

Synapse Engine:

A research oriented
modern AAA game engine

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

2026

Introduction I.

About me:

- ▶ 23 years old
- ▶ **M.Sc. Student** in Computer Engineering at Budapest University of Technology 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 exploring 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.



Introduction 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 enthusiasts understand how modern game engines 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.

Introduction 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 implementations.
- Developed my first data-oriented OpenGL engine as my B.Sc. Thesis, which is publicly available on GitHub and received a special award.
- Started exploring Vulkan during my M.Sc. Studies and built the first version of Synapse Engine while learning modern techniques. (The original 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 significantly accelerated using AI (Gemini Pro Student), but only after establishing a deep understanding of the underlying concepts and a clear architectural vision.



Introduction IV.

Acknowledgements:

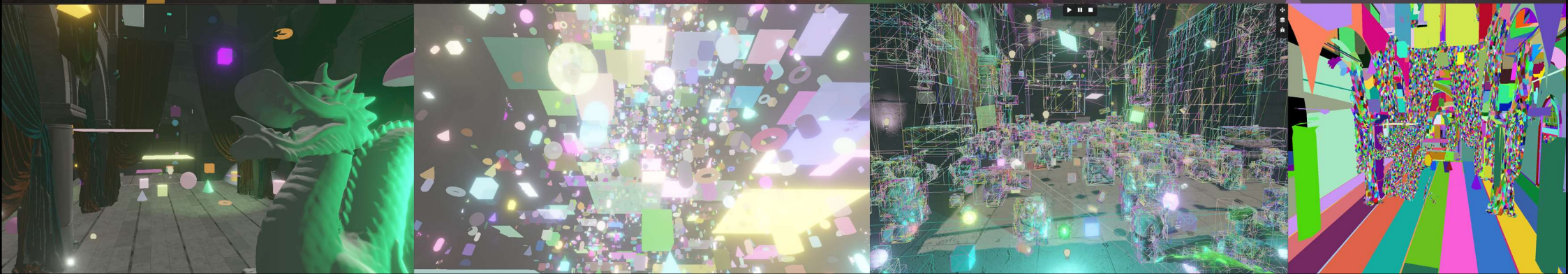
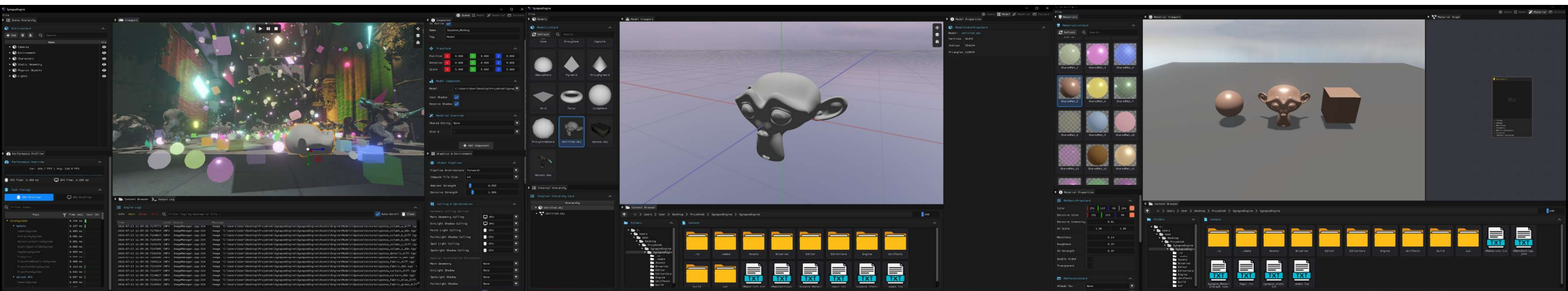
- ▶ **LearnOpenGL** (<https://learnopengl.com/>): An outstanding introduction to OpenGL and real-time graphics.
- ▶ **Vulkan Tutorial** (<https://vulkan-tutorial.com/>): A fantastic resource for learning Vulkan from the ground up.
- ▶ **Vulkan Guide** (<https://vkguide.dev/>): An invaluable reference 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 fundamentally 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

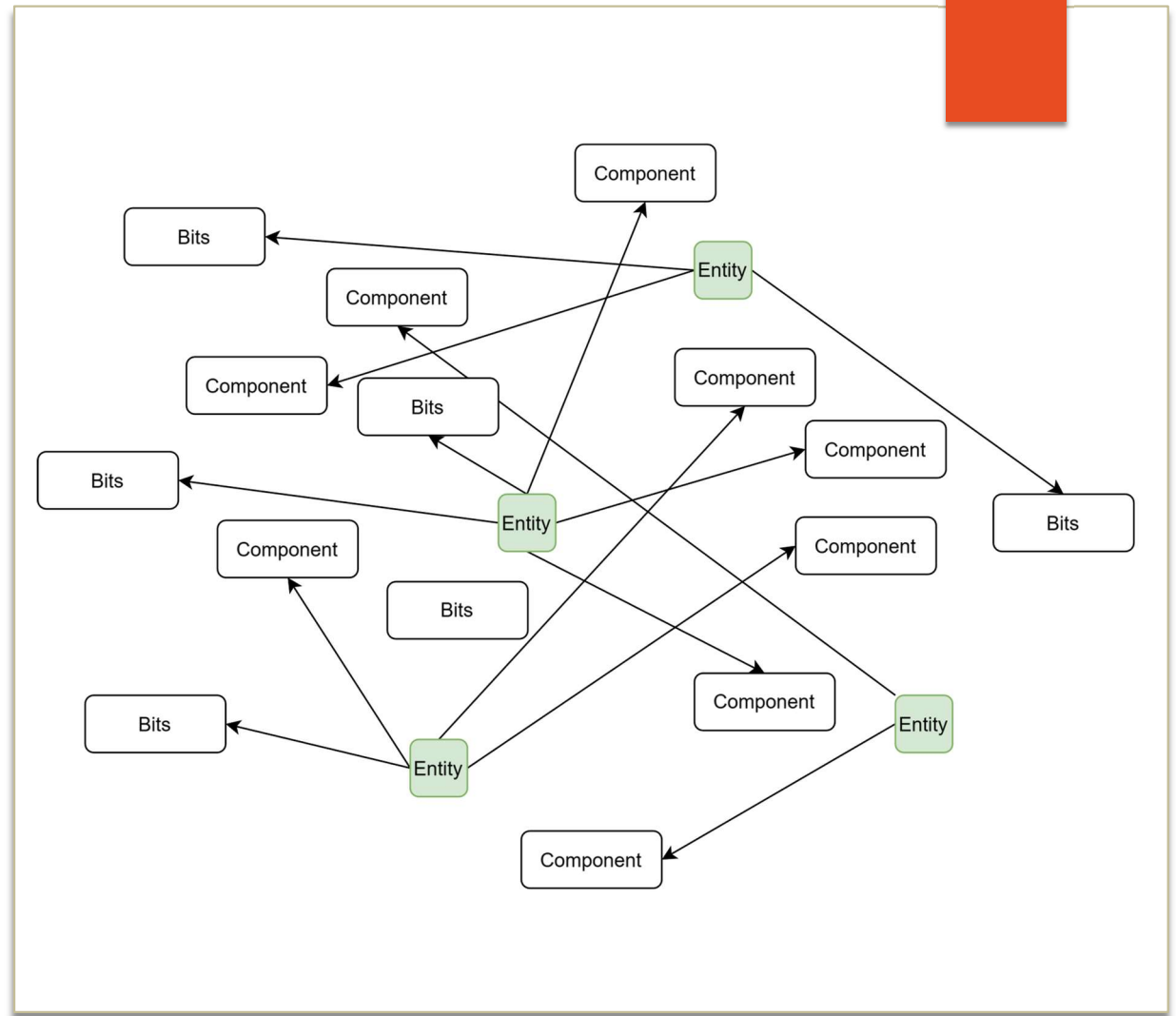
Created By: Péter Tamás
Github: <https://github.com/TamasPetii/SynapseEngine>

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 effectively across multiple CPU cores



Object-Oriented Entity-Component-System

Problems II:

- ▶ Matches how developers think, not how modern CPUs execute code
- ▶ Significantly slower data access
- ▶ Complex CPU/GPU synchronization
- ▶ Object-by-Object update model is difficult 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 logic
- ▶ **Memory Model**: Components of the same type are stored contiguously in memory!!

```
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];

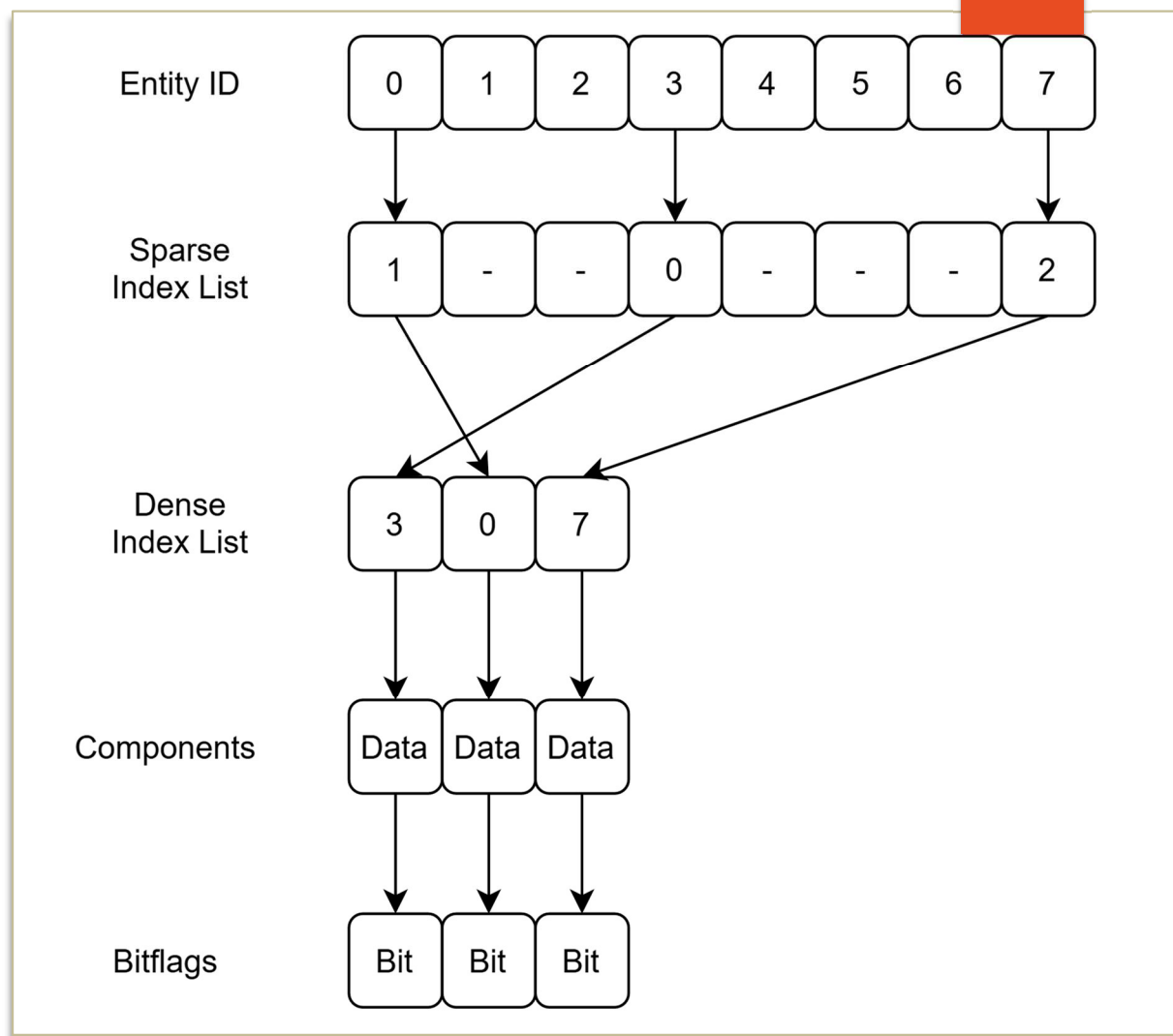
    glm::mat4 localMat = glm::translate(glm::mat4(1.0f), transformComponent.translation);
    localMat = glm::rotate(localMat, glm::radians(transformComponent.rotation.z), glm::vec3(0, 0, 1));
    localMat = glm::rotate(localMat, glm::radians(transformComponent.rotation.y), glm::vec3(0, 1, 0));
    localMat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

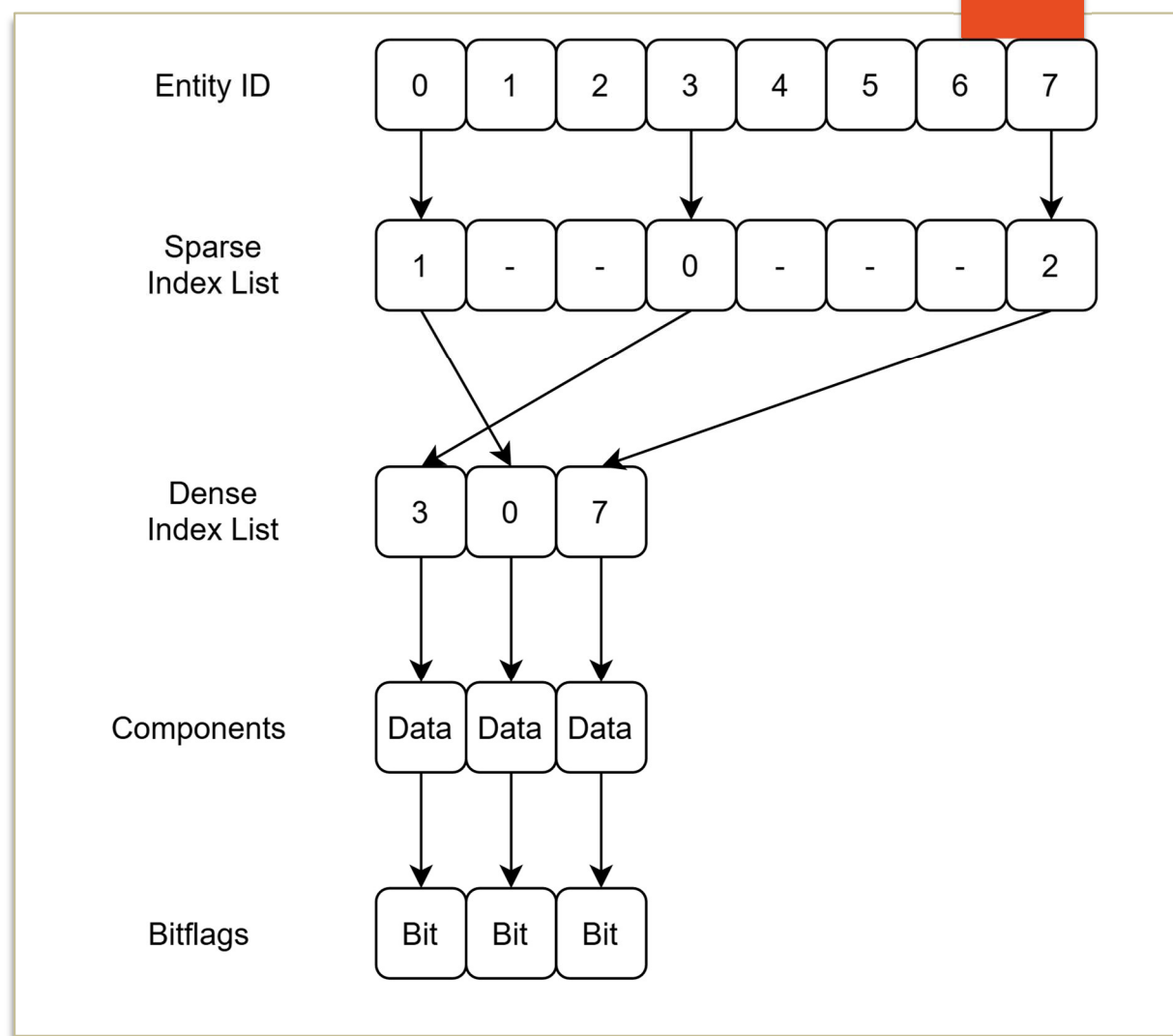
- ▶ Sparse Index Array with gaps, and a tightly packed Dense index array
- ▶ Bidirectional mapping between Sparse and Dense arrays
- ▶ Entity Ids index directly into the Sparse array
- ▶ The dense array can be paired 1:1 with any arbitrary data array
- ▶ In my case: Component Storage + Bitmask



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:

- ▶ Replacing `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 work 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);

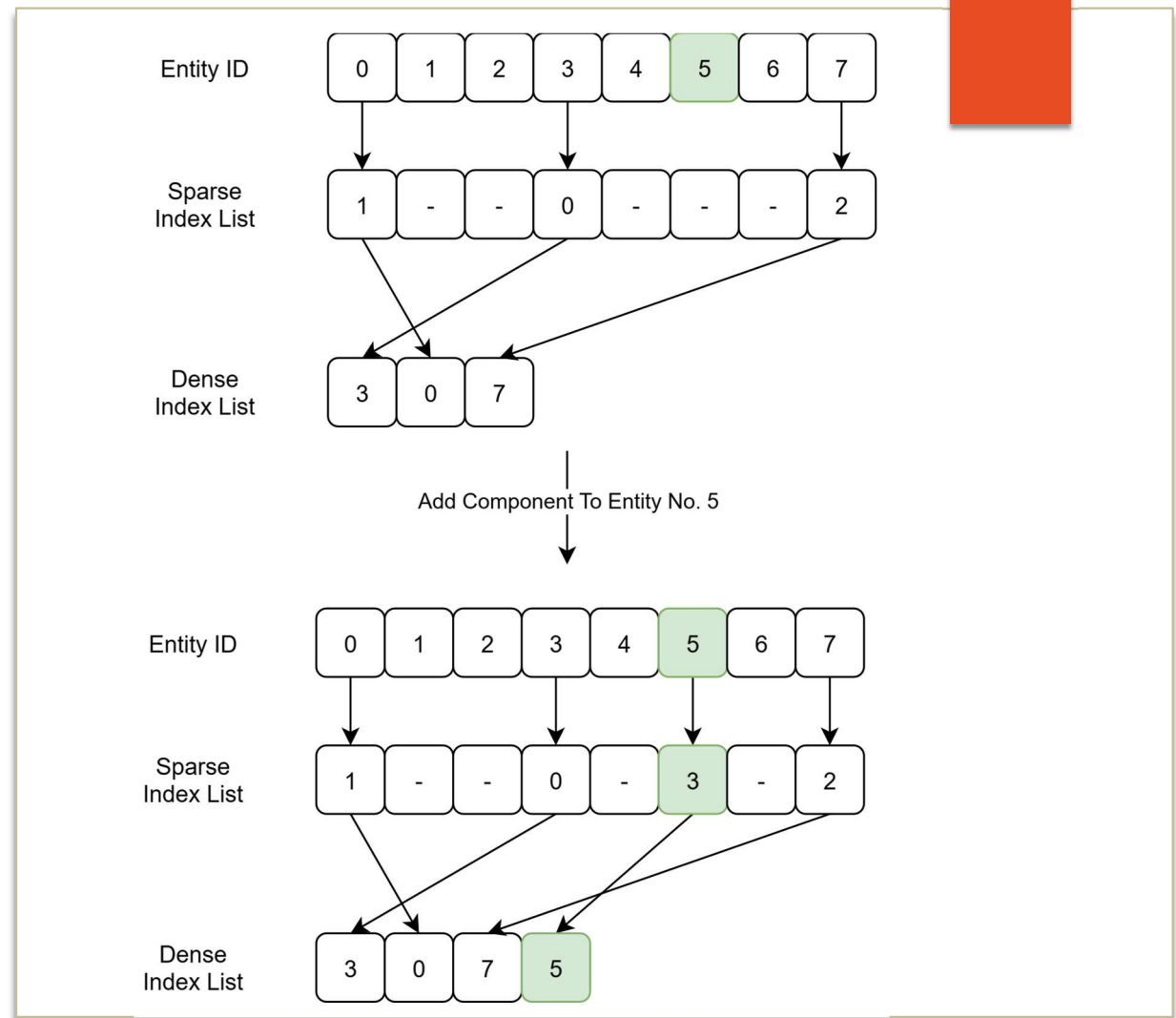
    transformComponent.transform = localMat;
});
```

<https://github.com/taskflow/taskflow>

Data-Oriented Entity-Component-System V.

Sparse Set – Add operation:

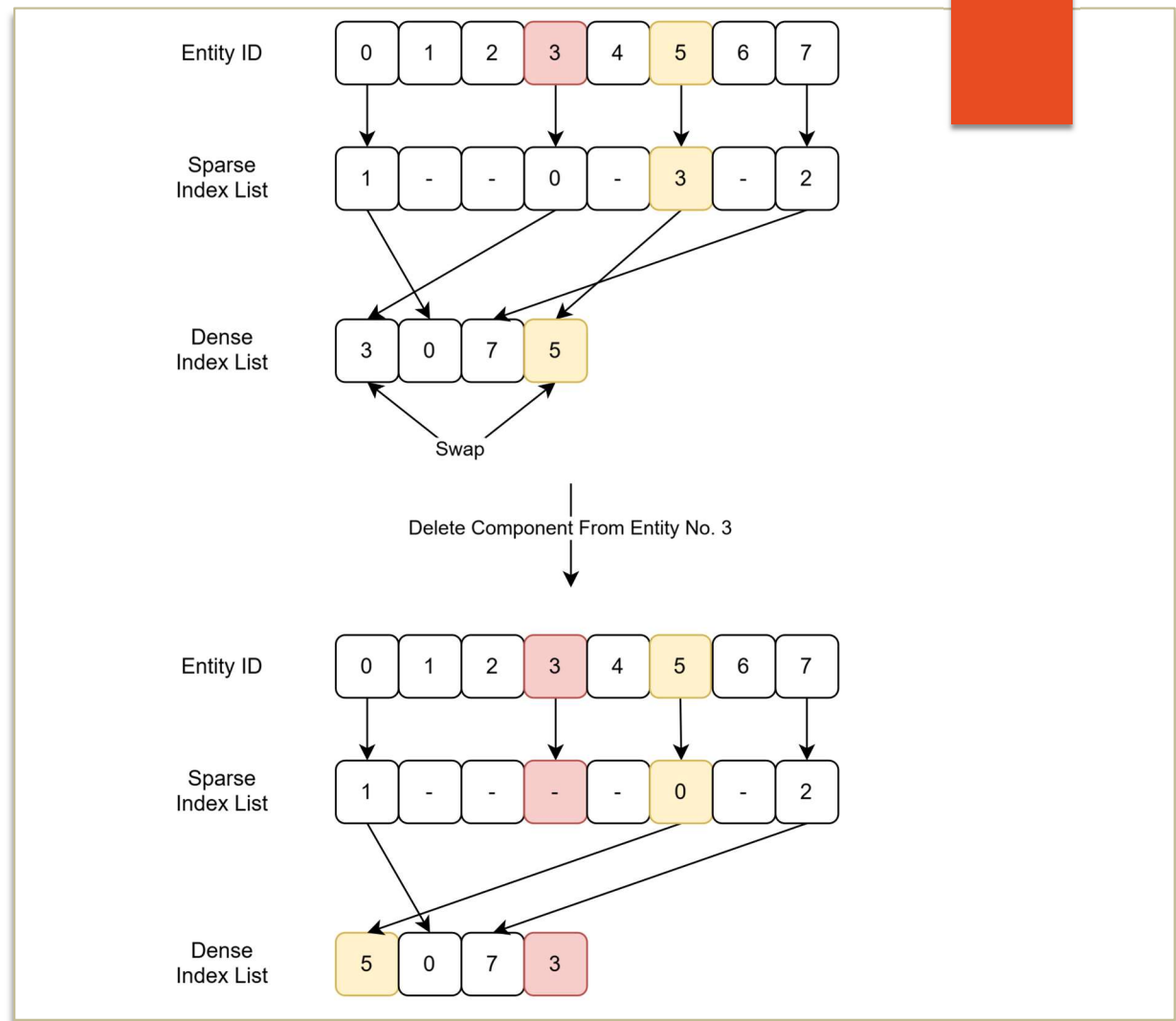
- ▶ Append the new component and entity to the end of the dense arrays.
- ▶ Update the corresponding entry in the sparse array
- ▶ Insertion is true $O(1)$ (excluding reallocation)
- ▶ The dense array remains compact and cache-friendly
- ▶ No shifting or reordering of existing elements is required!!



Data-Oriented Entity-Component-System VI.

Sparse Set – Remove operation:

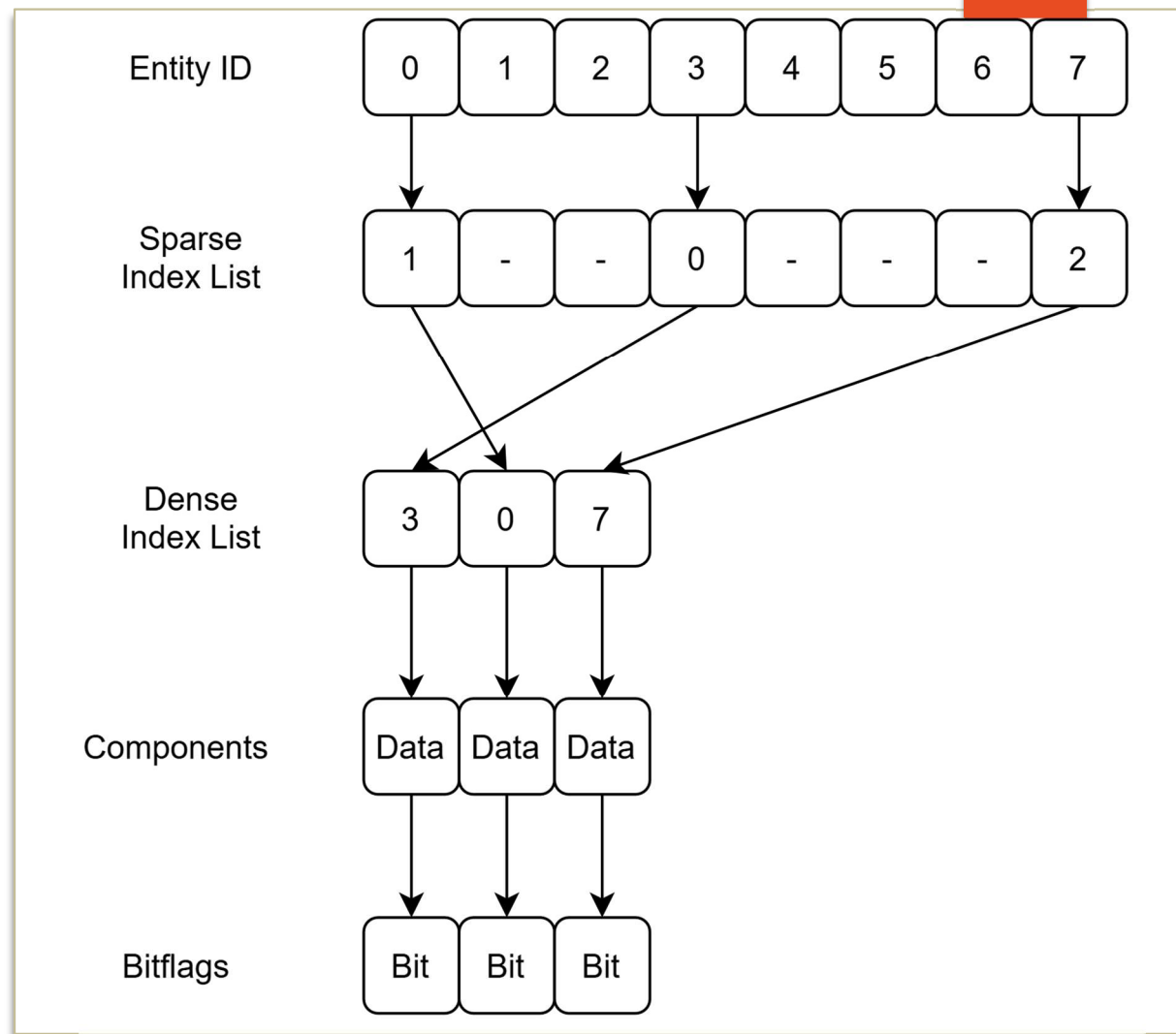
- ▶ Swap the element to remove with the last element in the dense arrays.
- ▶ The removed element is now at the end and can be erased with a single pop_back
- ▶ Update the sparse array: Mark the removed entity as invalid and remap the swapped entity to its new dense index.
- ▶ Removal is also true $O(1)$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



Data-Oriented Entity-Component-System VII.

Sparse Set – Remaining Challenges:

- ▶ The sparse array **grows with the maximum entity ID**, leaving unused memory for gaps
- ▶ This is more noticeable 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.



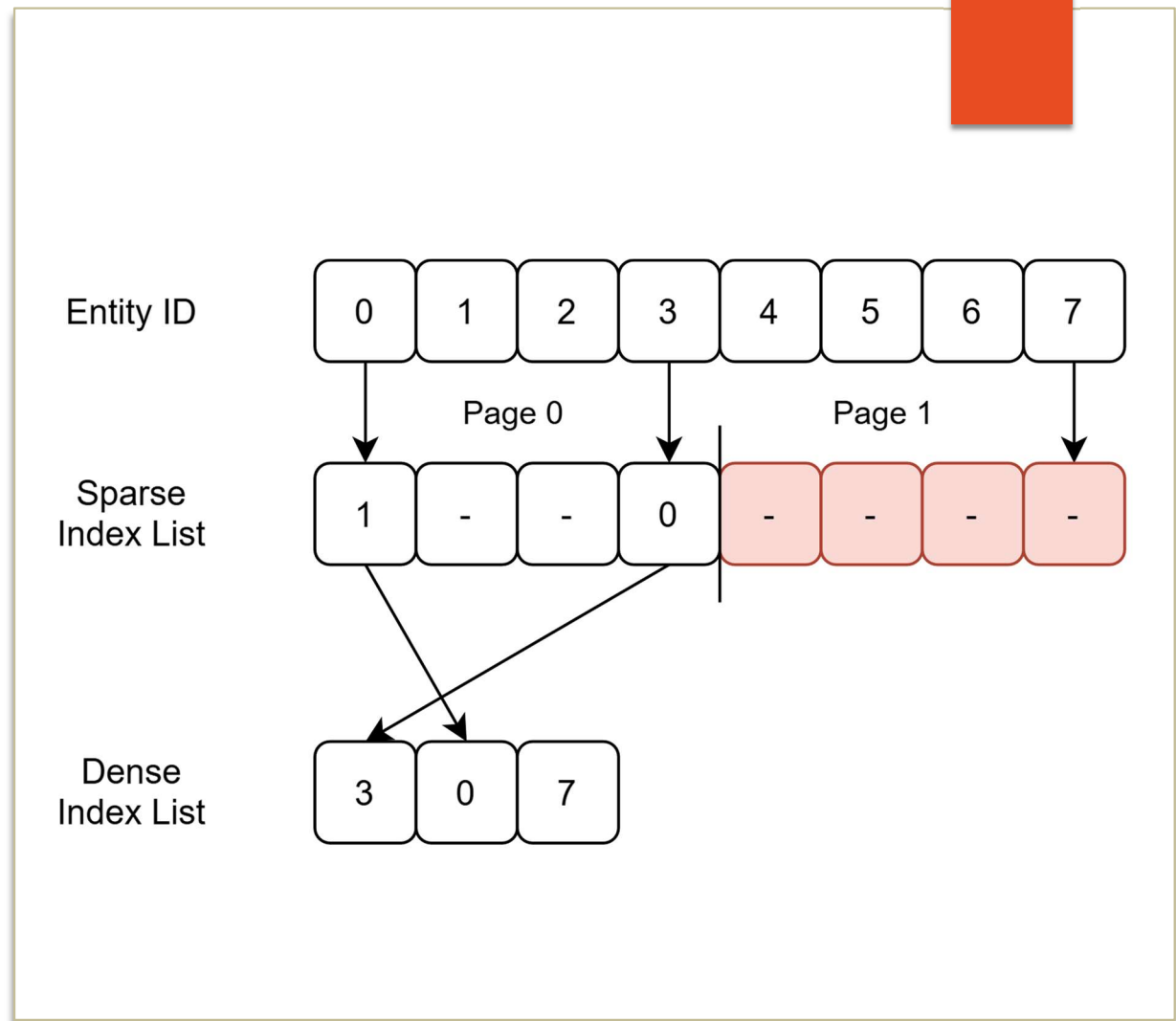
Data-Oriented Entity-Component-System VIII.

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!!**



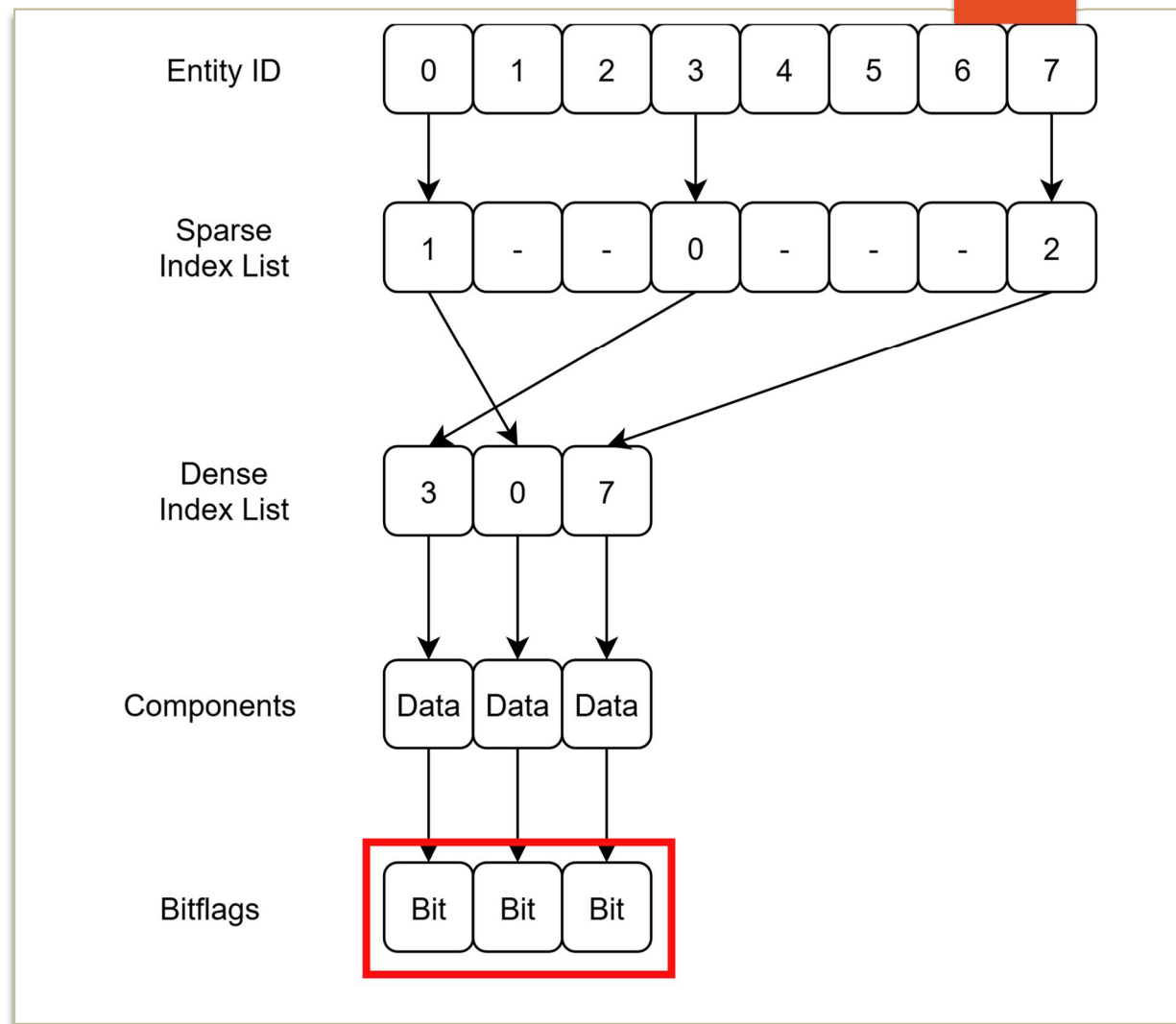
Data-Oriented Entity-Component-System IX.

