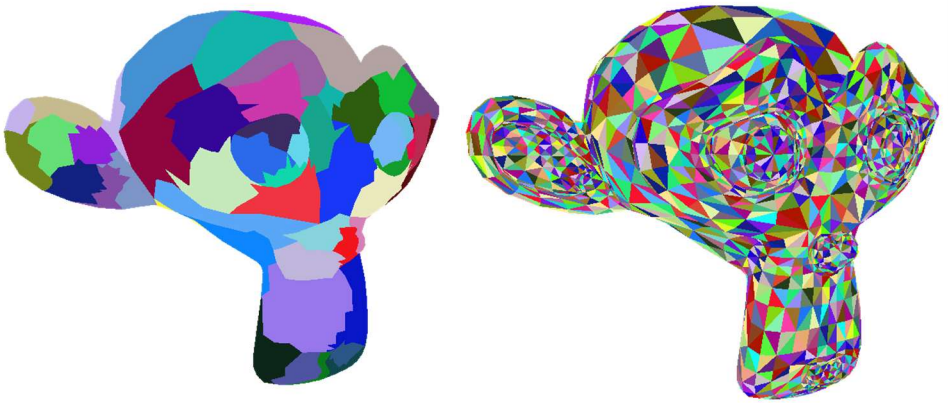
```
struct SYN_API GpuVertexPosition
{
    glm::vec3 position;
    uint32_t packedIndex; // nodeIndex, meshIndex

    void SetMeshAndNodeIndex(uint16_t meshIndex, uint16_t nodeIndex);
};

struct SYN_API GpuVertexAttributes {
    glm::vec3 normal;
    float uv_x;
    glm::vec3 tangent;
    float uv_y;
};

void main()
{
    //NO VERTEX ATTRIB READ!!

    uint realVertexIndex = GET_INDEX(addrs.indices, gl_VertexIndex);
    GpuVertexPosition v = GET_VERTEX_POS(addrs.vertexPositions, realVertexIndex);

    // Fetch Transform and Camera
    TransformComponent transform = GET_TRANSFORM(ctx.transformBufferAddr, transformDenseIndex);

    // Evaluate Static Hierarchy (Default pose)
    uint nodeIndex = UNPACK_UINT16_X(v.packedIndex);
    uint meshIndex = UNPACK_UINT16_Y(v.packedIndex);

    GpuNodeTransform staticNodeTransform = GET_NODE_TRANSFORM(addrs.nodeTransforms, nodeIndex);
    mat4 finalModelMat = staticNodeTransform.globalTransform;

    // Resolve Directional Light Shadow component
    mat4 viewProj = GET_DIRECTION_LIGHT_SHADOW(ctx.directionLightShadowDataBufferAddr, lightShadowDenseIndex);

    // Calculate Final World Position and Outputs
    vec4 clipPos = viewProj * transform.transform * finalModelMat * vec4(v.position, 1.0);
    gl_Position = clipPos;
}
```

https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/Mesh/Data/Gpu/GpuVertexData.h

Model Representation III.

Meshlet representation

- A meshlet is a compact partition of a mesh's geometry, built on top of the original vertex and index data.
- Meshlets reference the existing vertex data through a dedicated vertex index buffer and triangle index buffer.
- The original mesh index data is used as the input for meshlet generation and paritioning.
- Each mesh is processed independently, with meshlets typically limited to 64 vertices and 64 triangles in the current implementation.
- Meshlet generation is performed using **meshoptimizer**.
- Meshlets are generated separately for every mesh and later consumed by the Task and Mesh Shader pipeline.
- The meshlet representation enables fine-grained GPU culling and highly parallel meshlet processing.



```
size_t meshlet_count = meshopt_buildMeshlets(
    meshoptMeshlets.data(),
    meshoptVertexIndices.data(),
    meshoptTriangleIndices.data(),
    lod.indices.data(),
    lod.indices.size(),
    &mesh.vertices[0].position.x,
    mesh.vertices.size(),
    sizeof(Vertex),
    _maxVertices,
    _maxTriangles,
    _coneWeight
);
```

https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/Mesh/Processor/MeshProcessor/Meshlet/MeshoptimizerMeshletProcessor.cpp

Model Representation IV.

LOD Generation

- Multiple LOD levels are generated automatically using meshoptimizer.
- The vertex data remains unchanged across LOD levels, only the index data is progressively simplified.
- The current implementation uses four LOD levels: 100%, 50%, 25%, 12.5% of the original geometric complexity.
- Each mesh therefore has its own set of LOD-specific index data.
- Since meshlets depend on the index topology, meshlets are regenerated independently for evey LOD level.
- The final GPU representation therefore contains LOD-specific index data and LOD-specific meshlet data.
- **The memory layout is critical**: the GPU most be able to dynamically resolve the correct LOD and its associated meshlet data directly from compact LOD descriptors.
- This representation forms the foundation for GPU-driven, compute-based dynamic LOD selection

LOD0 – 100%

LOD3 – 12.5%

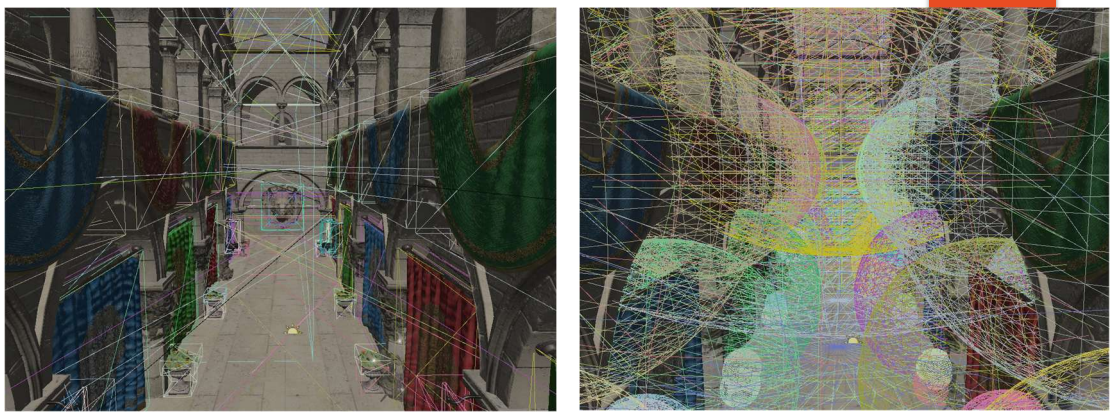
https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/Mesh/Processor/MeshProcessor/Lod/MeshoptimizerLodProcessor.cpp

Model Representation V.

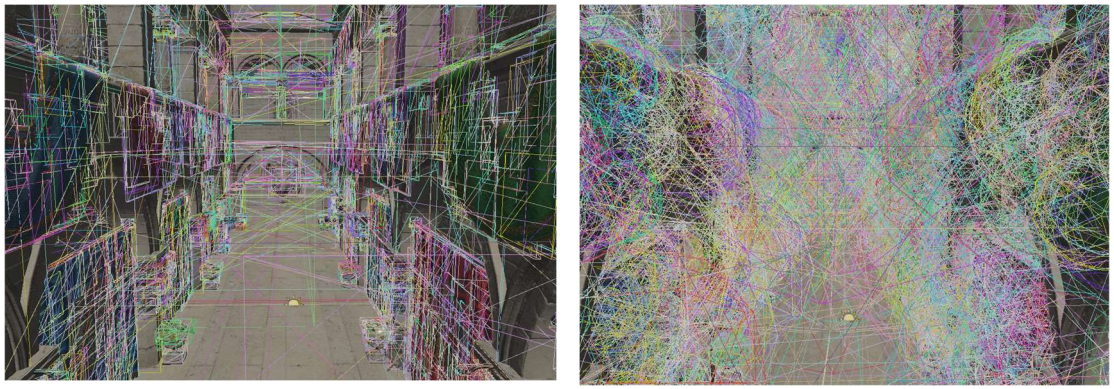
Collider generation

- Every mesh generates AABB and Sphere colliders from its vertices.
- Every meshlet also generates AABB, Sphere and Cone bounds, generated by meshoptimizer.
- A model-level collider is generated from the colliders of all meshes belonging to the model.
- **Colliders are stored in model space, with node transformations already applied** during asset processing to avoid unnecessary transform calculations during GPU culling.
- The data-oriented mesh processor pipeline allows collider generation to run fully in parallel across meshes and meshlets!
- OBB colliders are intentionally avoided, AABBs and spheres provide better trade-off between culling accuracy, memory usage and computational cost for GPU-driven rendering.
- **Mesh-level colliders are generated only for LOD0**, since LOD selection is performed dynamicall on the GPU, generating separate colliders for each LOD would require knowing the selected LOD beforehand.

Mesh – AABB/Sphere Collider



Meshlet – AABB/Sphere Collider



https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/Mesh/Processor/MeshProcessor/Geometry/ColliderProcessor.cpp

Model Representation VI.

Batched Buffers & Allocation

- The model is a single logical asset composed of multiple meshes, so GPU resources are batched across all meshes into shader buffers.
- A per-mesh buffer allocation would result in an excessive number of GPU buffers…
- Fully batching the entire model into a single renderable unit is also undesirable, as it prevents mesh-level culling and limits rendering flexibility.
- **The solution is a hybrid representation**: buffers are fully batched, while each mesh has dedicated descriptors that define its data is located within the shared buffers.
- The mesh descriptors allow the renderer to reconhstruct each mesh' vertices, indices, meshlets, colliders.
- Although the ECS and high-level logic operate on models, **the renderer operates on individual meshes**, with each mesh corresponding to its own draw command.

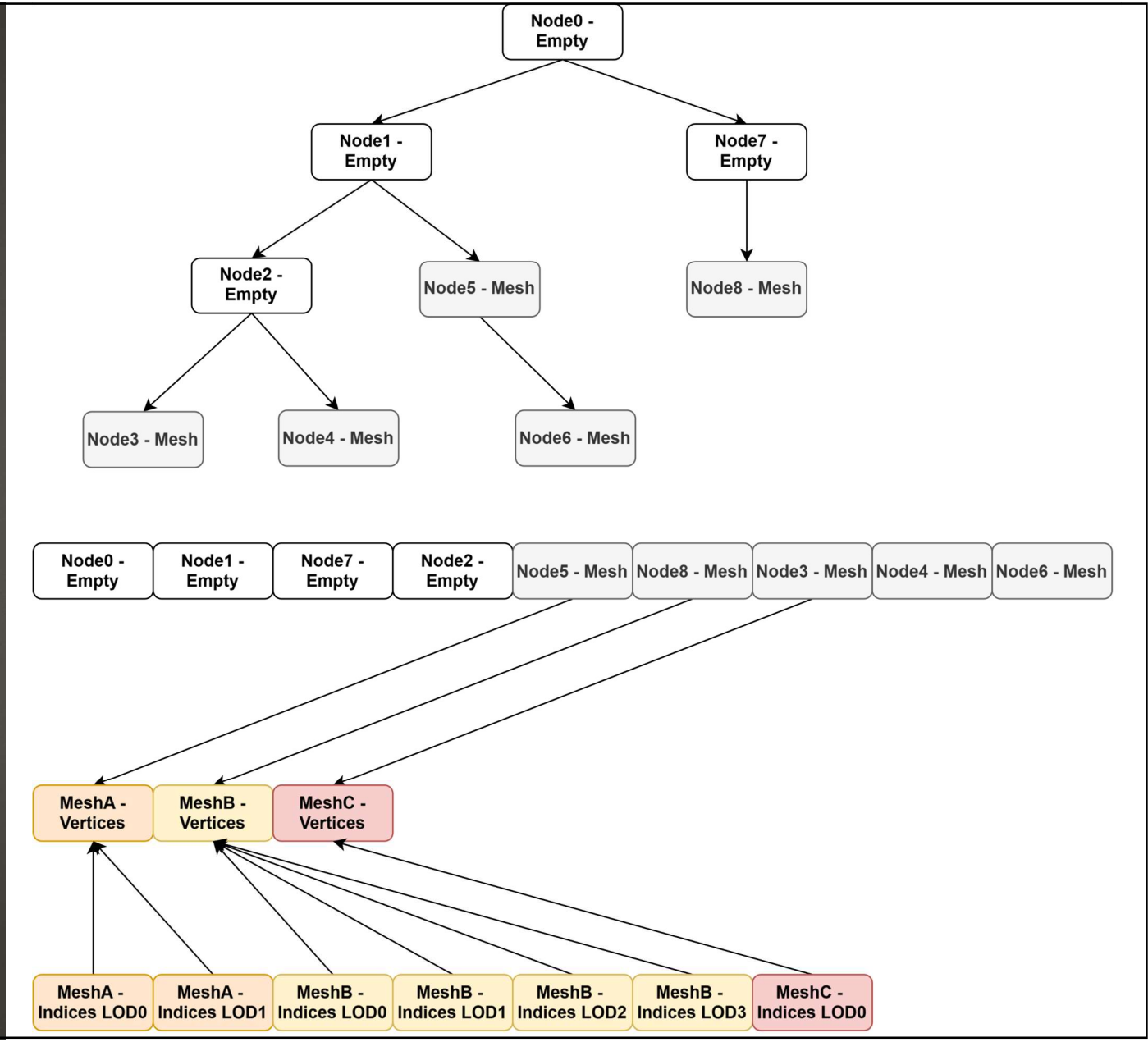
```
struct SYN_API GpuModelBuffers
{
    // Geometry Buffers
    std::unique_ptr<Vk::Buffer> vertexPositions;
    std::unique_ptr<Vk::Buffer> vertexAttributes;
    std::unique_ptr<Vk::Buffer> indices;
    std::unique_ptr<Vk::Buffer> meshMaterialIndices;

    // Traditional Pipeline Buffers
    std::unique_ptr<Vk::Buffer> meshDescriptors;
    std::unique_ptr<Vk::Buffer> meshColliders;
    std::unique_ptr<Vk::Buffer> lodDescriptors;

    // Mesh Shader Buffers
    std::unique_ptr<Vk::Buffer> meshletVertexIndices;
    std::unique_ptr<Vk::Buffer> meshletTriangleIndices;
    std::unique_ptr<Vk::Buffer> meshletDescriptors;
    std::unique_ptr<Vk::Buffer> meshletDrawDescriptors;
    std::unique_ptr<Vk::Buffer> meshletColliders;

    //Hierarchy Buffers
    std::unique_ptr<Vk::Buffer> nodeTransforms;
};
```

https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/Mesh/Uploader/DefaultGpuModelUploader.cpp

Model Representation VII.