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 states or engine-specific flags.

Optimization – Pool-level dirty flag

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



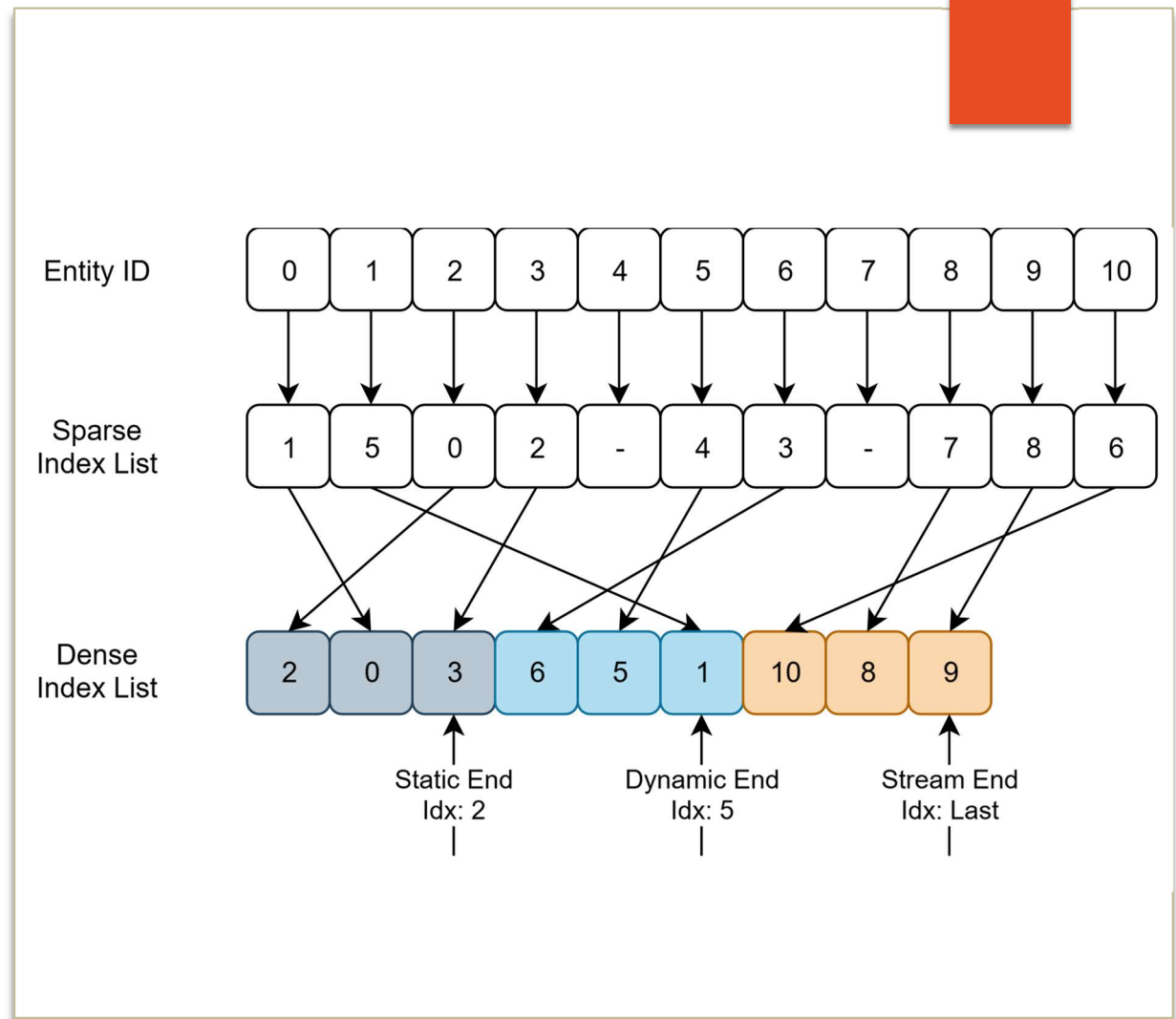
Data-Oriented Entity-Component-System X.

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.



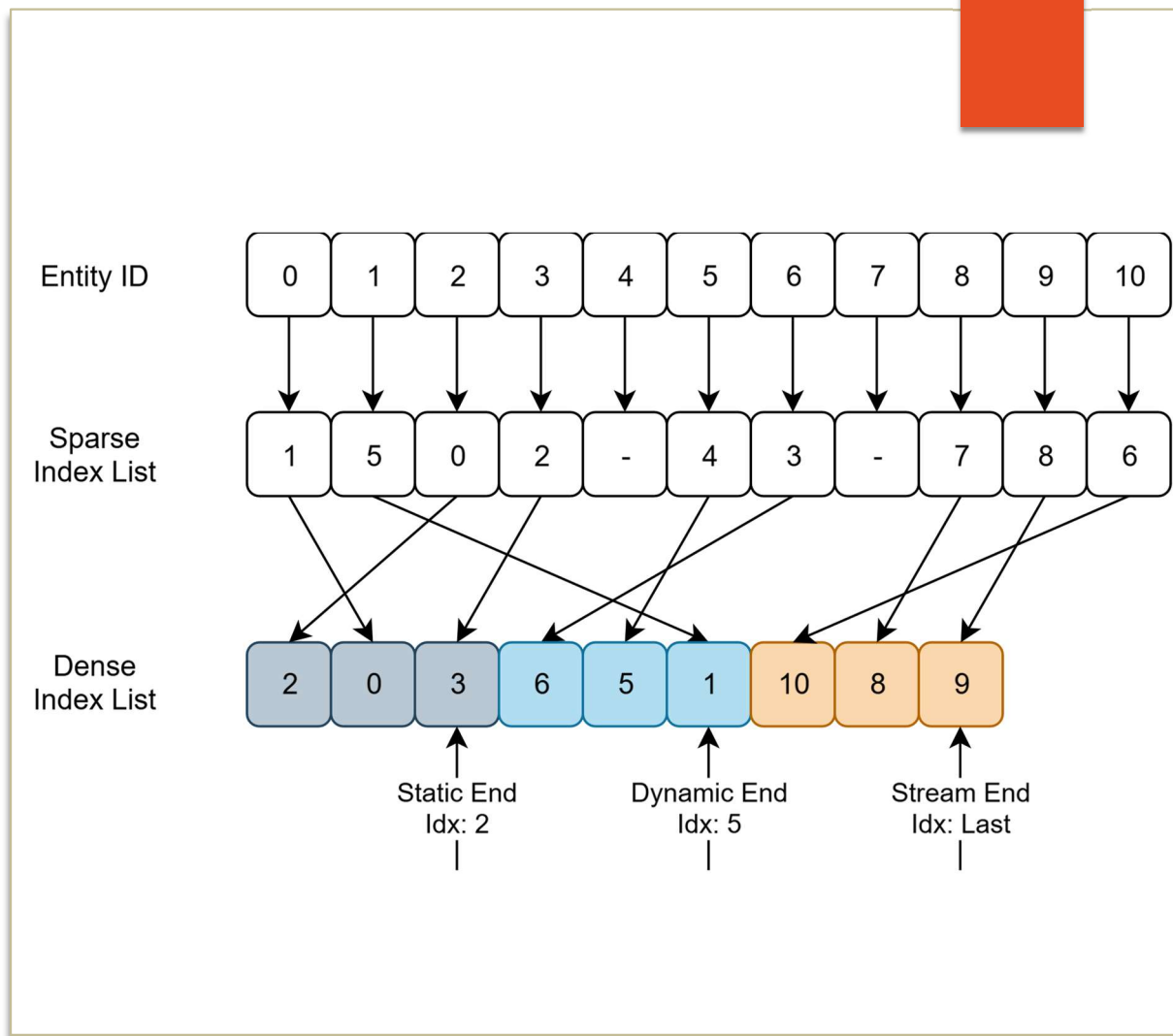
Data-Oriented Entity-Component-System XI.

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 updates!!

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



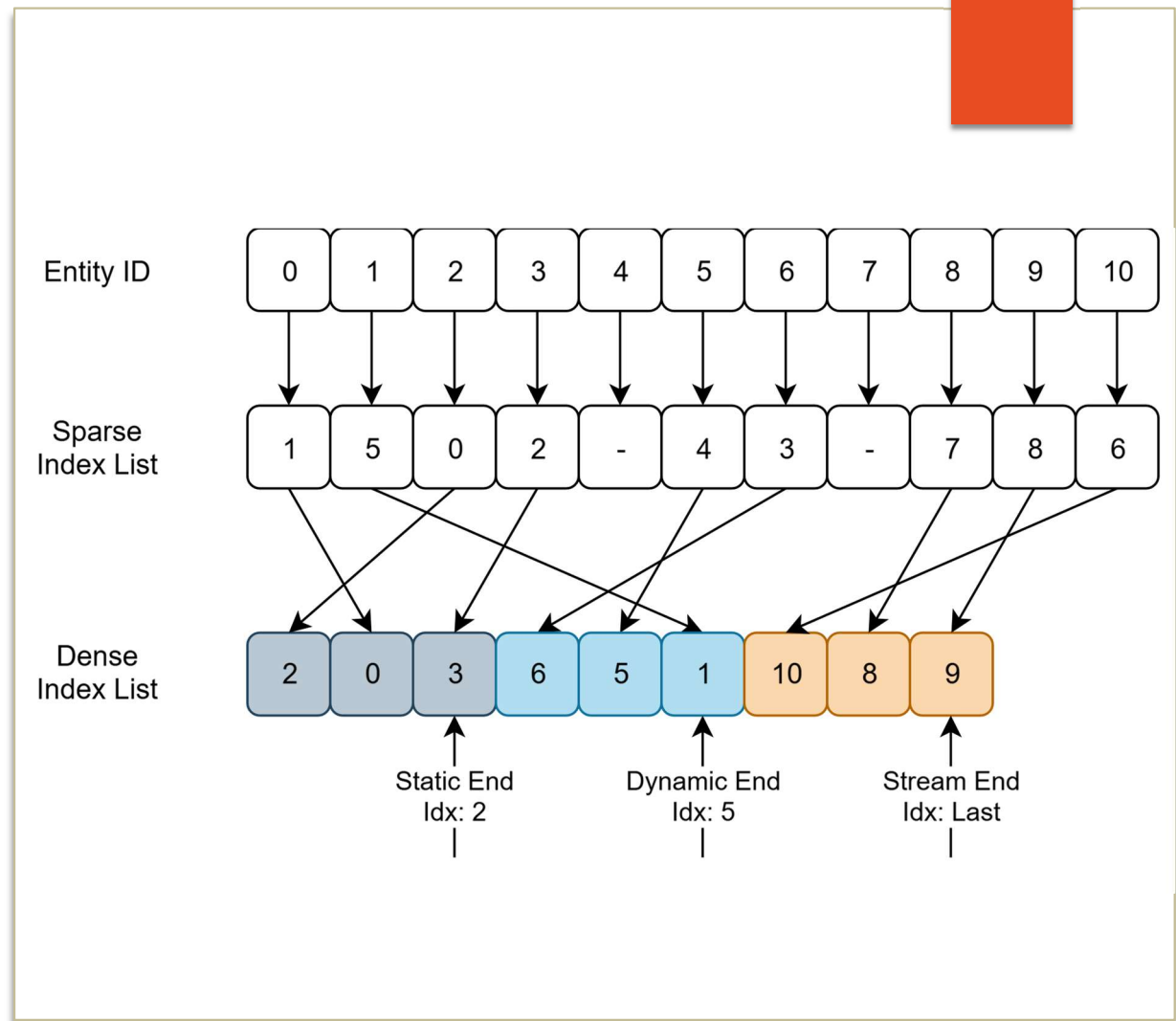
Data-Oriented Entity-Component-System XII.

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

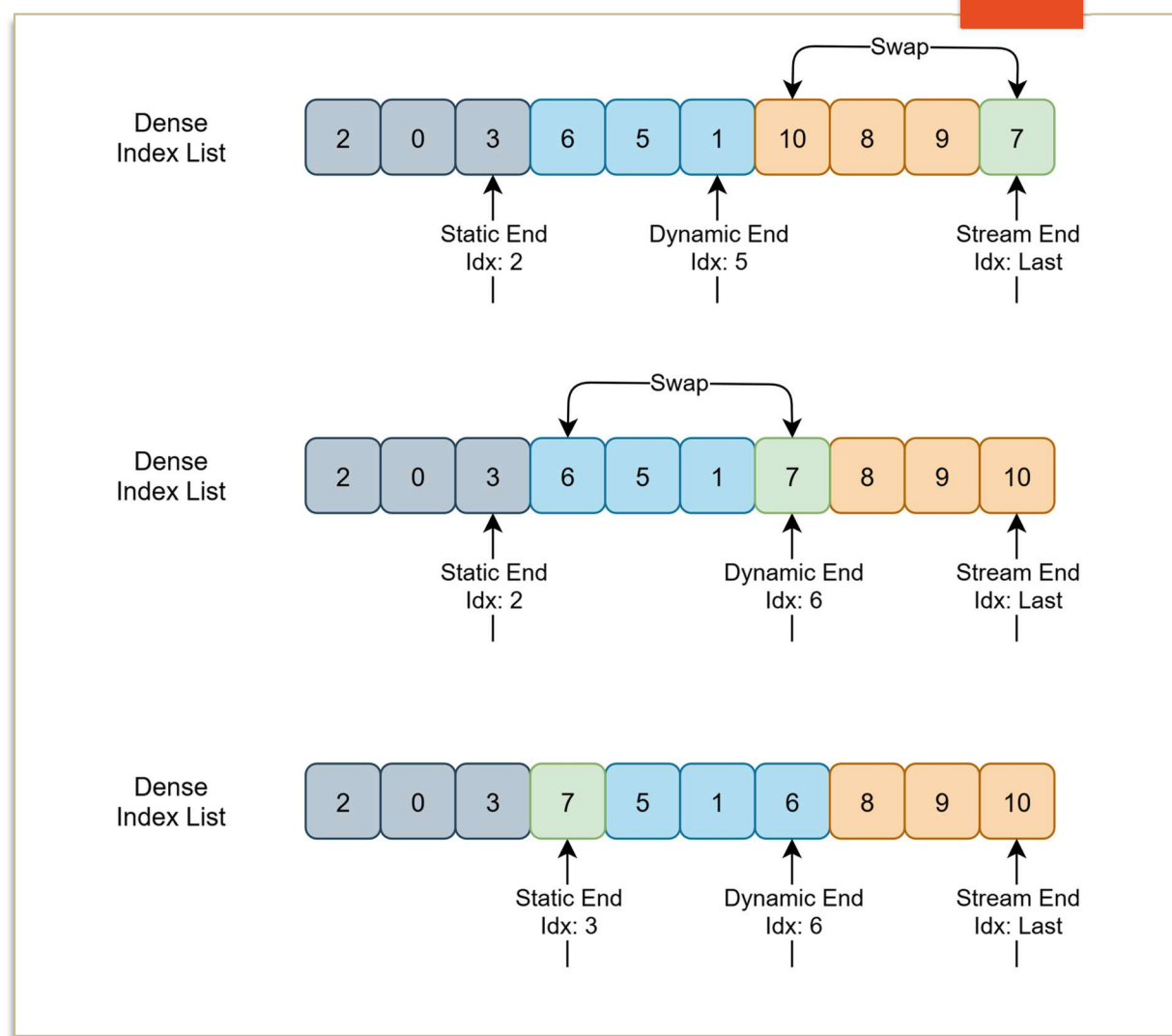
- ▶ Two boundary indices, **static-end** and **dynamic-end** define the segment ranges
- ▶ These **boundaries are updated** when components are **added**, **removed** or **change update frequency**.



Data-Oriented Entity-Component-System XIII.

Segmented Sparse Set – Add OP

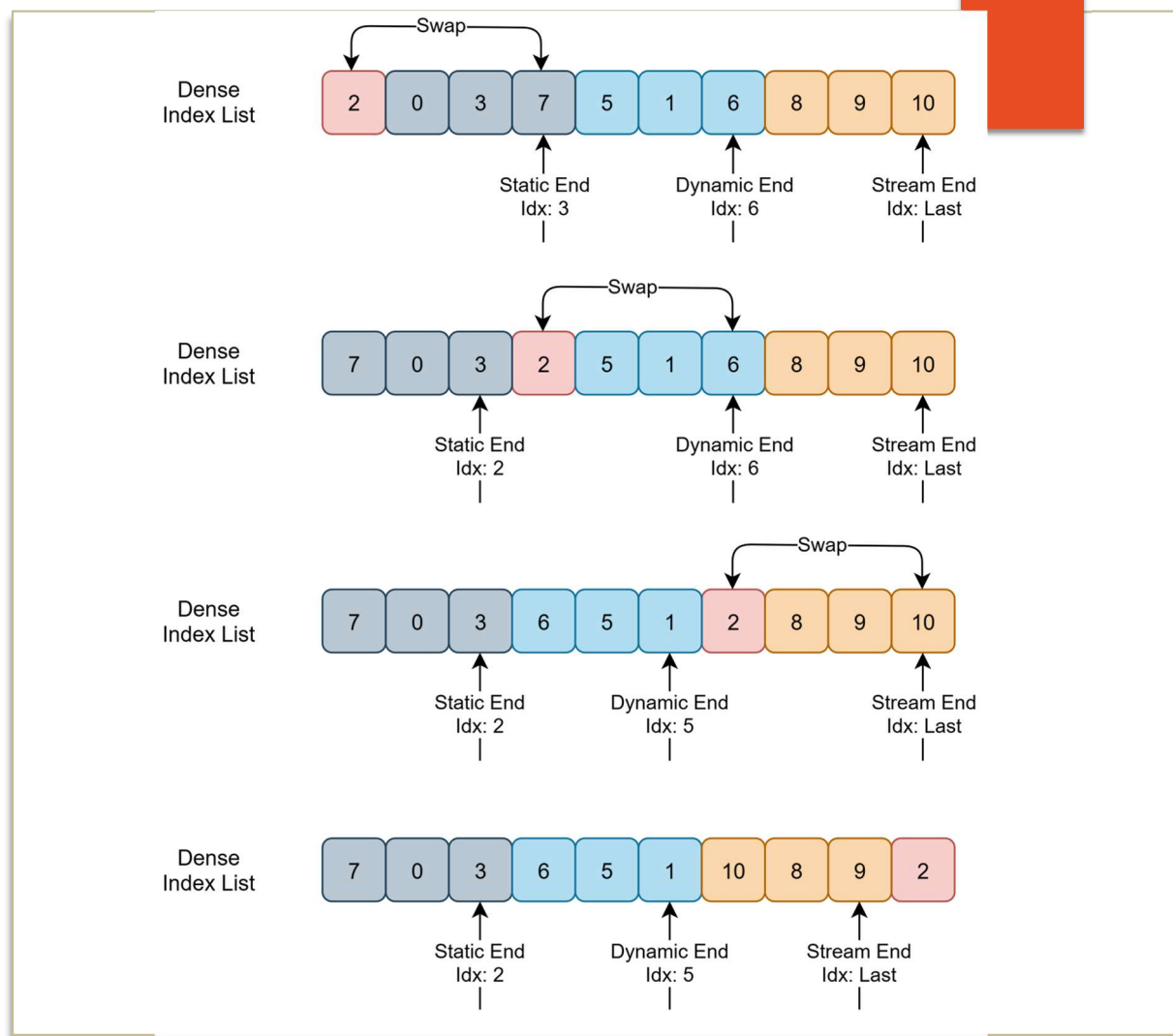
- ▶ **Worst case:** only 2 swaps are required
- ▶ **Insert the new component at the end** of the dense array (end of stream region)
- ▶ **Swap it with the first element** of the **stream region** (dynamic-end + 1) and update the boundary
- ▶ **Swap it with the first element** of the **dynamic region** (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 against 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 purpose-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).

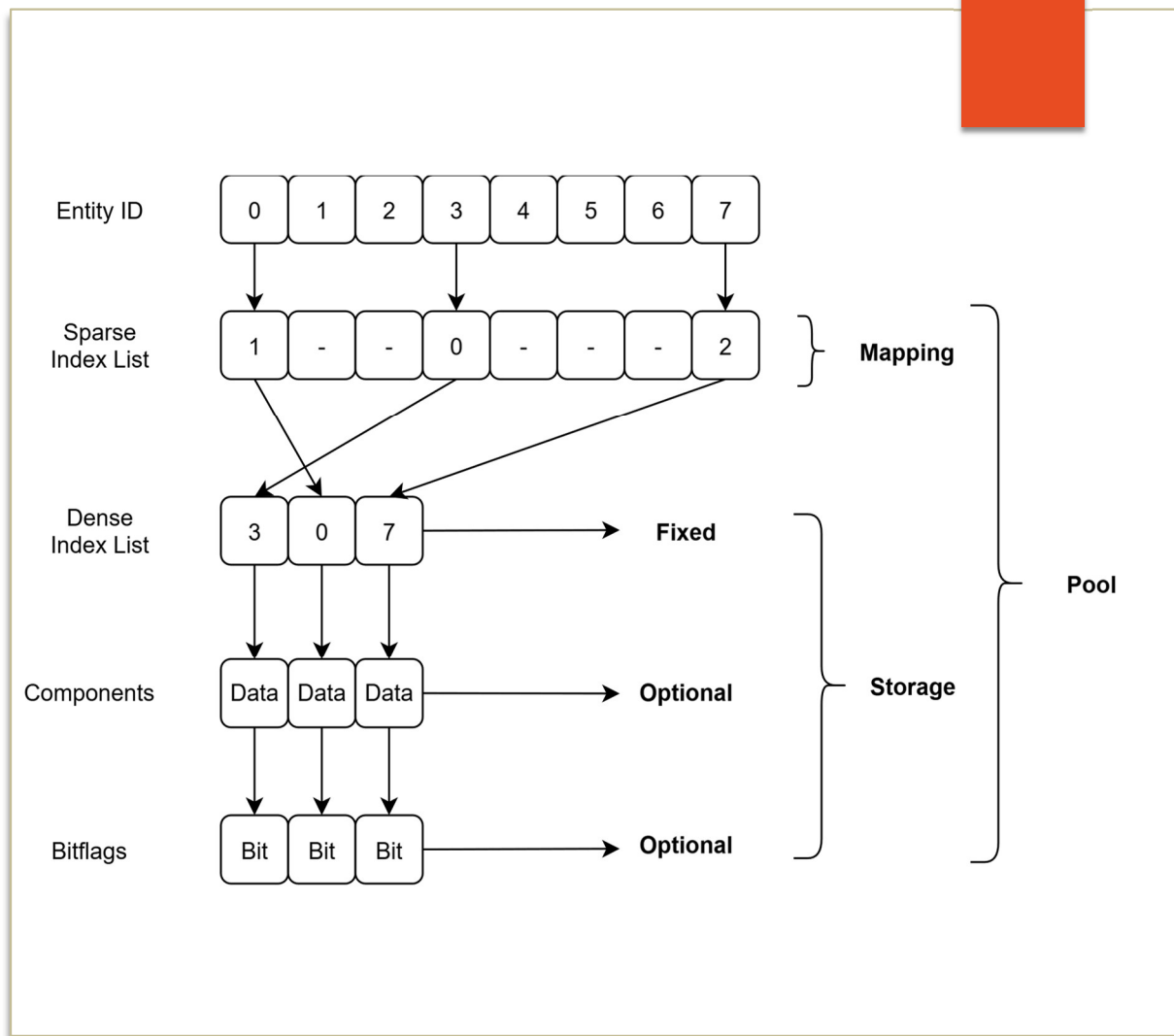
Operation	Entities	Synapse (ms)	EnTT (ms)
Add	100k	6.08	2.99
	1M	59.66	35.59
Remove	100k	2.31	3.29
	1M	26.18	37.01
Iterate (Seq)	100k	0.24	0.20
	1M	2.46	2.25
Iterate (Par)	100k	0.25	0.25
	1M	2.15	2.26
Engine Update	100k	0.11	0.14
	1M	1.15	1.76

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 replaceable independently
- ▶ Support multiple implementations:
 - ▶ Mapping: Flat/Paged
 - ▶ Storage: Flat/Segmented
- ▶ Support different use cases:
 - ▶ Sparse Set indexing layer only
 - ▶ Data storage 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 compile time**, zero runtime overhead.
- ▶ **Concepts** enforce the required interface at compile 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 configured with any combinations of features:
Data-Only / Flags-Only
Data + Flags / No extra storage
- ▶ **Unused functionality** is replaced with empty implementation, compiler **eliminates** these calls during optimization!!

Storage Backend

- ▶ **DataMixin<T>**: provides component storage
- ▶ **FlagMixinPolicy**: 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 overhead
- ▶ **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!!

```
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, unified interface while remaining fully modular.

Remaining Challenge

- ▶ Different Mapping and Storage **implementations expose their own specialized functionality**.
- ▶ Ideally, these functions should also be accessible through 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 challenge

- ▶ 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 accesses are **true O(1)**, with no map lookup or string hashing!

```
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/TamasPettii/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**: Previous 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;

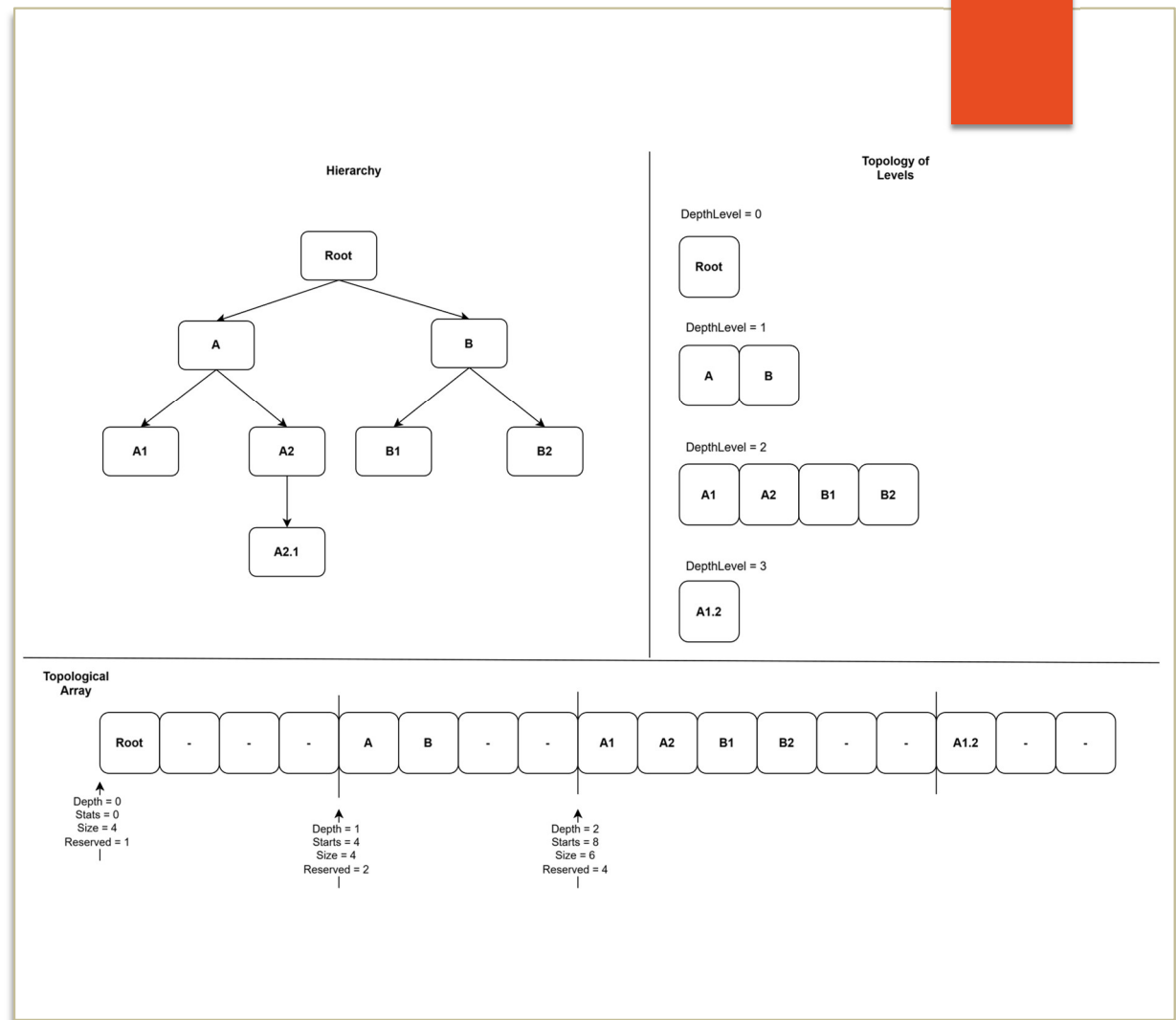
    uint32_t depthLevel = 0;
    uint32_t topoIndex = NULL_INDEX;
};
```

<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 Challenge

- ▶ 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 challenge 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**.
- ▶ Iterate 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.

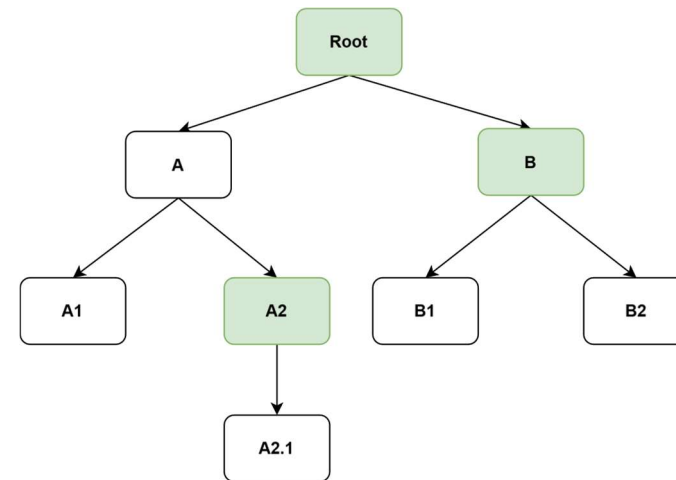
Update Category Rules

- ▶ **Static parent**: can have static/dynamic/stream child
- ▶ **Dynamic 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



WorkQueue - Depth 0
Counter: 1



WorkQueue - Depth 1
Counter: 1



WorkQueue - Depth 2
Counter: 1



WorkQueue - Depth 3
Counter: 0

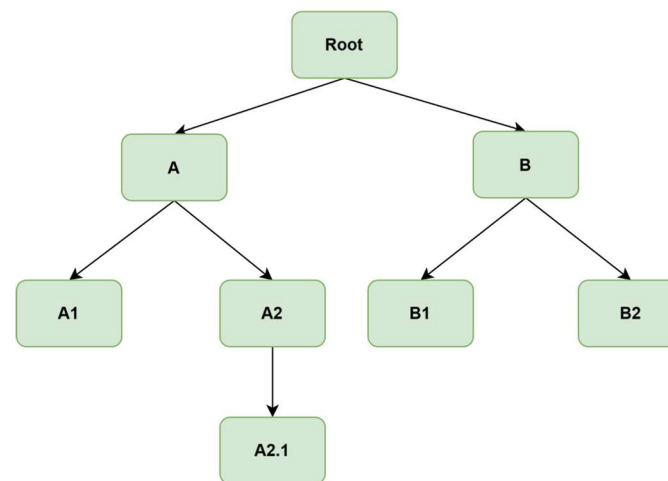


<https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/Scene/Hierarchy/HierarchyWorkQueue.h>

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 entity 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.



WorkQueue - Depth 0
Counter: 1



WorkQueue - Depth 1
Counter: 2



WorkQueue - Depth 2
Counter: 4



WorkQueue - Depth 3
Counter: 1

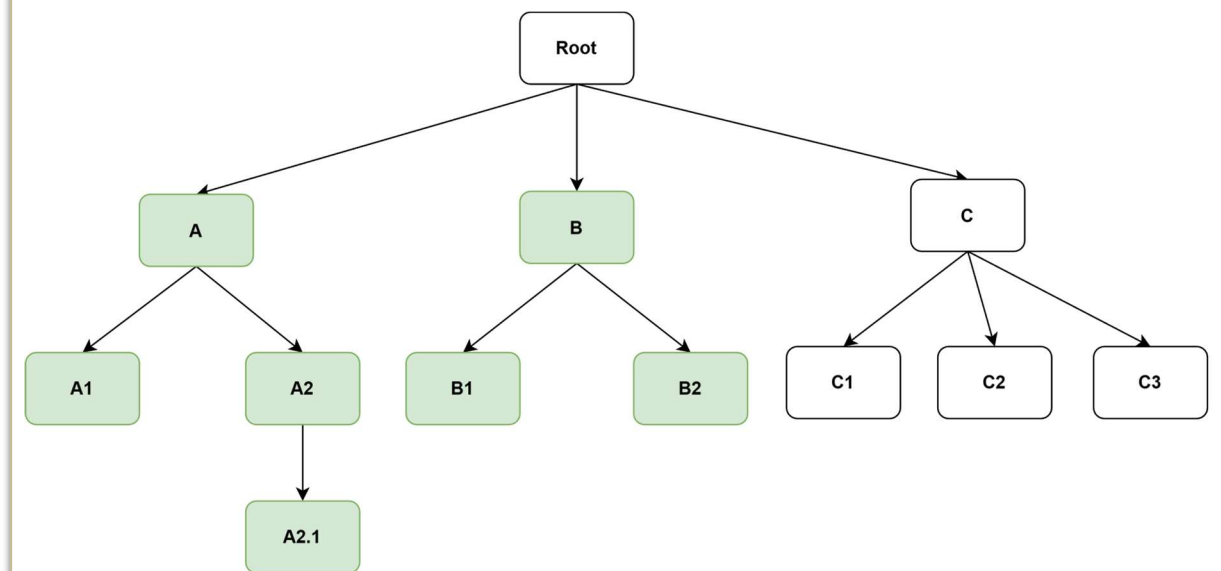


<https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/System/Core/TransformSetupSystem.cpp>

Data-Oriented Entity-Component- System XXIX.

Duplicate Prevention & Scalability

- ▶ Each entity can be **inserted into the 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!**

```
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);
    }
};
```

```
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]

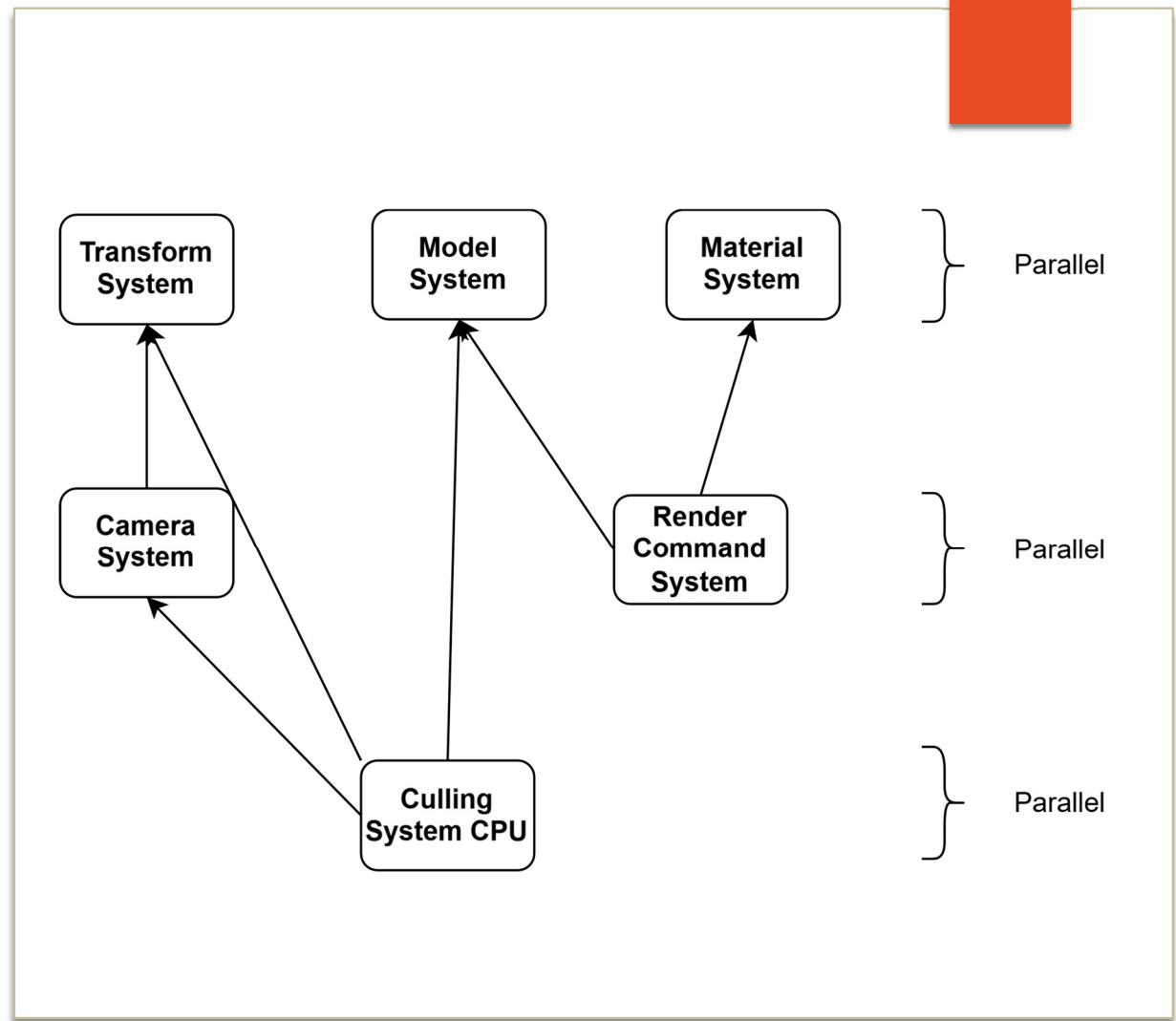
//Shader Code:
uint transformDenseIndex = GET_SPARSE_INDEX(ctx.transformSparseMapBufferAddr, entityId);
TransformComponent transform = GET_TRANSFORM(ctx.transformBufferAddr, transformDenseIndex);
```

<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 stealing!
- ▶ **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 ~90.000 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 interchangeable Pool policies.
- ▶ Clear architecture reduces complexity and accelerates 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 **modify 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();

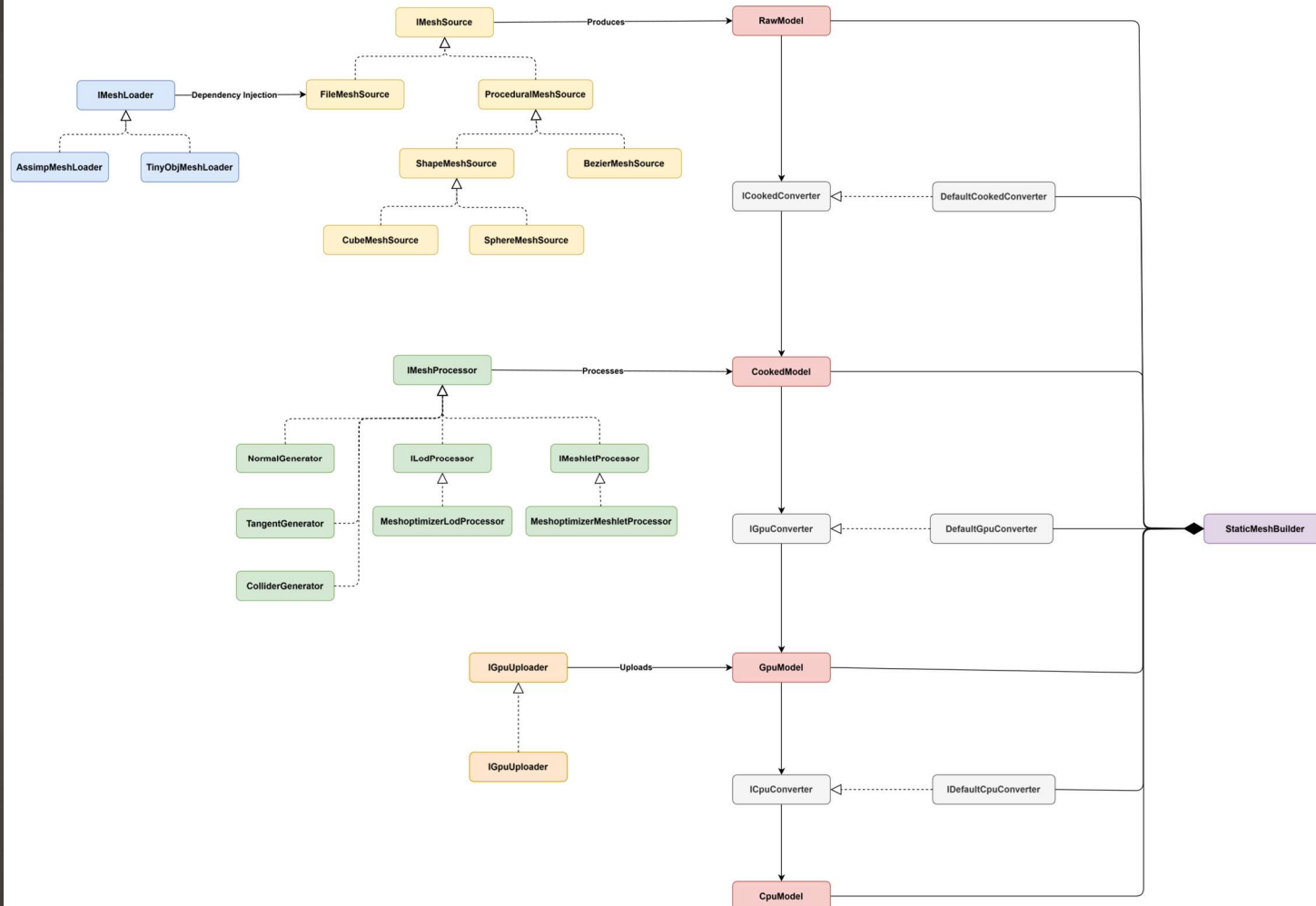
    return true;
}
```

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

Software Architecture III.

Mesh Pipeline – Design 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.

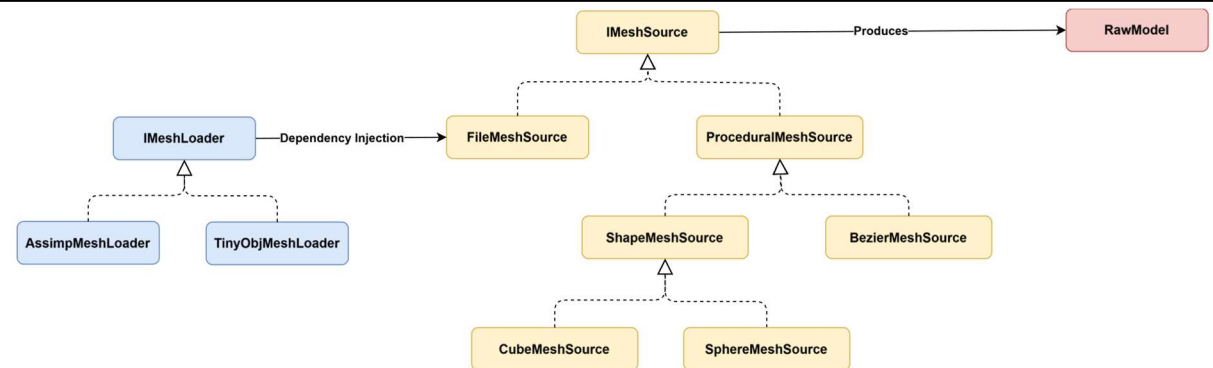
Raw Model & Mesh Sources

► 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.**



```
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>

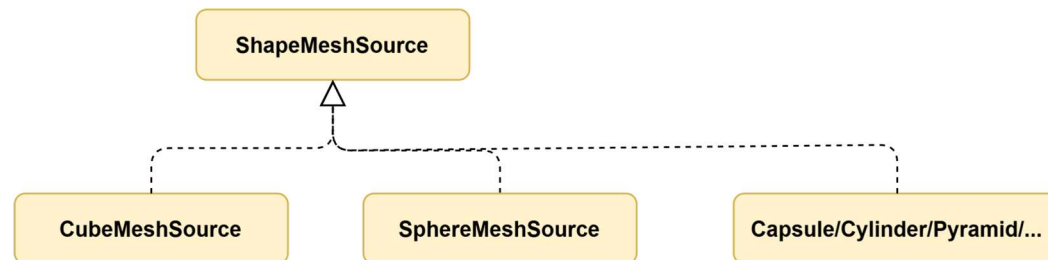
Software Architecture V.

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 functions** which are going to be overridden.

Benefits

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



```
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));

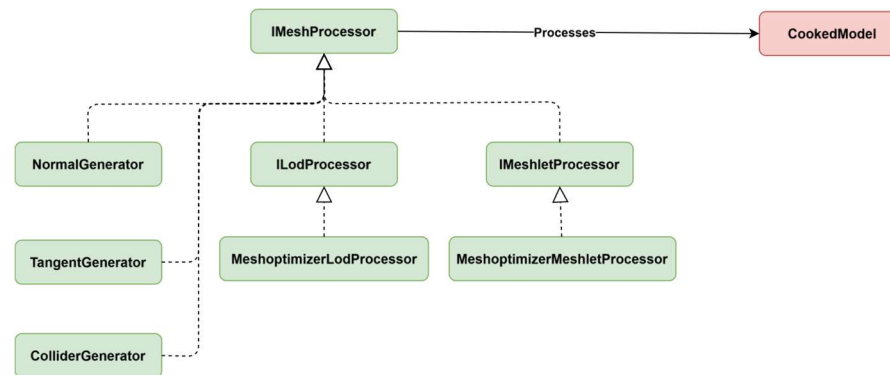
    //...
}
```

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

Software Architecture VI.

Cooked Representation

- ▶ The **second stage** of the mesh pipeline 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 organization** 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>

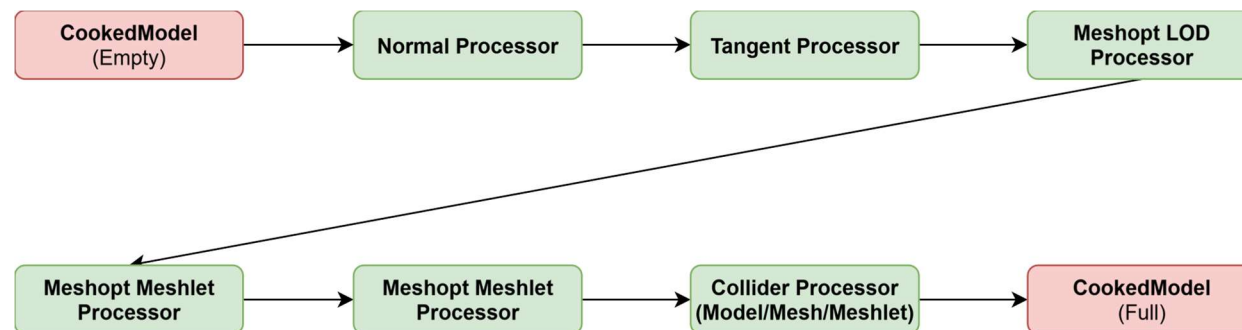
Software Architecture VII.

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 **independently** 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 **transformed 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 Representation

- ▶ 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;
};

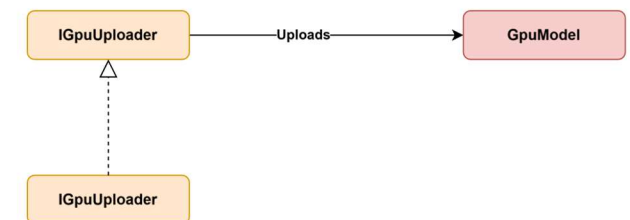
struct SYN_API GpuIndexedDrawData
{
    std::vector<uint32_t> indices;

    //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. Everything is dependency injected.
- ▶ GPU upload is managed by the ModelManager, not the builder since models are loaded async and GPU upload require synchronization, 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 accessible via centralized buffer addresses.
- ▶ All model resources become globally accessible to every shaders.

```
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 applied 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 unit 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 happen, the **manager only decides when** to invoke it.
- ▶ Eliminates unnecessary dependencies between subsystems.
- ▶ Supports 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.

```
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**

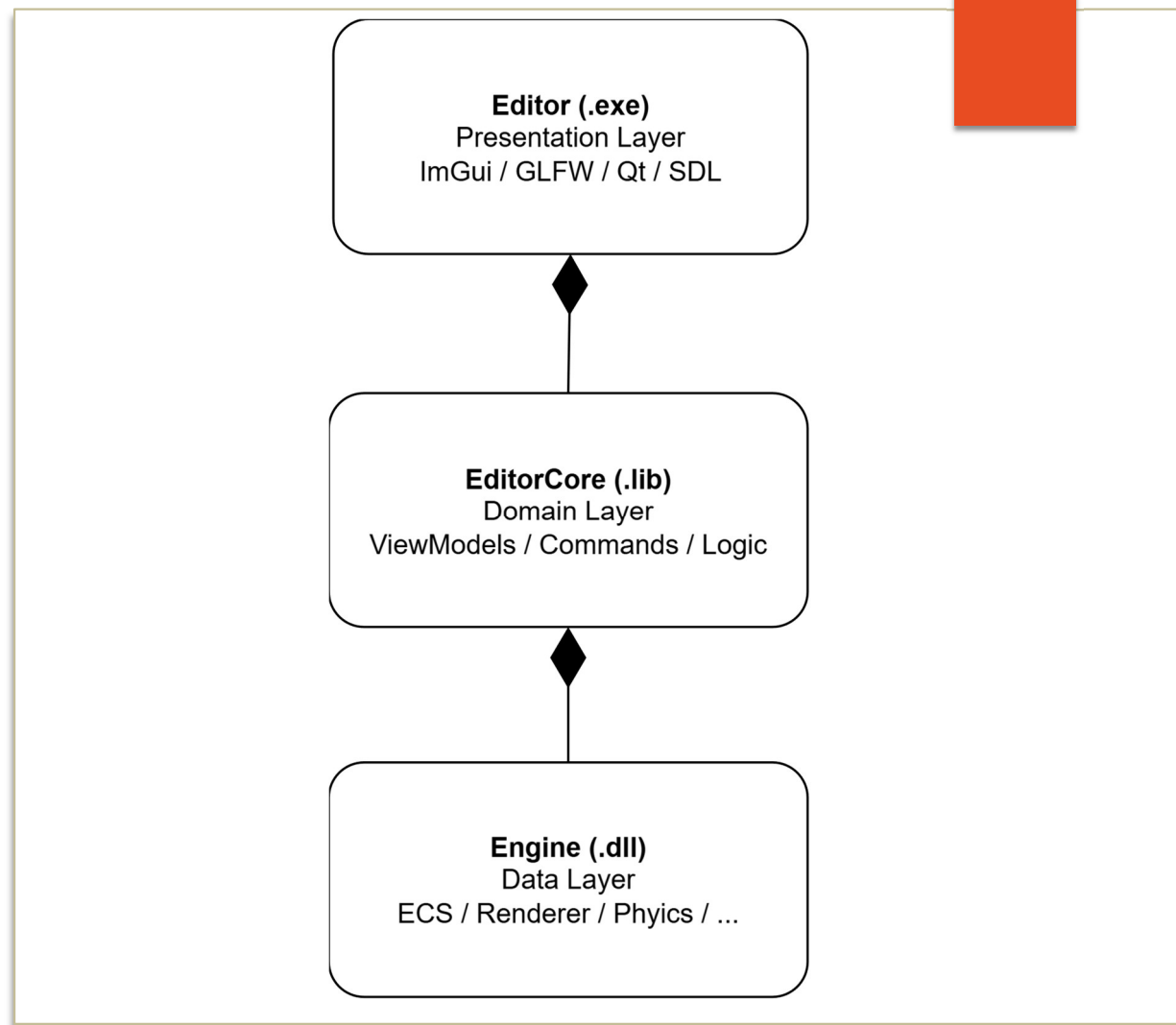
- ▶ Contains the core engine implementation.
- ▶ ECS, Renderer, Asset System, Physics, Algorithms, Shaders, Manager
- ▶ **Completely independent from the Editor!!**

▶ **EditorCore (.lib) – Domain Layer**

- ▶ Contains 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.



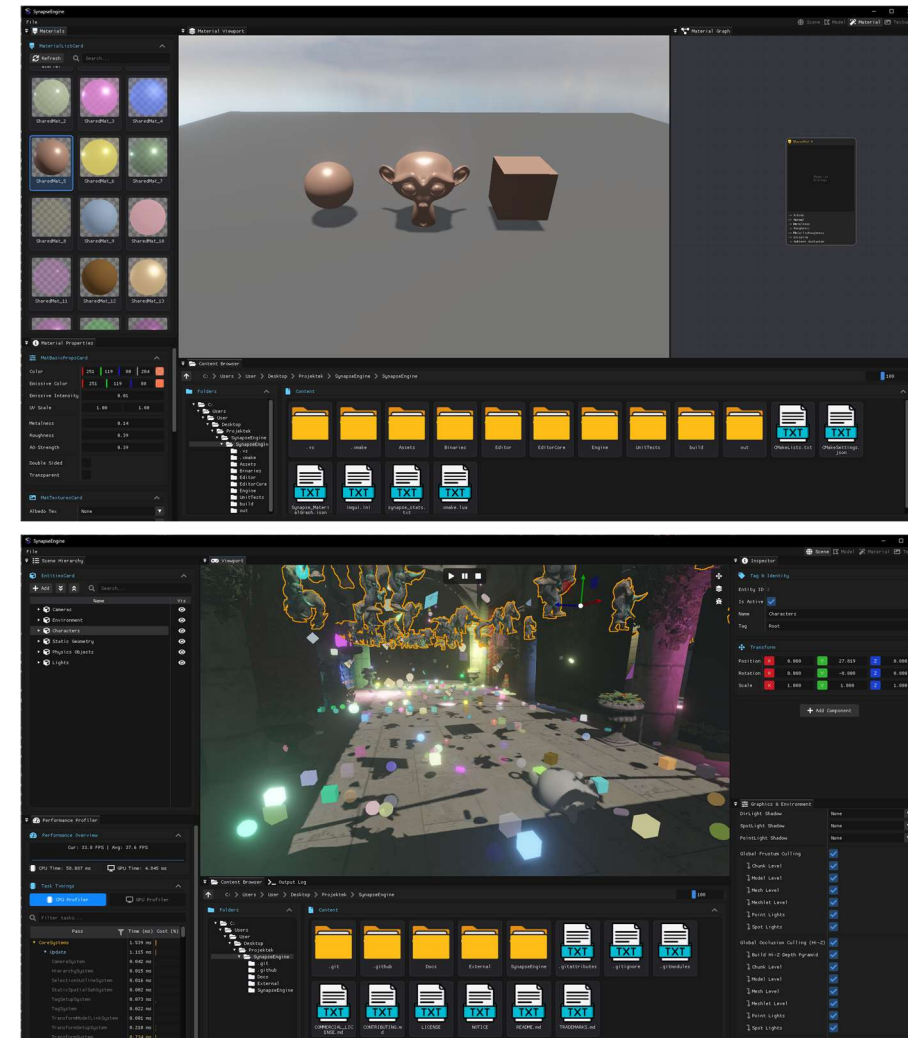
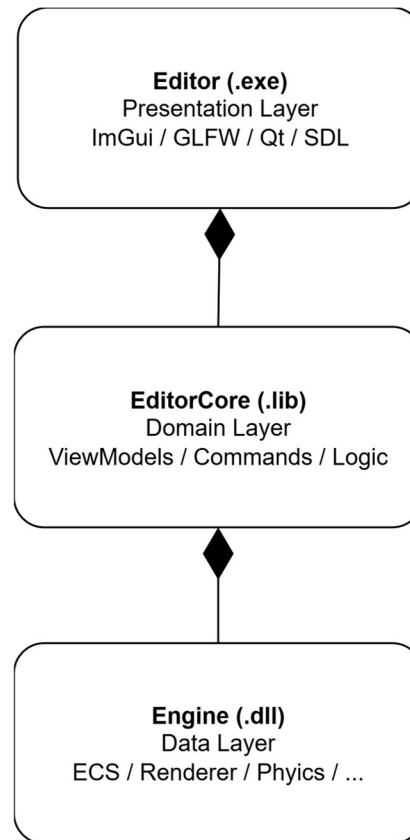
Software Architecture XVI.

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 modifying 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.

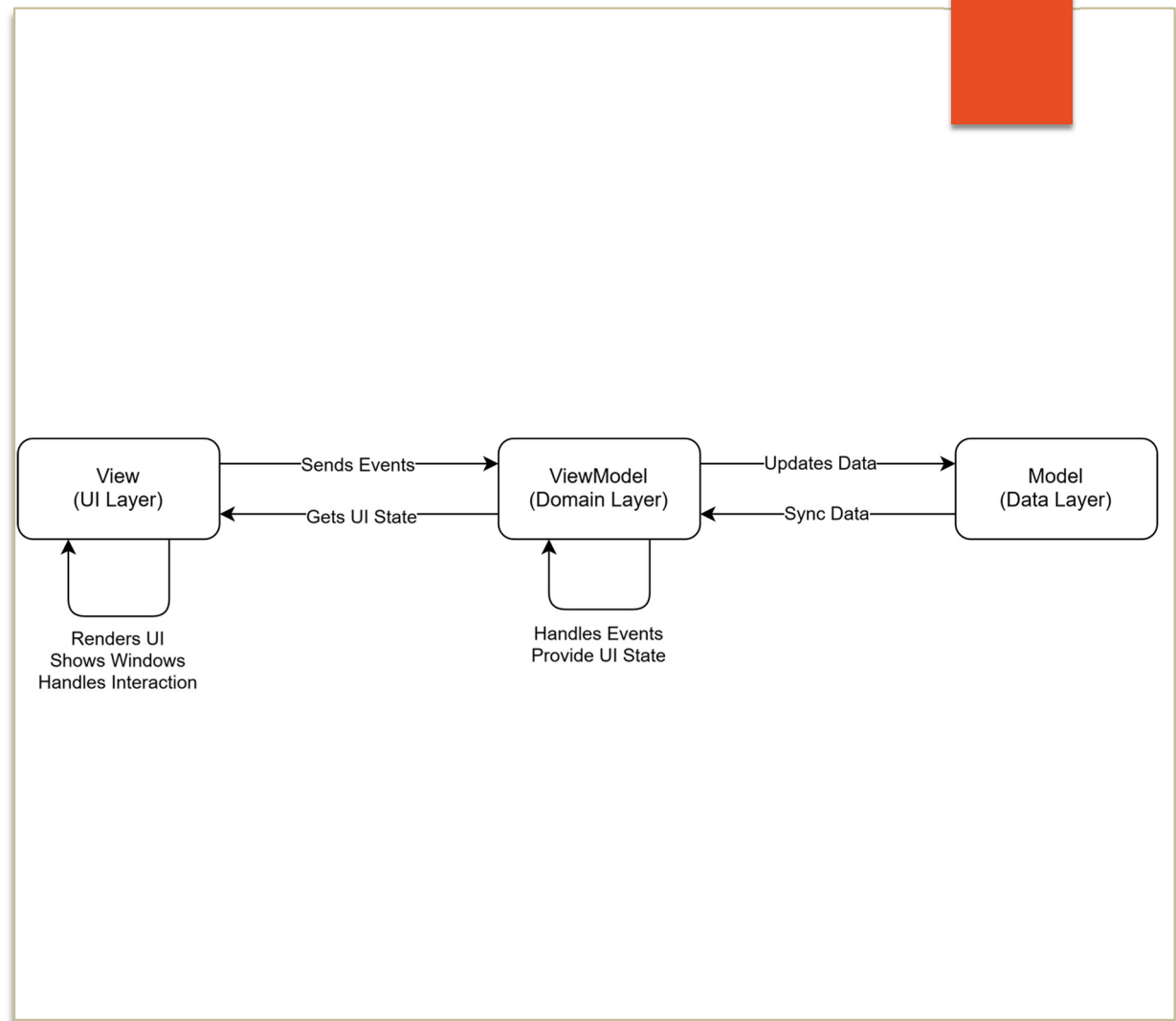
Model-View-Intent (MVI) Architecture

▶ Three-Layer Architecture

- ▶ **Data Layer (Engine):** The single source of truth containing all runtime engine data.
- ▶ **Domain Layer (EditorCore):** Business logic, editor state, viewmodels and intents.
- ▶ **UI Layer (Editor):** ImGui windows responsible onl 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 from 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 interface.
- **Completely independent** from 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 exclusively 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!

```
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 from the ViewModel and converts user interactions into intent objects.
- ▶ Forwards every Intent to the ViewModels for processing.
- ▶ The view remains completely stateless apart from local UI-only state.
- ▶ Different user interactions can produce different Intent types, for example **dragging a value and releasing** it may generate separate Intents
- ▶ This enables features such as **Undo/Redo, command history**, validation or even deferred execution without modifying the View.

```
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 Principle)

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 updating dirty flags, scheduling refreshes.
- ▶ **API objects can be stored, passed around and reused by higher-level systems such as Undo/Redo commands!**

```
//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**, eliminating duplicated editing logic.
- ▶ **Only finalized user actions create undoable commands**, continuous 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 from the UI!

```
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);
    }

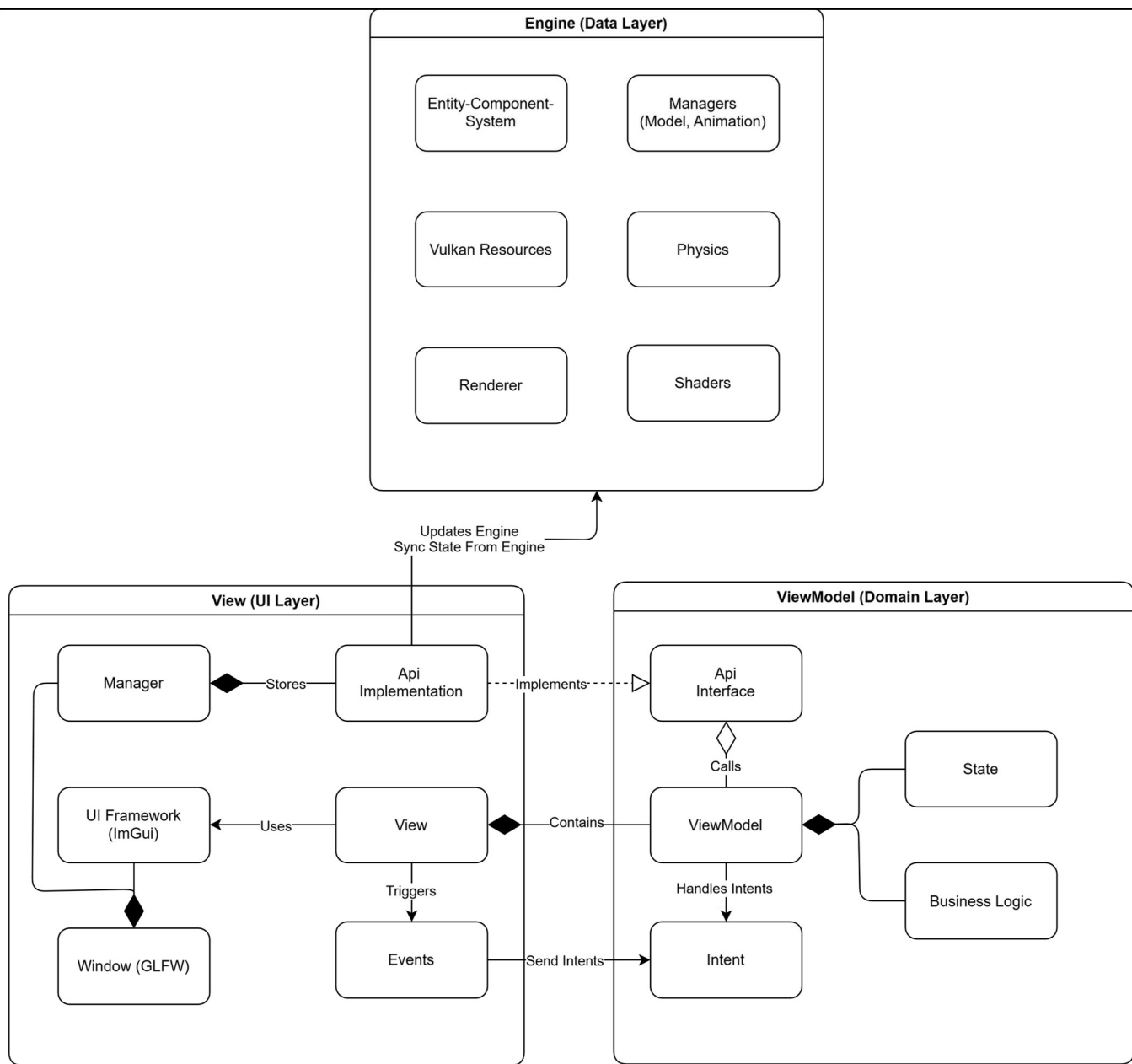
private:
    CommandRegistry() = default;
    std::vector<std::shared_ptr<ICommand>> _undoStack;
    std::vector<std::shared_ptr<ICommand>> _redoStack;
};
```

<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 source of truth** for all runtime data.
- ▶ The UI is purely declarative and communicates exclusively 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.
- ▶ **Architecture follows Single Responsibility, Dependency Inversion and layered design, resulting in a modular, scalable and maintainable editor architecture.**



Software Architecture XXV.

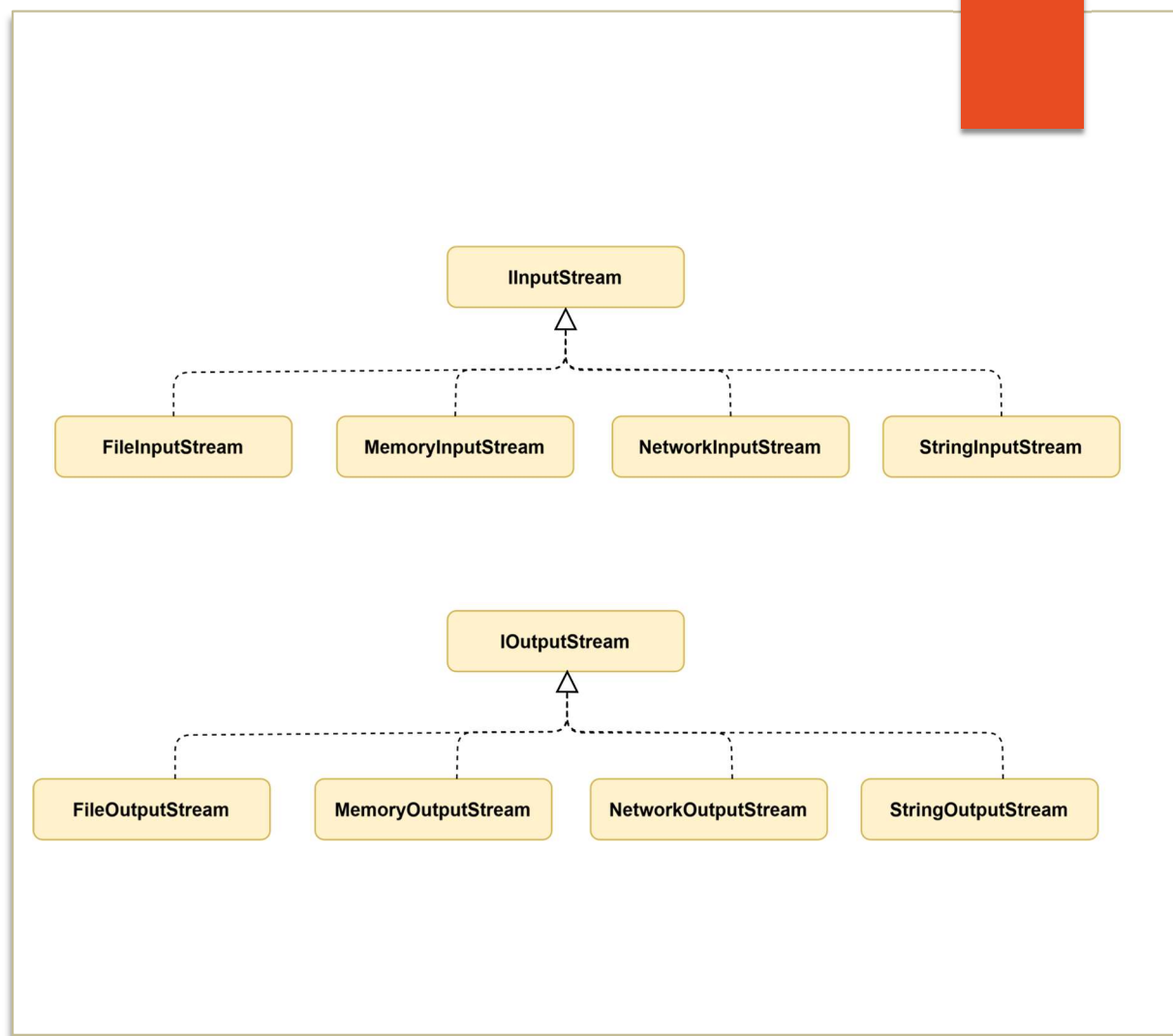
Serialization Architecture – Design Goals