Batched Mesh-Based Allocation

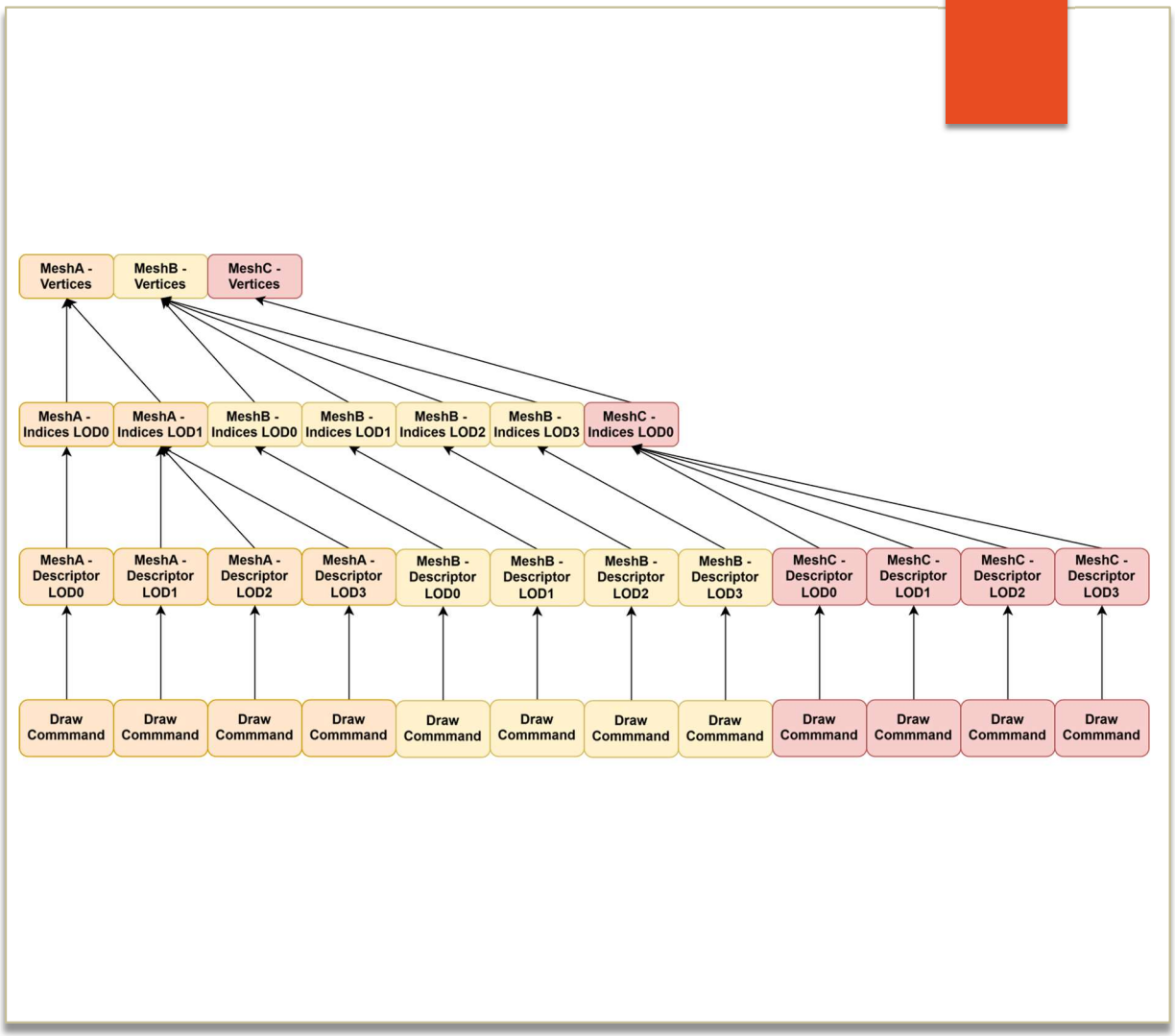
- Flatten the node hierarchy into a level-contiguous, data-oriented vector representation, preserving parent-child relationships.
- Traverse the nodes and cocatenate each mesh' vertices, indices, collider, meshlets and other data into the corresponding global model buffers.
- Record the starting offsets and sizes to build mesh descriptors and draw commands.
- Store each mesh's LOD data as well, with dedicated draw commands for every LOD level.
- The batching itself is straightforward, the real challange is uniquely identifying each mesh and its LOD data.
- The GPU must be able to reconstruct these data ranges efficiently using compact buffers and compute shaders.
- **The final layout is designed around GPU-driven access and dynamic LOD selection**



Model Representation VIII.

LOD Representation Challange

- Each mash can have up to 4 LOD levels, but not every mesh can be simplified to all four levels.
- Simple geometry, such as a cube, may only have one valid LOD, while more complex meshes may have two, three, or all four levels.
- To enable true O(1) GPU indexing, every mesh is assigned a fixed number of 4 LOD descriptors and draw commands, regardless of how many valid LOD levels it actually contains.
- The LOD descriptor buffer is therefore a compact, GPU-friendly array where each mesh occupies excatly four consecutive entries.
- Any missing LOD descriptors simply references the last valid LOD level.
  - 1 valid: **LOD0, LOD0, LOD0, LOD0**
  - 3 valid: LOD0, LOD1, **LOD2**, **LOD2**
- Any mesh and LOD descriptors can then be resolved in O(1) using **meshIndex * 4 + lodIndex**
- The trade-off is minor rendering overhead: meshes with fewer LODs may generate multiple draw commands that ultimately reference the same geometry, but the GPU indexing remains simple and fully predictable

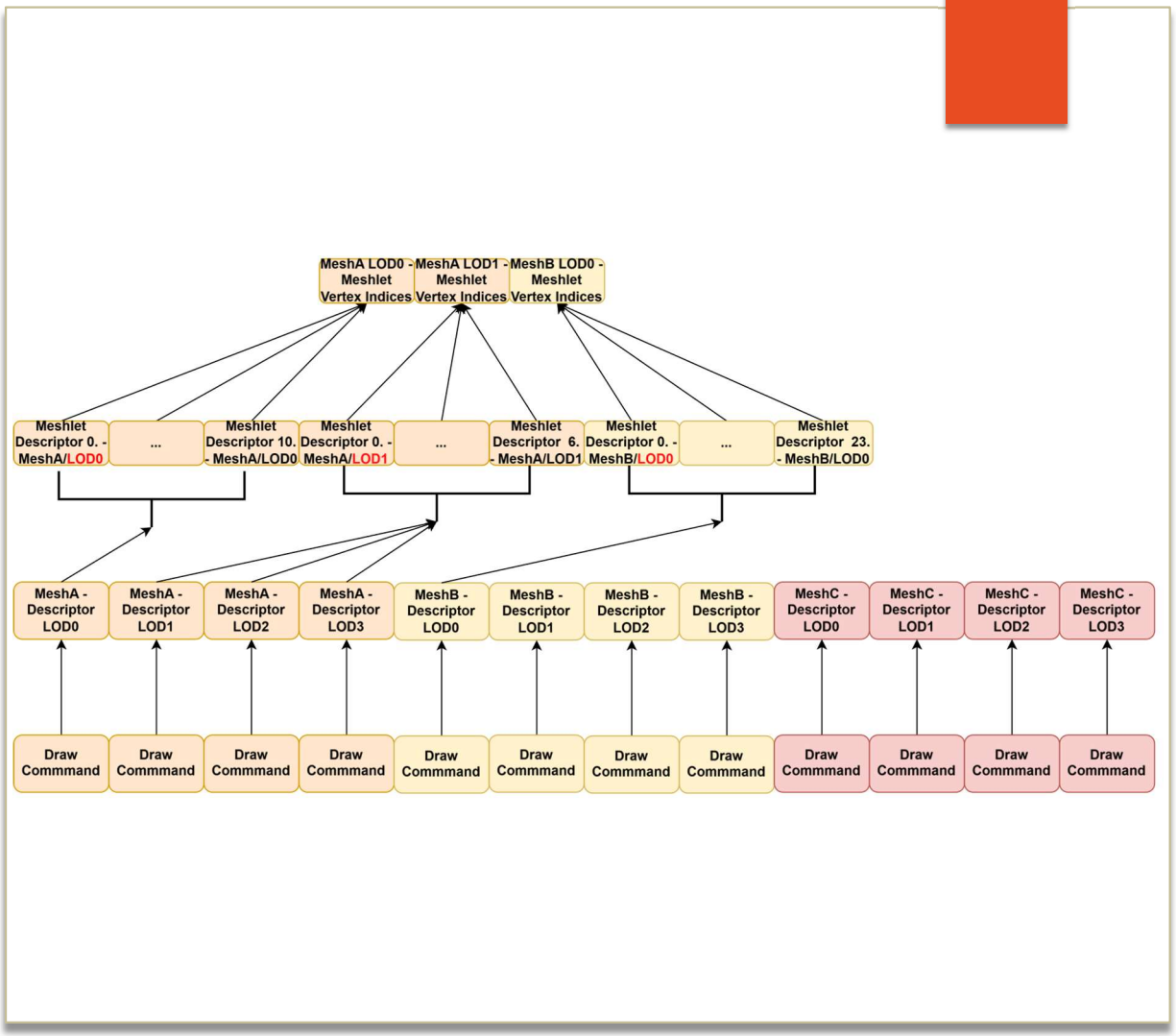


https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/Mesh/Converter/DefaultGpuModelConverter.cpp

Model Representation IX.

Meshlet Descriptor

- The same descriptor-based approach is used for meshlets, as the Mesh Shader pipeline operates on meshlet-level geometry rather than entire meshes.
- Meshlet data, including vertex indices and triangle indices is fully batched into shared GPU buffers.
- During allocation, each meshlet records its offset and count, forming a dedicated meshlet descriptor.
- Every mesh and every LOD level has its own set of meshlet descriptors, generated from the corresponding data.
- Meshlet descriptors are stored in a compact batched array, with the descriptors belonging to each mesh and LOD stored consecutively.
- If a mesh has fewer than four valid LOD levels, the missing LODs simply reuse the meshlets of the last valid LOD.
- Task and Mesh shaders can then **map meshlets directly to shader invocations** using their built-in invocation Ids, enabling highly parallel meshlet processing.

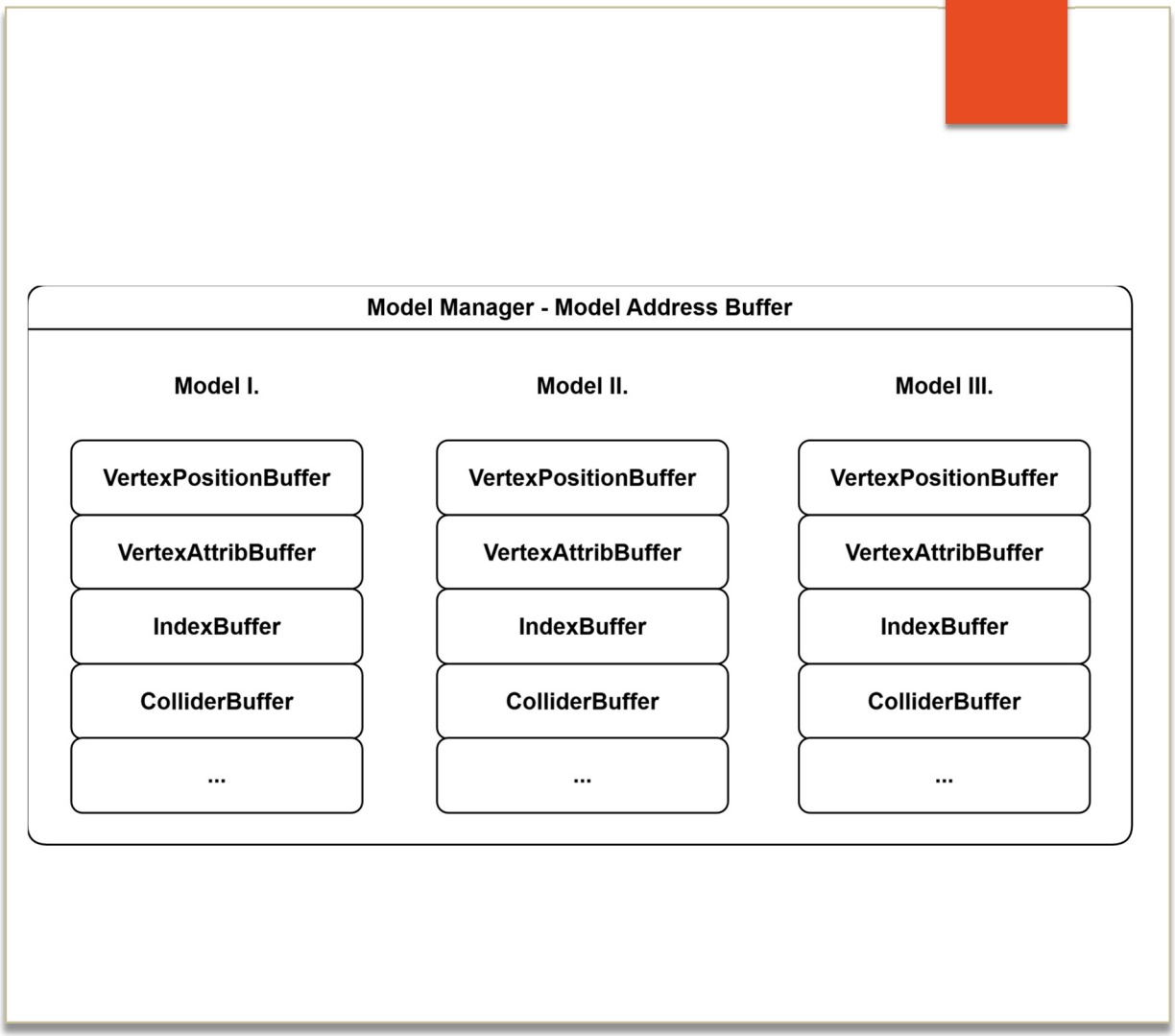


https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/Mesh/Converter/DefaultGpuModelConverter.cpp

Model Representation X.

Model Manager & Global Resource Access

- A single model has 10+ GPU buffers, and many models can be loaded simultaneously.
- The renderer requires global GPU access to every model and all of its resources.
- The ModelManager stores all loaded models in a compact, data-oriented vector so a global GPU buffer stores the addresses of all model-specific buffers, index by the ModelIndex.
- Shaders can use the ModelIndex to dynamically resolve and access the corresponding model resources.
- This creates a hierarchical resource structure: a global buffer contains model-level addresses, which point to the actual GPU data buffers.
- **The challange**: how can shaders access a dynamically growing set of GPU buffers without binding each resource individually?

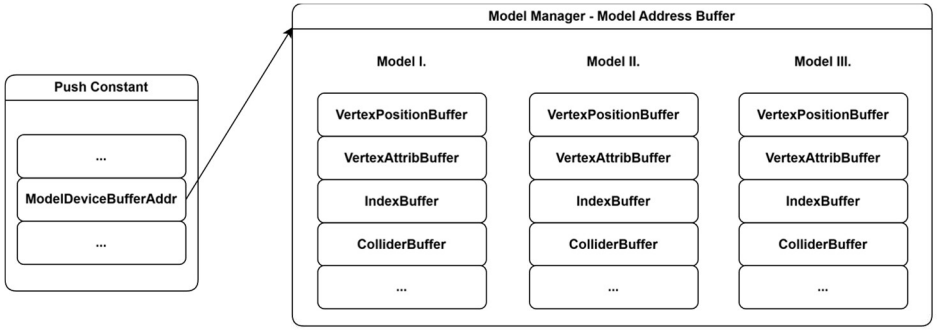


https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/Manager/AddressResourceManager.h

Model Representation XI.