- ▶ 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 independently 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 structures and efficient binary persistence.
- ▶ 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 modifying 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 separated 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 modifying** 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 remaining independent from the actual data types.
- ▶ Schemas define the **serializable structure** of complex engine types and explicitly select which properties are persistent
- ▶ **Template metaprogramming** recursively resolves types.
- ▶ Complex types are decomposed recursively through their **Schema<T>** specialization.
- ▶ Data structures remain **serialization-agnostic**, components do not contain JSON, XML, YAML or binary-specific logic.
- ▶ Serialization logic is **reusable across archive** formats.

```
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 exchange and **networking scenarios**.
- ▶ **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.**

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

    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;
};

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

    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;
};
```

```
"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
  }
},
```

```
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 Design Principles, Closing Thoughts

- ▶ **Layered Architecture** separates responsibilities across systems and allows individual implementations to evolve independently.
- ▶ **Single 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 interfaces.
- ▶ **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 goal is not to simply 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-oriented 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 rather than a sequence of CPU-issued commands.

Meshlet-Based Rendering

- ▶ Large meshes are decomposed into small, independently processable meshlets.
- ▶ Meshlets provide a natural granularity for GPU culling and rendering.
- ▶ Task and Mesh shaders allow geometry amplification and processing 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 dynamically 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

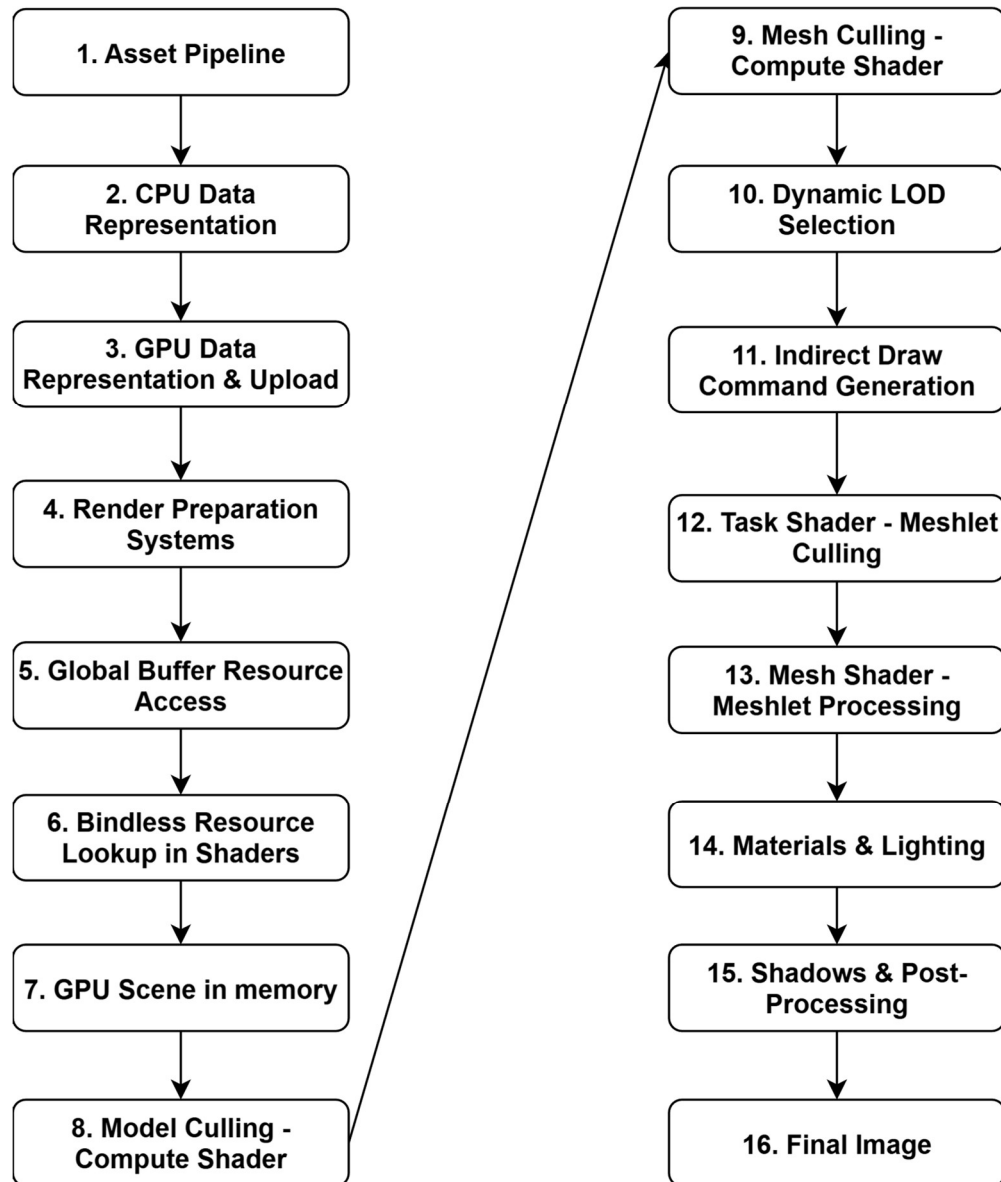
- ▶ Rendering workloads should expose as much parallel work as possible.
- ▶ GPU workloads are structured around massively parallel execution.
- ▶ CPU-side preparation is also parallelized where applicable.
- ▶ Task dependencies should be minimized and explicit.

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, Descriptor Buffers, Mesh Shaders, and Indirect Execution work together as part of the architecture.
- ▶ API decisions are driven by rendering requirements rather than feature availability 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 remain extensible as new rendering techniques are introduced.

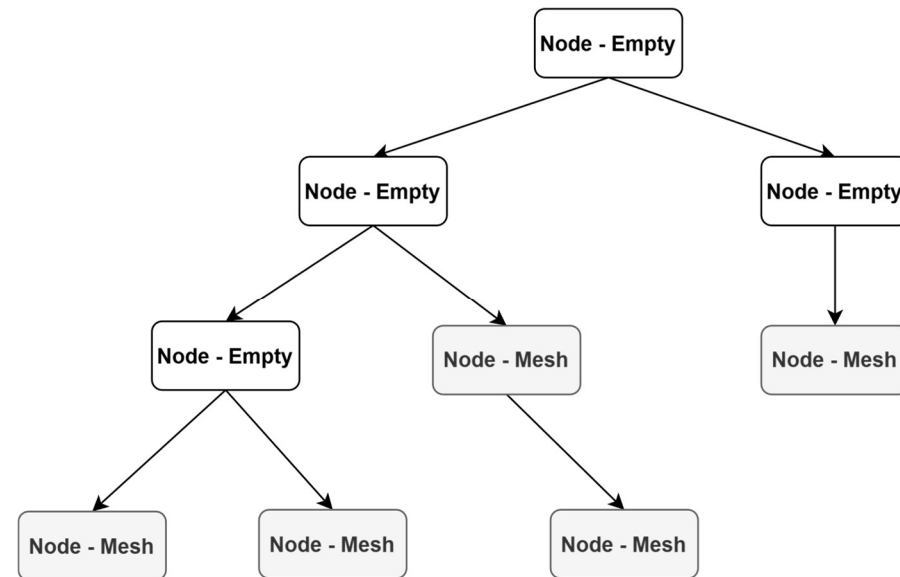


Rendering Architecture IV.

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.
- ▶ This representation separates geometry from hierarchical transform, 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, tangents, uvs, so this **improves cache locality, memory bandwidth 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(addr.indices, gl_VertexIndex);
    GpuVertexPosition v = GET_VERTEX_POS(addr.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(addr.nodeTransforms, nodeIndex);
    mat4 finalModelMat = staticNodeTransform.globalTransform;

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

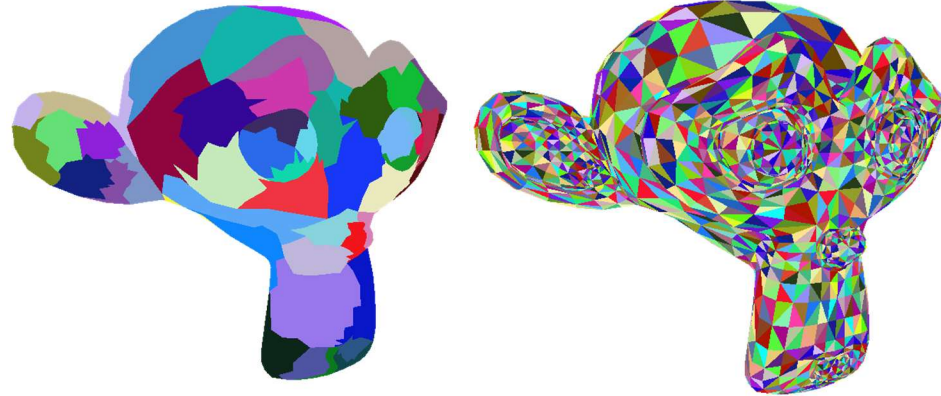
    // Calculate Final World Position and Outputs
    vec4 clipPos = viewProj * transform.transform * finalModelMat * vec4(v.position, 1.0);
    gl_Position = clipPos;
}
```

<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 partitioning.
- ▶ 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
);
```

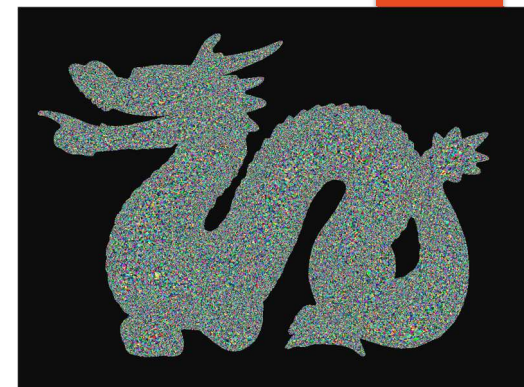
<https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/Mesh/Processor/MeshProcessor/Meshlet/MeshoptimizerMeshletProcessor.cpp>

Model Representation IV.

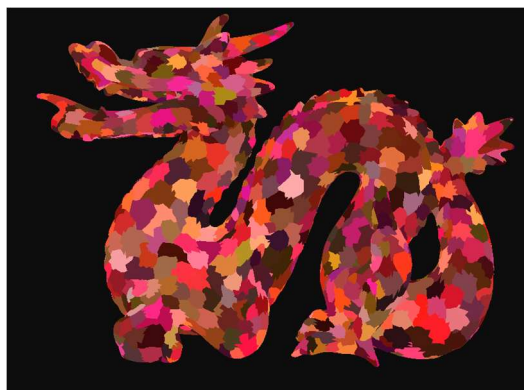
LOD Generation

- ▶ 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 every LOD level**.
- ▶ The final GPU representation therefore contains LOD-specific index data and LOD-specific meshlet data.
- ▶ **The memory layout is critical:** the GPU must 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%



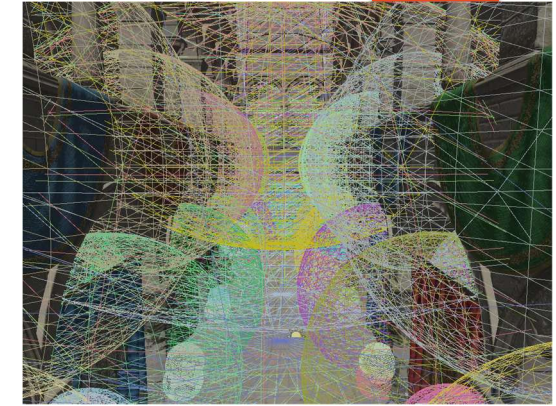
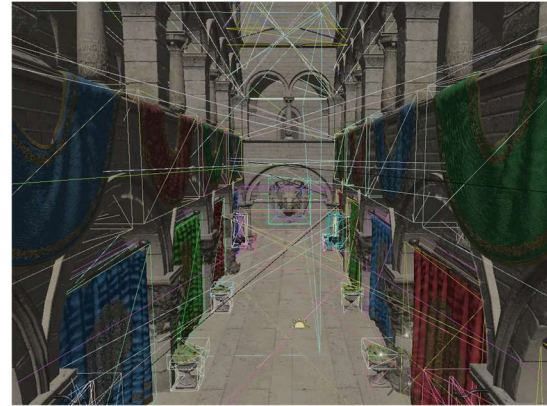
<https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/Mesh/Processor/MeshProcessor/Lod/MeshoptimizerLodProcessor.cpp>

Model Representation V.

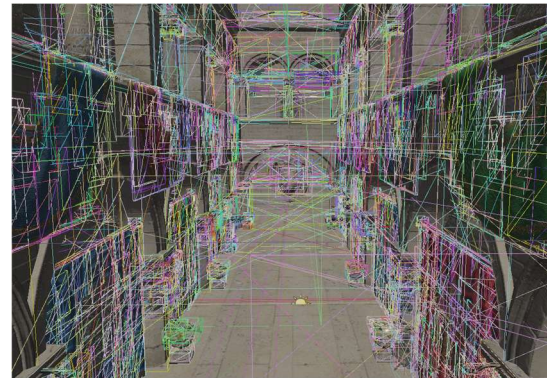
Collider generation

- ▶ 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 dynamically 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 reconstruct each mesh's 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;

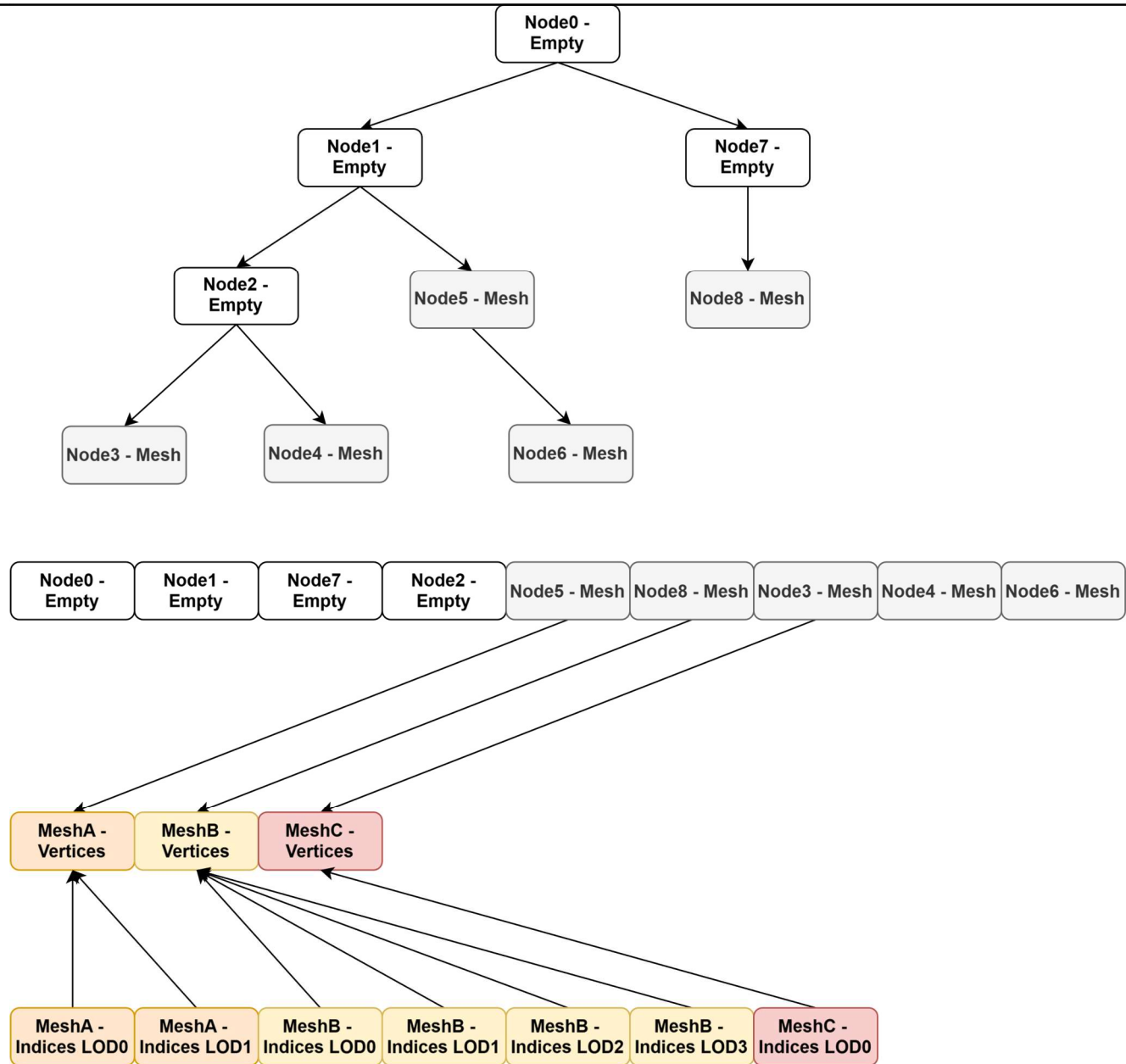
    //Hierarchy Buffers
    std::unique_ptr<Vk::Buffer> nodeTransforms;
};
```

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

Model Representation VII.

Batched Mesh-Based Allocation

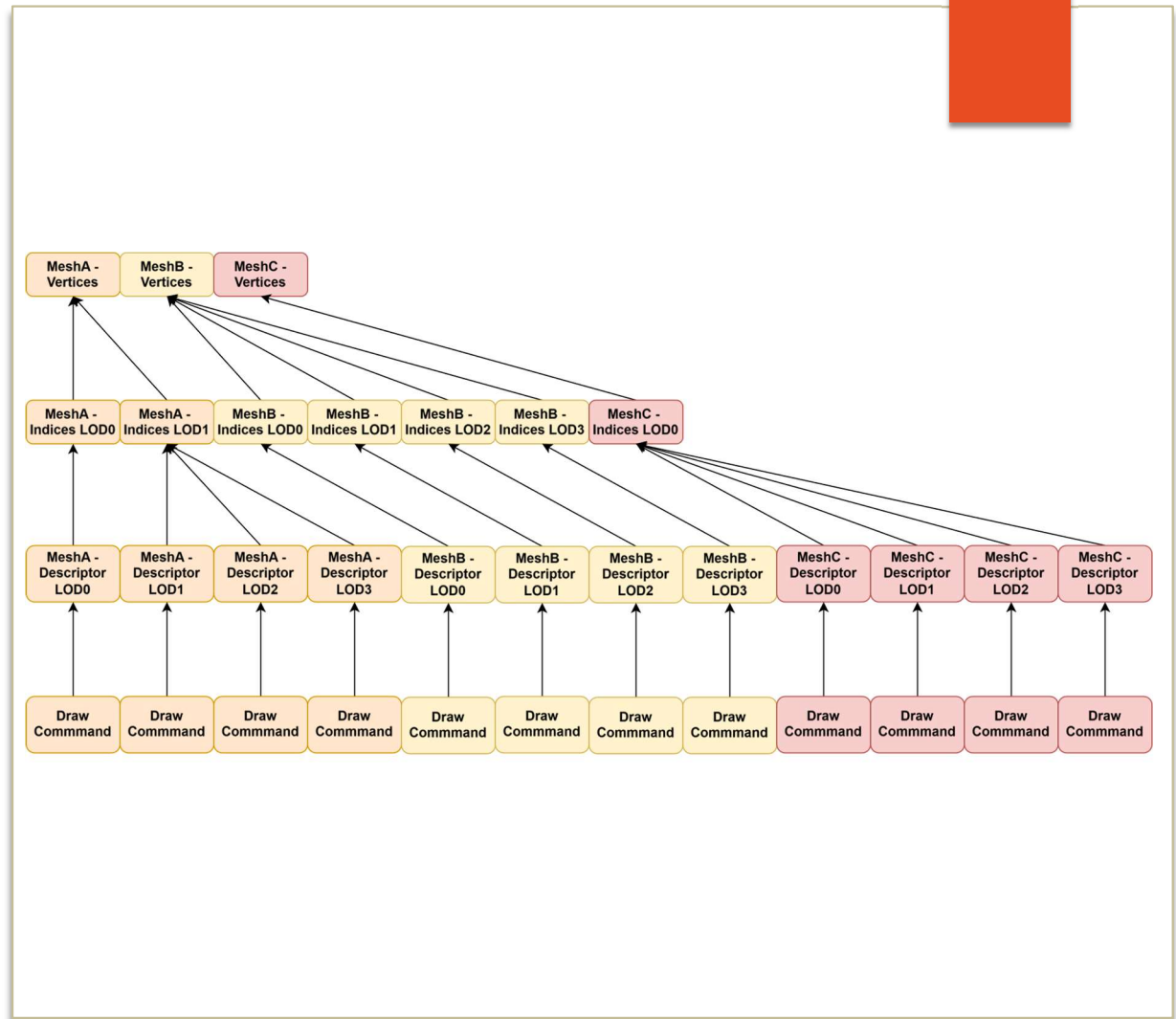
- ▶ Flatten the node hierarchy into a level-contiguous, data-oriented vector representation, preserving parent-child relationships.
- ▶ Traverse the nodes and concatenate each mesh's vertices, indices, collider, meshlets and other data into the corresponding global model buffers.
- ▶ Record the starting 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 challenge 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 Challenge

- ▶ Each mesh 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 exactly 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

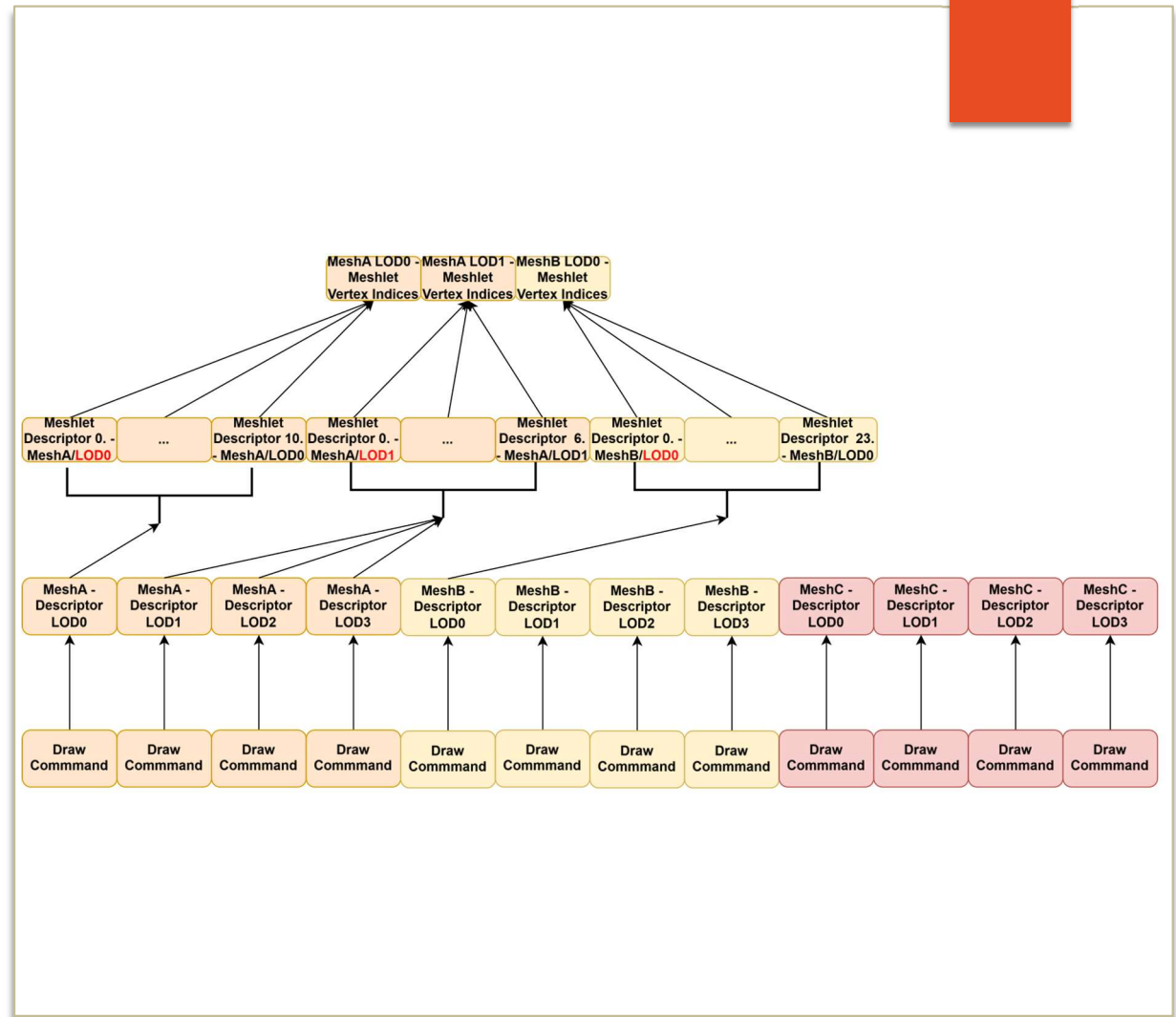


<https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/Mesh/Converter/DefaultGpuModelConverter.cpp>

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.

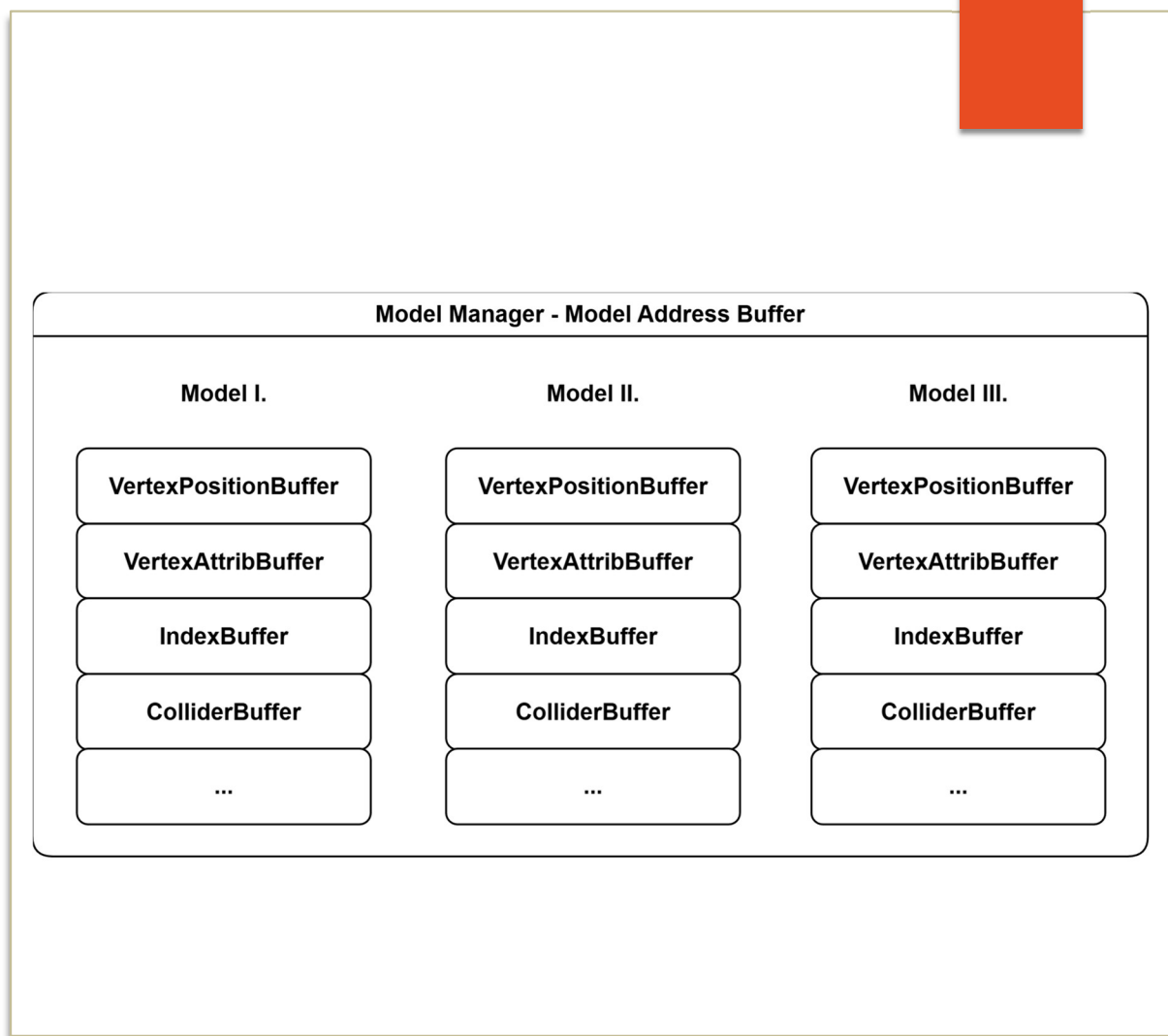


<https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/Mesh/Converter/DefaultGpuModelConverter.cpp>

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, indexed 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 challenge:** how can shaders access a dynamically growing set of GPU buffers without binding each resource individually?

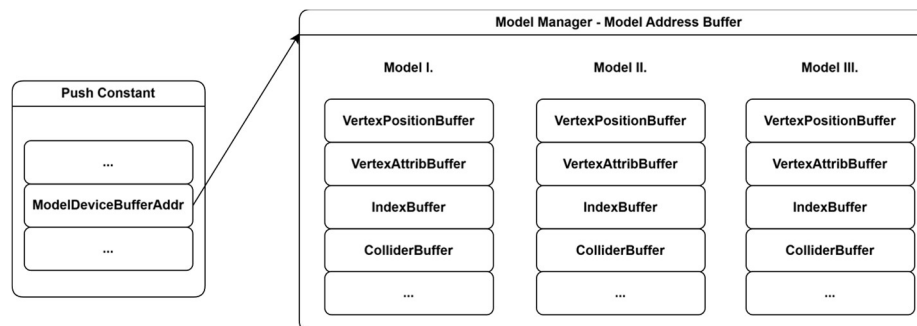


<https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/Manager/AddressResourceManager.h>

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 per-resource binding increasingly complex 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 hierarchical 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 rendered 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 continuously 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 different 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 such 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 corresponding 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 **fetches 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];

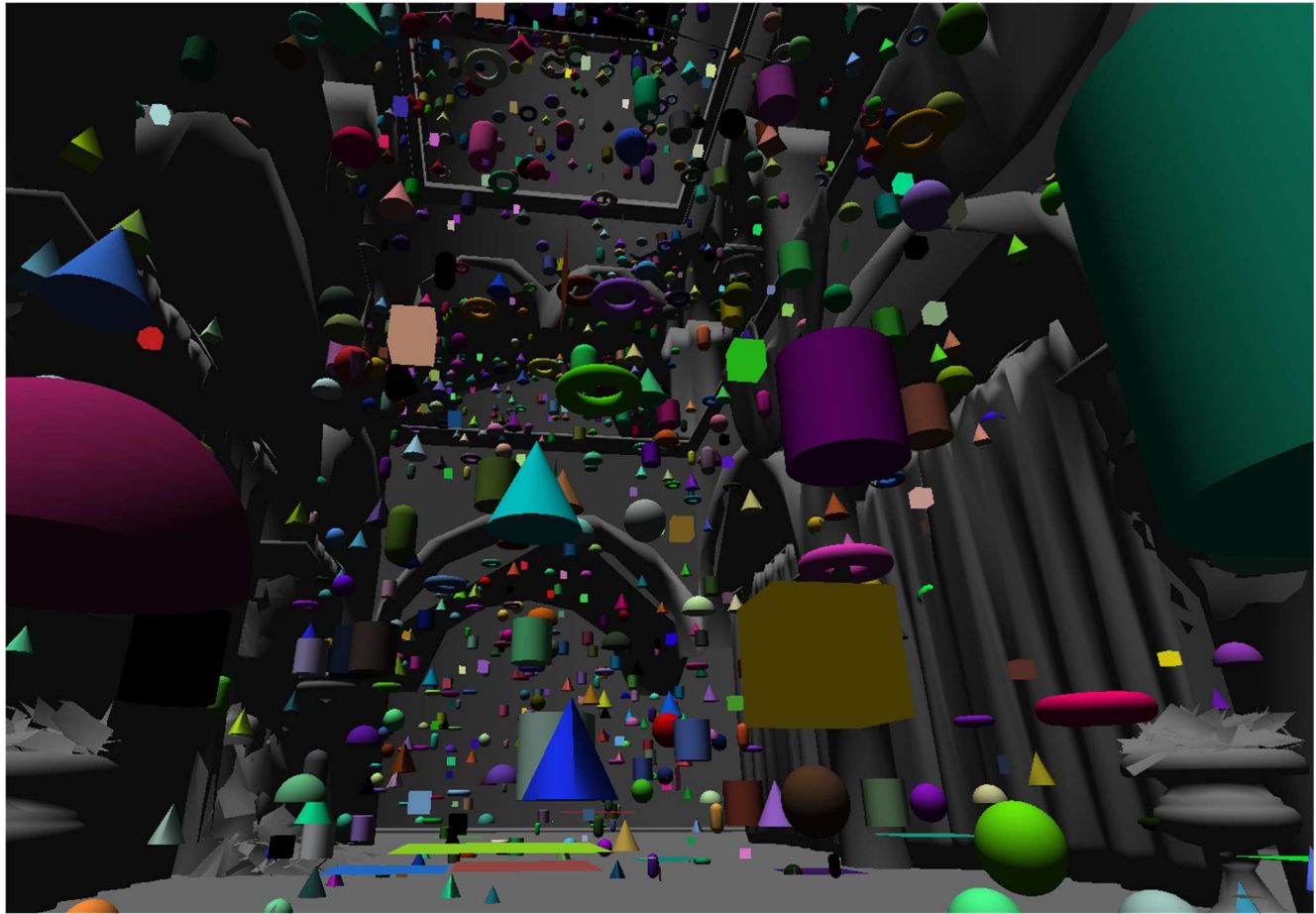
    vec3 position = vertex.position;

    // ...
}
```


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 dynamically 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.

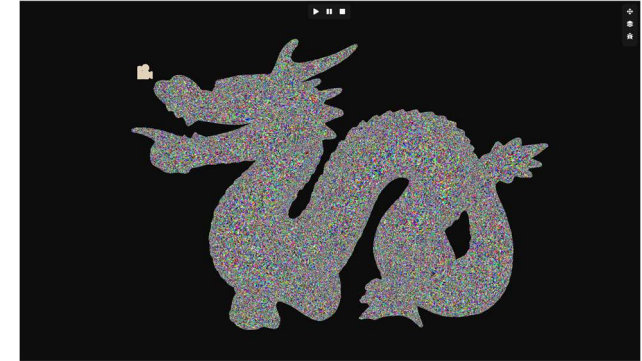
Why Vertex Shaders Are Not Enough?

- ▶ 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 image.
- ▶ 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 too coarse.

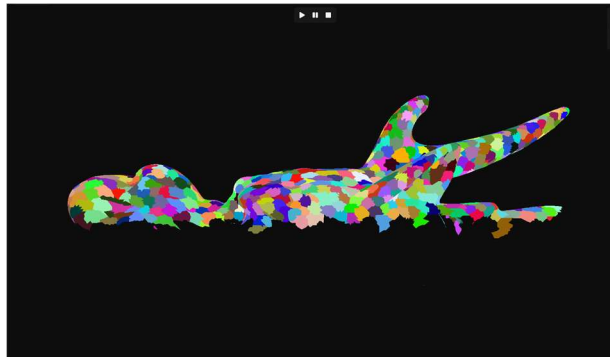
The Scene: Dragon partially visible



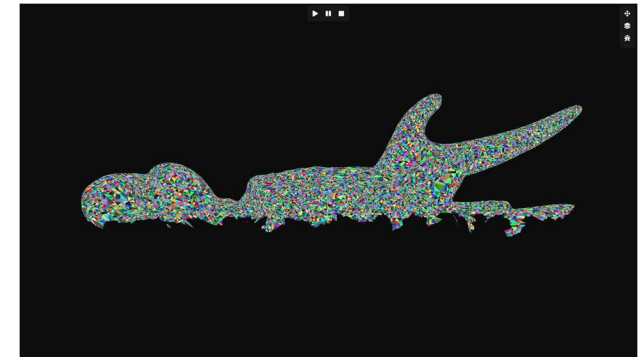
The Vertex Shader: Processes all triangles



The Task Shader: Culls Invisible Meshlets



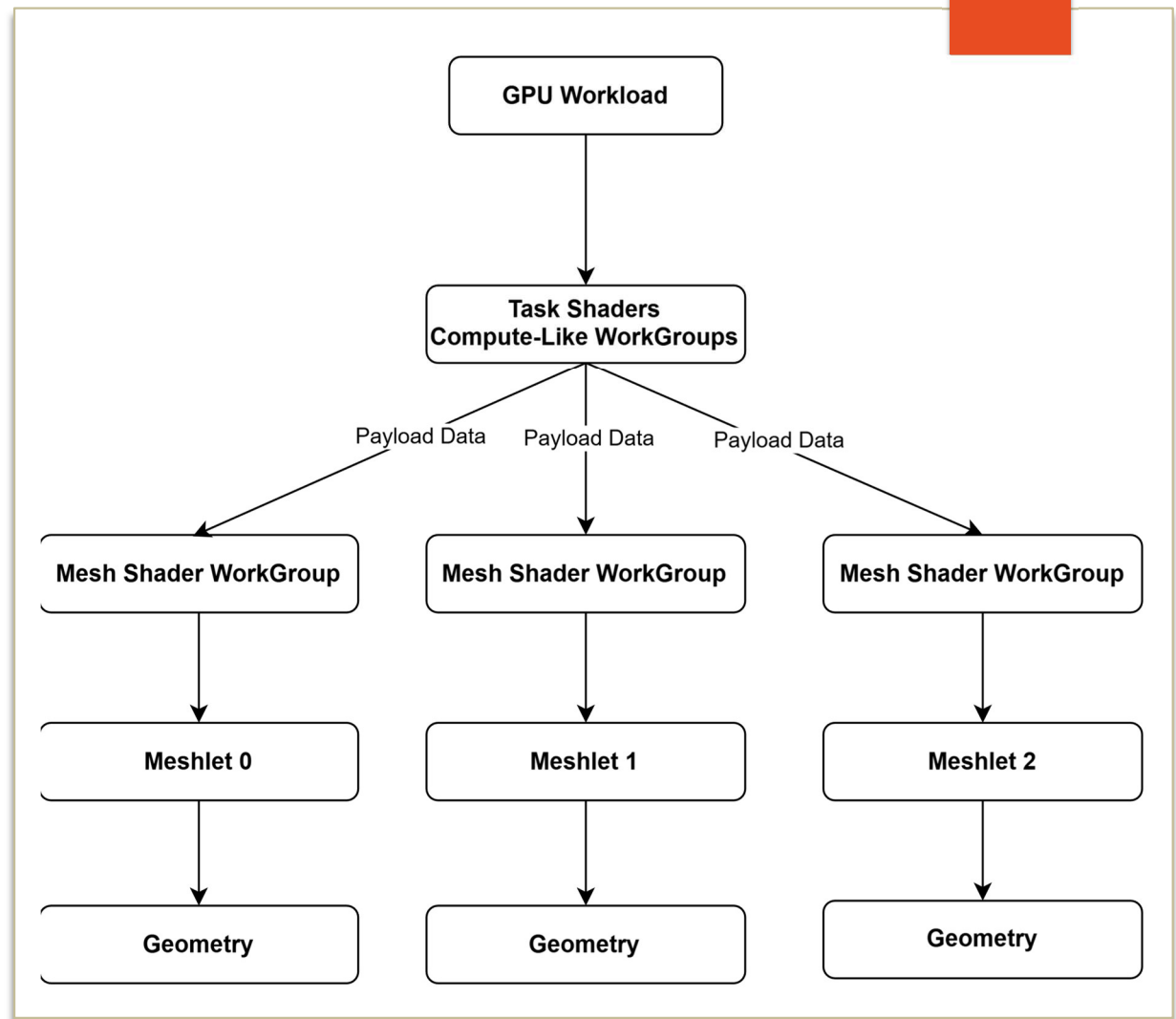
The Mesh Shader: Processes only visible triangles



Rendering IX.

Task & Mesh Shaders – A Compute-Based Graphics Pipeline

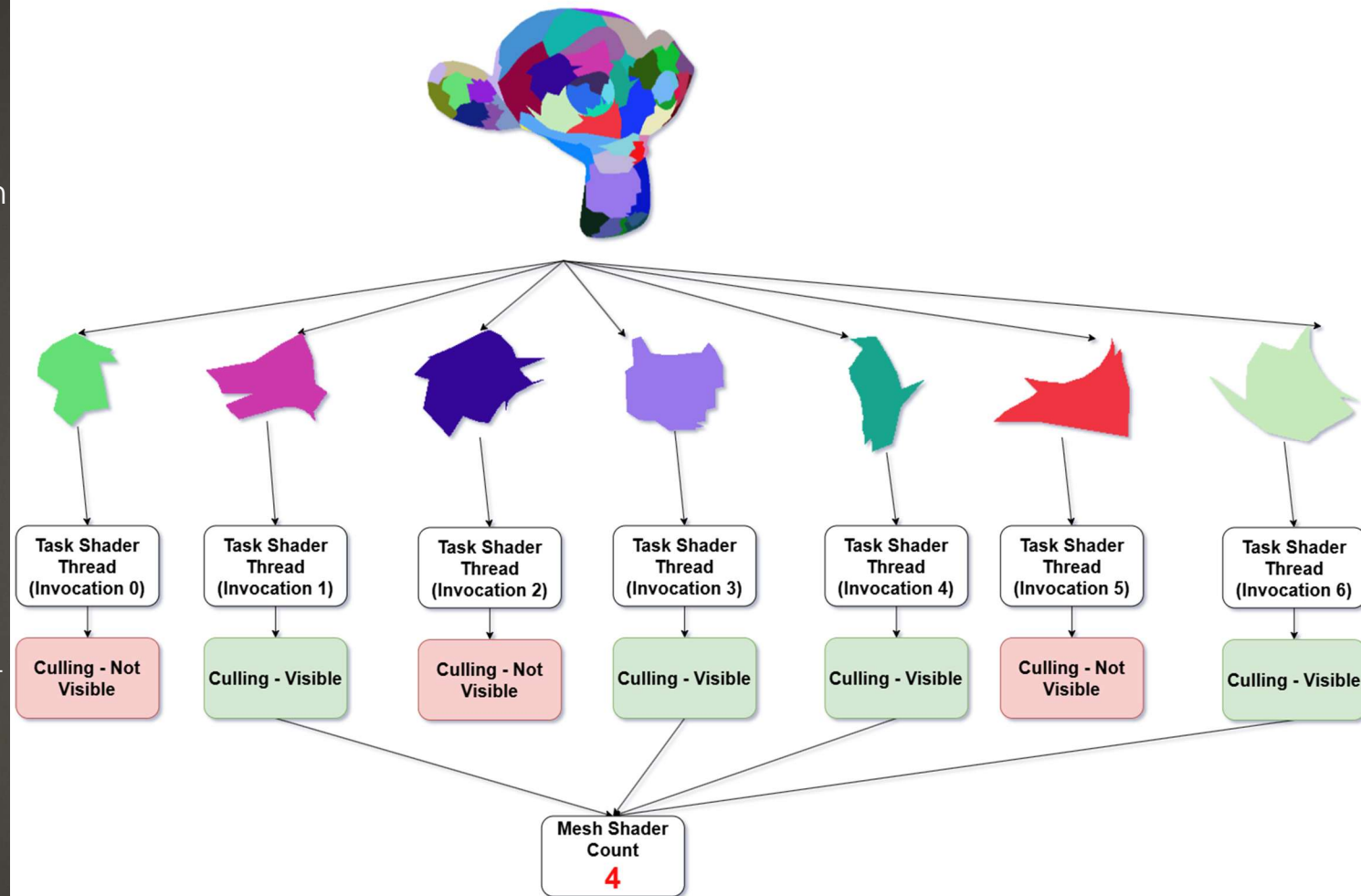
- ▶ Task and Mesh shaders are fundamentally **compute-like shader stages**, using workgroups and built-in invocation IDs to distribute work across threads.
- ▶ The **Task Shader** behaves similarly to a compute shader, but **can dynamically launch a variable number of Mesh Shader workgroups**.
- ▶ The **Mesh Shader** also follows the compute shader execution 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)`

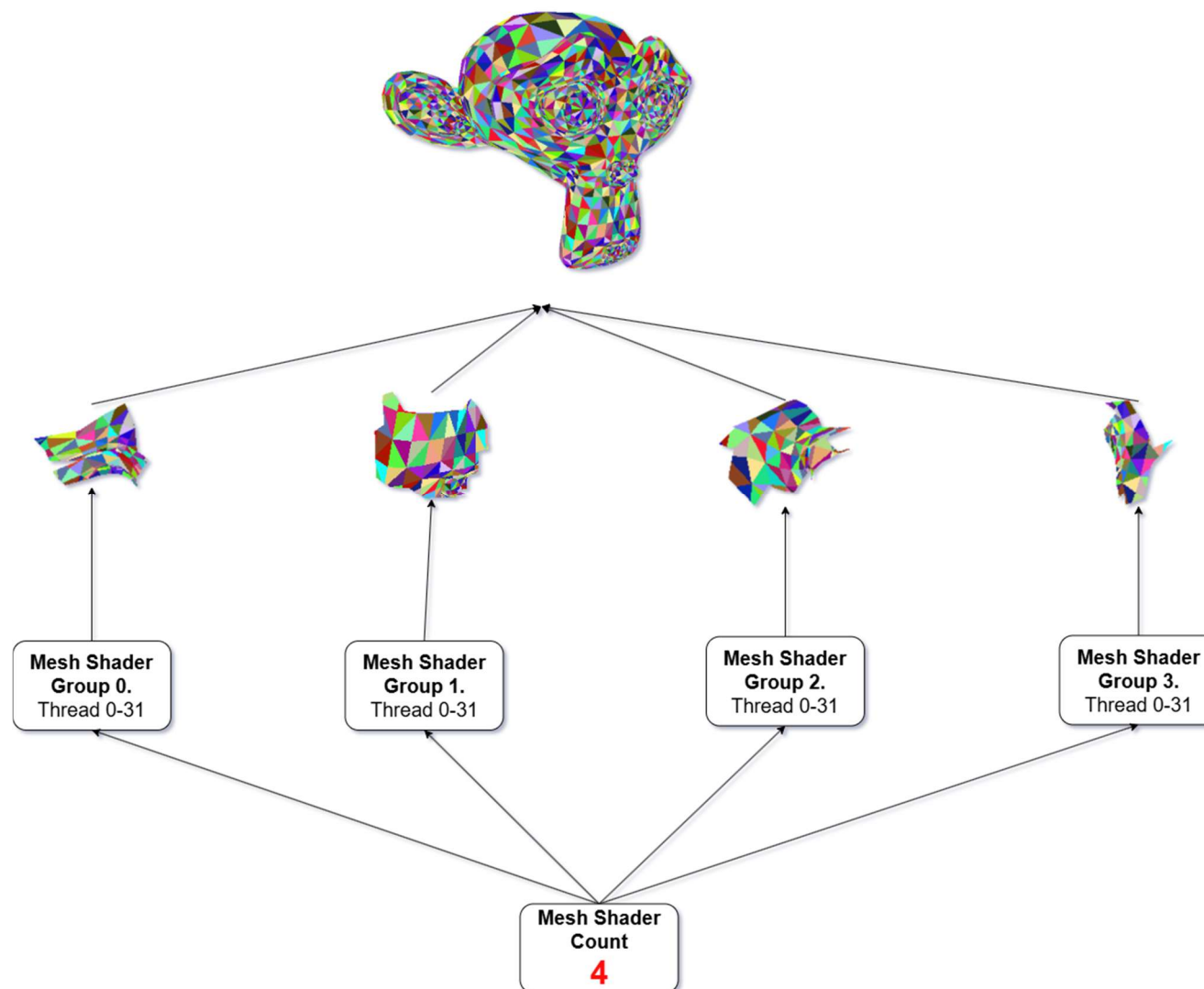


<https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/Shaders/Passes/Shading/Common/Meshlet.task>

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.

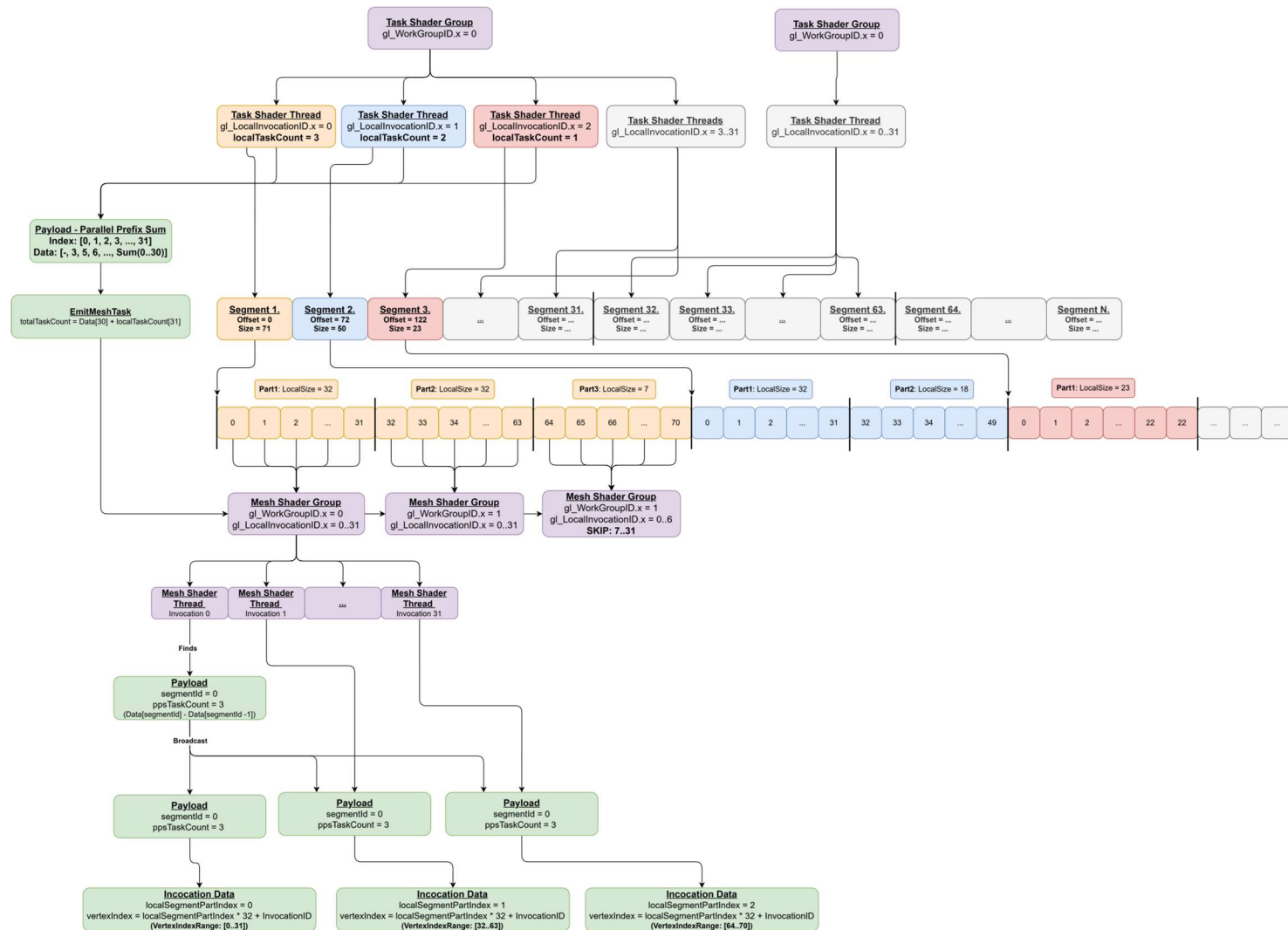


<https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/Shaders/Passes/Shading/Common/Meshlet.mesh>

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;
};
```



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 respective indices.
- ▶ Materials therefore **store only compact texture and sampler IDs**, which are resolved through 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 introduces 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

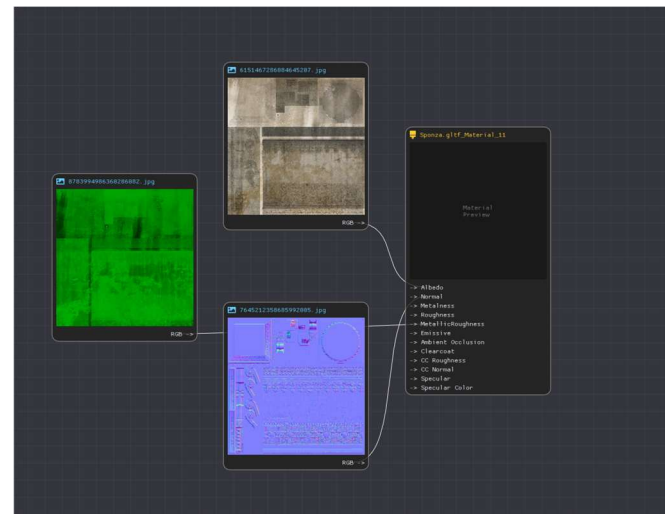
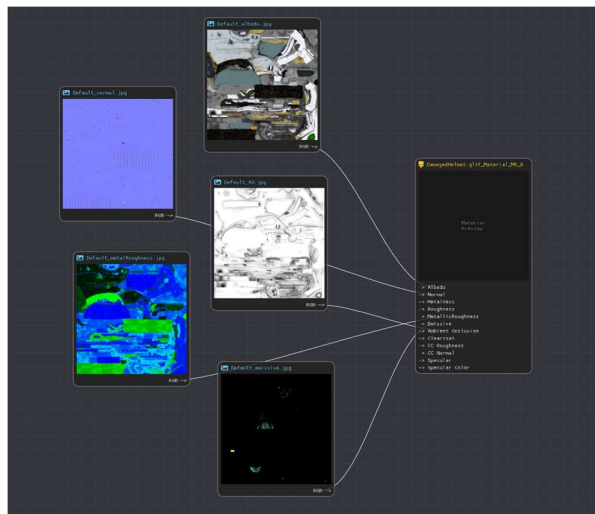
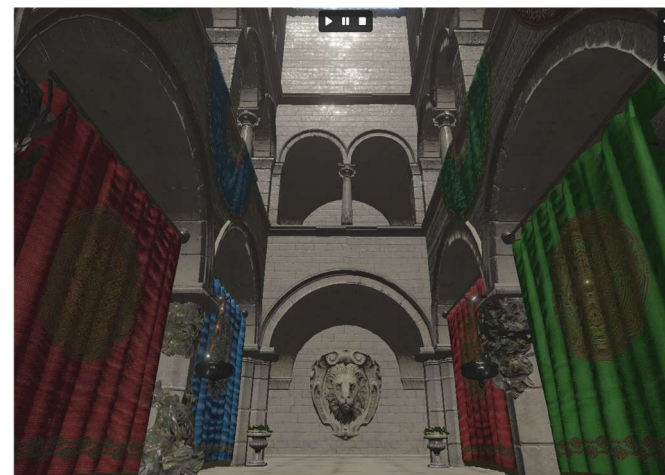
```
void DescriptorWriter::UpdateSet(VkDescriptorSet set) {  
    auto device = ServiceLocator::Get<Vk::Context>()->GetDevice();  
  
    std::vector<VkWriteDescriptorSet> writes;  
    writes.reserve(_imageWrites.size() + _bufferWrites.size());  
  
    for (const auto& imgData : _imageWrites) {  
        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 significantly 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**.

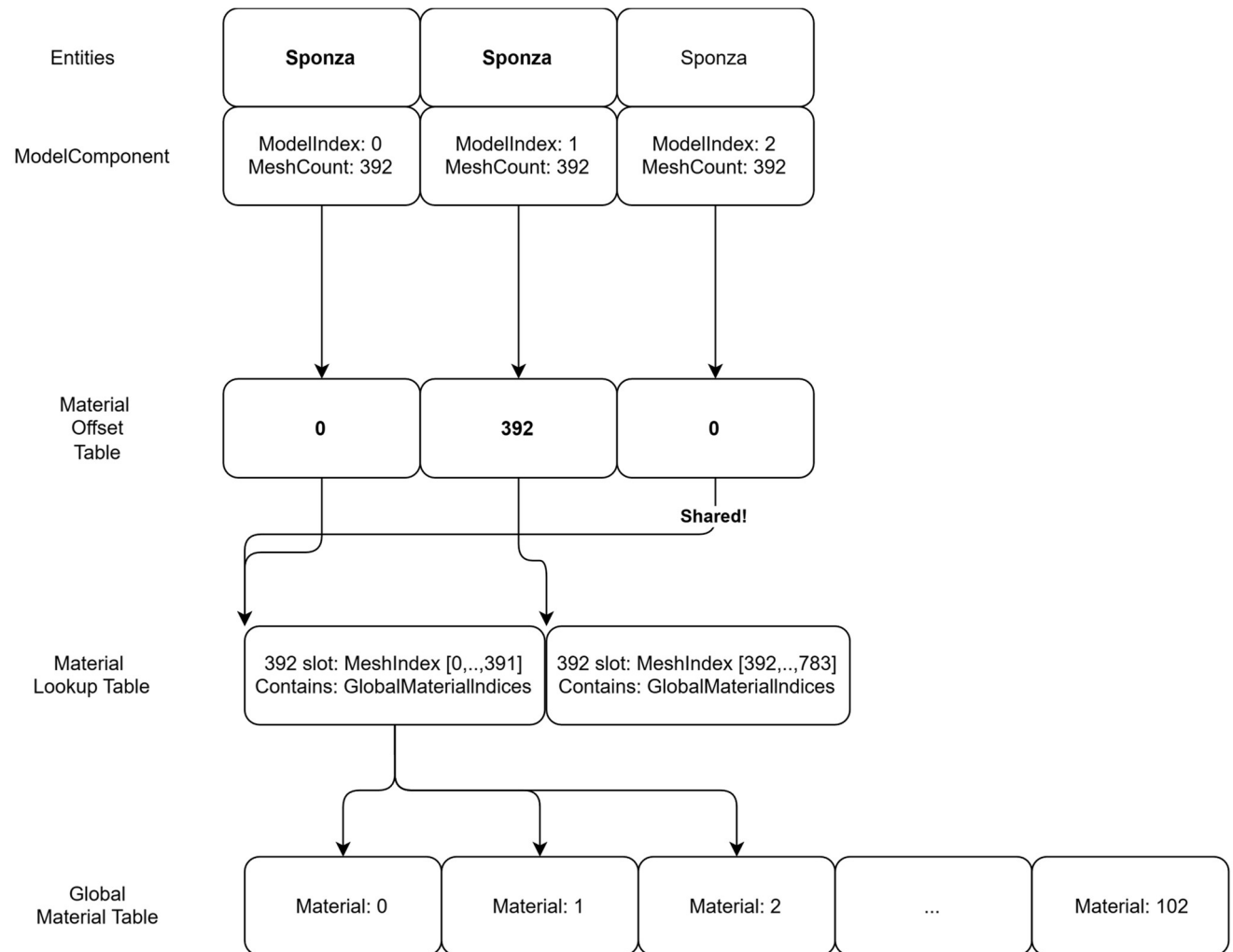


<https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/Vk/Descriptor/DescriptorBuffer.cpp>

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 corresponding material in the Global Material Table.
- ▶ Each **vertex stores its 16-bit mesh index**, which is combined with the model's material offset to resolve the final material: **materialOffset + meshIndex**

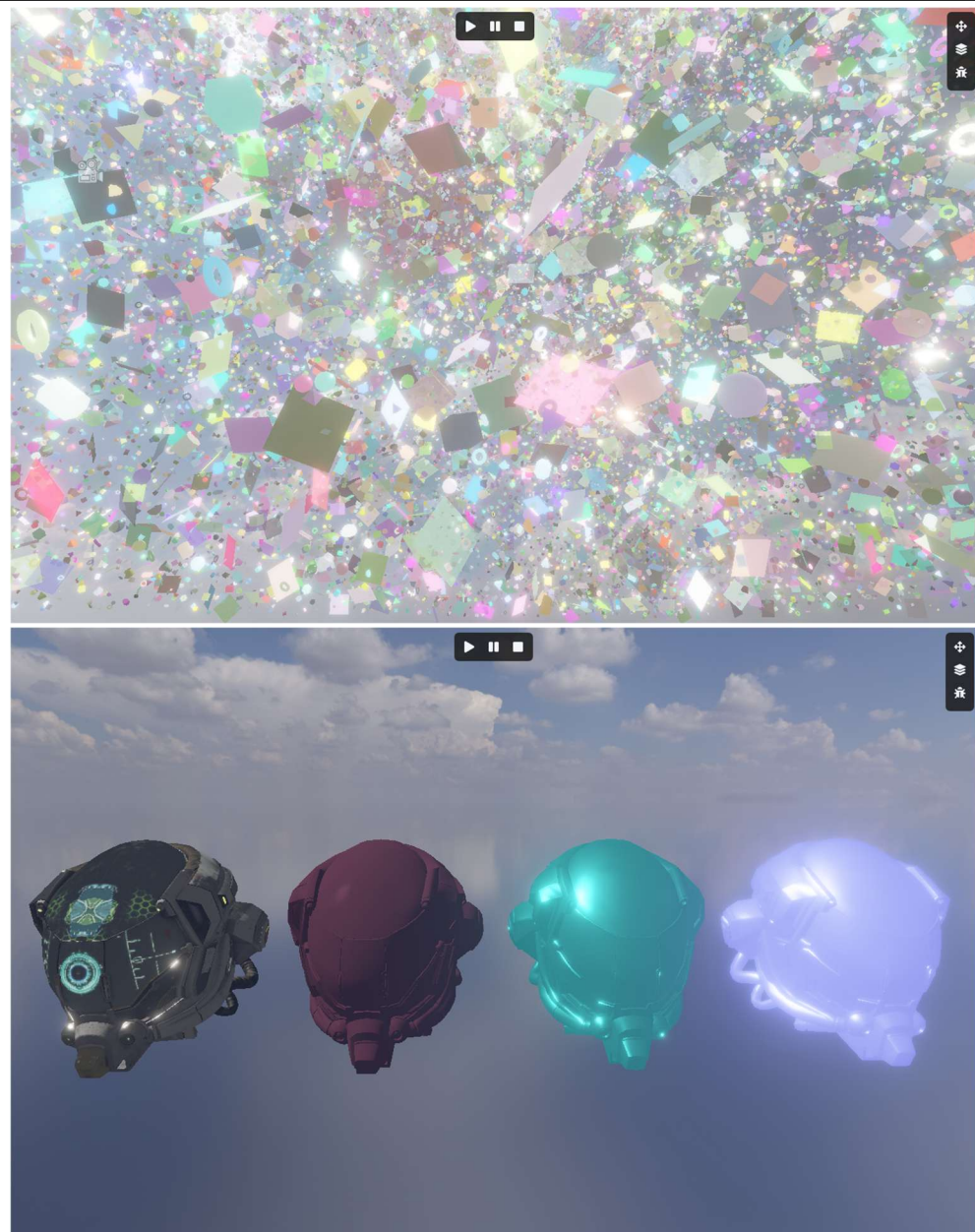


<https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/System/Rendering/MaterialSystem.cpp>

Rendering XVII.

Material Override & Flexibility

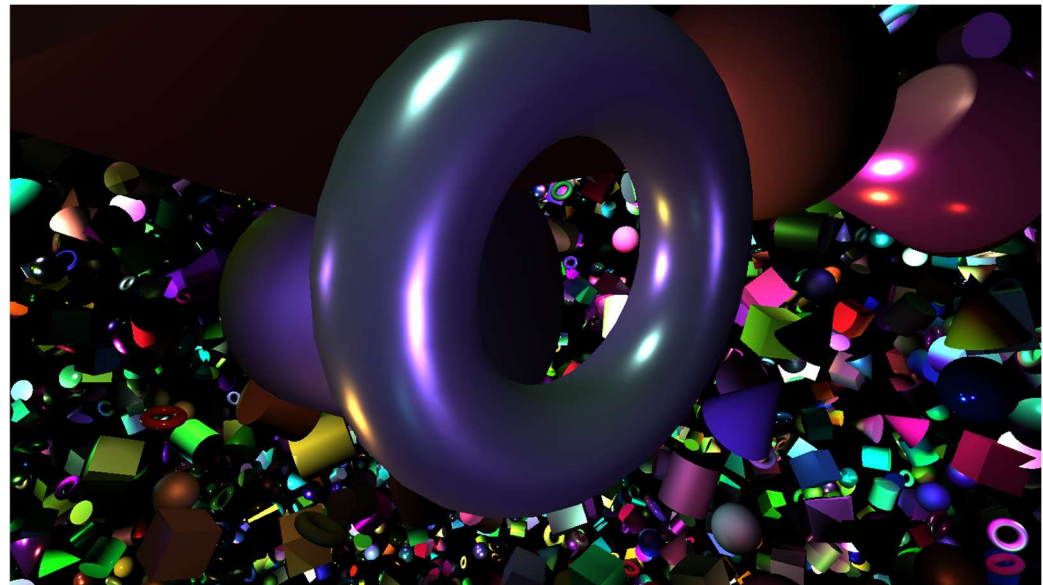
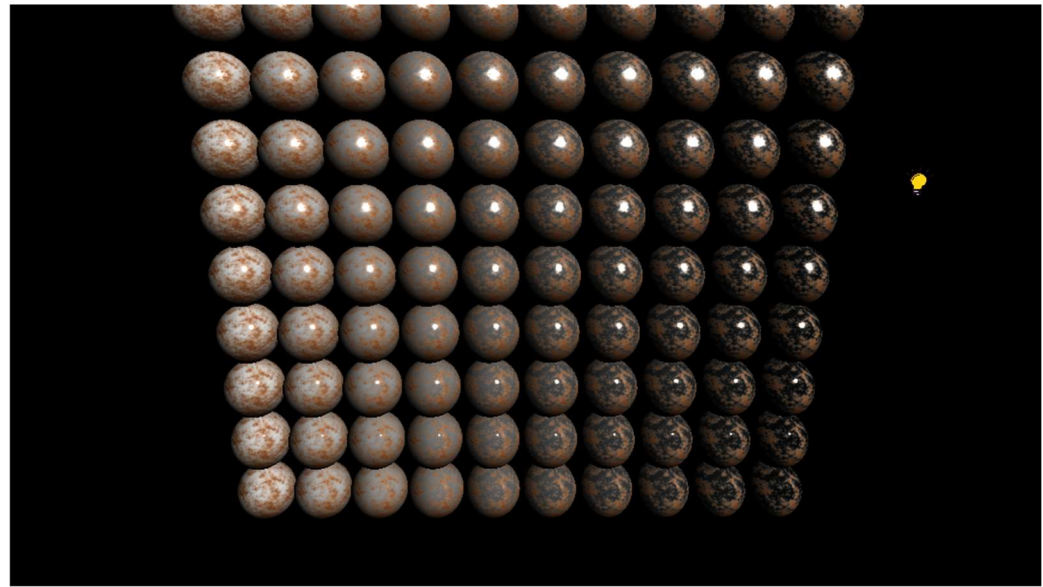
- ▶ Fully indirect rendering allows 100,000 different procedural shape instances to be rendered with a single draw call, while each instance can still reference a different material.
- ▶ The two-level material indirection enables per-mesh material assignment without breaking the GPU-driven rendering model.
- ▶ Complex models can be overridden 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 currently 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 diffuse and specular reflections**, with metallic surfaces transitioning from diffuse-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.

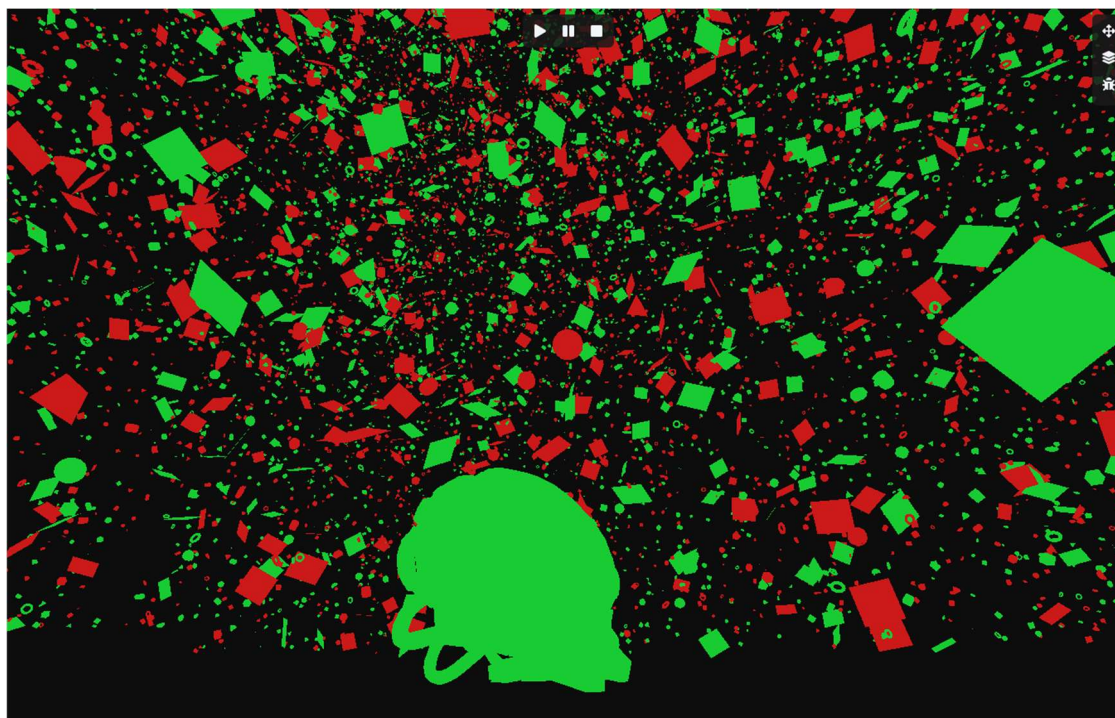
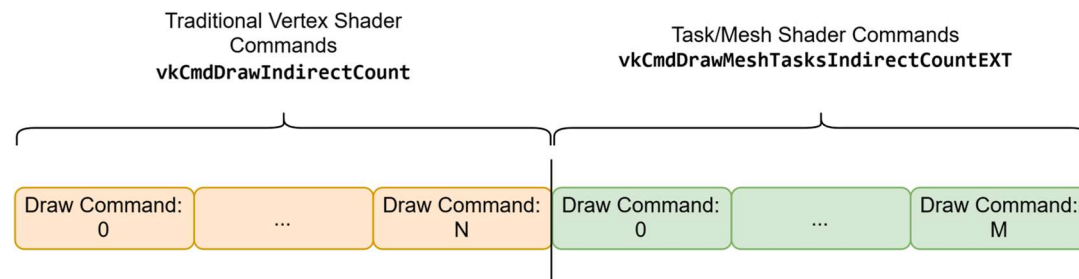


<https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/Shaders/Includes/Utils/PbrMath.glsl>

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 `vkCmdDrawMeshTasksIndirectCountEXT`.
- ▶ 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.

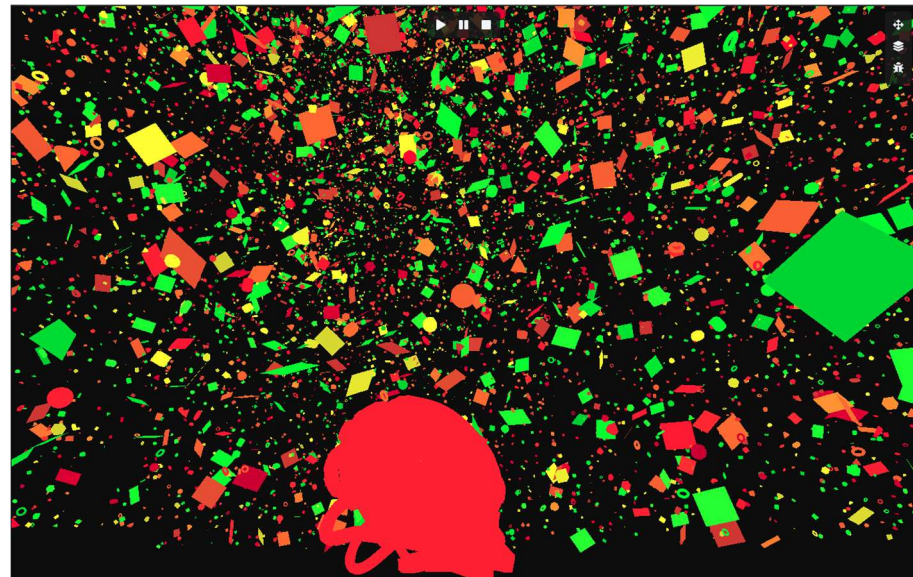
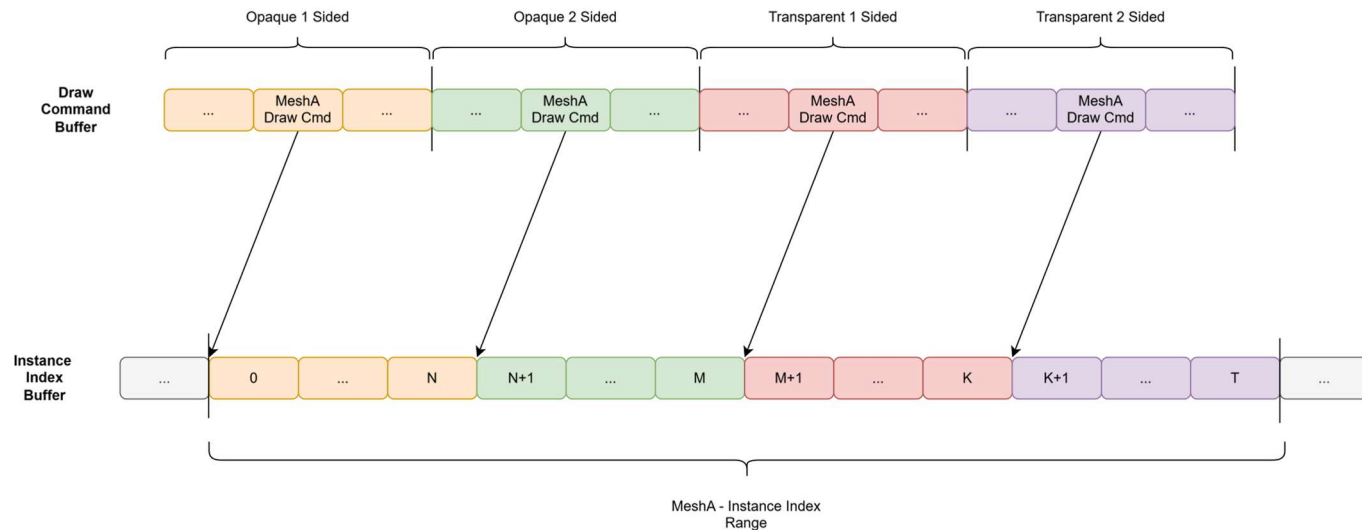


Red = Vertex Shader Meshes | Green = Mesh Shader Meshes

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.

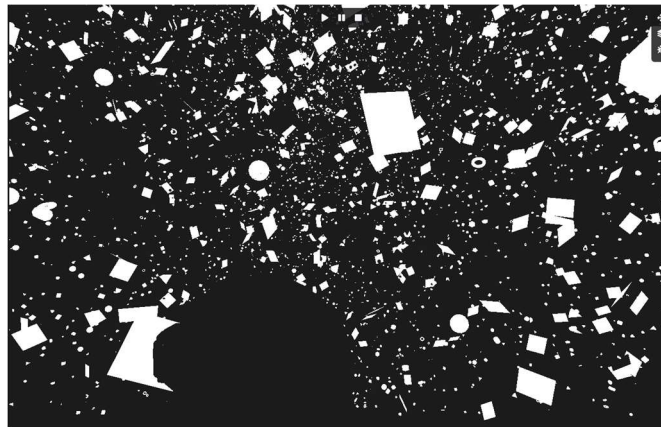
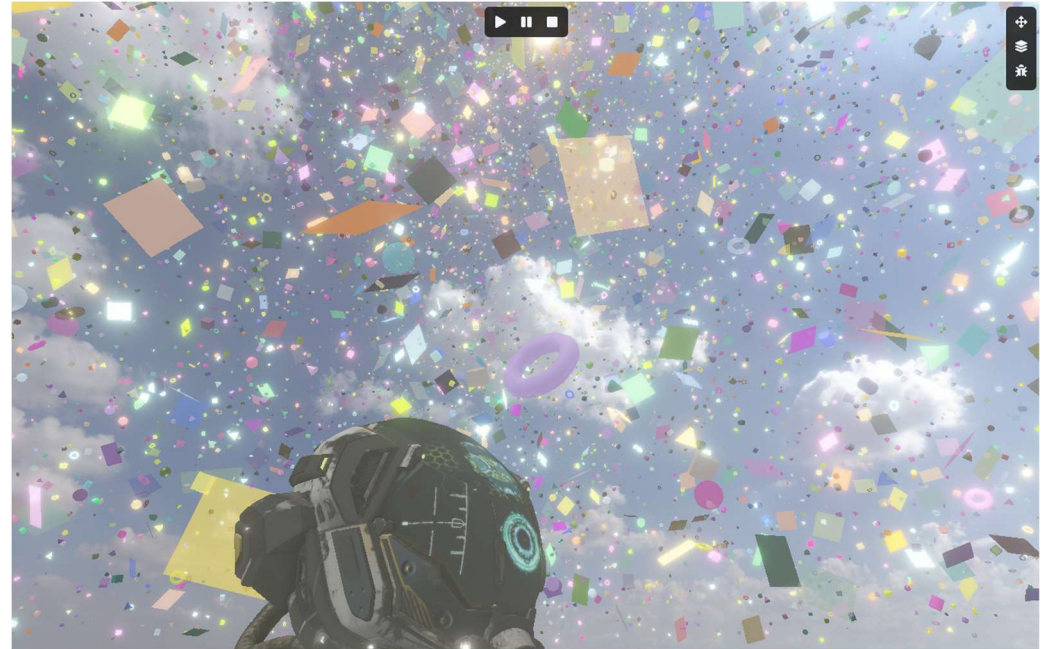


Red/Orange/Yellow/Green = Opaque1/Opaque2/Trans1/Trans2

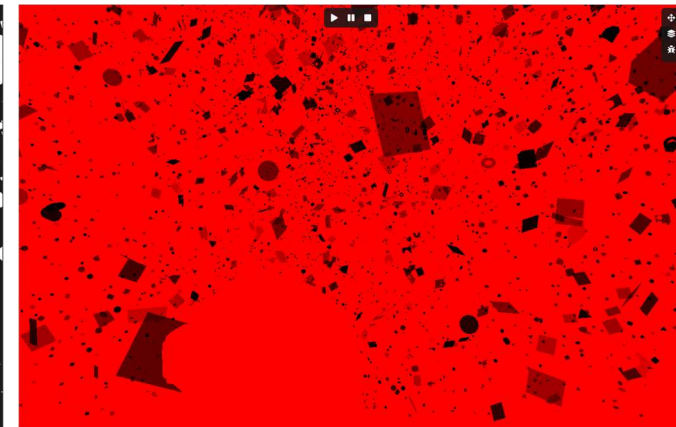
Rendering XXI.

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 because it exceeds 1.0f)

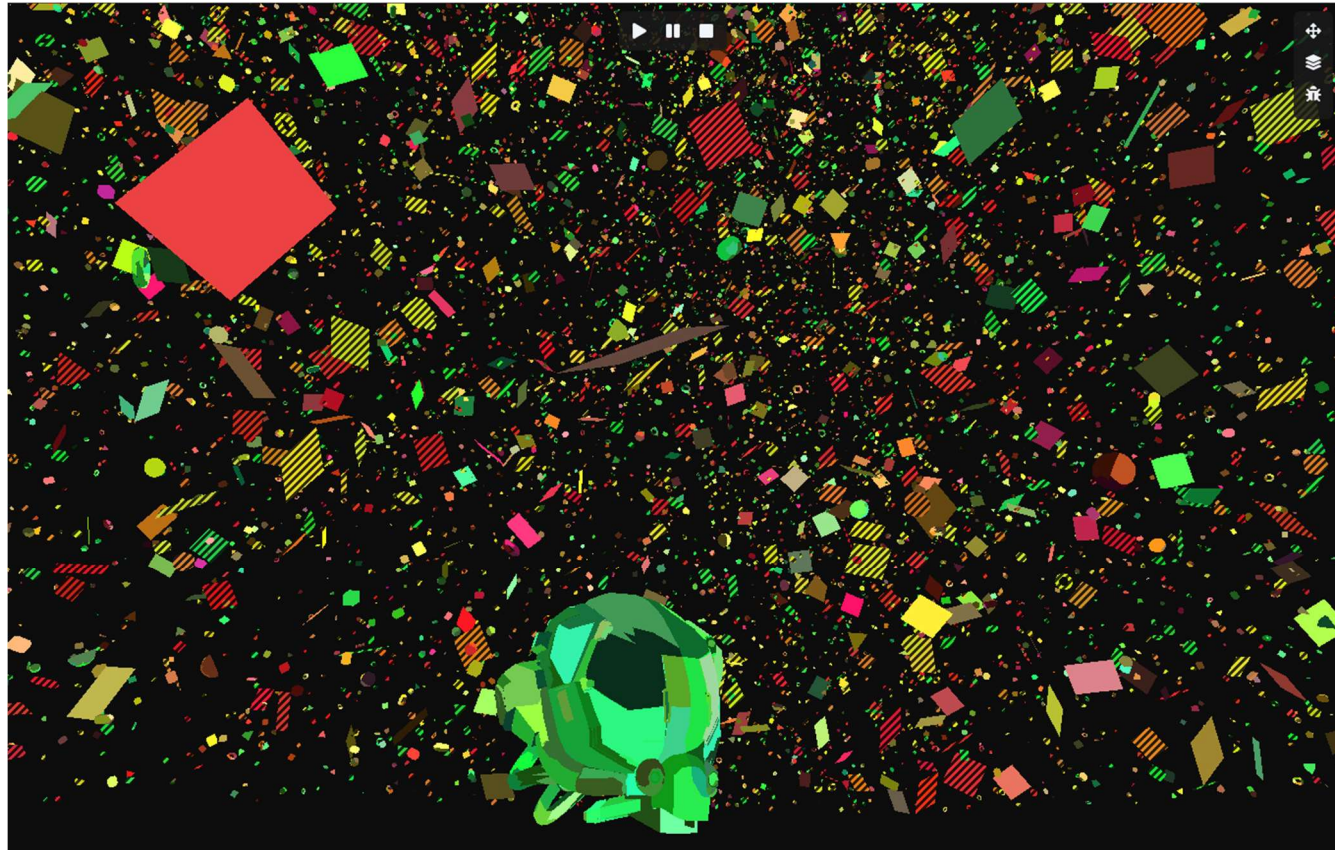


Revealage Texture

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;

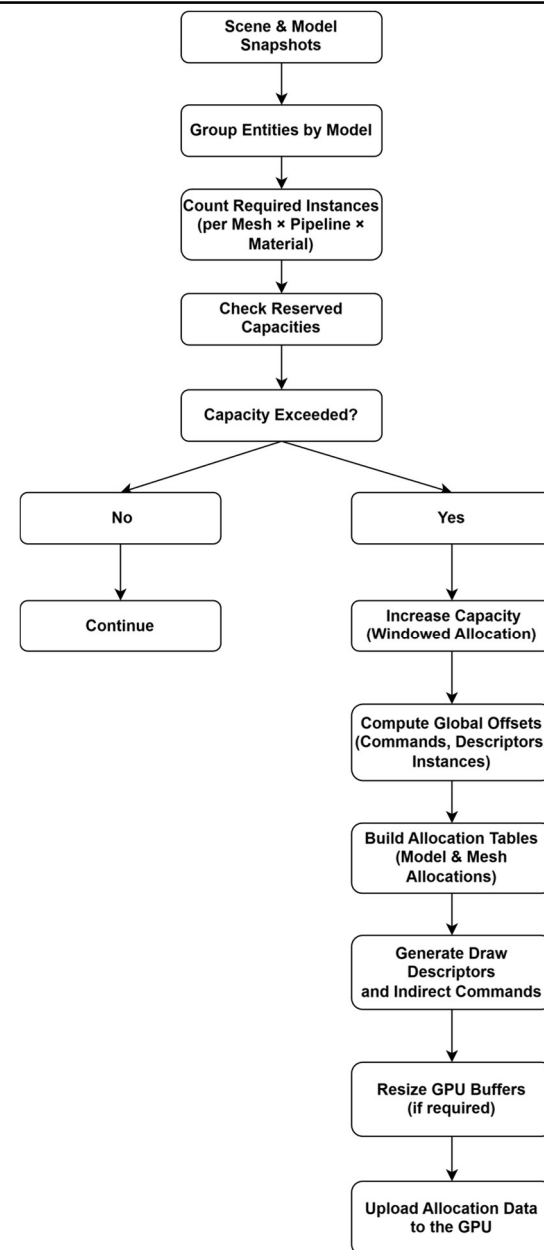
        // 2. Evaluate Albedo & Alpha
        vec4 albedoAlpha = EvaluateAlbedoAlpha(ctx.textureMetadataBufferAddr, mat, finalUV);
        if (IS_ALPHA_TESTED(mat) && albedoAlpha.a < ctx.alphaLimitDiscard) {
            discard;
        }

    #endif
}
```