Vulkan Bindless

- Vulkan's Buffer Device Address *(VK_KHR_buffer_device_address extension)* functionality provides a 64-bit GPU address for a buffer, allowing shaders to access GPU memory through device addresses.
- This removes the need to explicitly bind every individual buffer through traditional descriptor bindings.
- In a GPU-driven renderer, many models and resources must be accessible simultaneously, making traditional pre-resource binding increasingly comples and inflexible.
- The ModelManager therefore maintains a global buffer of GPU device addresses, where each model entry contains the addresses of its GPU resources.
- The address of this global resource table is passed to the GPU once, for example through a push constant.
- **This enables hierarchial GPU resource storage**, where buffers can contain the device address of other buffers, allowing complex GPU data structures to be represented and traversed entirely on the GPU.




```
#extension GL_EXT_buffer_reference : require

struct GpuModelAddresses
{
    uint64_t vertexPositionsAddress;
    uint64_t vertexAttributesAddress;
    uint64_t indexBufferAddress;
    uint64_t meshCollidersAddress;
    // ...
};

layout(buffer_reference, std430) readonly restrict buffer ModelAddressBuffer { GpuModelAddresses data[]; };
layout(buffer_reference, std430) readonly restrict buffer PositionBuffer { GpuVertexPosition data[]; };
layout(buffer_reference, std430) readonly restrict buffer AttributeBuffer { GpuVertexAttributes data[]; };

layout(push_constant) uniform PushConstants
{
    uint64_t modelAddressBufferAddress;
} pc;

void main()
{
    // Model selected by the GPU-driven rendering workload
    uint modelIndex = ...;

    // Resolve the model's resource addresses
    GpuModelAddresses model =  ModelAddressBuffer(pc.modelAddressBufferAddress).data[modelIndex];

    // Resolve the actual GPU buffers
    PositionBuffer positions = PositionBuffer(model.vertexPositionsAddress);
    AttributeBuffer attributes = AttributeBuffer(model.vertexAttributesAddress);

    // Access model data directly through GPU addresses
    GpuVertexPosition vertex = positions.data[gl_VertexIndex];
    GpuVertexAttributes attribute = attributes.data[gl_VertexIndex];

    // ...
}
```

Model Representation XII.

Model Representation - Summary

- Models are represented through a layered GPU-oriented structure built from nodes, meshes, LODs, meshlets and descriptors.
- Vertex data is split into positions and attributes, allowing shaders to fetch only the data required by a particular rendering pass.
- Vertex positions remain in local mesh space, while colliders are stored in model space already containing the node transformation.
- Mesh data is stored in large batched GPU buffers, while per-mesh descriptors provide the offsets and counts required to reconstruct individual meshes from the packed storage.
- Each mesh supports up to four LOD levels, with a fixed-size LOD descriptor array enabling O(1) GPU indexing even when a mesh has fewer valid LODs
- Every LOD has its own meshlet representation, with a batched meshlet descriptors and index data enabling fine-grained meshlet-level processing and culling.
- AABB and Sphere bounds are generated for meshes and models, while meshlets additionally use Cone bounds for efficient GPU-driven culling.
- The same data-oriented representation supports both traditional mesh rendering and mesh-shader pipelines, allowing the renderer to operate at mesh or meshlet granularity.
- The resulting representation is designed for GPU-driven rendering and bindless resource access, using compact descriptors, batched buffers, and GPU addresses to dynamically resolve model, mesh, LOD, and meshlet data without CPU-side resource rebinding.

Rendering I.

Rendering: The Worst Way

- **Fully CPU-driven rendering**: the CPU iterates over every renderable entity and processes each model individually.
- Visibility is checked on the CPU before rendering each entity.
- For every visible model, the CPU iterates through all meshes and prepares the required rendering state.
- Vertex and index buffers are bound, material resources and transform data are updated, then *vkCmdDrawIndexed* is issued for each mesh.
- **Each draw call processes the complete vertex/index range of the selected mesh**, even when the same mesh is rendererd repeatedly across many instances.
- Rendering the same large mesh thousands of times can therefore generate enormous amounts of redundant GPU work.
- The CPU becomes responsible for both scene traversal and rendering workload generation, creating a major scalability bottleneck.
- CPU iterates → CPU culls → CPU binds → CPU submits → GPU renders

```
for (EntityID entity : scene.GetEntities())
{
    if (!scene.IsVisible(entity))
        continue;

    const Model& model = scene.GetModel(entity);
    const glm::mat4& transform = scene.GetTransform(entity);

    for (const Mesh& mesh : model.GetMeshes())
    {
        // Update per-object state
        pushConstants.transform = transform;
        pushConstants.materialIndex = mesh.GetMaterialIndex();

        vkCmdPushConstants(commandBuffer, pipelineLayout,
            VK_SHADER_STAGE_VERTEX_BIT | VK_SHADER_STAGE_FRAGMENT_BIT,
            0, sizeof(PushConstants), &pushConstants);

        // Bind mesh-specific index buffer
        VkBuffer indexBuffer = mesh.GetIndexBuffer();

        vkCmdBindIndexBuffer(commandBuffer, indexBuffer, 0, VK_INDEX_TYPE_UINT32);

        // Render the mesh
        vkCmdDrawIndexed(
            commandBuffer,
            mesh.GetIndexCount(),
            1,              // instanceCount
            0,              // firstIndex
            0,              // vertexOffset
            0               // firstInstance
        );
    }
}
```

Rendering II.

Instance Rendering

- Visible instances of the same mesh are grouped together and rendered with a single draw call.
- Each mesh maintains an **instance index buffer containing the EntityID** of every visible instance.
- *vkCmdDrawIndexed* is issued once per mesh, with *instanceCount* set to the number of visible instances.
- ***gl_InstanceIndex*** directly indexes the instance index buffer, resolving the corresponding EntityID for each instance.
- The EntityID is then used to retrieve the instance's transform and other per-entity data from the Sparse/Dense ECS GPU buffers.
- All instances reuse the same vertex and index data, avoiding redundant geometry processing and buffer binding.
- The GPU processes the shared mesh data once and executes the rendering workload across all visible instances.
- One Mesh → Many Instances → One Draw Call

```
// CPU: collect visible instances for each mesh
for (EntityID entity : scene.GetEntities())
{
    if (!scene.IsVisible(entity))
        continue;

    const Model& model = scene.GetModel(entity);

    for (const Mesh& mesh : model.GetMeshes())
    {
        // Store the entity associated with this mesh instance
        meshInstanceIndices[mesh.GetID()].push_back(entity);
    }
}

// Render each mesh once
for (const Mesh& mesh : scene.GetMeshes())
{
    const auto& instances = meshInstanceIndices[mesh.GetID()];

    if (instances.empty())
        continue;

    // Upload visible EntityIDs for this mesh
    UploadInstanceIndices(mesh, instances);

    vkCmdBindIndexBuffer(commandBuffer, mesh.GetIndexBuffer(), 0, VK_INDEX_TYPE_UINT32);

    vkCmdDrawIndexed(commandBuffer, mesh.GetIndexCount(), static_cast<uint32_t>(instances.size()), 0, 0, 0);
}

//-----------------------------------------------------------------------------------------

layout(buffer_reference, std430) readonly buffer InstanceIndexBuffer
{
    uint entityIDs[];
};

layout(push_constant) uniform PushConstants
{
    uint64_t instanceIndexBufferAddress;
} pc;

void main()
{
    // Resolve the EntityID of the current instance
    uint entityID = InstanceIndexBuffer(pc.instanceIndexBufferAddress).entityIDs[gl_InstanceIndex];

    // Resolve per-entity data through the ECS
    Transform transform = GetTransform(entityID);

    // Resolve the vertex through the index buffer
    uint vertexIndex =IndexBuffer(pc.indexBufferAddress).indices[gl_VertexIndex];

    // Fetch shared mesh geometry
    vec3 position = VertexBuffer(pc.vertexBufferAddress).vertices[vertexIndex];

    // Transform the shared vertex for this instance
    vec3 worldPosition = (transform.matrix * vec4(position, 1.0)).xyz;

    // ...
}
```

Rendering III.

CPU Bottleneck

- The fundamental limitation of traditional and instanced rendering is that the CPU still generates the rendering workload every frame.
- Consider a scene with thousands of different meshes, each potentially having thousands of visible instances.
- The CPU must continously iterate over entities, perform visibility checks, group instances, and update instance index buffers.
- Even with parallel processing and atomic counters, repeatedly rebuilding this data on the CPU introduces significant synchronization and memory overhead.
- The CPU must still iterate over thousands of meshes, bind their resources, and issue thousands of *vkCmdDrawIndexed* calls every frame.

- Most of this work is redundant when the scene structure and mesh resources remain relatively stable between frames.
- The CPU should not have to repeatedly describe what needs to be rendered, it should primarily provide scene data and orchestrate the rendering pipeline.
- A better approach is to predefine or generate rendering commands in a GPU-accessible buffer, allowing the CPU to submit the workload with minimal per-mesh interaction.
- This leads naturally to **indirect rendering**, where the GPU can consume a buffer of draw commands instead of the CPU issuing every draw individually.
- Instead of the CPU issuing thousands of draw calls, the CPU should provide the GPU with the data required to generate and execute the rendering workload.

Rendering IV.

Indirect Rendering

- Draw parameters are stored in a GPU-accessible indirect command buffer instead of being provided directly by the CPU.
- A single *vkCmdDrawIndexedIndirect* call can execute a large number of draw commands stored sequentially in the buffer.
- The CPU no longer needs to issue an individual draw call for every mesh, significantly reducing CPU-side rendering overhead.
- The command buffer can contain draw commands for all meshes and LOD levels in the scene.
- The entire scene can therefore be submitted with a single indirect rendering call, covering all entities, models, meshes, and their instances.
- However, all commands share the same bound rendering state, creating a fundamental problem: **How can different meshes use differenct index buffers without binding?**
- Solving this limitation requires moving away from traditional per-draw index buffer binding and introducing **Vertex Pulling**.

```
struct VkDrawIndexedIndirectCommand
{
    uint32_t indexCount;
    uint32_t instanceCount;
    uint32_t firstIndex;
    int32_t  vertexOffset;
    uint32_t firstInstance;
};

// GPU-visible indirect command buffer
VkBuffer indirectBuffer = CreateIndirectBuffer();
VkDrawIndexedIndirectCommand* commands = MapBuffer<VkDrawIndexedIndirectCommand>(indirectBuffer);

for (const Mesh& mesh : scene.GetMeshes())
{
    commands[mesh.GetDrawIndex()] = {
        .indexCount    = mesh.GetIndexCount(),
        .instanceCount = mesh.GetInstanceCount(),
        .firstIndex    = mesh.GetFirstIndex(),
        .vertexOffset  = mesh.GetVertexOffset(),
        .firstInstance = mesh.GetFirstInstance()
    };
}

// One CPU command executes all indirect draw commands
vkCmdDrawIndexedIndirect(
    commandBuffer,
    indirectBuffer,
    0,
    scene.GetDrawCount(),
    sizeof(VkDrawIndexedIndirectCommand)
);
```

Rendering V.

Indirect Draw Count

- *vkCmdDrawIndirectCount* allows the GPU to control how many indirect draw commands are actually executed.
- The indirect command buffer can therefore be larger than the current number of active draw commands, avoiding frequent reallocations as the scene changes.
- The actual draw count is stored in a separate GPU-resident count buffer and can be updated dynamically by GPU workloads.
- A draw command with **instanceCount of 0 is already efficiently skipped** by the GPU, so explicitly compacting suck commands is usually overhead.
- Since the renderer no longer binds individual index buffers, the pipeline uses non-indexed indirect draw commands instead of *VkDrawIndexedIndirectCommand*
- **gl_DrawID** identifies the currently executed draw command, allowing the shader to resolve the curresponding mesh and its associated GPU data.
- If all meshes are rendered through one indirect command buffer, **where does the vertex and index data come from**?

```
// GPU-generated number of active draw commands
uint drawIndex = atomicAdd(drawCount, 1);

// Append a new draw command
drawCommands[drawIndex] = DrawIndirectCommand(
    vertexCount,
    instanceCount,
    firstVertex,
    firstInstance
);

vkCmdDrawIndirectCount(
    commandBuffer,
    drawCommandBuffer,
    0,
    drawCountBuffer,
    0,
    maxDrawCount,
    sizeof(VkDrawIndirectCommand)
);

//Vertex Shader:

void main()
{
    // Identifies the current indirect draw command
    uint meshIndex = gl_DrawID;

    // Resolve the mesh descriptor
    MeshDescriptor mesh = meshDescriptors[meshIndex];

    // ...
}
```

Rendering VI.

Vertex Pulling

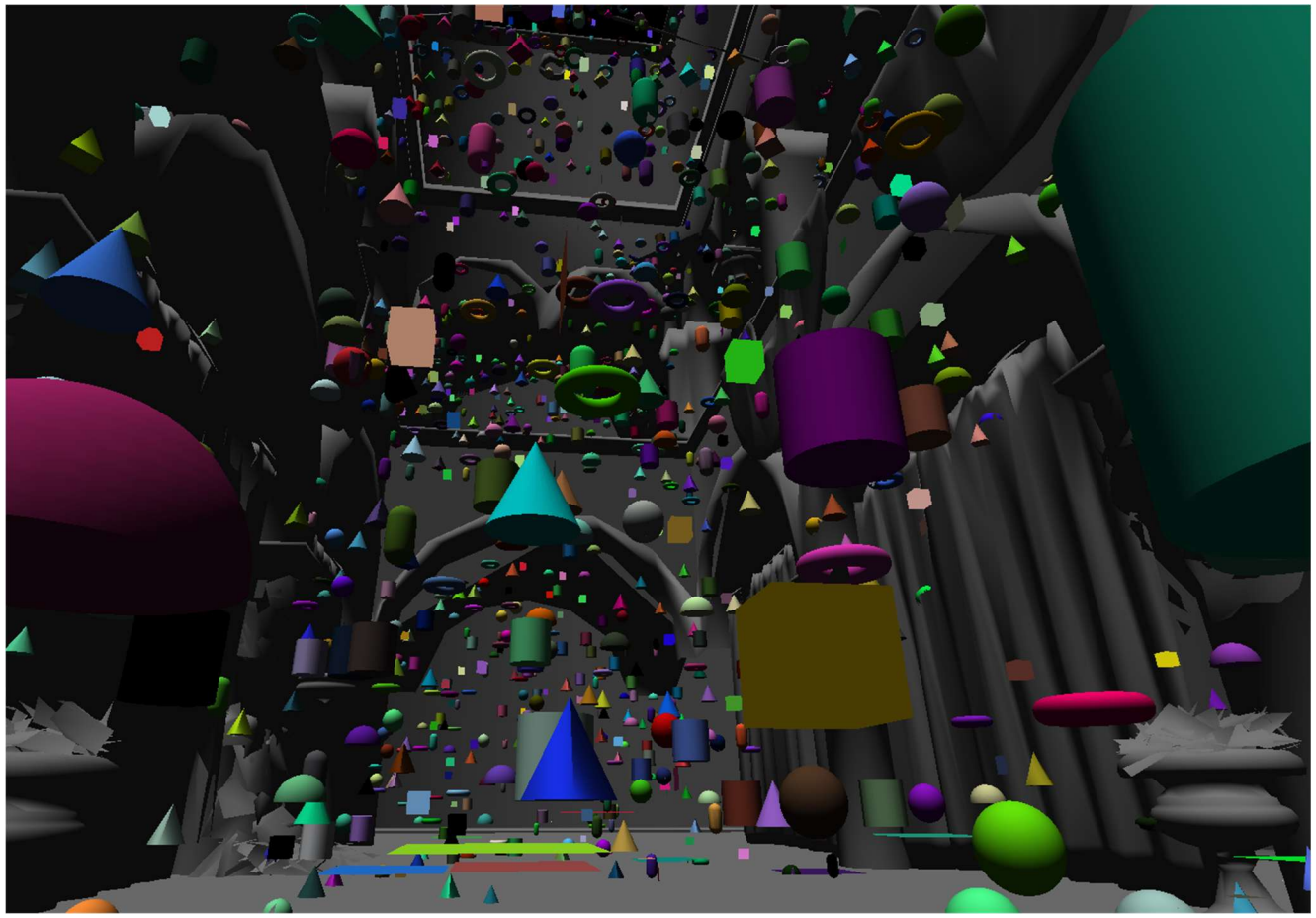
- **No vertex and index buffer is explicitly bound** to the graphics pipeline.
- All model buffers are already resident on the GPU and globally accessible through the bindless resource architecture introduced earlier.
- The indirect draw command's **vertexCount corresponds to the number of indices** that need to be processed.
- *gl_VertexIndex* identifies the current index position, which is used to manually fetch the actual index from the model's index buffer.
- The fetched index is then used to retrieve the corresponding vertex from the model's vertex buffer.
- Vertex pulling replaces traditional vertex input fetching with explicit shader-side buffer access.
- This allows a single indirect rendering pass to process meshes from many different models without rebinding vertex or index buffers between draw commands.

```
// Traditional pipeline: Vertex and index buffers are bound before drawing.

uint index = indexBuffer[gl_VertexIndex];
vec3 position = vertexBuffer[index];

// Vertex Pulling:

#extension GL_EXT_buffer_reference : require

layout(buffer_reference, std430) readonly buffer IndexBuffer
{
    uint indices[];
};

layout(buffer_reference, std430) readonly buffer VertexBuffer
{
    GpuVertexPosition vertices[];
};

layout(buffer_reference, std430) readonly buffer ModelAddressBuffer
{
    GpuModelAddresses data[];
};

layout(push_constant) uniform PushConstants
{
    uint64_t modelAddressBuffer;
} pc;

void main()
{
    // One draw command corresponds to one mesh
    uint meshIndex = gl_DrawID;

    // Resolve the resources belonging to this mesh/model
    GpuModelAddresses model = ModelAddressBuffer(pc.modelAddressBuffer).data[meshIndex];

    // Resolve GPU buffers through their device addresses
    // gl_VertexIndex now represents the index-buffer position
    uint vertexIndex = IndexBuffer(model.indexBufferAddress).indices[gl_VertexIndex];

    // Resolve the actual vertex
    GpuVertexPosition vertex = VertexBuffer(model.vertexPositionsAddress).vertices[vertexIndex];

    vec3 position = vertex.position;

    // ...
}
```

Rendering VII.

GPU-Driven Rendering

- Minimize CPU involvement in the rendering process and move rendering workload management to the GPU wherever possible.
- The CPU primarily provides scene data and submits the rendering workload instead of iterating over individual meshes and instances.
- The entire scene is represented through GPU-resident, indirect rendering command so **all models, meshes, instances, and LOD levels can be processed through a unified GPU-driven rendering pipeline**.
- The renderer uses a single indirect draw submission to execute the complete rendering workload.
- Resource binding and per-mesh data access are resolved dynamiocall through the bindless GPU resource architecture.
- The result is a rendering pipeline where the CPU no longer needs to explicitly issue individual draw calls or manage per-mesh rendering state.



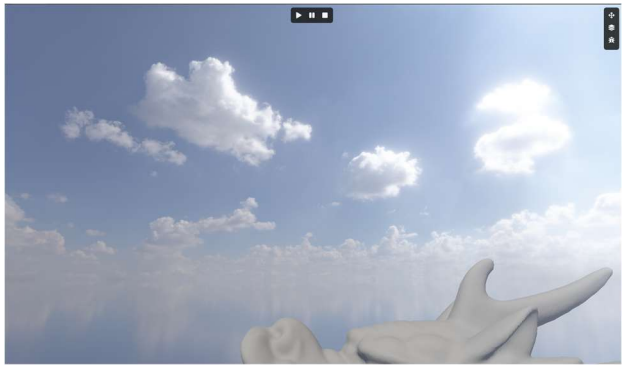
This scene was rendered with only 1 vkCmdDrawIndirectCount call (1 sponza + 13 unique shapes + 4 LOD levels)

Rendering VIII.

Why Vertex Shaders Are Not Enough?

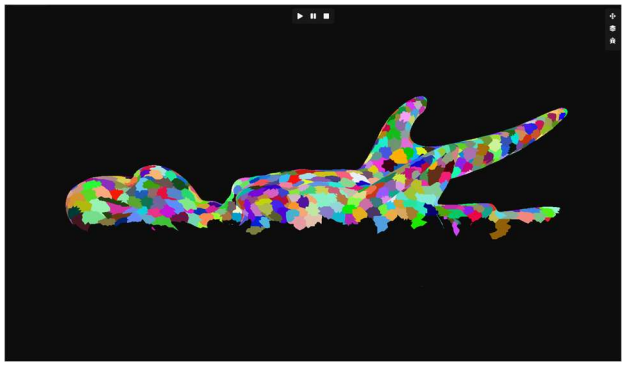
- Traditional vertex processing operates at mesh-level granularity, the entire mesh is processed even when only a small portion is visible.
- A large mesh with millions of vertices may execute vertex shading for almost the entire mesh, even if only a small fraction contributes to the final iamge.
- With a 2-million-vertex mesh where only 1% is visible, the remaining 99% of vertex processing is completely wasted.
- Vertex Pulling also removes the traditional index-buffer-driven vertex reuse mechanism, making hardware's vertex cache less effective.
- **Mesh Shaders introduce meshlet-level granularity**, allowing each meshlet to be processed independently as a small unit of geometry.
- **The primary motivation is fine-grained geometry processing**. The problem is not that Vertex Shaders are inefficient, the problem is that their granularity is to coarse.

The Scene: Dragon partially visible

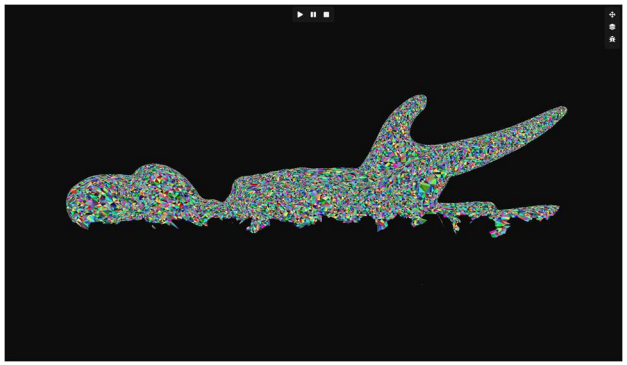


The Vertex Shader: Processes all triangles

The Task Shader: Culls Invisible Meshlets



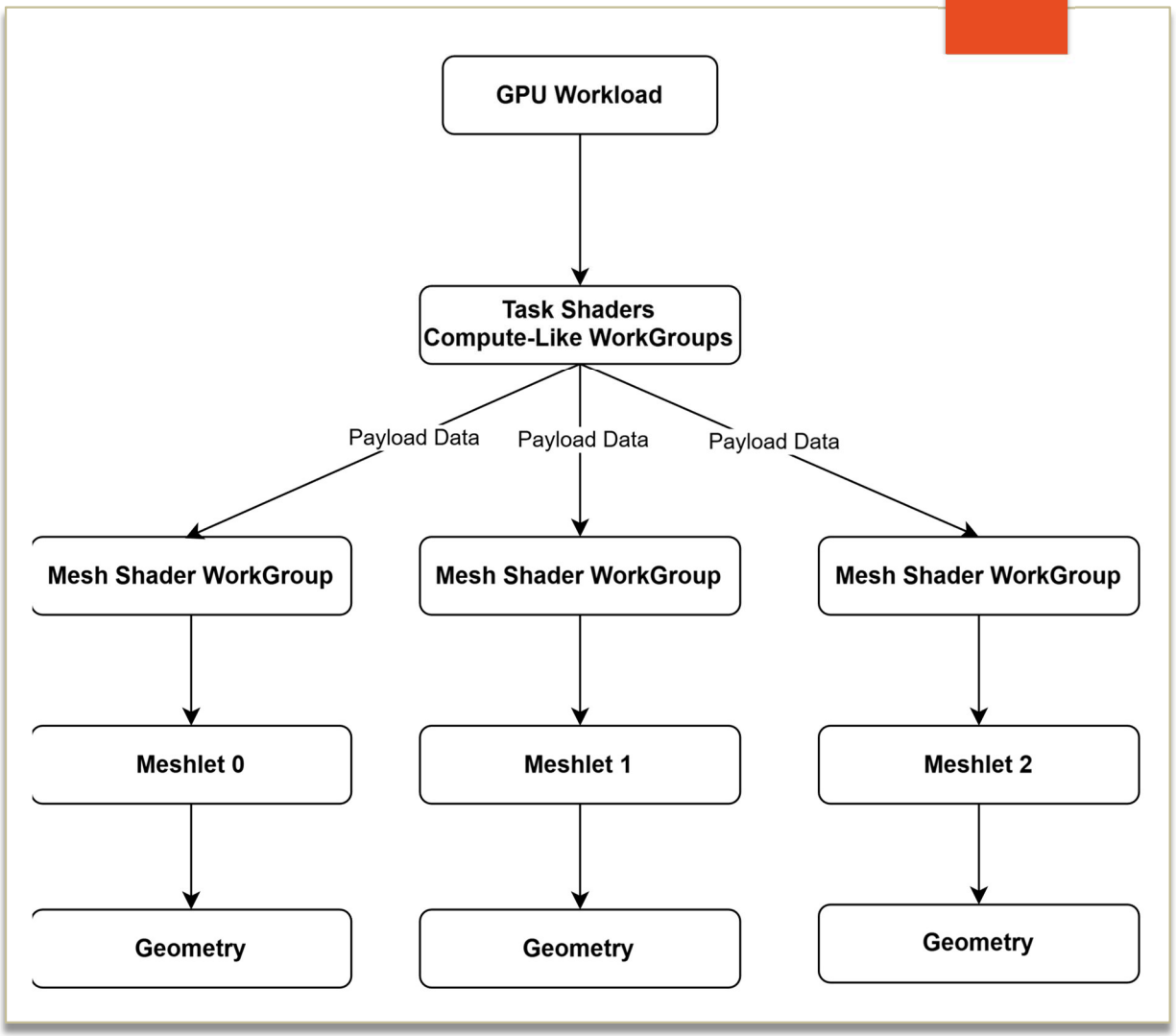
The Mesh Shader: Processes only visible triangles



Rendering IX.

Task & Mesh Shaders – A Compute-Based Graphics Pipeline

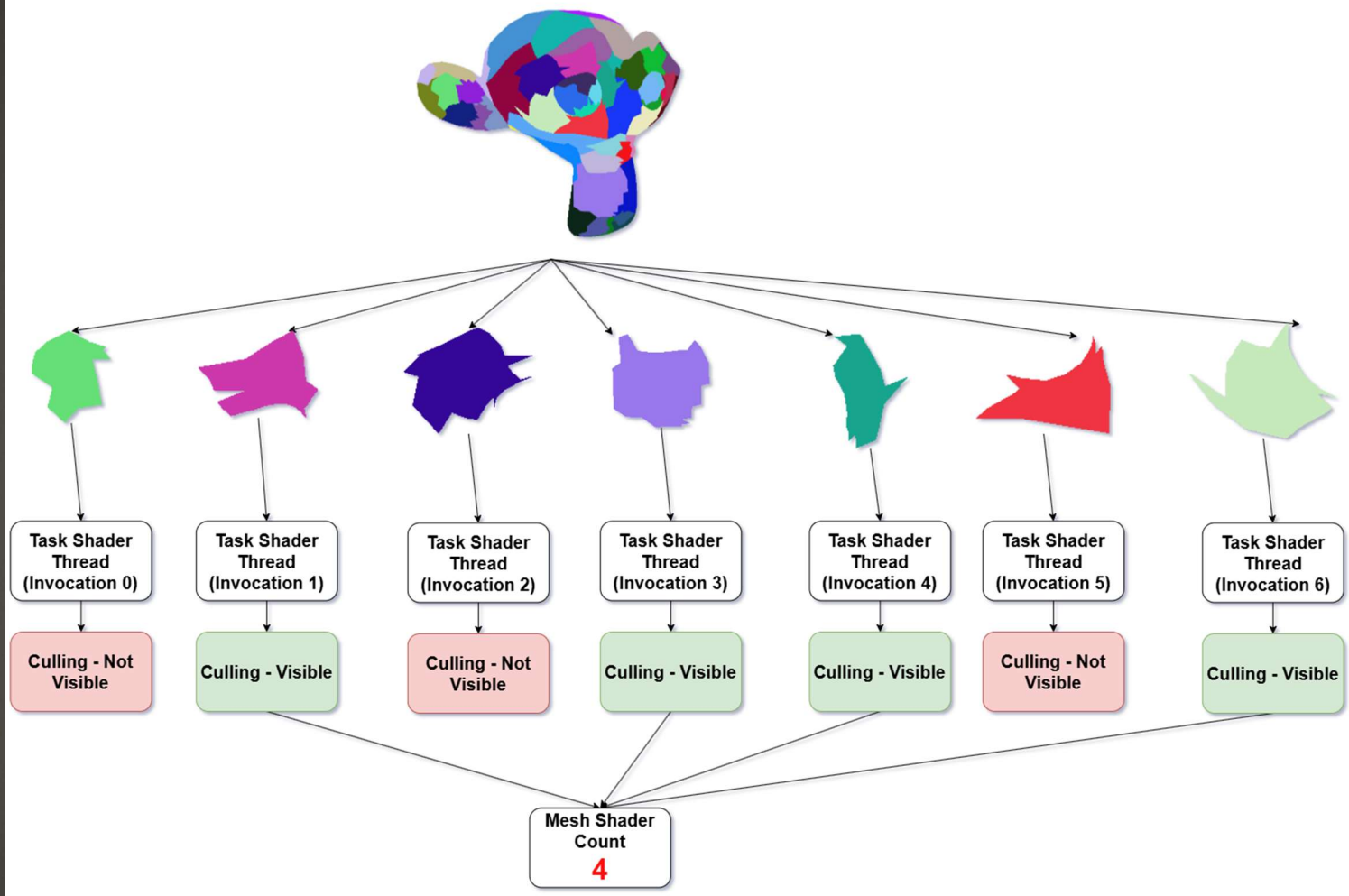
- Task and Mesh shaders are fundamentally compute-like shader stages, using workgroups and built-in invocation IDs to distribute work across threads.
- The **Task Shader** behaves similarly to a compute shader, but **can dynamically launch a variable number of Mesh Shader workgroups**.
- The **Mesh Shader** also follows the compute shader executuion model, but **can directly generate geometry by emitting vertices and primitives** instead of writing to a traditional vertex buffer.
- Together, they form a **hierarchical and dynamically expanding graphics pipeline**.
- Task Shaders can dynamically determine how many Mesh Shader workgroups should be launched making the pipeline naturally suited for GPU-driven workloads.
- Data can be passed from Task Shader groups to their spawned Mesh Shader groups through a Payload, typically containing compact information required to identify and process the assigned workload.
- This model enables fine-grained GPU control over geometry processing, where work can be dynamically distributed at meshlet granularity.



Rendering X.