Rendering XXIV.

Render System

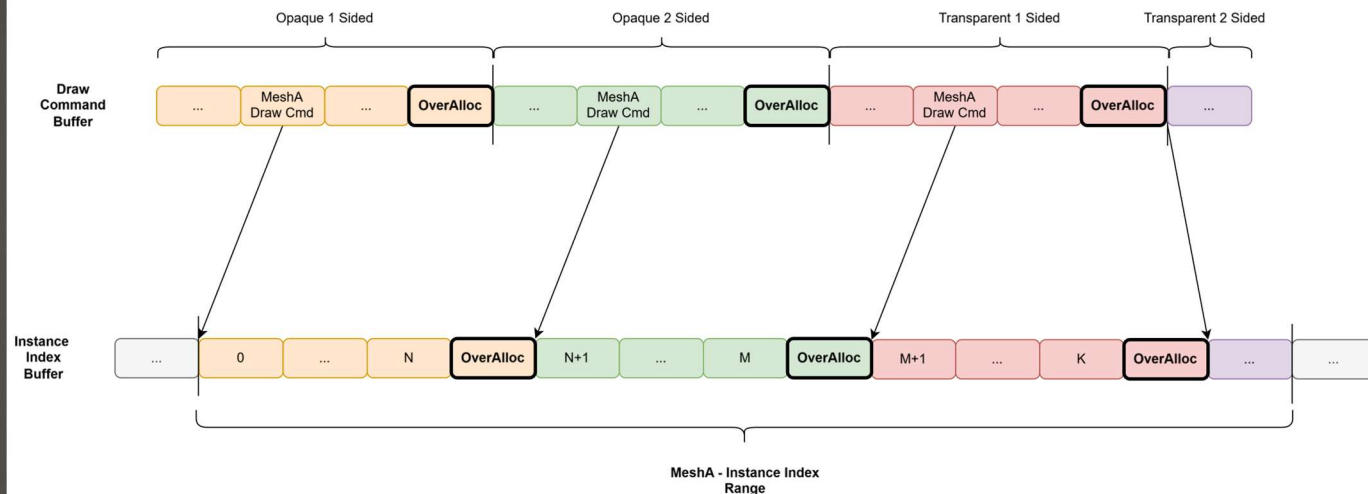
- ▶ The Render System maintains a **single batched draw command buffer**, partitioned into the 16 rendering buckets.
- ▶ A single **batched instance index buffer** stores instance data for all meshes and buckets.
- ▶ Each mesh is assigned a **predefined region** in the instance buffer, allowing the GPU 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.

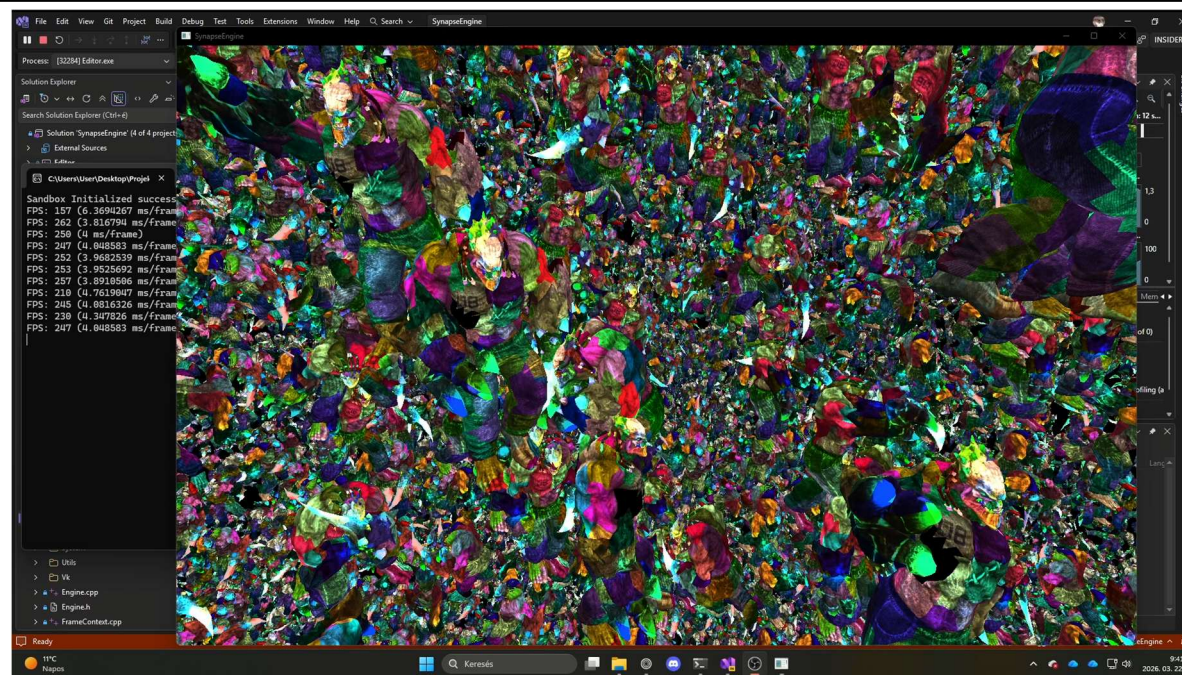


<https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/System/Rendering/RenderSystem.cpp>

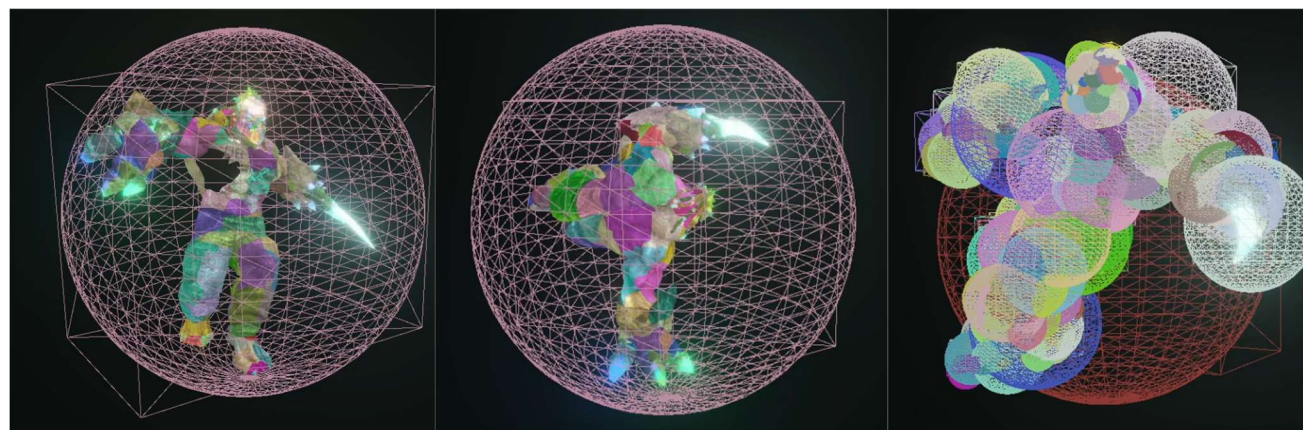
Rendering XXVI.

Unified Static and Animation Rendering

- ▶ The engine supports **skeletal animation through bone indices and bone weights**, allowing individual vertices to be dynamically transformed by the corresponding bone matrices.
- ▶ Static and animated objects do **not require separate rendering pipelines or shaders**. Both are processed by the same GPU-driven rendering architecture.
- ▶ Every animated asset retains its original static mesh representation, which acts as the default bind/T-pose geometry.
- ▶ If no animation is associated with an entity, the original static vertex data is rendered directly.
- ▶ When animation data is available, the **shader dynamically resolves the entity's animation state** and accesses the corresponding bone transformation data.
- ▶ The vertex bone indices and weights are then used to apply skeletal skinning before the vertex continues through the regular rendering pipeline.
- ▶ From the renderer's perspective, an **animated object is therefore handled exactly like any other instance**: the difference is resolved dynamically during vertex processing.



https://www.youtube.com/watch?v=TXt_5GbGFfc



<https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/Shaders/Passes/Shading/Common/Traditional.vert>

Synapse Engine: Culling

Created By: Péter Tamás

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

2026

Culling I.

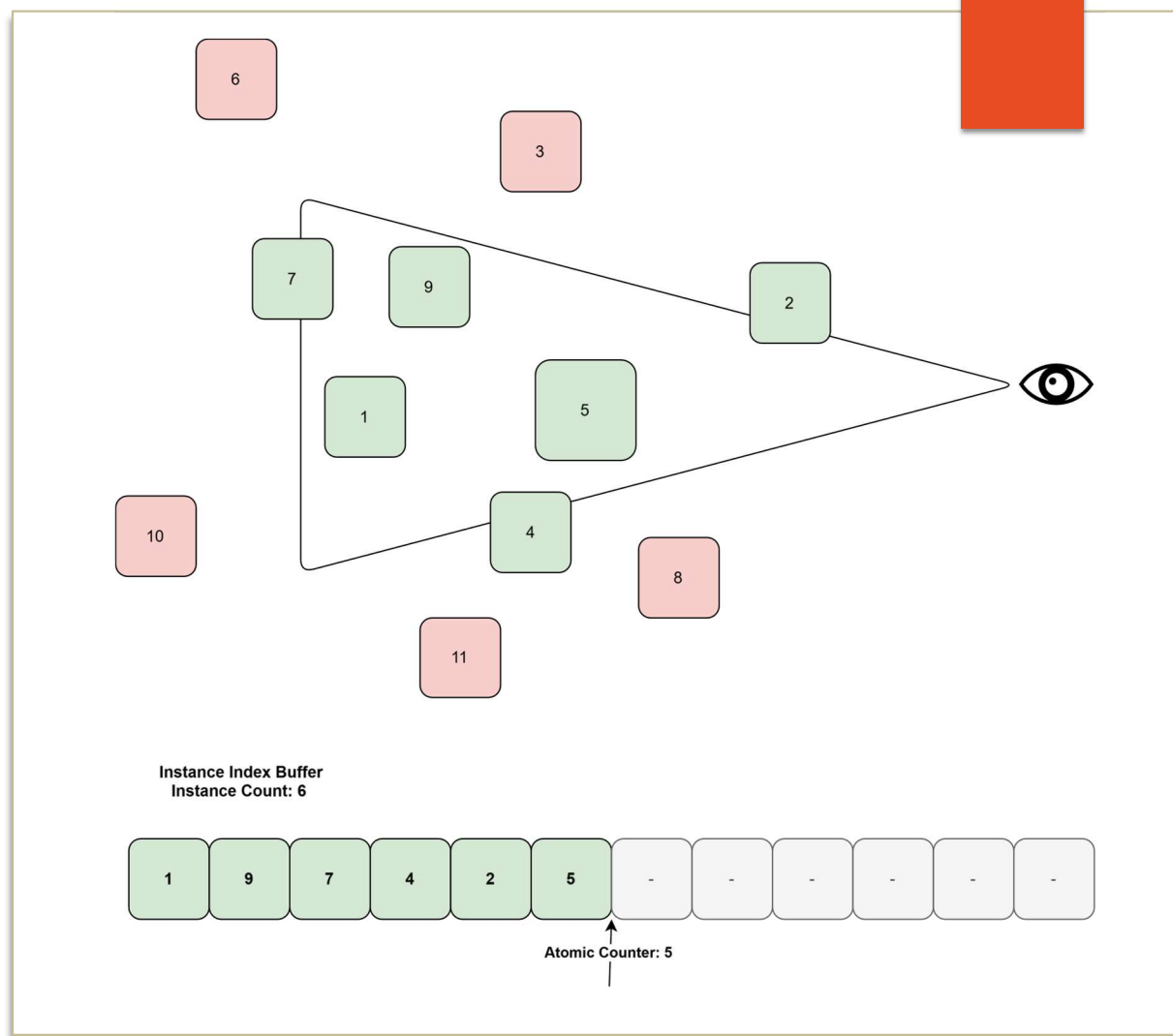
Culling Architecture

- ▶ **The primary goal of culling is to minimize rendering workload** by preventing invisible objects from reaching the geometry processing stages.
- ▶ **Culling determines visibility** by evaluating whether an object can contribute to the final rendered image. Visibility may depend on multiple conditions, such as whether an entity is enabled, inside the camera frustum, or passes additional visibility tests.
- ▶ **Culling can be performed on the CPU or GPU.** CPU-side culling is straightforward to implement but becomes increasingly expensive as scene complexity and instance counts grow.
- ▶ **GPU-driven culling moves visibility determination to the GPU**, allowing thousands or millions of instances to be processed in parallel without requiring the CPU to iterate over the entire scene every frame.
- ▶ **The output of culling is not simply a visibility flag.** The culling process must generate the rendering workload required by the GPU-driven renderer.
- ▶ **For each mesh, the corresponding draw command is updated**, primarily by determining its instanceCount based on the number of visible instances.
- ▶ **The Instance Index Buffer must remain synchronized with the generated draw commands.** Every visible instance is written into the appropriate mesh and bucket-specific region of the instance index buffer.
- ▶ **Each instance is represented by its entity ID**, providing an indirection layer through which the GPU can resolve the entity's transform and other required component data.
- ▶ **The rendering pipeline can therefore process only visible instances.** Invisible objects do not contribute to the generated instance lists and therefore do not launch unnecessary geometry processing work.
- ▶ **The final objective is a fully GPU-driven visibility pipeline**, where the GPU determines what is visible, builds the corresponding instance lists and draw commands, and directly feeds the resulting workload into the rendering pipeline.

Culling II.

Draw Command & Instance Index Management

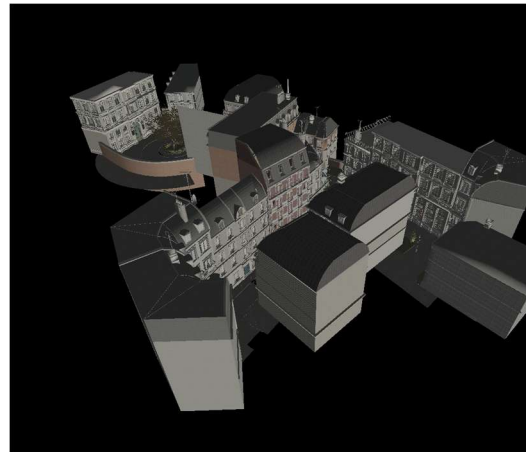
- ▶ For each mesh, the culling stage must determine the number of visible instances and update the corresponding **instanceCount** in the draw command.
- ▶ The instance index buffer must be populated with the **entity IDs of the visible instances**.
- ▶ Each mesh has its own pre-allocated region in the instance index buffer.
- ▶ An **atomic counter** is used as the instanceCount of the corresponding draw command.
- ▶ When a visible instance is found, the counter is atomically incremented.
- ▶ The atomic increment **returns the previous value of the counter**, which directly provides the next free position in the mesh's instance index range.
- ▶ The entity ID is then written to this position in the instance index buffer.
- ▶ At the beginning of each frame, the **counters are reset to zero**. The instance index buffer itself does not need to be cleared, since only the range generated by the current frame's counter is consumed during rendering.



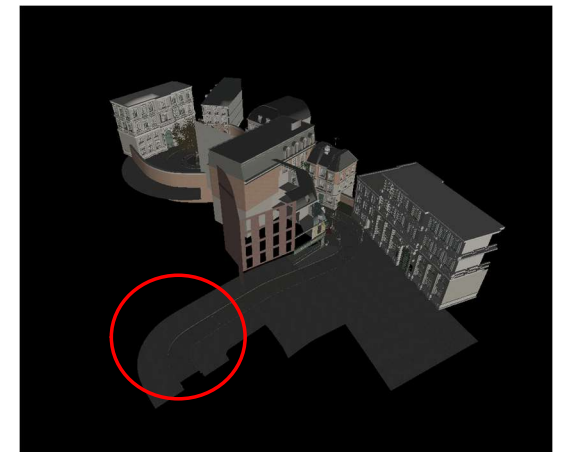
Culling III.

Hierarchical Culling

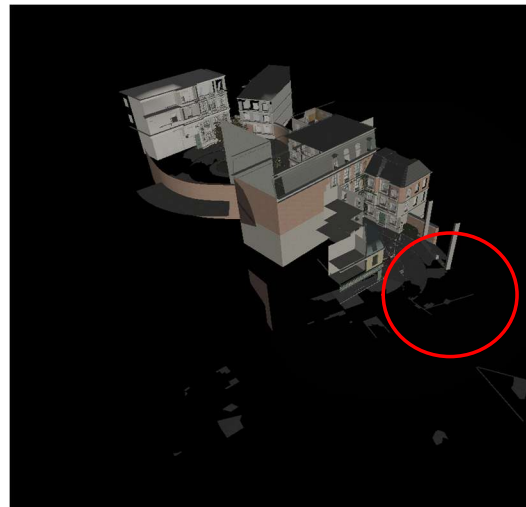
- ▶ The scene is organized into multiple levels of geometric granularity.
- ▶ The **ECS and high-level engine logic** operate primarily on **models**, while a model may contain multiple meshes.
- ▶ The **GPU-driven renderer**, however, operates at **mesh granularity**, since each mesh is represented by its own draw commands within the multi-draw indirect pipeline.
- ▶ Therefore, **visibility can be evaluated hierarchically** at multiple levels:
 - ▶ **Model-level culling**: eliminates entire models that cannot contribute to the final image.
 - ▶ **Mesh-level culling**: eliminates individual meshes within visible models.
 - ▶ **Meshlet-level culling**: provides fine-grained visibility at the geometry level.
- ▶ Each level **progressively reduces the amount of work** that reaches the next stage of the rendering pipeline.



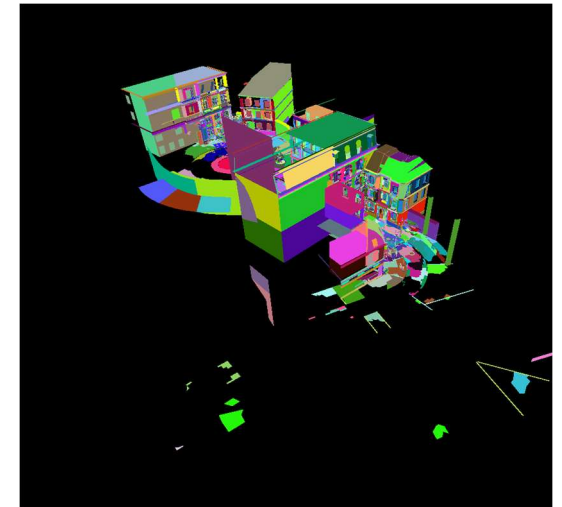
Model-Level Culling



Mesh-Level Culling



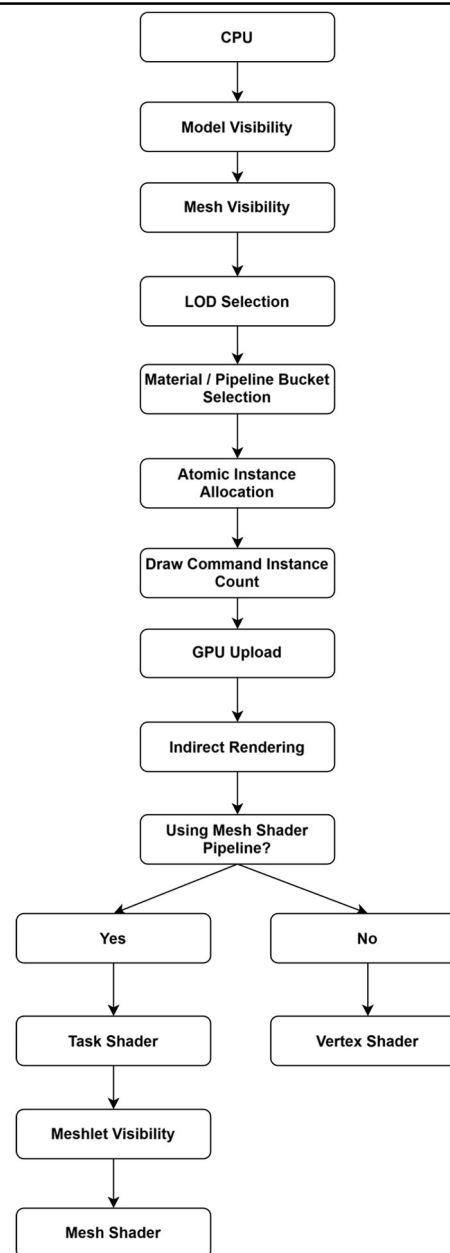
Meshlet-Level Culling



Culling IV.

CPU-Driven Culling Pipeline

- ▶ The CPU performs the complete visibility determination for the scene.
- ▶ The culling workload is executed **fully in parallel using Taskflow**, processing static, dynamic and stream entities concurrently.
- ▶ For each entity, the **CPU evaluates its visibility and performs hierarchical culling**:
 - ▶ Model-level frustum culling
 - ▶ Mesh-level frustum culling
 - ▶ Screen-space size evaluation and LOD selection
- ▶ For every visible mesh, the CPU determines the appropriate **pipeline and material bucket** and updates the corresponding draw command.
- ▶ An atomic counter is used to reserve an instance slot for the mesh and bucket.
- ▶ The entity ID is then written into the corresponding region of the **Instance Index Buffer**.
- ▶ **The final rendering stage remains GPU-driven**: the generated commands are executed through **indirect rendering or indirect mesh-task rendering**, followed by task-shader meshlet culling.



<https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/System/Rendering/ModelFrustumCullingSystem.cpp>

Culling V.

CPU-Driven Culling: The Cost

- ▶ The main limitation is that **the CPU must process the entire visibility workload every frame.**
- ▶ Even with full parallelization, a scene containing **millions of entities** creates a significant CPU workload.
- ▶ Every frame, the CPU must iterate through the relevant entity sets and perform visibility tests, LOD selection and bucket assignment.
- ▶ The CPU must also update the instance counters and populate the Instance Index Buffer.
- ▶ The generated draw commands and instance data then have to be **synchronized and uploaded to the GPU.**
- ▶ This creates **additional CPU-to-GPU data transfer and synchronization** overhead every frame.
- ▶ As scene complexity increases, the CPU becomes the bottleneck even when many CPU cores are available.
- ▶ The key problem is therefore not only the visibility tests themselves, but the **continuous CPU-side generation and synchronization of the GPU rendering workload.**
- ▶ **GPU-driven culling** addresses exactly these limitations by moving **visibility determination and rendering workload generation from the CPU to the GPU.**

Culling VI.

GPU-Driven Culling

- ▶ Culling is implemented entirely in **compute shaders**, allowing all visibility tests to run massively in parallel.
- ▶ Each compute invocation processes one model instance and performs the required visibility tests.
- ▶ The **draw command buffer and Instance Index Buffer reside entirely on the GPU**, eliminating the CPU-side synchronization and per-frame upload required by CPU-driven culling.
- ▶ At the beginning of each frame, a compute pass resets the **instanceCount** of the draw commands.
- ▶ For every visible instance, **atomicAdd** atomically increments the corresponding **instanceCount** and returns the previous value, which is used as the write position in the Instance Index Buffer.
- ▶ The entity ID is written directly into the GPU-resident Instance Index Buffer, keeping the rendering workload entirely GPU-generated.
- ▶ The compute shader can determine the appropriate **material bucket and LOD level dynamically**, updating the corresponding draw command and instance range.
- ▶ The **data-oriented ECS provides the required GPU-friendly data layout**, allowing the culling workload to process large amounts of scene data efficiently in parallel.
- ▶ This makes the complete visibility → draw command generation pipeline **CPU-independent**.
- ▶ **The data-oriented architecture introduced at the beginning of the engine now becomes the foundation of the GPU-driven culling pipeline.**

Culling VII.