Task Shader

- The Task Shader is launched through *vkCmdDrawMeshTasksIndirectEXT*, with the indirect command generated by the GPU-driven rendering pipeline.
- GroupCountX represents the number of visible mesh instances, while GroupCountY covers the meshlets belonging to each mesh in fixed-size blocks.
- The Task Shader reconstructs the entity, mesh and meshlet range directly from the built-in workgroup and invocation IDs.
- **In this representation:**
  - **gl_WorkGroupID.x**: identifies the current entity instance
  - **gl_WorkGroupID.y**: identifies a meshlet block
  - **gl_LocalInvocationID.x**: identifies the individual meshlet processed by the current Task Shader workgroup.
- The surviving meshlets are dynamically forwarded to Mesh Shader workgroups using **EmitMeshTaskEXT(count, 1, 1)**

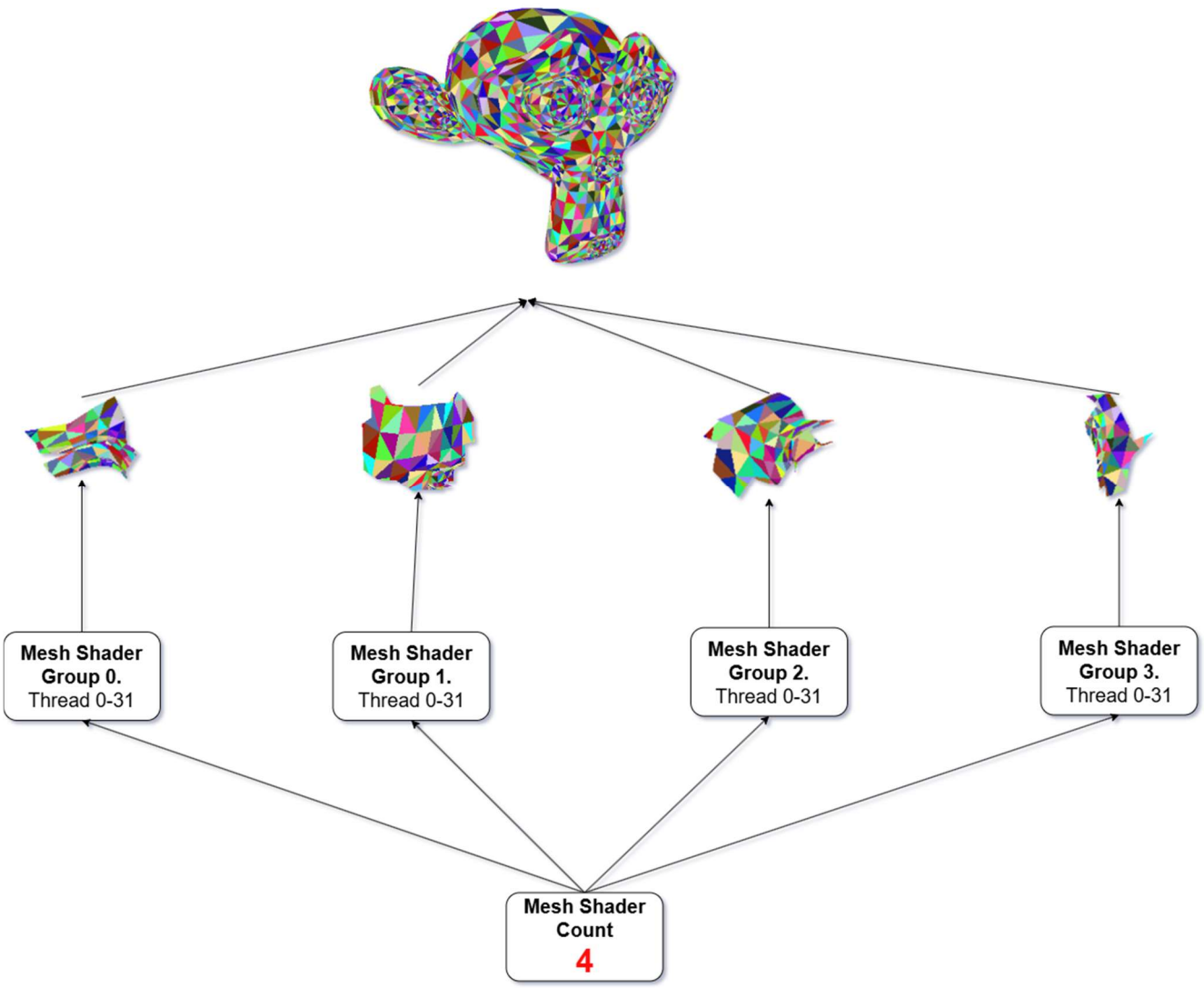


https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/Shaders/Passes/Shading/Common/Meshlet.task

Rendering XI.

Mesh Shader

- Each **Mesh Shader workgroup processes one meshlet collaboratively**, with 32 shader invocations working together.
- Meshlets are generated with a maximum of 64 vertices and 64 triangles, allowing the entire meshlet to be processed efficiently within a single workgroup.
- The workgroup distributes vertex and triangle processing across its invocation using **strided iteration +=32**.
- **SetMeshOutputsEXT** dynamically defines the number of vertices and triangles emitted by the Mesh Shader based on the processed meshlet.
- Vertex positions are emitted through *gl_MeshVerticesEXT[i].gl_Position*
- Triangle topology is emitted through *gl_PrimitiveTriangleIndicesEXT[i]*
- *gl_MeshPrimitivesEXT[i].gl_PrimitiveID* can be assigned to preserve the original primitive identity for later processing
- The Mesh Shader therefore **reconstructs and emits the complete meshlet geometry**, which is then passed directly to the rasterization pipeline.

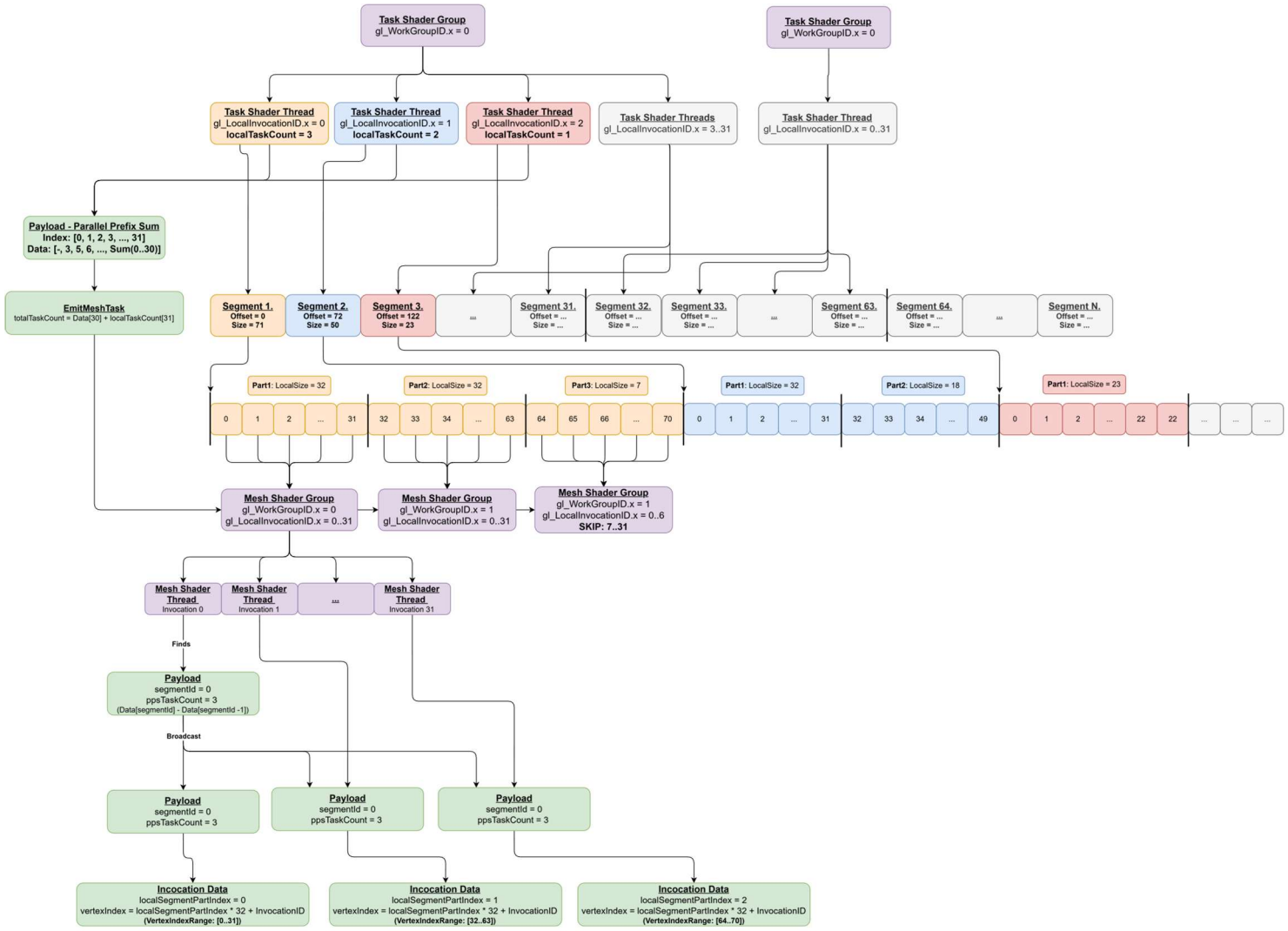


https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/Shaders/Passes/Shading/Common/Meshlet.mesh

Rendering XII.

Task & Mesh Shaders – Beyond Meshlet Rendering

- Task and Mesh Shaders are general-purpose programmable GPU stages, not limited to meshlet-based rendering.
- Mesh Shaders can optionally generate geometry, but the underlying mechanism can be used for arbitrary GPU-driven workloads.
- Possible applications include hair simulation, procedural terrain generation, dynamic geometry or GPU-based LOD generation, or grass and vegetation generation and more…
- Task and Mesh Shaders enable highly dynamic, hierarchical workloads where GPU threads can generate and process further GPU work.
- Understanding their execution model opens up a much broader design space for GPU-driven rendering and simulation.
- **When I first understood the Task and Mesh Shader model, it completely changed how I thought about GPU programming. I realized how powerful and general this mechanism actually is, and how many possibilities it opens up beyond traditional rendering.**



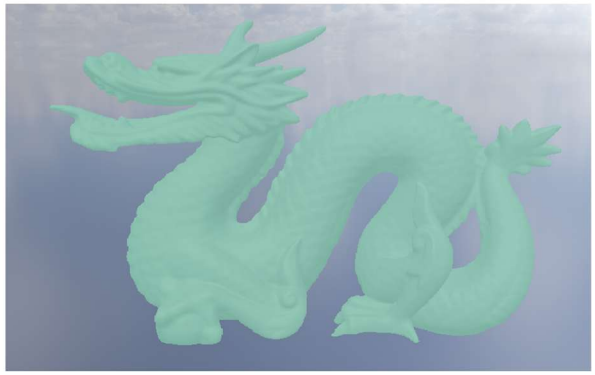
This is **Hair Generation** using Task/Mesh shader logic

Rendering XIII.

Material Management

- **Each material is loaded once and stored once**, avoiding duplicated material data across models.
- Materials are global resources, allowing multiple models to reference and reuse the same materials.
- A material contains both surface properties and references to its associated textures.
- Each texture can have its own sampler configuration, providing independent control over texture sampling.
- The **MaterialManager** stores all materials in a single global, data-oriented buffer accessible by the GPU.
- Meshes therefore only need to reference their material index, while the actual material data and resources remain globally managed.

```
struct SYN_API Material
{
    glm::vec4 color = glm::vec4(1.0f);
    glm::vec3 emissiveColor = glm::vec3(0.0f);
    float emissiveIntensity = 1.0f;
    glm::vec2 uvScale = glm::vec2(1.0f);
    float metalness = 0.0f;
    float roughness = 1.0f;
    float aoStrength = 1.0f;
    bool doubleSided = false;
    bool isTransparent = false;
    bool isAlphaTested = false;

    float clearcoatFactor = 0.0f;
    float clearcoatRoughness = 0.0f;
    float ior = 1.5f;
    float specularFactor = 1.0f;
    glm::vec3 specularColor = glm::vec3(1.0f);

    uint32_t albedoTexture = UINT32_MAX;
    uint32_t albedoSampler = UINT32_MAX;

    uint32_t normalTexture = UINT32_MAX;
    uint32_t normalSampler = UINT32_MAX;

    uint32_t metalnessTexture = UINT32_MAX;
    uint32_t metalnessSampler = UINT32_MAX;

    uint32_t roughnessTexture = UINT32_MAX;
    uint32_t roughnessSampler = UINT32_MAX;

    uint32_t metallicRoughnessTexture = UINT32_MAX;
    uint32_t metallicRoughnessSampler = UINT32_MAX;

    uint32_t emissiveTexture = UINT32_MAX;
    uint32_t emissiveSampler = UINT32_MAX;

    uint32_t ambientOcclusionTexture = UINT32_MAX;
    uint32_t ambientOcclusionSampler = UINT32_MAX;

    uint32_t opacityTexture = UINT32_MAX;
    uint32_t opacitySampler = UINT32_MAX;

    uint32_t clearcoatTexture = UINT32_MAX;
    uint32_t clearcoatSampler = UINT32_MAX;

    uint32_t clearcoatRoughnessTexture = UINT32_MAX;
    uint32_t clearcoatRoughnessSampler = UINT32_MAX;

    uint32_t clearcoatNormalTexture = UINT32_MAX;
    uint32_t clearcoatNormalSampler = UINT32_MAX;

    uint32_t specularTexture = UINT32_MAX;
    uint32_t specularSampler = UINT32_MAX;

    uint32_t specularColorTexture = UINT32_MAX;
    uint32_t specularColorSampler = UINT32_MAX;
};
```



https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/Material/Material.h

Rendering XIV.

Textures & Samplers

- **All textures and samplers are stored in a global descriptor arrays**, allowing materials to reference them through compact indices.
- **Descriptor Indexing** allows descriptor array *(VK_KHR_descriptor_indexing)* to be dynamically sized instead of requiring a fixed array size at shader compilation time.
- The engine uses traditional Vulkan descriptors only for global textures and samplers:
  - Set 0 | Binding 0 = Samplers
  - Set 0 | Binding 1 = Textures
- Textures and samplers are stored separately, allowing the shader to dynamically combine them at runtime using their repective indices.
- Materials therefore store only compact texture and sampler IDs, which are resolved thorugh the global descriptor arrays.
- Other shader-specific resources, such as deferred rendering targets, are handled separately using push descriptors.
- While this approach enables flexible bindless resource access, managing large descriptor arrays still introducces additional complexity, motivating a more direct GPU resource management approach

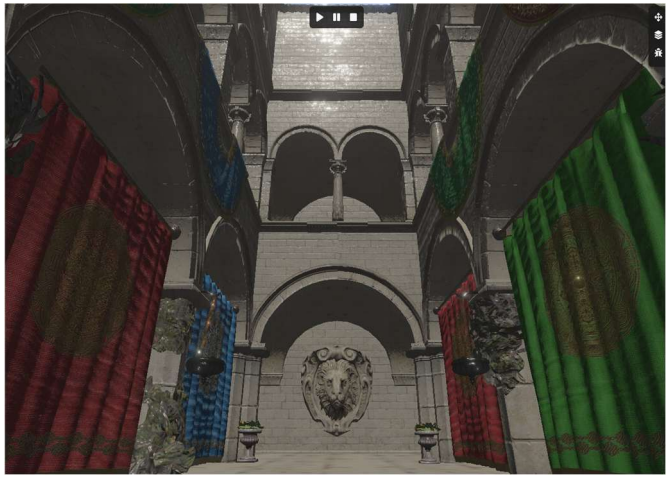
```
// Shader Side

layout(set = 0, binding = 0) uniform sampler globalSamplers[];
layout(set = 0, binding = 1) uniform texture2D bindlessTextures[];

#define SAMPLER_LINEAR_REPEAT        0
#define SAMPLER_LINEAR_CLAMP_EDGE    1
#define SAMPLER_NEAREST_REPEAT       2
#define SAMPLER_NEAREST_CLAMP_EDGE   3
#define SAMPLER_LINEAR_ANISO         4
#define SAMPLER_NEAREST_ANISO        5
#define SAMPLER_MAX_REDUCTION        6
#define SAMPLER_BLOOM                7
#define SAMPLER_SHADOW               8

vec4 SampleTexture2D(uint textureID, uint samplerID, vec2 uv) {
    return texture(
        sampler2D(
            bindlessTextures[nonuniformEXT(textureID)],
            globalSamplers[nonuniformEXT(samplerID)]
        ),
        uv
    );
}

// Cpp Side

void DescriptorWriter::UpdateSet(VkDescriptorSet set) {
    auto device = ServiceLocator::Get<Vk::Context>()->GetDevice();

    std::vector<VkWriteDescriptorSet> writes;
    writes.reserve(_imageWrites.size() + _bufferWrites.size());

    for (const auto& imgData : _imageWrites) {
        VkWriteDescriptorSet write{ VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET };
        write.dstSet = set;
        write.dstBinding = imgData.binding;
        write.dstArrayElement = imgData.arrayElement;
        write.descriptorCount = 1;
        write.descriptorType = imgData.type;
        write.pImageInfo = &imgData.info;
        writes.push_back(write);
    }

    //...

    if (!writes.empty()) {
        vkUpdateDescriptorSets(device->Handle(), static_cast<uint32_t>(writes.size()), writes.data(), 0, nullptr);
    }

    _imageWrites.clear();
    _bufferWrites.clear();
}
```

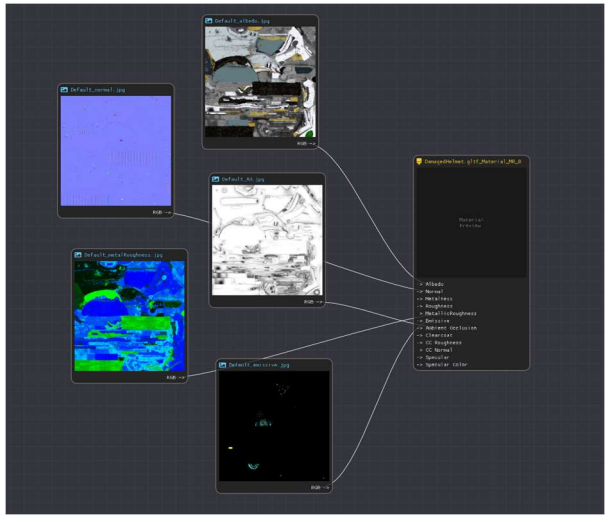
https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/Vk/Descriptor/DescriptorWriter.cpp

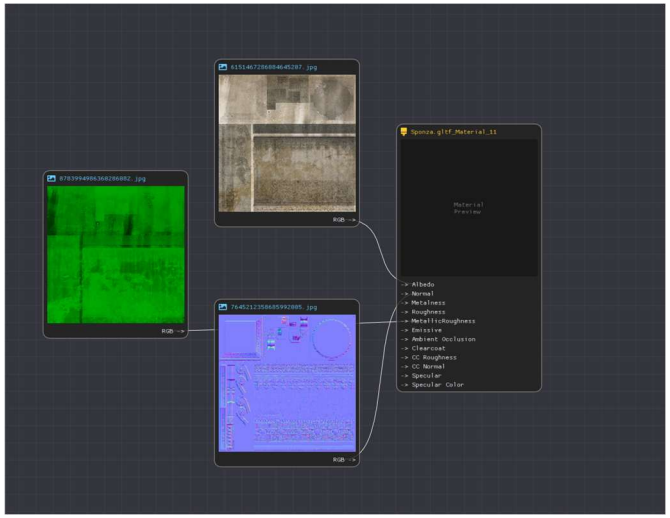
Rendering XV.

Descriptor Buffer & Descriptor Heap

- ***VK_EXT_descriptor_buffer*** makes descriptors directly resident in shader-accessible memory, allowing descriptors data to be managed explicitly alongside regular GPU buffers.
- Descriptor data can be updated through direct memory operations, removing the need for traditional descriptor pool and set management.
- This signlificantly simplifies resource binding and fits naturally into a data-oriented, GPU-driven architecture.
- ***VK_EXT_descriptor_heap*** introduces a newer descriptor management model with explicit resource and sampler heap, conceptually closer to the Directx 12 descriptor heap model.
- The engine currently uses the descriptor buffer approach, while the descriptor heap architecture is planned as the long-term direction for the Vulkan backend and future DirectX 12 backend integration.
- The overall goal is to move resource binding away from driver-managed descriptor objects **toward explicit, GPU-accessible descriptor memory management**.





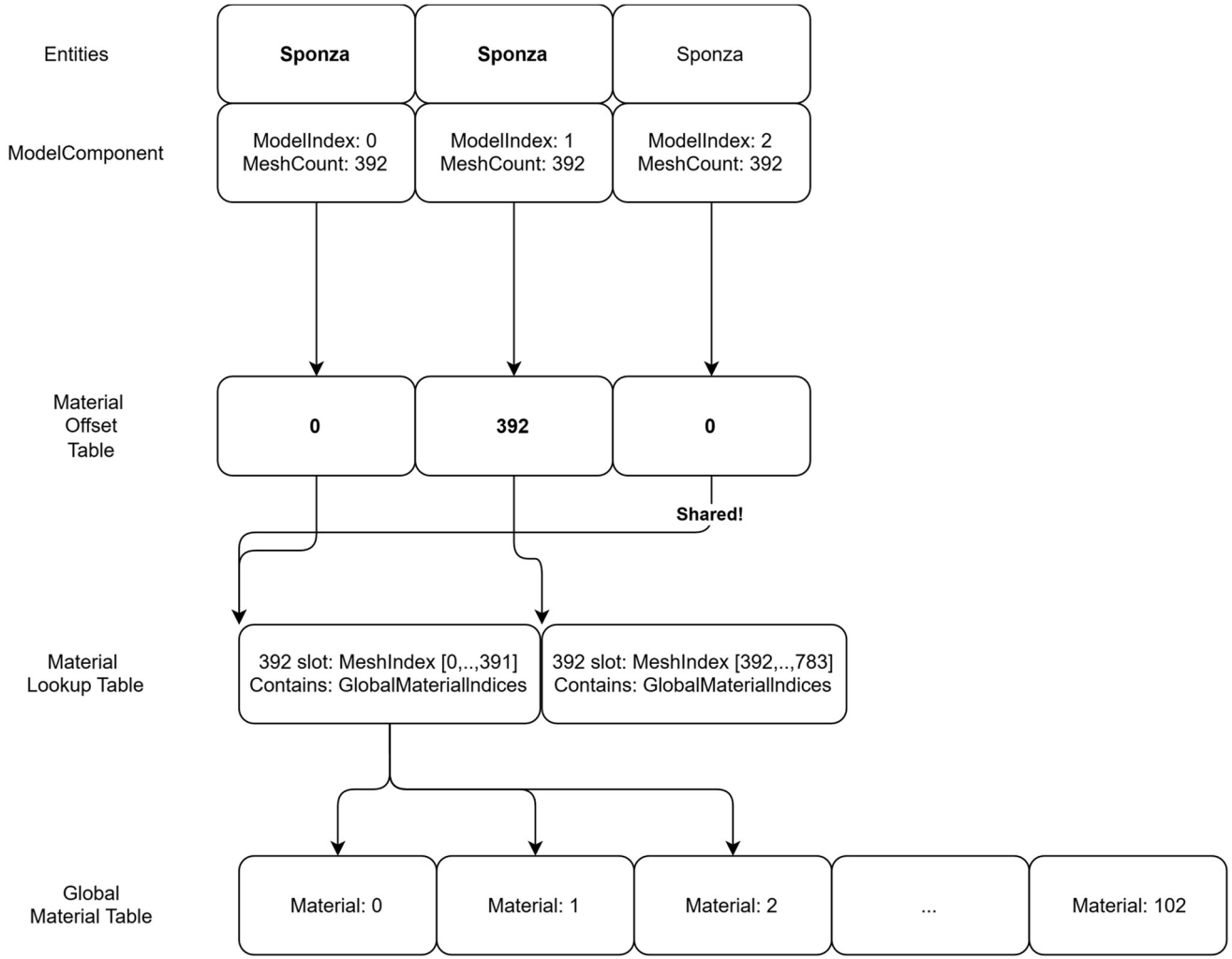


https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/Vk/Descriptor/DescriptorBuffer.cpp

Rendering XVI.

Bindless Material Table

- A **two-level indirection table** provides a fully generalized and flexible Mesh to Material relationship.
- The system allows per-mesh material overrides, so individual meshes of a model can easily use completely different materials.
- Each ModelComponent (ECS) stores a material offset pointing into the Material Lookup Table.
- The lookup table reserves a contiguous range of entries for each model, based on its total mesh count.
- Multiple models can **share the same lookup range by using the same offset**, while different offsets provide dedicated material mappings.
- Each lookup entry contains a global material index, resolving the mesh to the correspoing material in the Global Material Table.
- Each **vertex stores its 16-bit mesh index**, which is comined with the model's material offset to resolve the final material: materialOffset + meshIndex

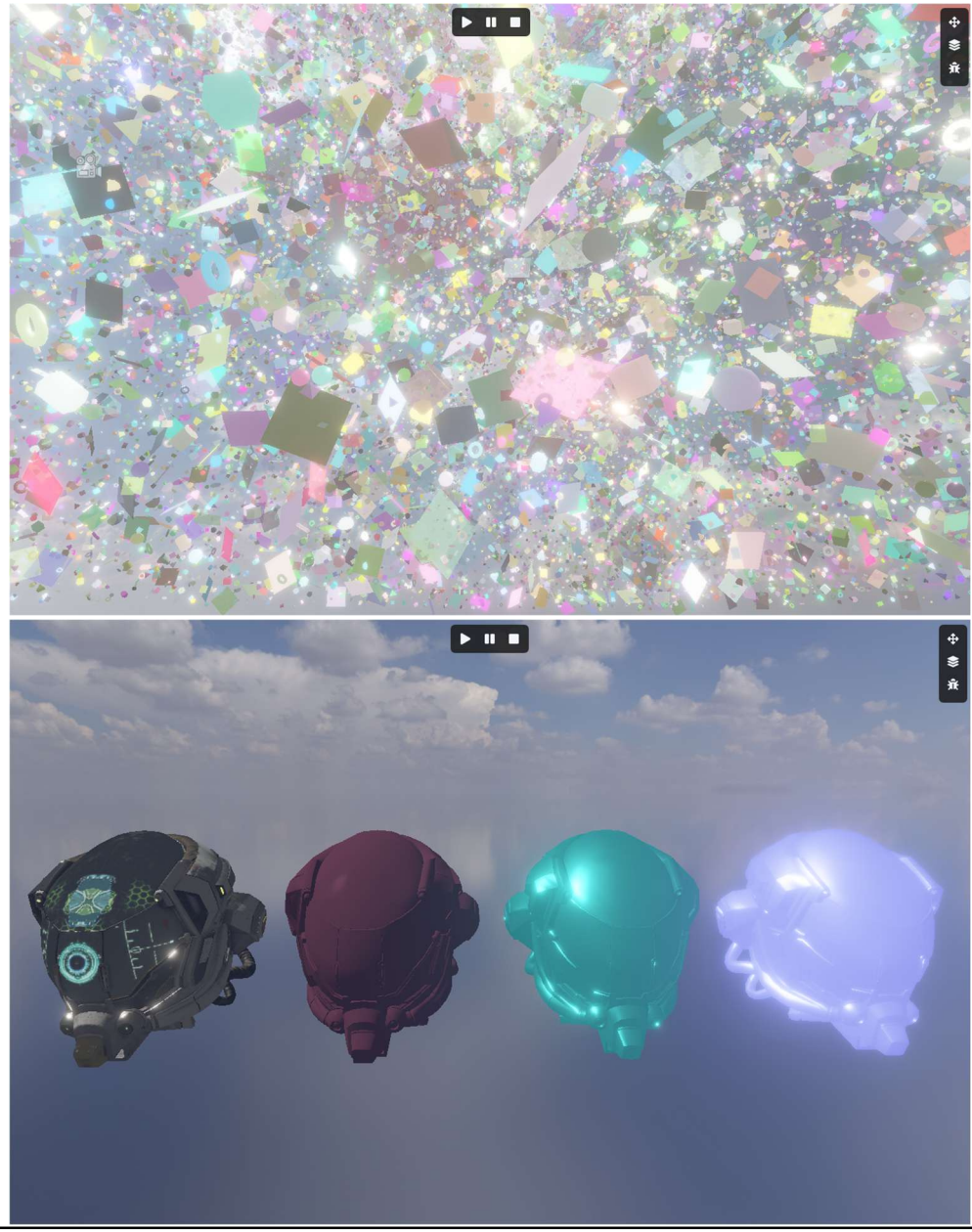


https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/System/Rendering/MaterialSystem.cpp

Rendering XVII.