Model Culling Compute Shader

- ▶ The first stage of the hierarchical culling pipeline operates at **model level**.
- ▶ A dedicated **Transform-Model Link Buffer** connects each entity's transform and model data through their respective ECS pool indices.
- ▶ Each compute invocation processes one model instance and retrieves its transform and model data through this indirection.
- ▶ For visibility testing, each model provides a **model-level AABB and Sphere collider**, enclosing all meshes belonging to the model.
- ▶ The collider is transformed into **world space**, after which the visibility tests are performed.
- ▶ The result determines whether the model can contribute to the final image and therefore requires further processing.

```
void main()
{
    FrameGlobalContext ctx = GET_FRAME_CONTEXT(pc.frameGlobalContextBufferAddr);

    // Resolve dense index for non-static (dynamic/stream) entities
    uint offset = ctx.staticTransformCount * ctx.enableGeometryBvhCulling;
    uint transformDenseIndex = gl_GlobalInvocationID.x + offset;
    uint transformCount = ctx.allTransformCount - offset;

    // 1. Thread Boundary Check
    if (gl_GlobalInvocationID.x >= transformCount) return;

    TransformModelLink link = GET_TRANSFORM_MODEL_LINK(ctx.transformModelLinkBufferAddr, transformDenseIndex);
    if (link.modelDenseIndex == INVALID_INDEX) return;
    uint entityId = link.entityIndex;

    uint tagSparseIndex = GET_SPARSE_INDEX(ctx.tagSparseMapBufferAddr, entityId);
    if (tagSparseIndex == INVALID_INDEX) return;
    uint entityFlags = GET_TAG_DATA(ctx.tagDataBufferAddr, tagSparseIndex);

    bool isActive = (entityFlags & (1u << 0)) != 0;
    if (!isActive) return;

    // 2. Fetch Model component and Entity ID
    ModelComponent modelComp = GET_MODEL_COMP(ctx.modelBufferAddr, link.modelDenseIndex);
    if (modelComp.modelIndex == INVALID_INDEX) return;

    // 3. Resolve Transform and Main Camera data
    TransformComponent transform = GET_TRANSFORM(ctx.transformBufferAddr, transformDenseIndex);
    uint cameraIdx = GET_SPARSE_INDEX(ctx.cameraSparseMapBufferAddr, ctx.mainCameraEntity);
    CameraComponent camera = GET_CAMERA(ctx.cameraBufferAddr, cameraIdx);

    // 4. Resolve Model metadata and addresses
    ModelAllocationInfo mAlloc = GET_MODEL_ALLOC(ctx.globalModelAllocationBufferAddr, modelComp.modelIndex);
    GpuModelAddresses addrs = GET_MODEL_ADDRESSES(ctx.modelAddressBufferAddr, modelComp.modelIndex);
    GpuMeshCollider localGlobalCollider = addrs.globalCollider;

    /// Animation...

    // Transform global collider to world space
    GpuMeshCollider worldCollider = TransformCollider(localGlobalCollider, transform.transform);

    uint visibility = INTERSECTION_INTERSECT;
    if (ctx.enableModelFrustumCulling == 1)
    {
        visibility = TestFrustum(worldCollider, camera);
        if (visibility == INTERSECTION_OUTSIDE) return;
    }

    // 6. Occlusion Culling Test
    float screenSizePixels = 0.0;
    vec2 screenRes = vec2(ctx.screenWidth, ctx.screenHeight);
    bool enableDepthOcclusion = (ctx.enableModelOcclusionCulling == 1);

    if (IsSphereOccludedPerspective(worldCollider.center, worldCollider.radius, camera.view, camera.projVulkan,
        camera.params.x, camera.params.y, depthPyramid, screenRes, enableDepthOcclusion, screenSizePixels)) {
        return;
    }

    ///...
}
```

Transform-Model Range To Cull
(This allows static/dynamic/stream partitioning more on that later!!)

Cull Inactive Entities

Get Component Data

Calculate World Collider On-The-Fly

Camera Frustum Culling Test

HiZ Occlusion Culling Test

<https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/Shaders/Passes/Culling/Geometry/GeometryModelCulling.comp>

Culling VIII.

Model Culling Compute Paths

► Fast Path - Few Meshes

- For models containing only a small number of meshes, mesh-level culling is performed **directly inside the model culling shader**.
- The shader immediately generates the corresponding draw command and Instance Index Buffer entries.
- This avoids launching another mesh-culling pass for a very small amount of work.

► Slow Path - Many Meshes

- Models containing a large number of meshes, such as **Sponza**, take a different path.
- Their model IDs are atomically appended to a **visible-model buffer**.
- This buffer becomes the input of the next **mesh culling compute pass**.
- Instead of processing every mesh in every model, the next stage only processes meshes belonging to models that actually survived model-level culling.

- **The goal of hierarchical culling is not simply to perform more culling tests, but to avoid launching work for geometry that has already been eliminated at a higher level.**

```
void main()
{
    // Previous Code ...

    // 7. Fast-path: Process simple/small models directly
    uint safeMeshCount = mAlloc.meshAllocationCount / 4;
    if (!IsModelSimpleEnoughForFastPath(safeMeshCount, addr.vertexCount))
    {
        // Calculate LOD
        uint lod = CalculateLodFromScreenSize(screenSizePixels);

        for(uint m = 0; m < safeMeshCount; ++m) {
            uint matIdx = GET_MATERIAL_INDEX(ctx.materialLookupBufferAddr, modelComp.materialOffset + m);
            Material mat = GET_MATERIAL(ctx.materialBufferAddr, matIdx);

            // Determine Material Render Type (Opaque/Transparent + 1/2 Sided)
            uint matType = GetMaterialRenderType(mat);

            MeshAllocationInfo meshAlloc = GET_MESH_ALLOC(ctx.globalMeshAllocationBufferAddr, mAlloc.meshAllocationOffset + (m * 4) + lod);
            uint pipeIdx = GET_PIPELINE_INDEX(ctx.pipelineLookupBufferAddr, modelComp.pipelineOffset + m);

            if (meshAlloc.activeTypes[pipeIdx][matType] == 1) {
                uint slotIndex = 0;
                uint indirectIdx = meshAlloc.indirectIndices[pipeIdx][matType];

                // Append instance to the appropriate indirect draw command using atomic additions
                if (pipeIdx == PIPELINE_MESHLET) {
                    uint64_t meshletBaseAddr = ctx.globalIndirectCommandBufferAddr + (ctx.globalTraditionalCommandsCount * 16ul);
                    slotIndex = atomicAdd(GET_MESHLET_CMD(meshletBaseAddr, indirectIdx).groupCountX, 1);
                } else {
                    slotIndex = atomicAdd(GET_TRADITIONAL_CMD(ctx.globalIndirectCommandBufferAddr, indirectIdx).instanceCount, 1);
                }

                // Write Instance ID (and encode full visibility flag in the MSB)
                uint payload = SET_BIT_TO(entityId, 31, visibility == INTERSECTION_INSIDE);
                GET_INSTANCE(ctx.globalInstanceIndexBufferAddr, meshAlloc.instanceOffsets[pipeIdx][matType] + slotIndex) = payload;
            }
        }
        return;
    }

    // 8. Slow-path: Defer large/complex models to the Mesh Culling pass
    uint needsWrite = 1;
    uint subgroupWriteCount = subgroupAdd(needsWrite);
    uint localOffset = subgroupExclusiveAdd(needsWrite);
    uint baseOffset = 0;

    // Elect one thread per subgroup to perform the atomic memory allocation
    if (subgroupElect()) {
        baseOffset = atomicAdd(GET_DISPATCH_CMD(ctx.modelCountBufferAddr).groupCountX, subgroupWriteCount);
    }

    baseOffset = subgroupBroadcastFirst(baseOffset);

    if (needsWrite == 1) {
        uint finalIndex = baseOffset + localOffset;
        VisibleModelData outData;

        // Encode full visibility flag into entity ID
        outData.entityId = SET_BIT_TO(entityId, 31, visibility == INTERSECTION_INSIDE);
        outData.modelIndex = modelComp.modelIndex;

        GET_VISIBLE_MODEL(ctx.modelVisibleIndexBufferAddr, finalIndex) = outData;
    }
}
```

Fast Path - Few
Meshes

Slow Path - Many
Meshes

<https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/Shaders/Passes/Culling/Geometry/GeometryModelCulling.comp>

Culling IX.

Vulkan Subgroup Operations

- ▶ Modern GPUs expose **subgroup operations** that allow threads within a subgroup to cooperate without going through global memory.
- ▶ These operations perform collective computations across the subgroup, such as counting, **broadcasting** and **prefix-sum** operations.
- ▶ In the slow path, multiple threads may need to append visible models to the same global buffer.
- ▶ Performing one global atomicAdd per thread would create heavy **atomic contention** and can become a serialization bottleneck.
- ▶ Instead, the subgroup first determines **how many threads actually need to write**.
- ▶ A single thread then performs one global atomicAdd, reserving the required contiguous range for the entire subgroup.
- ▶ The subgroup distributes the reserved indices locally using a **prefix-sum / exclusive-add operation**.
- ▶ This reduces the number of global atomic operations from **one per thread to one per subgroup**.
- ▶ The resulting writes are also **contiguous**, improving the organization of the generated data and making the technique useful for other GPU-generated workloads.

```
// 8. Slow-path: Defer large/complex models to the Mesh Culling pass
uint needsWrite = 1;
uint subgroupWriteCount = subgroupAdd(needsWrite);
uint localOffset = subgroupExclusiveAdd(needsWrite);
uint baseOffset = 0;

// Elect one thread per subgroup to perform the atomic memory allocation
if (subgroupElect()) {
    baseOffset = atomicAdd(GET_DISPATCH_CMD(ctx.modelCountBufferAddr).groupCountX, subgroupWriteCount);
}

baseOffset = subgroupBroadcastFirst(baseOffset);

if (needsWrite == 1) {
    uint finalIndex = baseOffset + localOffset;
    VisibleModelData outData;

    // Encode full visibility flag into entity ID
    outData.entityId = SET_BIT_TO(entityId, 31, visibility == INTERSECTION_INSIDE);
    outData.modelIndex = modelComp.modelIndex;

    GET_VISIBLE_MODEL(ctx.modelVisibleIndexBufferAddr, finalIndex) = outData;
}
```

<https://www.khronos.org/blog/vulkan-subgroup-tutorial>

Culling X.

Mesh Culling Compute Shader

- ▶ Complex models that survive model-level culling are stored in the **Visible Model Buffer**.
- ▶ A mesh culling workgroup is dispatched for each visible model.
- ▶ Each workgroup contains **32 threads**, which collaboratively process all meshes belonging to one model.
- ▶ **gl_WorkGroupID.x** identifies the model being processed, while **gl_LocalInvocationID.x** identifies the thread within the workgroup.
- ▶ Each thread processes multiple meshes using a strided loop when the model contains more meshes than available threads.
- ▶ At this level, the **individual mesh colliders** are tested against the camera.
- ▶ Visible meshes update their corresponding **draw commands and Instance Index Buffer** entries.

```
void main()
{
    // Thread mapping: 1 WorkGroup = 1 Deferred Model, Local Threads = Meshes
    uint jobId = gl_WorkGroupID.x;
    uint localThreadId = gl_LocalInvocationID.x;

    //... Resolve data, preparations

    // 6. Collaborative Loop: Process all meshes of the current model
    for (uint m = localThreadId; m < addr.meshCount; m += gl_WorkGroupSize.x) {

        // Resolve local mesh collider (Handle animation frame data if skinned)
        GpuMeshCollider localMeshCollider = GET_MESH_COLLIDER(addr.meshColliders, m);

        //... Animation

        // Transform mesh collider to world space
        GpuMeshCollider worldCollider = TransformCollider(localMeshCollider, transform.transform);

        // 7. Precise Frustum Test (Optimization: Skip if parent model is already fully inside)
        uint visibility = parentFullyInside ? INTERSECTION_INSIDE : INTERSECTION_INTERSECT;
        if (!parentFullyInside && ctx.enableMeshFrustumCulling == 1) {
            visibility = TestFrustum(worldCollider, camera);
            if (visibility == INTERSECTION_OUTSIDE) continue;
        }

        // 8. Occlusion Culling using Main Camera HZB
        float screenSizePixels = 0.0;
        if (!IsSphereOccludedPerspective(worldCollider.center, worldCollider.radius, camera.view, camera.projVulkan,
            camera.params.x, camera.params.y, depthPyramid, screenRes, enableDepthOcclusion, screenSizePixels)) {
            continue;
        }

        // 9. Determine LOD based on screen size
        uint lod = CalculateLodFromScreenSize(screenSizePixels);

        // Resolve Material
        uint matIdx = GET_MATERIAL_INDEX(ctx.materialLookupBufferAddr, modelComp.materialOffset + m);
        Material mat = GET_MATERIAL(ctx.materialBufferAddr, matIdx);

        // 10. Resolve material render type bitmask
        uint matType = GetMaterialRenderType(mat);

        MeshAllocationInfo meshAlloc = GET_MESH_ALLOC(ctx.globalMeshAllocationBufferAddr, mAlloc.meshAllocationOffset + (m * 4) + lod);
        uint pipeIdx = GET_PIPELINE_INDEX(ctx.pipelineLookupBufferAddr, modelComp.pipelineOffset + m);

        if (meshAlloc.activeTypes[pipeIdx][matType] == 1) {
            uint slotIndex = 0;
            uint indirectIdx = meshAlloc.indirectIndices[pipeIdx][matType];

            // 11. Append instance to the appropriate indirect draw command
            if (pipeIdx == PIPELINE_MESHLET) {
                uint64_t meshletBaseAddr = ctx.globalIndirectCommandBufferAddr + (ctx.globalTraditionalCommandsCount * 16ul);
                slotIndex = atomicAdd(GET_MESHLET_CMD(meshletBaseAddr, indirectIdx).groupCountX, 1);
            } else {
                slotIndex = atomicAdd(GET_TRADITIONAL_CMD(ctx.globalIndirectCommandBufferAddr, indirectIdx).instanceCount, 1);
            }

            // Repack payload with precise mesh-level visibility flag and write instance data
            uint payload = SET_BIT_TO(pureEntityId, 31, visibility == INTERSECTION_INSIDE);
            GET_INSTANCE(ctx.globalInstanceIndexBufferAddr, meshAlloc.instanceOffsets[pipeIdx][matType] + slotIndex) = payload;
        }
    }
}
```

Collaborative Mesh Culling Iteration

Camera Frustum Culling Test

Occlusion Culling Test

Calculate LOD based on projected size

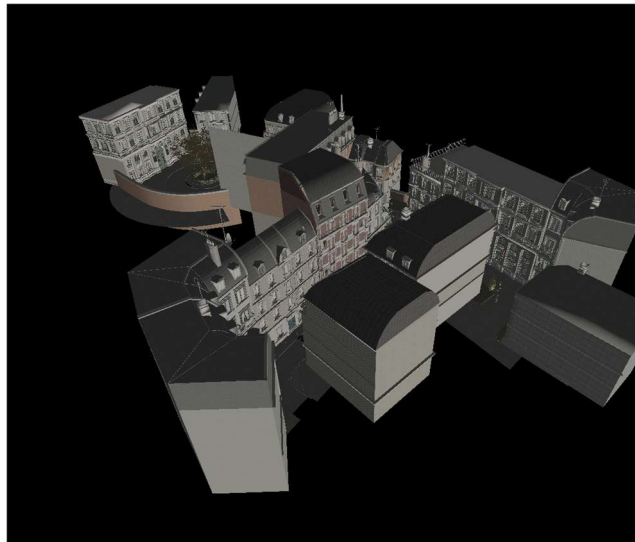
Determine Material Bucket Type

Generate Draw Command and Instance Index Buffer Payload

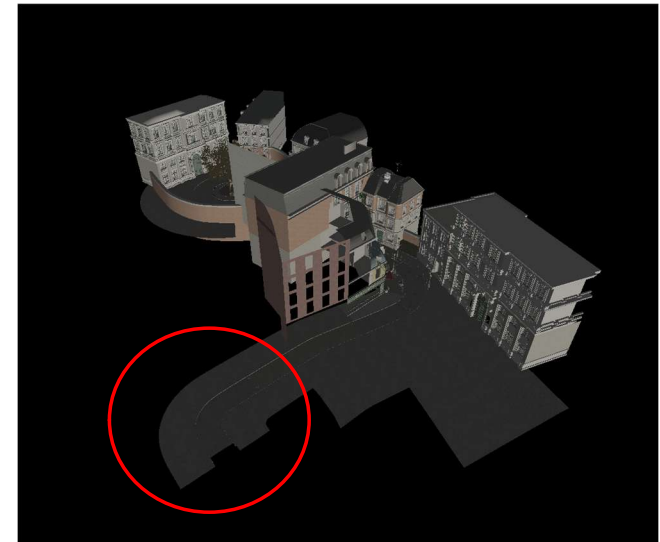
Culling XI.

Why Mesh-Level Culling?

- ▶ The **renderer operates at mesh granularity**: every mesh has its own draw command and visibility state.
- ▶ The **ECS**, however, operates at **model granularity**, since a model represents a logical collection of meshes.
- ▶ Therefore, model-level visibility alone is not sufficient, the renderer must ultimately determine visibility **for each individual mesh**.
- ▶ Hierarchical culling exploits the relationship between these two levels:
 - ▶ If an entire model is invisible, **none of its meshes need to be processed further**.
 - ▶ If the model is visible, its individual meshes are tested to determine which draw commands need to be generated.
- ▶ This provides a natural transition from the coarse **model level** to the finer **mesh level**.
- ▶ The hierarchy does not stop at meshes: the Task/Mesh shader pipeline enables the same principle to continue down to **meshlet-level culling**.



Model-Level Culling



Mesh-Level Culling

Culling XII.

Work-Graph

- ▶ **Work Graphs** provide a GPU-driven mechanism for **dynamically generating** and **scheduling compute work**.
- ▶ The current hierarchical culling pipeline is inherently **producer → consumer**:
 - ▶ Model Culling produces visible models.
 - ▶ Mesh Culling consumes those models and produces visible meshes.
 - ▶ Meshlet Culling consumes the visible meshes.
- ▶ With the conventional approach, these stages are separated by **GPU-resident buffers and dispatches**.
- ▶ The problem is that the output size of each stage is **data-dependent** and **cannot be known beforehand**.
- ▶ Therefore, intermediate buffers and dispatch dimensions typically have to be allocated for a **worst-case workload**, even though most frames require significantly less work.
- ▶ A Work Graph allows the GPU to **dynamically create downstream work as upstream nodes produce results**.
- ▶ Instead of first generating a large visibility buffer and then launching a separate dispatch, a Model Culling node can **directly enqueue Mesh Culling work** for the models that survived.
- ▶ The GPU dynamically manages the required execution resources, allowing the workload to **expand and shrink according to the actual scene visibility**.
- ▶ Data can be passed between nodes through GPU-managed payload mechanisms, conceptually similar to the producer → consumer relationship of Task/Mesh shaders.

```
struct MeshCullingPayload {  
    // Bit-packing layout:  
    // GEOMETRY PASS: [Bit 31: Inside Frustum] [Bits 0-30: EntityID]  
    uint entityId;  
    uint modelIndex;  
};  
  
// Output node declaration for the Work Graph  
layout(coalescing_node_amdx) out node_type GeometryWorkGraphMeshCullingNode;  
layout(location = 0) out MeshCullingPayload payloadOut;  
  
void main() {  
    FrameGlobalContext ctx = GET_FRAME_CONTEXT(pc.frameGlobalContextBufferAddr);  
  
    // 1-5. Resolve data and preparation  
    //...  
  
    // 6. Fast-path: Process simple/small models directly  
    //...  
  
    // 7. Slow-path: WORK GRAPH ENQUEUE (Directly spawn a Mesh Culling Node)  
    allocateNodeRecordAMD(GeometryWorkGraphMeshCullingNode, 1);  
  
    payloadOut.entityId = SET_BIT_TO(entityId, 31, visibility == INTERSECTION_INSIDE);  
    payloadOut.modelIndex = modelComp.modelIndex;  
  
    submitNodeRecordAMD(GeometryWorkGraphMeshCullingNode);  
}
```

} **Enqueue Mesh Culling Work**

<https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/Shaders/Passes/Culling/Geometry/GeometryWorkGraphModelCulling.comp>

Culling XIII.

Task Shader Meshlet Culling

- ▶ The Task Shader receives an indirect mesh-task dispatch where:
 - ▶ **X = instance count**: the number of visible entities using the mesh.
 - ▶ **Y = meshlet counts**: the meshlet range processed by each instance.
- ▶ Each Task Shader workgroup therefore corresponds to **one mesh instance and one block of meshlets**.
- ▶ The meshlet and instance identity is reconstructed from the **built-in workgroup** and local **invocation IDs**.
- ▶ Each thread identifies and processes one meshlet within the current meshlet block.

```
taskPayloadSharedEXT TaskPayload payload;
shared uint survivingMeshletCount;

void main() {
    uint threadIdx = gl_LocalInvocationID.x; // Meshlet
    uint localInstanceID = gl_WorkGroupID.x; // Entity
    uint meshletGroupOffset = gl_WorkGroupID.y * 32; // Meshlet Block

    // 2. Fetch Draw Descriptor and Instance Data
    // 3. Resolve Camera and Transform
    // 4. Resolve Meshlet range for this group
    // ...

    if (isVisible) {
        uint globalMeshletIdx = submeshData.meshletOffset + currentMeshletIdx;
        GpuMeshletCollider localCollider = GET_MESHLET_COLLIDER(addr.meshletColliders, globalMeshletIdx);

        // 5. Evaluate Animation frame collider if applicable
        // ...

        GpuMeshletCollider worldCollider = TransformCollider(localCollider, transform.transform, transform.transformIT);

        // 7. Backface Cone Culling
        if ((pc.disableConeCulling == 0 && ctx.enableMeshletConeCulling == 1)) {
            if (TestConeCulling(worldCollider, camera.eye.xyz)) {
                isVisible = false;
            }
        }

        // 8. Frustum Culling (Skip if parent model is already fully inside)
        if (isVisible && !isParentFullyInside && ctx.enableMeshletFrustumCulling == 1) {
            if (TestFrustum(worldCollider.center, worldCollider.radius, worldCollider.aabbMin, worldCollider.aabbMax, camera) == INTERSECTION_OUTSIDE) {
                isVisible = false;
            }
        }

        // 9. Occlusion and Sub-pixel Culling using Depth Pyramid
        float screenSizePixels = 0.0;
        if (isVisible) {
            vec2 screenRes = vec2(ctx.screenWidth, ctx.screenHeight);
            bool enableDepthOcclusion = (ctx.enableMeshletOcclusionCulling == 1);

            if (IsSphereOccludedPerspective(worldCollider.center, worldCollider.radius, camera.view, camera.projVulkan,
                camera.params.x, camera.params.y, depthPyramid, screenRes, enableDepthOcclusion, screenSizePixels)) {
                isVisible = false;
            }
        }

        // 10. Append surviving meshlet index to payload
        if (isVisible) {
            uint slot = atomicAdd(survivingMeshletCount, 1);
            payload.meshletIndices[slot] = currentMeshletIdx;
        }

        barrier();

        // 11. Emit the count of surviving meshlets for the Mesh Stage
        if (threadIdx == 0) {
            EmitMeshTasksEXT(survivingMeshletCount, 1, 1);
        }
    }
}
```

Map Task Shader Thread to Meshlet

Resolve Meshlet Collider (World Space)

Cone Culling Test

Camera Frustum Culling Test

HiZ Occlusion Culling Test

Meshlet Survived: Populate Mesh Shader Payload Data

Start Mesh Shader Work Groups Dynamically

Culling XIV.

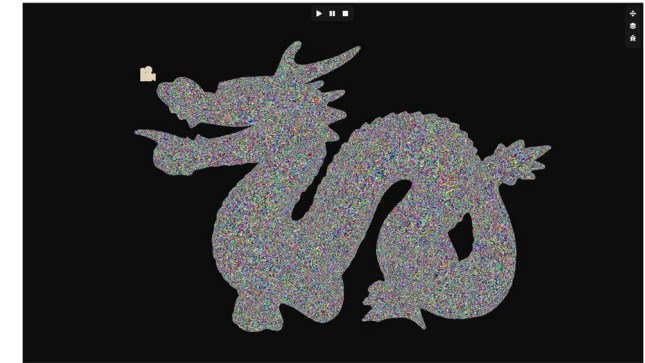
Task Shader Meshlet Culling

- ▶ The Task Shader computes the meshlet's **world-space bounds** and performs the visibility tests:
 - ▶ Cone Culling
 - ▶ Frustum Culling
 - ▶ Occlusion Culling
- ▶ Meshlets that survive all tests are written into the **Task Shader payload**.
- ▶ The payload contains the information required by the Mesh Shader to identify and process the surviving meshlets.
- ▶ After synchronization, the Task Shader **dynamically launches the required number of Mesh Shader workgroups** based on the surviving meshlets.
- ▶ **The Task Shader transforms a coarse mesh-level workload into a fine-grained, visibility-filtered meshlet workload.**

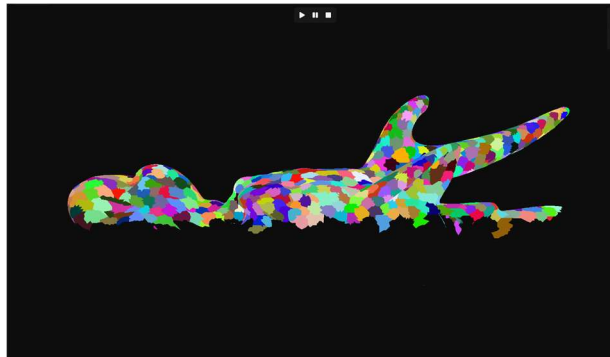
The Scene: Dragon partially visible



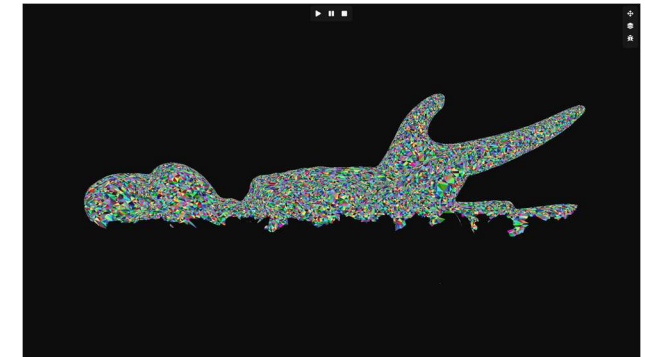
The Vertex Shader: Processes all triangles



The Task Shader: Culls Invisible Meshlets



The Mesh Shader: Processes only visible triangles



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

Culling XV.

Hierarchical Culling Benchmark

- ▶ **~20× faster** GPU-driven culling and rendering compared to the traditional CPU-driven approach.
- ▶ **2 million entities processed in <1 ms** for the complete GPU culling + rendering pipeline.
- ▶ **1 million entities**: only **0.677 ms** total GPU time.
- ▶ Model-level culling scales efficiently even at very high entity counts
- ▶ The GPU workload is fully parallelized, with no CPU-side synchronization or per-frame visibility data transfer.
- ▶ **NVIDIA Resizable BAR** is strongly recommended.

(Earlier implementations showed a severe performance degradation without it (~10–20 ms). The current engine version is designed to work without Resizable BAR, but enabling it is still recommended for optimal performance.)

Table 2: CPU vs. GPU culling performance (1M entities).

Scenario	CPU (ms)	GPU (ms)	FPS
CPU-Driven	12.244	-	79
– Full Mesh	-	3.481	-
– Mesh (no occ.)	-	4.504	-
– Traditional	-	7.687	-
GPU-Driven	0.153	0.677	1344

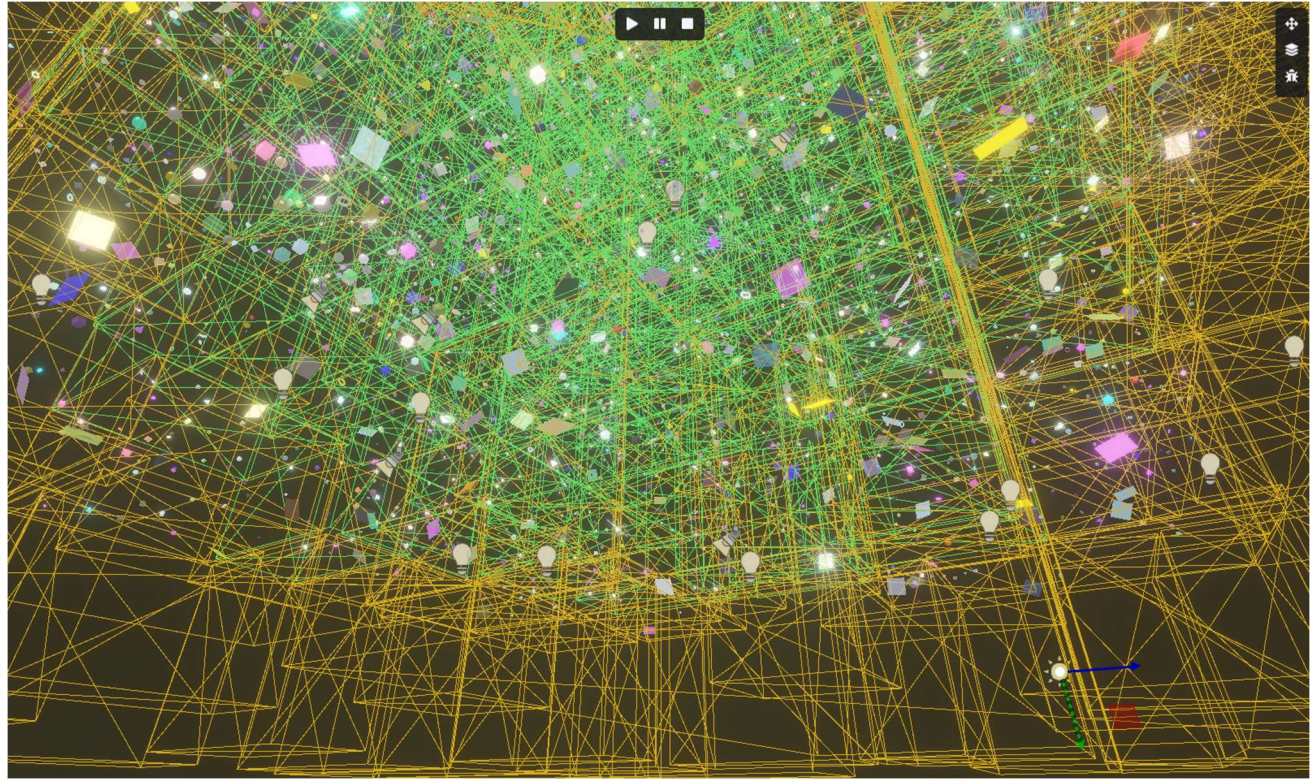
Table 4: Hardware scalability on RTX 4060.

Entities	Model Culling (ms)	Total GPU (ms)
10,000	0.013	0.253
100,000	0.050	0.549
1,000,000	0.360	0.671
2,000,000	0.680	0.992

Culling XVI.

Four-Level Hierarchical Culling

- ▶ The current engine uses a **four-level hierarchical culling architecture**.
- ▶ An additional **BVH Chunk layer** groups spatially close entities before model-level culling.
- ▶ The hierarchy can be generated using both **CPU- and GPU-based approaches**.
- ▶ Spatially nearby entities are grouped into approximately **32-element chunks**.
- ▶ Each chunk represents a coarse spatial region containing multiple models.
- ▶ The hierarchy therefore becomes: **Chunk – Model – Mesh – Meshlet**



**This is the AABB collider
visualization of the BVH
chunks**

Culling XVII.

Four-Level Hierarchical Culling – Static Entities

- ▶ The ECS already separates entities into **Static, Dynamic and Stream** storage regions.
- ▶ Static entities are particularly suitable for a persistent spatial hierarchy **because their transforms rarely change**.
- ▶ The static entities are grouped into spatial **chunks of approximately 32 entities**, each with its own **AABB and bounding sphere**.
- ▶ Culling is first performed at the Chunk level.
- ▶ Rejecting a single chunk therefore eliminates the processing of approximately **32 models, all of their meshes, and all of their meshlets**.
- ▶ Rebuilding the entire Chunk/BVH hierarchy every frame would introduce unnecessary overhead **even when generated on the GPU**.
- ▶ Therefore, the persistent spatial hierarchy is **built primarily over the Static region**.
- ▶ Dynamic and Stream entities continue through the regular **Model → Mesh → Meshlet** culling pipeline.
- ▶ A **Work Graph** could make this hierarchy even more efficient by allowing surviving chunks to dynamically generate the next level of culling work directly on the GPU, avoiding explicit intermediate buffers and dispatches between levels.

```
void main() {
    FrameGlobalContext ctx = GET_FRAME_CONTEXT(pc.frameGlobalContextBufferAddr);

    uint chunkId = gl_GlobalInvocationID.x;
    if (chunkId >= ctx.staticChunkCount) return;

    // 1. Fetch Chunk Data and Bounding Sphere
    StaticChunk chunk = GET_STATIC_CHUNK(ctx.staticChunkDataBufferAddr, chunkId);

    // 2. Resolve Main Camera data
    uint cameraIdx = GET_SPARSE_INDEX(ctx.cameraSparseMapBufferAddr, ctx.mainCameraEntity);
    CameraComponent camera = GET_CAMERA(ctx.cameraBufferAddr, cameraIdx);

    // 3. Frustum Culling Test
    uint visibility = INTERSECTION_INTERSECT;
    if (ctx.enableChunkFrustumCulling == 1)
    {
        visibility = TestAABBFrustum(chunk.minBounds, chunk.maxBounds, camera);
        if (visibility == INTERSECTION_OUTSIDE) return;
    }

    // 4. Occlusion Culling Test
    float screenSizePixels = 0.0;
    vec2 screenRes = vec2(ctx.screenWidth, ctx.screenHeight);
    bool enableDepthOcclusion = (ctx.enableChunkOcclusionCulling == 1);

    vec3 sphereCenter = (chunk.minBounds + chunk.maxBounds) * 0.5;
    float sphereRadius = length(chunk.maxBounds - sphereCenter);

    if (IsSphereOccludedPerspective(sphereCenter, sphereRadius, camera.view, camera.projVulkan,
        camera.params.x, camera.params.y, depthPyramid, screenRes, enableDepthOcclusion, screenSizePixels)) {
        return;
    }

    // 5. Subgroup-optimized writes to the visible chunk list
    uint needsWrite = 1;
    uint subgroupWriteCount = subgroupAdd(needsWrite);
    uint localOffset = subgroupExclusiveAdd(needsWrite);
    uint baseOffset = 0;

    if (subgroupElect() && subgroupWriteCount > 0) {
        baseOffset = atomicAdd(GET_DISPATCH_CMD(ctx.staticChunkCountBufferAddr).groupCountX, subgroupWriteCount);
    }

    baseOffset = subgroupBroadcastFirst(baseOffset);

    // 6. Encode Payload and write to visible list
    if (needsWrite == 1) {
        uint finalIndex = baseOffset + localOffset;

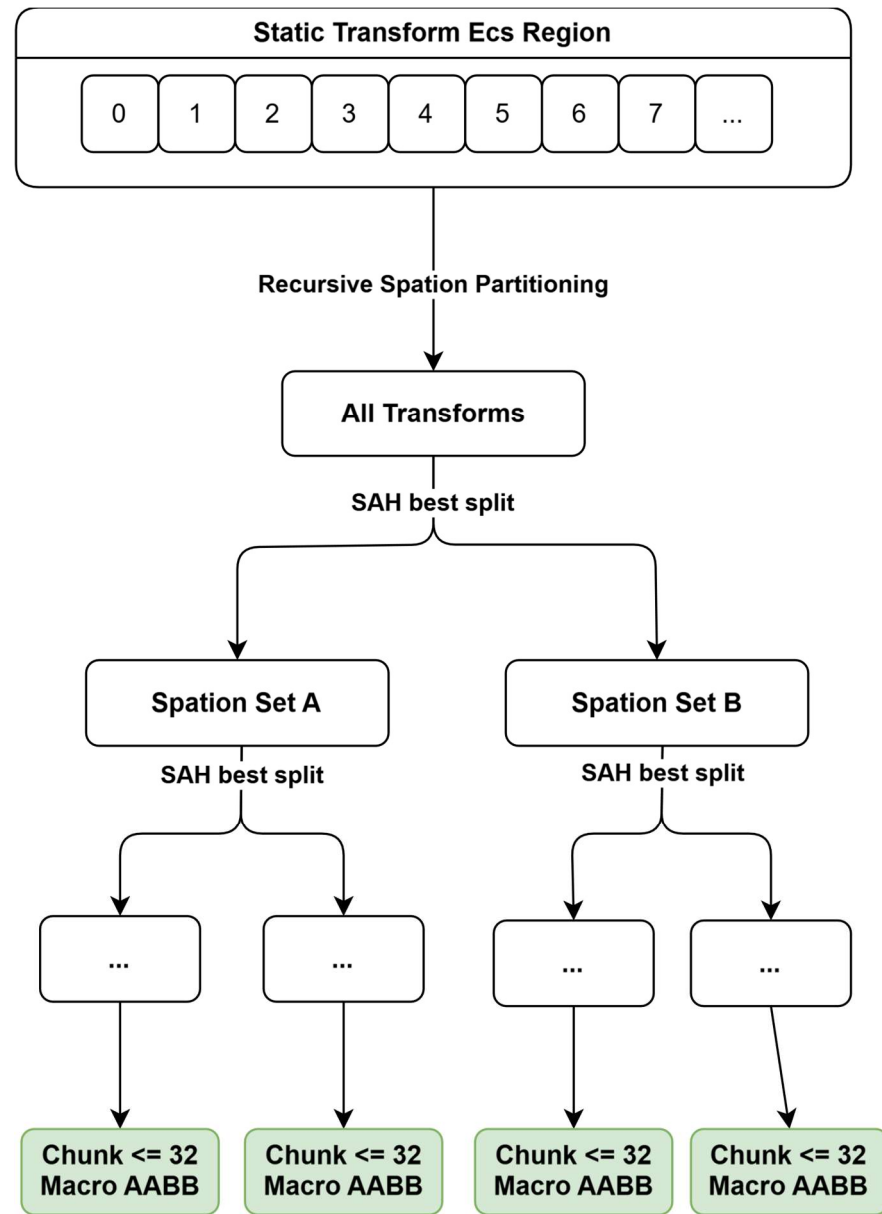
        // Encode chunk ID with visibility flag
        uint payload = SET_BIT_TO(chunkId, 31, visibility == INTERSECTION_INSIDE);
        GET_VISIBLE_CHUNK(ctx.staticChunkVisibleIndexBufferAddr, finalIndex) = payload;
    }
}
```

<https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/Shaders/Passes/Culling/Geometry/GeometryStaticChunkCulling.comp>

Culling XVIII.