Material Override & Flexibility

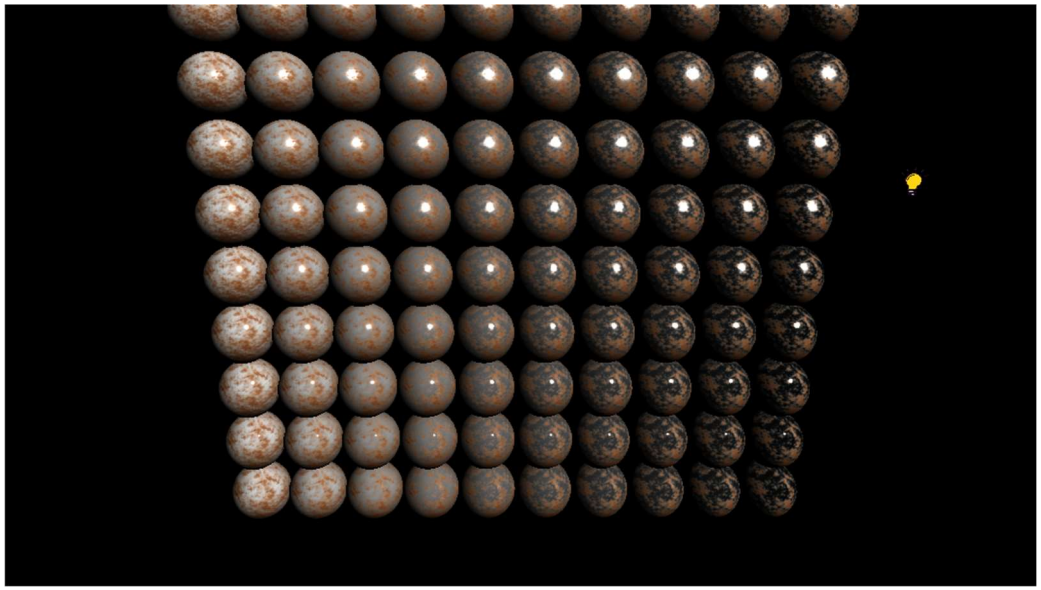
- Fully indirect rendering allows 100.000 diffrerrent procedural shape instances to be rendered with a single draw call, while each instance can still reference a different material.
- The two-level material indirection enables per-mesh material assignment without breaking the GPU-driven rendering model.
- Complex models can be overriden at mesh granularity, allowing individual parts of a model to use completely different materials.
- Material overrides require no changes to the underlying model or mesh data, only the material lookup mapping is modified.
- The main overhead is maintaining the lookup table when material assignments are added, removed, or changed.
- This approach provides a powerful separation between geometry and material assignment, making the rendering architecture highly flexible

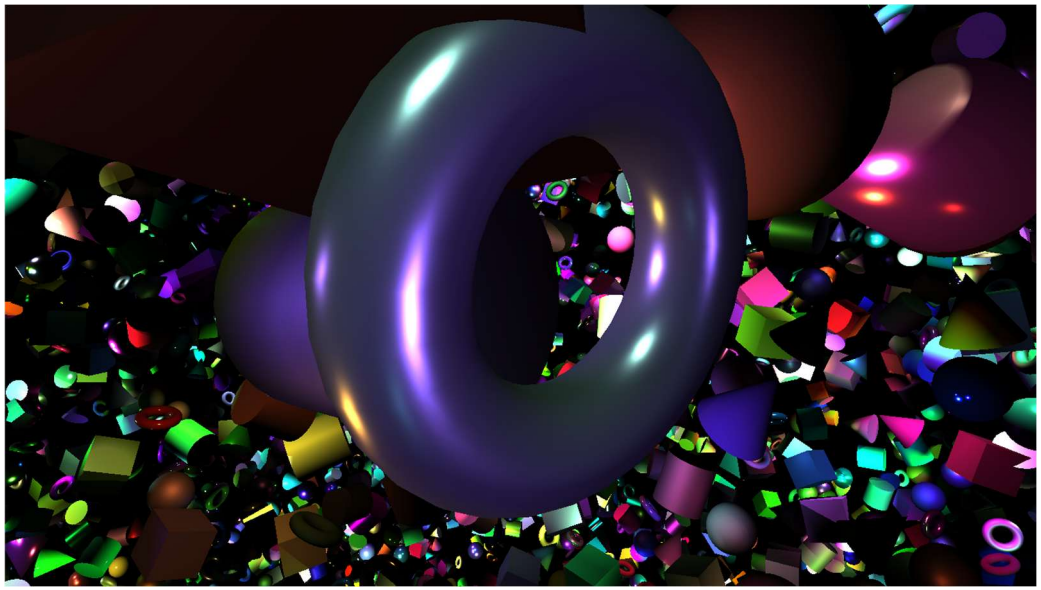


Rendering XVIII.

Physically Based Rendering

- The renderer currenty uses a **metallic-roughness based PBR** material model.
- **Fresnel-Schlick approximation** is used to model view-dependent specular reflectance and determine the surface reflectance at grazing angles.
- The microfacet specular BRDF uses **GGX Normal Distribution Function** for realistic rough-surface reflection.
- **Smith correlated visibility** is used to account for microfacet masking and shadowing effects in specular response.
- The base layer combines energy-conserving diffure and specular reflections, with metallic surfaces transitioning from diffure-dominant to specular-dominant behaviour.
- The material model supports an additional Clearcoat layer, following the layered-surface approach used by modern PBR models such as the Disney Principled BSDF.
- The clearcoat layer uses its own GGX distribution, Fresnel response, roughness and normal, allowing materials such as coated plastics, varnished surfaces and painted materials to be represented.



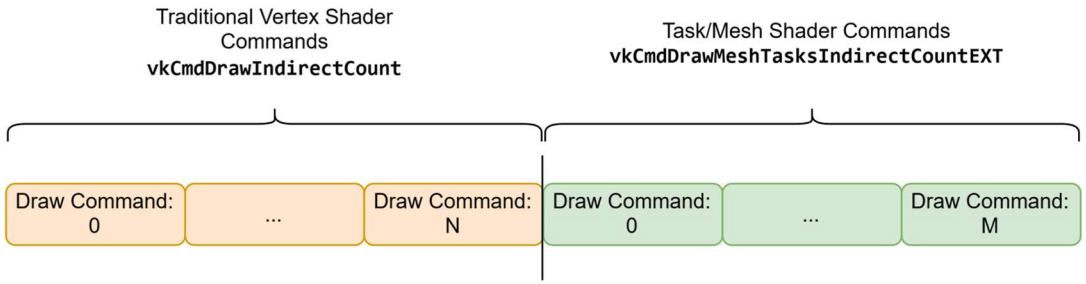


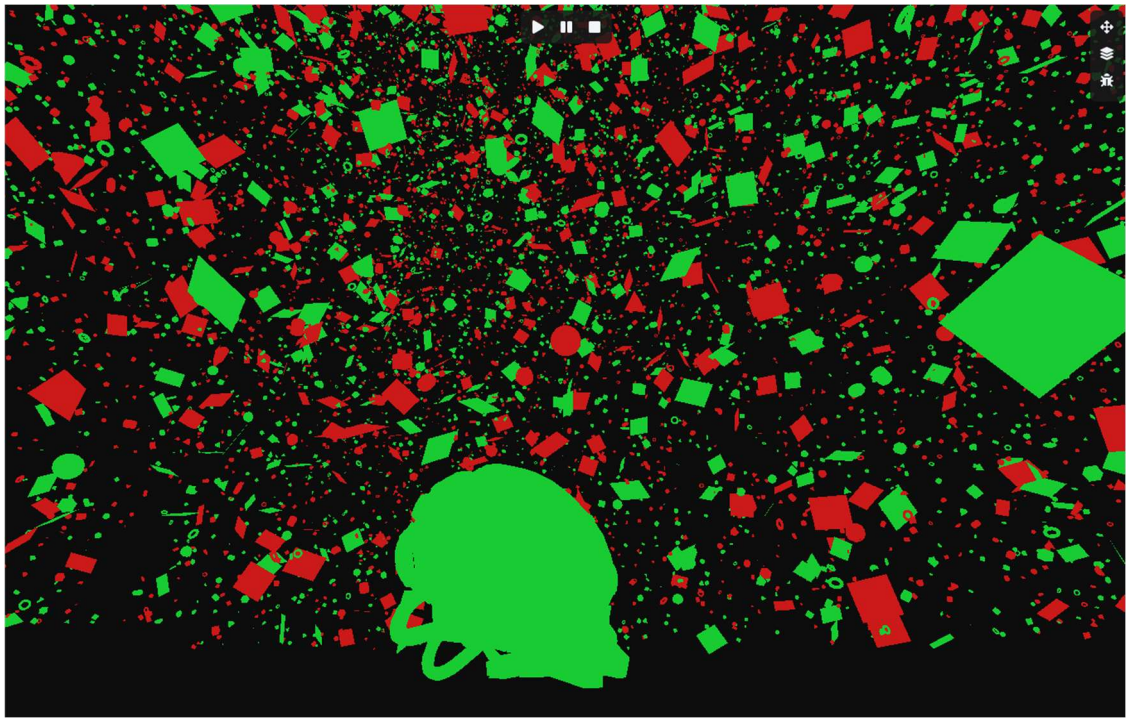
https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/Shaders/Includes/Utils/PbrMath.glsl

Rendering XIX.

2-Bucket Rendering

- **Buckets group meshes with identical rendering properties** that affect the rasterization pipeline and therefore require separate rendering paths.
- Each bucket still represents one complete GPU-driven render pass: all models, meshes, instances, and LOD levels assigned to the bucket are rendered with single indirect draw call.
- The **two buckets** separate the traditional Vertex Shader pipeline from the Task/Mesh Shader pipeline, as they require fundamentally different rendering commands and shader stages.
  - **Vertex Shader Bucket:** uses *vkCmdDrawIndirectCount*.
  - **Task/Mesh Shader Bucket:** uses *vkCmdDrawMeshTasksIndirectCount EXT*.
- The engine allows the rendering pipeline to be selected per mesh and per LOD level, enabling different geometry pipelines within the same scene.
- Draw commands and their corresponding descriptors are batched independently per bucket, allowing each rendering path to process its workload with a single GPU-driven render call.



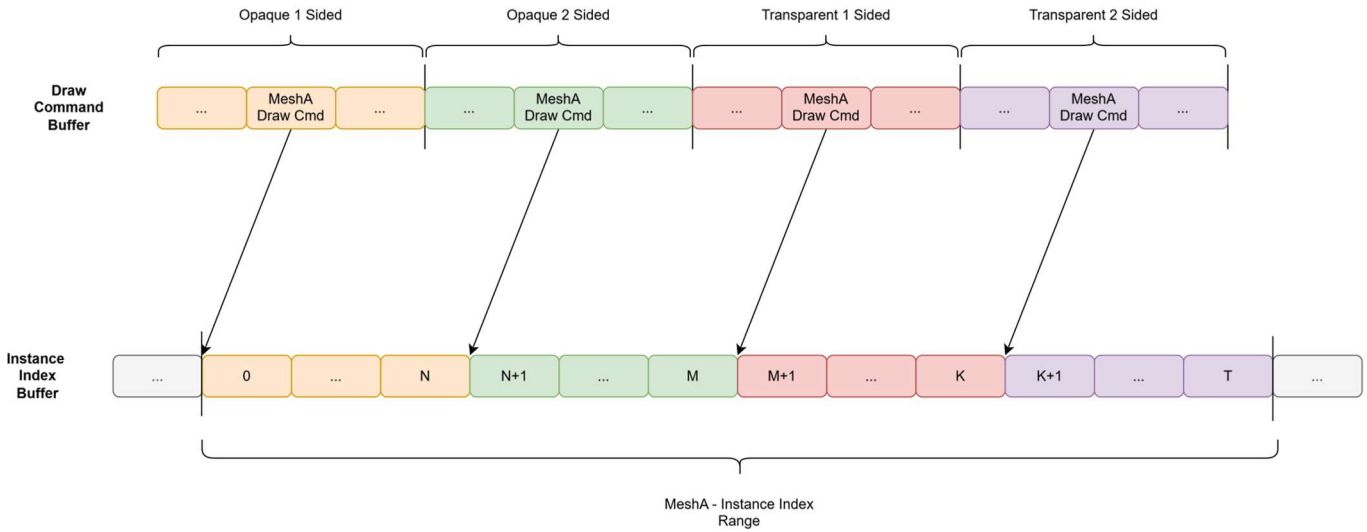


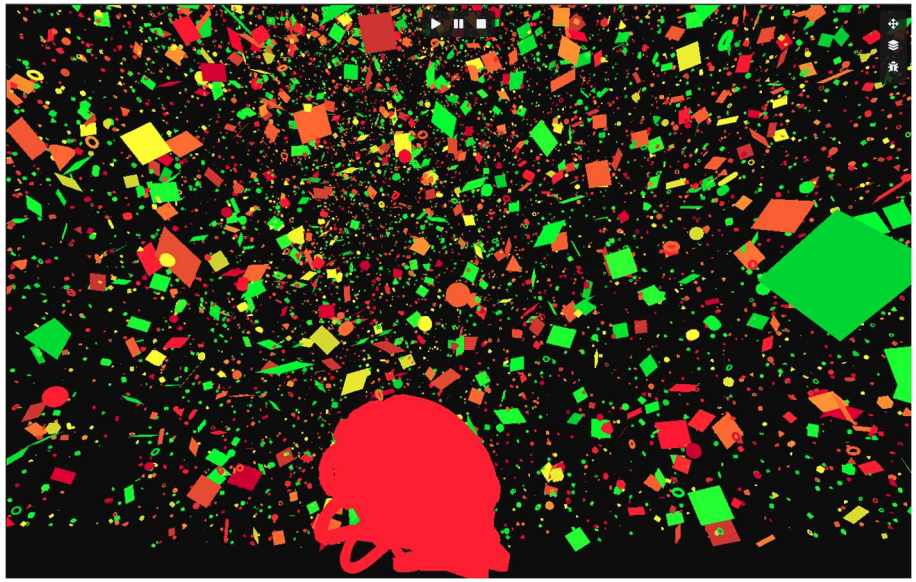
Red = Vertex Shader Meshes | Green = Mesh Shader Meshes

Rendering XX.

4-Bucket Rendering

- Materials are divided into **four rendering categories** based on material properties that require different pipeline and rasterization states.
  - **Opaque 1-Sided:** Back-face culling enabled with *VK_CULL_MODE_BACK_BIT*.
  - **Opaque 2-Sided:** Back-face culling disabled with *VK_CULL_MODE_NONE*.
  - **Transparent 1-Sided:** Uses a separate **WBOIT** pipeline with *VK_CULL_MODE_BACK_BIT*.
  - **Transparent 2-Sided:** Uses the **WBOIT** pipeline with *VK_CULL_MODE_NONE*.
- Draw commands remain fully batched, but are stored in four independent ranges, allowing each bucket to be rendered separately.
- The instance buffer is **pre-partitioned per mesh and bucket** based on the maximum expected model distribution, enabling memory-efficient allocation and fully parallel instance insertion using atomic counters.



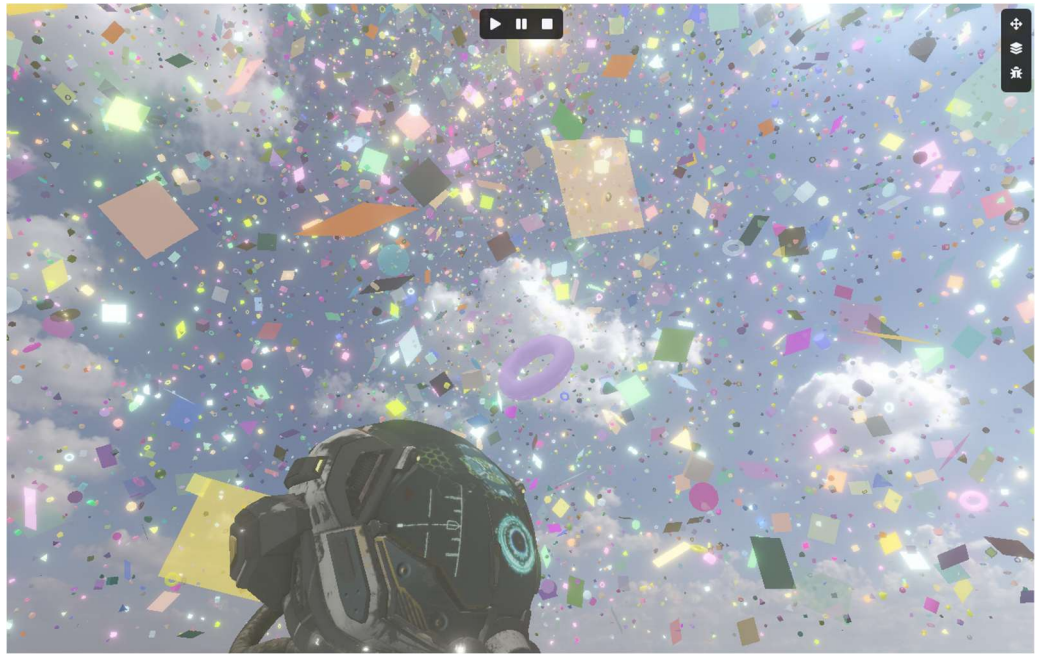


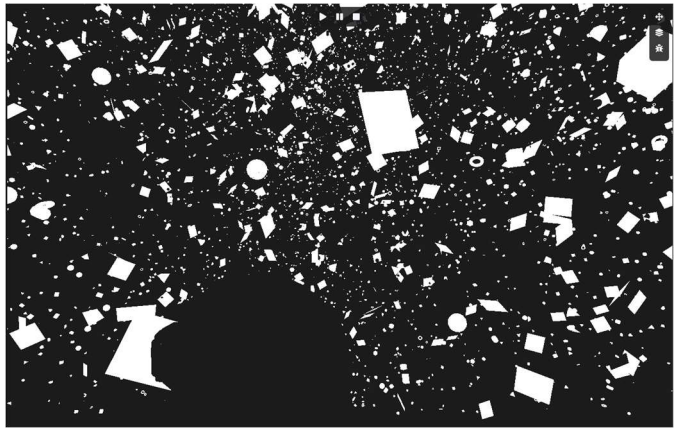
Red/Orange/Yellow/Green = Opaqe1/Opaque2/Trans1/Trans2

Rendering XXI.

Transparent Rendering - WBOIT

- **Transparent rendering is inherently difficult**, as transparent fragments must be composited in the correct depth order.
- Traditional depth sorting is expensive and does not scale well, especially with large numbers of overlapping transparent objects.
- **Weighted Blended Order-Independent Transparency (WBOIT)** avoids explicit sorting by accumulating transparent fragments in parallel using weighted blending.
- Two render targets are used:
  - **Accumulation Buffer:** accumulates the weighted transparent colors and their contributions.
  - **Revealage Buffer:** tracks how much of the underlying opaque surface remains visible through the transparent layers.
- The transparent fragments are therefore processed independently of their exact draw order, making the approach highly scalable for GPU-driven rendering.
- A final composite pass combines the accumulated transparent result with the opaque scene color to produce the final image.
- WBOIT provides a very good quality-to-performance trade-off, although it is an approximation and does not reproduce exact depth-sorted transparency.





Color Accumulation Texture

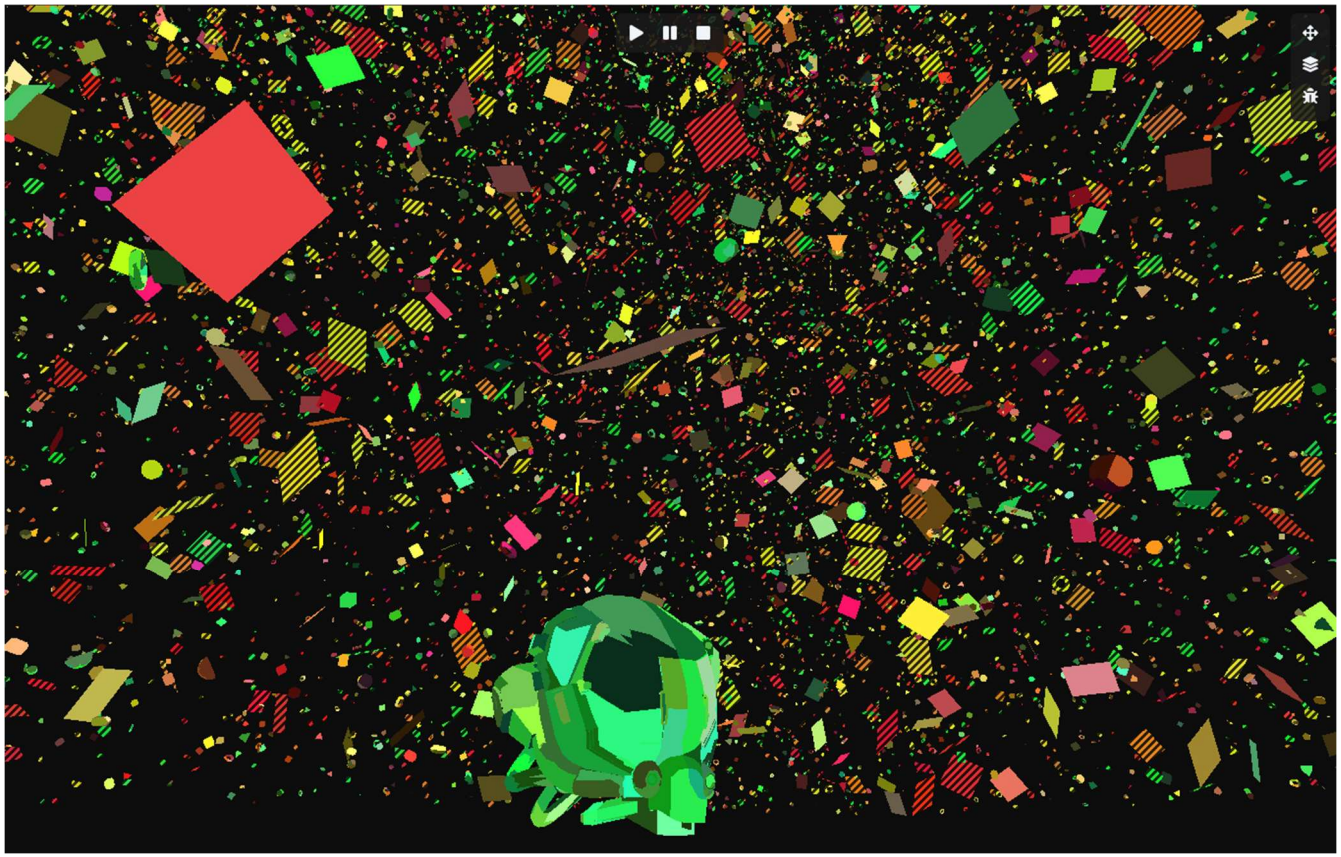
(Its white becouse it exceeds 1.0f)

Revelage Texture

Rendering XXII.

8-Bucket Rendering

- Combines the previous **2-bucket** and **4-bucket** rendering architectures.
- The first-level split separates **Traditional Vertex Shader** and **Task/Mesh Shader** rendering, keeping identical draw commands and descriptors grouped together.
- Each rendering pipeline is then further divided into **four material-based buckets**, resulting in eight independent buckets overall.
- The instance index buffer is partitioned into **eight** independent regions, one for each bucket.



Hatched regions represent the Traditional Vertex Shader pipeline, solid regions represent the Mesh Shader pipeline, and colors indicate the material type.

Rendering XXIII.

16-Bucket Rendering

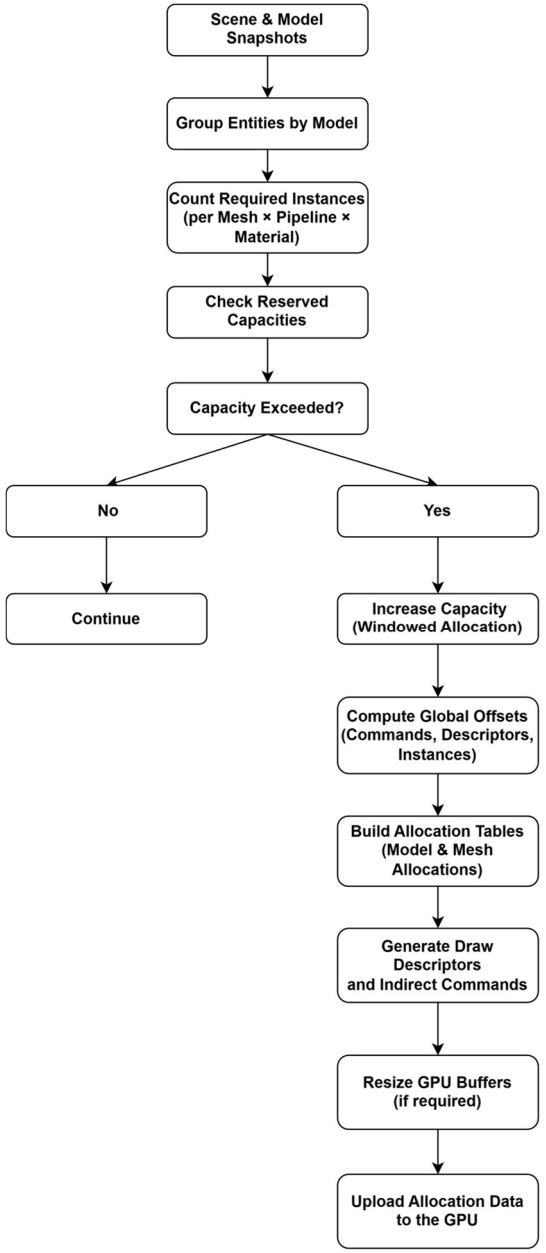
- Extends the **8-bucket architecture** by further separating **Alpha-Tested** and **non-Alpha-Tested** materials.
- Alpha-tested materials, such as foliage, require a discard operation in the fragment shader to reject transparent fragments.
- Fragment shader discard can prevent the GPU from fully exploiting **Early Fragment Tests**, potentially causing significantly more fragment shader invocations.
- For non-alpha-tested materials, this logic is unnecessary, allowing Early Fragment Tests to be explicitly enabled for better efficiency.
- The final **16-bucket architecture** therefore combines **shader pipeline type**, **material type**, and **alpha testing state**.
- The shader implementation remains shared, with a **compile-time define** selecting either the alpha-test discard path or explicitly enabling Early Fragment Tests.
- This keeps the shader code unified while ensuring that each bucket uses the most appropriate rendering path.

```
#version 460
#extension GL_EXT_mesh_shader : require
#extension GL_GOOGLE_include_directive : require

layout(location = 0) in vec2 inUV;
layout(location = 1) in flat uint inMaterialId;

#ifdef ENABLE_ALPHA_TEST
#include "../../../Includes/Core.glsl"
#include "../../../Includes/Common/FrameGlobalContext.glsl"
#include "../../../Includes/Common/Material.glsl"
#include "../../../Includes/Common/Texture.glsl"
#include "../../../Includes/Utils/MaterialMath.glsl"
#include "../../../Includes/PushConstants/DirectionLightShadowTraditionalMeshletPassPC.glsl"

layout(push_constant) uniform PushConstants {
   DirectionLightShadowTraditionalMeshletPassPC pc;
};
#endif

#ifndef ENABLE_ALPHA_TEST
layout(early_fragment_tests) in;
#endif

void main() {
    #ifdef ENABLE_ALPHA_TEST

    FrameGlobalContext ctx = GET_FRAME_CONTEXT(pc.frameGlobalContextBufferAddr);

    // 1. Fetch Material
    Material mat = GET_MATERIAL(ctx.materialBufferAddr, inMaterialId);
    vec2 finalUV = inUV * mat.uvScale;

    // 2. Evaluate Albedo & Alpha
    vec4 albedoAlpha = EvaluateAlbedoAlpha(ctx.textureMetadataBufferAddr, mat, finalUV);
    if (IS_ALPHA_TESTED(mat) && albedoAlpha.a < ctx.alphaLimitDiscard) {
        discard;
    }

    #endif
}
```

Rendering XXIV.

Render System

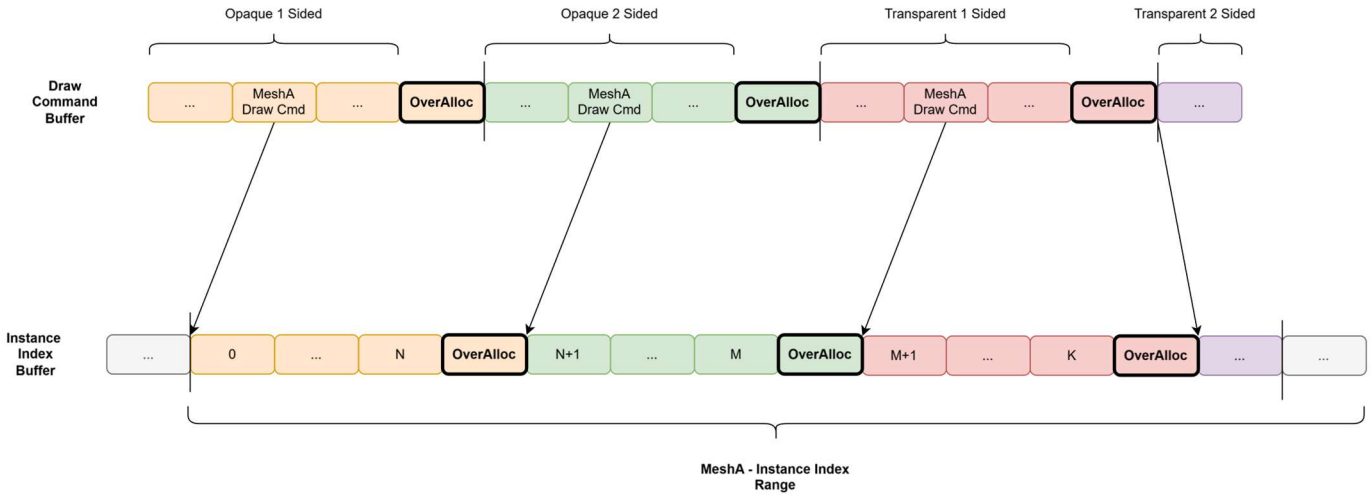
- The Render System maintains a **single batched draw command buffer**, partitioned into the 16 rendering buckets.
- A single **batched instance index buffer** stores instance data for all meshes and buckets.
- Each mesh is assigned a predefined region in the instance buffer, allowing the GPU to resolve visible instances without requiring a separate buffer per mesh.
- The Render System iterates over the active scene and tracks the required capacity for each mesh and rendering bucket.
- When a bucket exceeds its allocated capacity, the underlying buffers are reallocated and resized.
- Reallocation is intentionally infrequent: capacity is allocated using an **over-allocation strategy**, providing additional space for future growth.
- This keeps the steady-state rendering path stable while avoiding frequent GPU buffer reallocations.



Rendering XXV.

Windowed Buffer Allocation

- All allocations use **over-allocation windows** to avoid frequent buffer rebuilds.
- **Draw command regions** are over-allocated per bucket. If the material distribution changes, additional draw commands can be accommodated without immediately rebuilding the entire layout.
- *vkCmdDrawIndirectCount* allows the unused portion of the over-allocated command regions to remain inactive by limiting execution to the actual number of active commands.
- **Instance index regions** are also over-allocated:
  - A region is reserved for each mesh.
  - Within each mesh, the region is further divided between the individual rendering buckets.
- Adding or removing entities using a mesh therefore does not require rebuilding the global buffer layout as long as the reserved capacity is sufficient.
- The **over-allocation window can be tuned per mesh**:
  - Meshes with a stable, low instance count can use a small window.
  - Highly dynamic meshes, where instances are frequently added or removed, can use a larger window to avoid repeated reallocations.
- This provides a balance between **memory overhead and allocation stability**, adapting the buffer layout to the actual behavior of individual meshes.



https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/System/Rendering/RenderSystem.cpp