CPU-Built Bounding Volume Hierarchy

- ▶ The Static ECS region initially contains a large, flat collection of spatial entities.
- ▶ Each entity contributes its world-space bounds to the spatial hierarchy construction.
- ▶ The complete set is **recursively divided into smaller spatial partitions**. Each partition contains a contiguous subset of the spatial items.
- ▶ The subdivision continues until a **partition contains at most 32 entities**. At this point, the recursion terminates and the partition becomes a **leaf Chunk**.
- ▶ A tight **Macro-AABB** is generated around all entities contained in the Chunk.



Culling XIX.

Binned Surface Area Heuristic

- ▶ For the current spatial partition, the bounds of all entity centroids are calculated.
- ▶ The **axis with the largest centroid distribution** is selected as the split axis.
- ▶ The selected axis is divided into a fixed number of spatial bins.
- ▶ Each entity is **assigned to a bin** according to its centroid position.
- ▶ Every possible boundary between two bins becomes a candidate split position.
- ▶ A **forward sweep accumulates** the entity count and bounding area of the left partition.
- ▶ A **backward sweep performs** the same calculation for the right partition.
- ▶ The **Surface Area Heuristic evaluates all candidate splits** and selects the partition with the lowest estimated cost.

```
// Calculate the centroid bounds of the current partition
glm::vec3 centroidMin(FLT_MAX);
glm::vec3 centroidMax(-FLT_MAX);

for (const auto& item : items)
{
    centroidMin = glm::min(centroidMin, item.centroid);
    centroidMax = glm::max(centroidMax, item.centroid);
}

// Select the axis with the largest spatial distribution
glm::vec3 extent = centroidMax - centroidMin;

int splitAxis = 0;

if (extent.y > extent.x)
    splitAxis = 1;

if (extent.z > extent[splitAxis])
    splitAxis = 2;

//-----

SahBin bins[SAH_NUM_BINS];

float scale = SAH_NUM_BINS / extent[splitAxis];

// Assign each spatial item to a bin
for (const auto& item : items)
{
    int binIdx = std::min(
        static_cast<int>(SAH_NUM_BINS - 1),
        static_cast<int>(
            (item.centroid[splitAxis] -
             centroidMin[splitAxis]) * scale));

    bins[binIdx].count++;

    bins[binIdx].minBounds =
        glm::min(bins[binIdx].minBounds, item.minBounds);

    bins[binIdx].maxBounds =
        glm::max(bins[binIdx].maxBounds, item.maxBounds);
}

//-----

// Find the split with the minimum SAH cost
float minCost = FLT_MAX;
int bestSplit = -1;

for (int i = 0; i < SAH_NUM_BINS - 1; ++i)
{
    float cost =
        leftCount[i] * leftArea[i] +
        rightCount[i] * rightArea[i];

    if (cost < minCost)
    {
        minCost = cost;
        bestSplit = i;
    }
}
```

<https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/System/Core/StaticSpatialSahSystem.cpp>

Culling XX.

Recursive SAH Partitioning

- ▶ After the lowest-cost SAH split has been selected, the **spatial items are partitioned in-place**.
- ▶ Items on one side of the selected bin boundary are moved into the left partition. The remaining items form the right partition.
- ▶ **Both resulting partitions occupy contiguous memory** ranges inside the spatial item array.
- ▶ The **midpoint separates the two child partitions** without requiring additional temporary storage.
- ▶ Each child partition is then processed recursively using the same SAH procedure.
- ▶ Once a partition reaches the maximum chunk size, **recursion terminates and the final Chunk is created**.
- ▶ The left and right partitions are independent and can therefore be processed in parallel using **Taskflow**.

```
// Reorder items according to the selected SAH split
auto splitIter = std::partition(
    items.begin(),
    items.end(),
    [=](const SpatialItem& item)
    {
        int binIdx = std::min(
            static_cast<int>(SAH_NUM_BINS - 1),
            static_cast<int>(
                (item.centroid[splitAxis] -
                 centroidMin[splitAxis]) * scale));

        return binIdx <= bestSplit;
    });

size_t midIndex = std::distance(items.begin(), splitIter);

// Safety fallback if the partition fails
if (midIndex == 0 || midIndex == items.size())
{
    midIndex = items.size() / 2;
}

// Recursively process both spatial partitions
subflow.emplace(
    [this, scene, items, midIndex](tf::Subflow& sf)
    {
        BuildBinnedSahNodeTask(
            sf,
            scene,
            items.subspan(0, midIndex));
    })
    .name("SAH_Left");

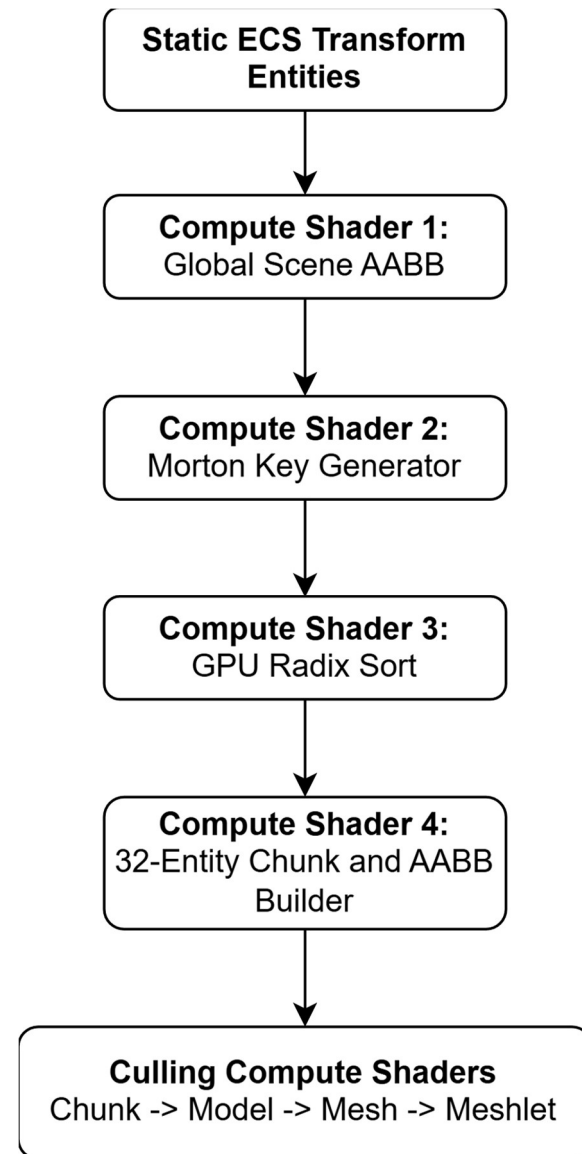
subflow.emplace(
    [this, scene, items, midIndex](tf::Subflow& sf)
    {
        BuildBinnedSahNodeTask(
            sf,
            scene,
            items.subspan(midIndex));
    })
    .name("SAH_Right");
```

<https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/System/Core/StaticSpatialSahSystem.cpp>

Culling XXI.

GPU-Generated Morton BVH

- ▶ The CPU-generated SAH BVH provides a high-quality spatial partition but **constructing it can become expensive for scenes** containing millions of entities.
- ▶ Although the Static region rarely changes, large scenes may still benefit from a **much faster GPU-generated spatial hierarchy**.
- ▶ The alternative approach is built entirely through a sequence of compute shader passes.
- ▶ Instead of recursively searching for optimal SAH split planes, every static entity is assigned a compact **Morton-based spatial key**.
- ▶ After sorting these keys, spatially nearby entities become adjacent in the resulting range.
- ▶ The sorted range is then divided into fixed-size groups of **32 entities**, creating the spatial chunks used by hierarchical culling.



Culling XXII.

Global Scene AABB Generation

- ▶ The first compute shader determines the complete **world-space bounds of the Static ECS region**.
- ▶ Each invocation processes the bounding data of its assigned entity.
- ▶ **Subgroup reduction** operations first calculate the local minimum and maximum bounds.
- ▶ Only one elected invocation per subgroup performs the global atomic update.
- ▶ However, **atomicMin** and **atomicMax** operate on integer values. **Native floating-point atomic min/max operations are not universally available** or suitable across all target hardware.
- ▶ Therefore, each floating-point value is converted into a **monotonically sortable unsigned integer representation** before performing the atomic operation.
- ▶ The global minimum and maximum are calculated using integer atomics and later converted back to floating-point values.
- ▶ This preserves the ordering of the original floating-point values, allowing integer **atomicMin and atomicMax to correctly determine the final Scene AABB**.

```
uint floatToSortableUint(float f)
{
    uint u = floatBitsToUint(f);

    // Preserve floating-point ordering in unsigned integer space
    uint mask = uint(-int(u >> 31)) | 0x80000000u;

    return u ^ mask;
}

float sortableUintToFloat(uint u)
{
    uint mask = ((u >> 31) - 1u) | 0x80000000u;

    return uintBitsToFloat(u ^ mask);
}

//-----

void main()
{
    //...

    vec3 sgMin = subgroupMin(localMin);
    vec3 sgMax = subgroupMax(localMax);

    if (subgroupElect())
    {
        atomicMin(sceneAABB.minX, floatToSortableUint(sgMin.x));
        atomicMin(sceneAABB.minY, floatToSortableUint(sgMin.y));
        atomicMin(sceneAABB.minZ, floatToSortableUint(sgMin.z));

        atomicMax(sceneAABB.maxX, floatToSortableUint(sgMax.x));
        atomicMax(sceneAABB.maxY, floatToSortableUint(sgMax.y));
        atomicMax(sceneAABB.maxZ, floatToSortableUint(sgMax.z));
    }
}
```

<https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/Shader/Passes/Morton/StaticSceneAABB.comp>

Culling XXIII.

Morton Encoding and Spatial Sorting

- ▶ Each entity position is **normalized relative to the global Scene AABB**.
- ▶ The normalized position is converted into a **27-bit Morton code**.
- ▶ The bounding radius is additionally converted into a compact **size classification**.
- ▶ Both values are packed into a single **32-bit spatial key**.
- ▶ The corresponding dense transform index is stored as the associated value.
- ▶ A **GPU radix sort then sorts** the complete key-value range.
- ▶ After sorting, entities that are spatially close to each other are **also located close to each other in memory**.

```
uint expandBits9(uint v)
{
    v = (v * 0x00010001u) & 0xFF000FFu;
    v = (v * 0x00000101u) & 0x0F00F0Fu;
    v = (v * 0x00000111u) & 0xC30C30C3u;
    v = (v * 0x0000005u) & 0x49249249u;
    return v;
}

uint CalculateMortonCode27(vec3 normalizedPos)
{
    uint x = min(max(uint(normalizedPos.x * 512.0), 0u), 511u);
    uint y = min(max(uint(normalizedPos.y * 512.0), 0u), 511u);
    uint z = min(max(uint(normalizedPos.z * 512.0), 0u), 511u);

    return expandBits9(x) |
           (expandBits9(y) << 1) |
           (expandBits9(z) << 2);
}

uint CalculateSizeBin(float radius)
{
    float bin = log2(radius) + 5.0;
    return clamp(uint(max(bin, 0.0)), 0u, 15u);
}

//-----

void main()
{
    //...

    vec3 normalizedPos = (center - sceneMin) / extent;

    uint mortonCode = CalculateMortonCode27(normalizedPos);
    uint sizeBin = CalculateSizeBin(radius);

    uint key = PackMorton32(sizeBin, mortonCode);

    GET_MORTON_KEY(
        ctx.mortonKeysBufferAddr,
        transformDenseIndex) = key;

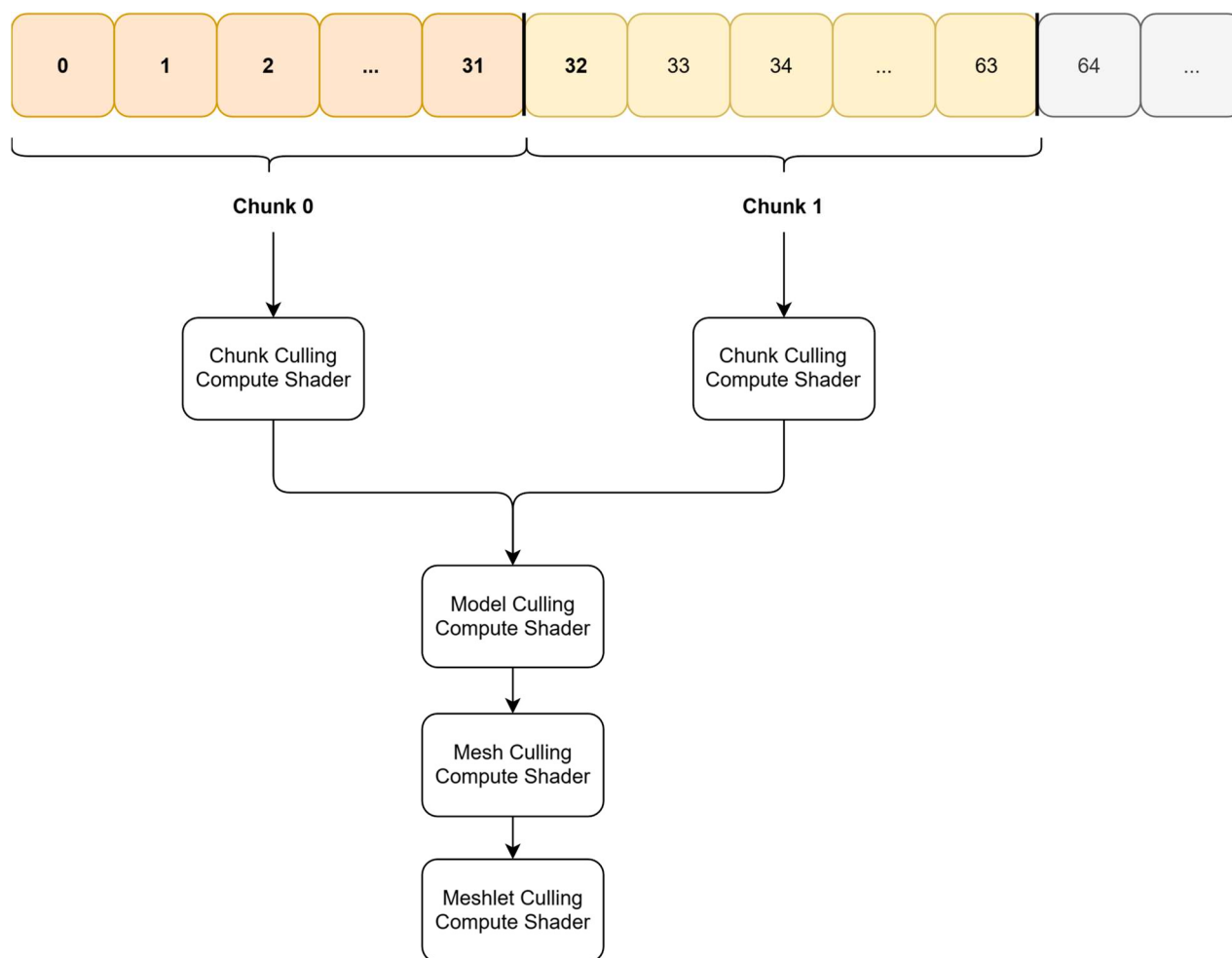
    GET_MORTON_VALUE(
        ctx.mortonValuesBufferAddr,
        transformDenseIndex) = transformDenseIndex;
}
```

<https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/Shaders/Passes/Morton/MortonGenerator.comp>

Culling XXIV.

Morton Chunk Construction

- ▶ The radix-sorted range is divided into fixed-size groups of **32 entities**. Each group becomes a single spatial chunk.
- ▶ During chunk generation, the combined **world-space bounds of all contained entities are calculated**. The resulting AABB becomes the bounding volume of the chunk.
- ▶ If a chunk is outside the camera frustum or occluded, all contained entities can be rejected immediately.
- ▶ The remaining visible entities continue through the existing **Model → Mesh → Meshlet** culling hierarchy.
- ▶ The **culling behavior is therefore almost identical to the CPU-generated SAH BVH approach**, only the spatial hierarchy construction method is different.



<https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/Shaders/Passes/Morton/ChunkBuilder.comp>

Culling XXV.

Spatial Partitioning Benchmark

- ▶ This benchmark was executed with **10 million randomly distributed static entities**.
- ▶ Without spatial partitioning, the GPU must launch **model culling work for every single entity**. The flat GPU-driven approach therefore scales approximately linearly with scene size.
- ▶ At 10 million entities, the **ModelCullingPass** alone requires **5.885 ms**.
- ▶ With the Static BVH / Chunk hierarchy enabled, large invisible spatial regions are rejected **before individual model culling begins**.
- ▶ Each rejected chunk eliminates up to **32 models at once**, together with all of their meshes and meshlets.
- ▶ **With Static BVH / Chunk Culling: 0.264 ms, ~22× faster total culling at 10 million entities.**

Table 5: GPU culling performance without static spatial partitioning (10 million entities).

Culling Stage	GPU Time (ms)
Culling Command Reset	0.006
Model Culling	5.885
Mesh Culling	0.003
Total	5.908

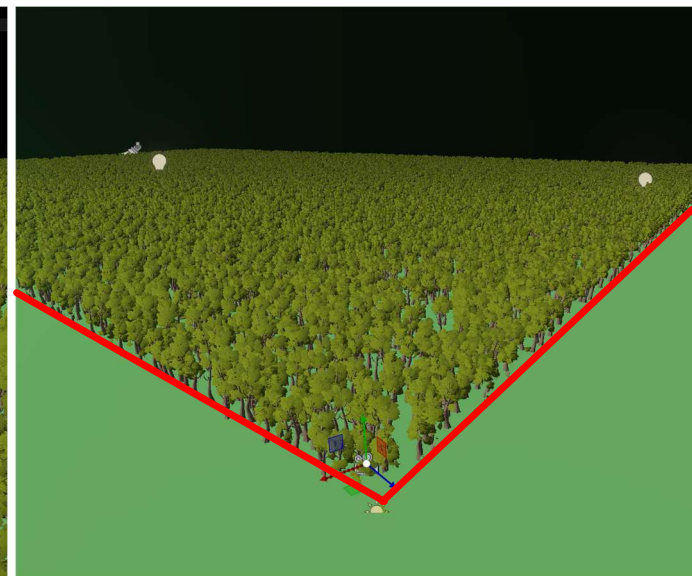
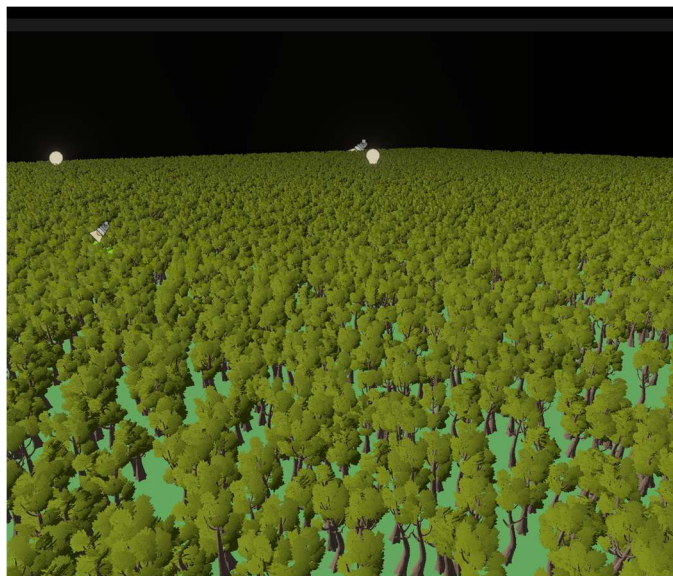
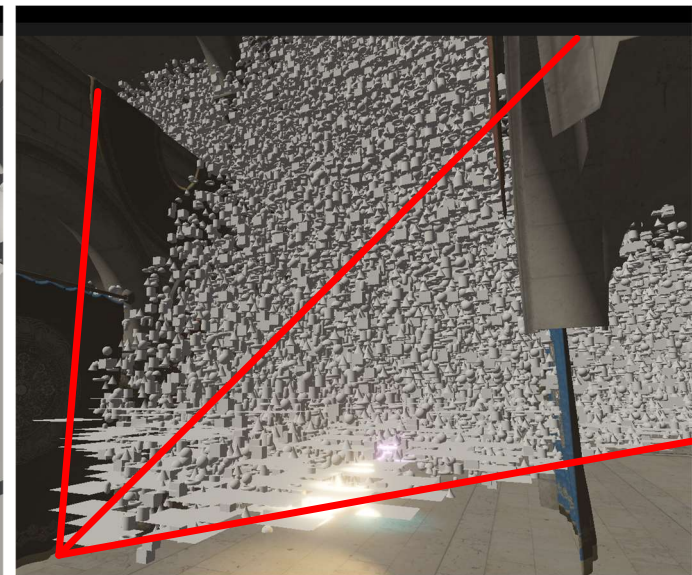
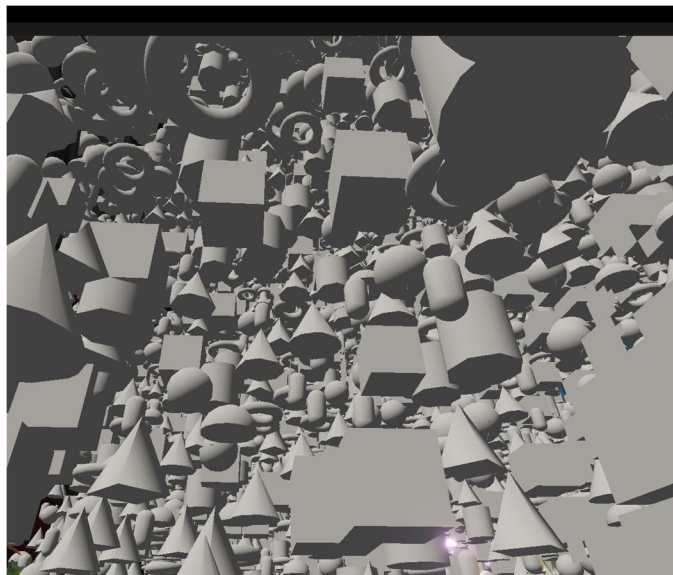
Table 4: GPU culling performance with static BVH/chunk partitioning (10 million entities).

Culling Stage	GPU Time (ms)
Culling Command Reset	0.005
Static Chunk Culling	0.069
Static Model Culling	0.164
Model Culling	0.005
Mesh Culling	0.003
Total	0.264

Culling XXVI.

Frustum Culling

- ▶ The camera view volume is represented by a **frustum bounded by six planes**: Near, Far, Left, Right, Top, and Bottom.
- ▶ Each bounding volume is tested against the frustum using **Sphere** → **AABB intersection tests**.
- ▶ The result has three possible states: **Outside, Intersecting, or Inside**.
 - ▶ Outside objects are immediately rejected from the rendering pipeline.
 - ▶ **Intersecting objects frustum culling continue** through the hierarchical Model → Mesh → Meshlet culling pipeline.
 - ▶ If a model is **completely Inside the frustum**, all of its meshes and meshlets are also guaranteed to be inside, **so further frustum tests can be skipped**.
- ▶ The Inside state is propagated through the hierarchy by encoding the **fully-inside flag into bit 31 of the entity ID**.



Culling XXVII.

Frustum Culling Implementation

- ▶ The frustum planes are ordered to maximize **early termination**, allowing objects behind the camera to be rejected as early as possible.
- ▶ A **sphere test is performed first** because it provides a fast conservative visibility test.
- ▶ If the sphere is completely Outside or Inside, the result is immediately returned.
- ▶ Only Intersecting spheres continue to the more accurate **AABB vs Frustum** test.
- ▶ Both tests evaluate the object against all six frustum planes and return the same three-state visibility result.

```
uint TestSphereFrustum(vec3 center, float radius, vec4 planes[6]) {
    bool intersecting = false;
    for(int i = 0; i < 6; ++i) {
        vec4 plane = planes[i];
        float dist = dot(plane.xyz, center) - plane.w;

        if(dist < -radius) return INTERSECTION_OUTSIDE;
        if(dist < radius) intersecting = true;
    }
    return intersecting ? INTERSECTION_INTERSECT : INTERSECTION_INSID
}

uint TestAABBFrustum(vec3 aabbMin, vec3 aabbMax, vec4 planes[6]) {
    vec3 extents = (aabbMax - aabbMin) * 0.5;
    vec3 center = (aabbMax + aabbMin) * 0.5;

    bool intersecting = false;
    for(int i = 0; i < 6; ++i) {
        vec4 plane = planes[i];
        float r = dot(extents, abs(plane.xyz));
        float dist = dot(plane.xyz, center) - plane.w;

        if (dist < -r) return INTERSECTION_OUTSIDE;
        if (dist < r) intersecting = true;
    }
    return intersecting ? INTERSECTION_INTERSECT : INTERSECTION_INSID
}

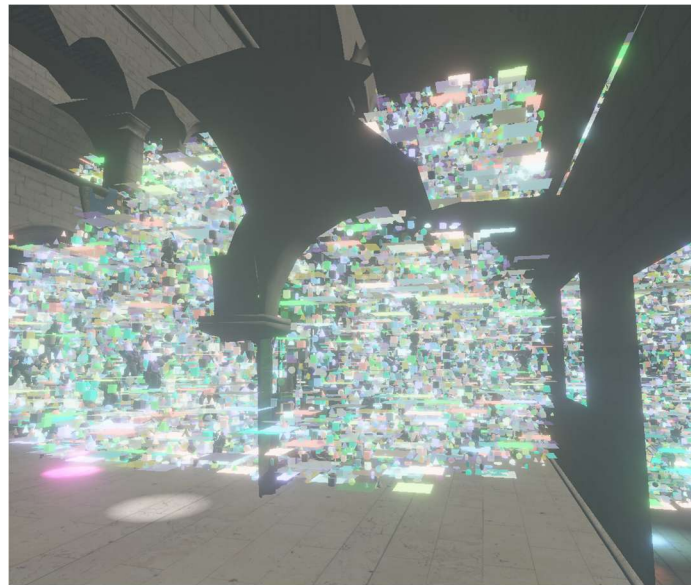
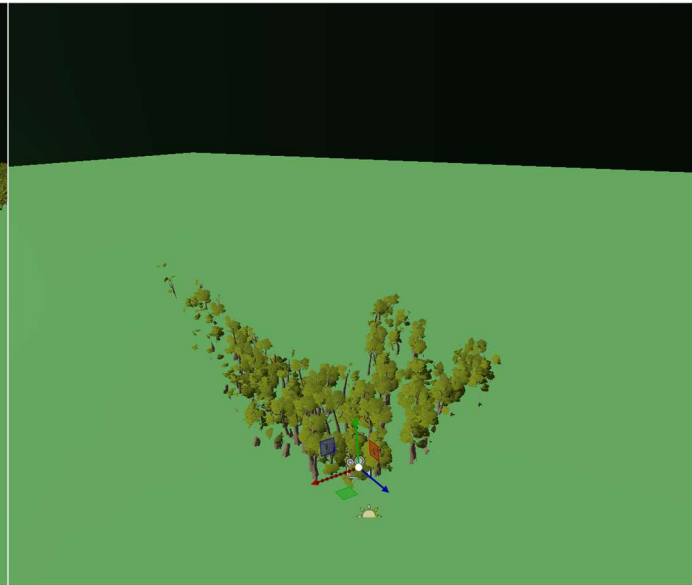
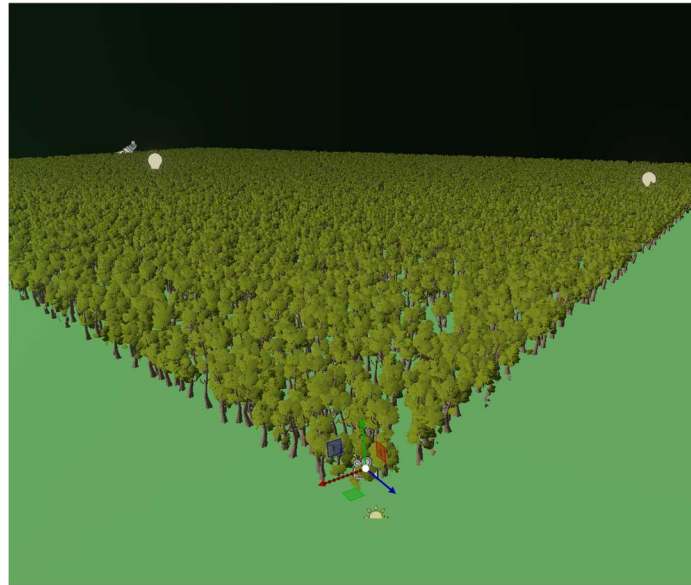
uint TestFrustum(GpuMeshCollider collider, vec4 planes[6]) {
    uint sphereResult = TestSphereFrustum(collider, planes);
    if (sphereResult != INTERSECTION_INTERSECT) return sphereResult;
    return TestAABBFrustum(collider, planes);
}
```

<https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/Shaders/Includes/Utils/CullingMath.glsl>

Culling XXVIII.

Occlusion Culling

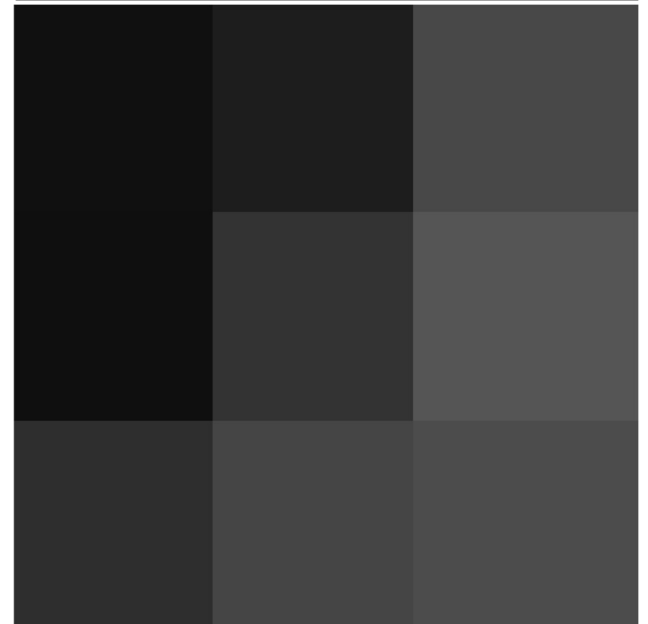
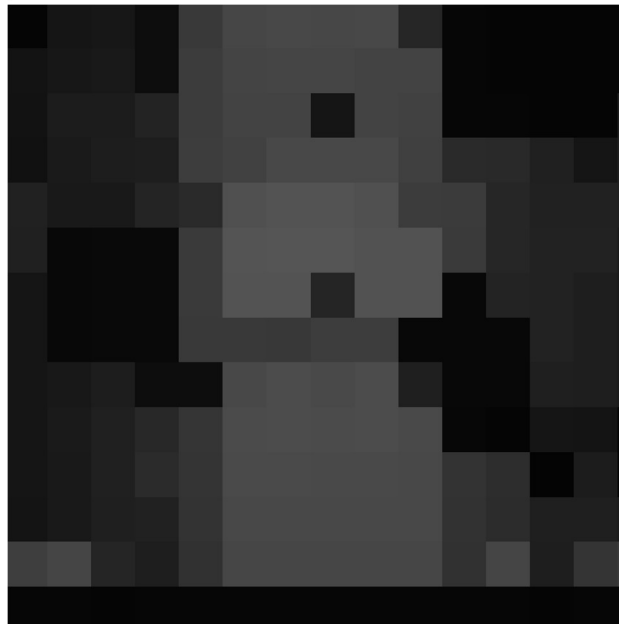
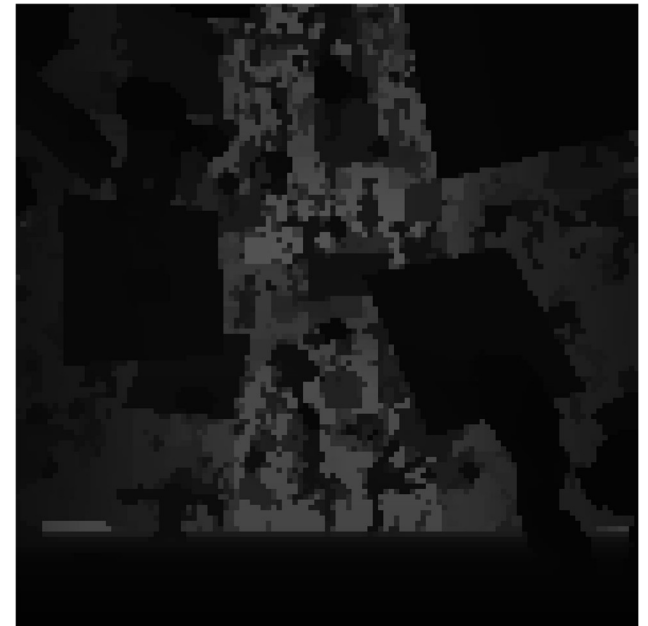
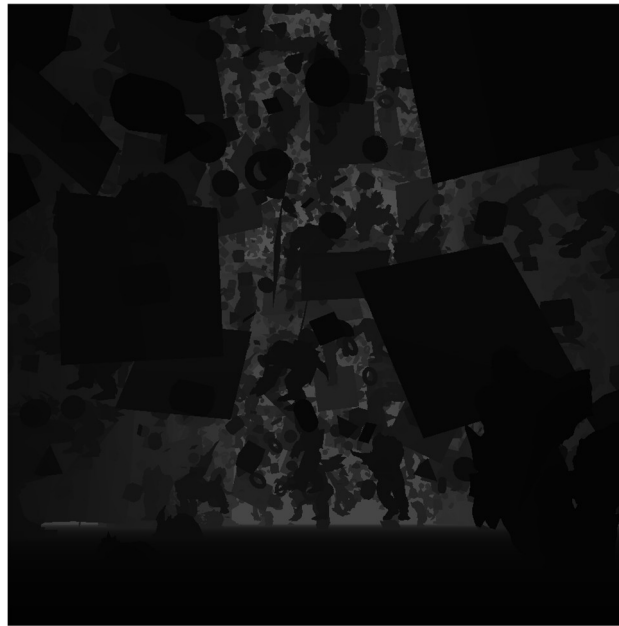
- ▶ Frustum culling removes objects that are outside the camera view, but it cannot detect objects that are **completely hidden behind other geometry**.
- ▶ Occlusion culling aims to **reject these hidden objects** before unnecessary geometry processing is performed.
- ▶ A typical example is an object located completely **behind a wall**: it may be inside the camera frustum, but it contributes nothing to the final image.
- ▶ Accurate occlusion testing is difficult, especially in a massively parallel GPU-driven pipeline.
- ▶ The approach used in the engine is **Hierarchical Z-Buffer (Hi-Z) Occlusion Culling**.
- ▶ The technique uses **previously generated depth information** to conservatively determine whether a bounding volume is hidden by already visible geometry.



Culling XXIX.

Hierarchical Z-Buffer (Hi-Z)

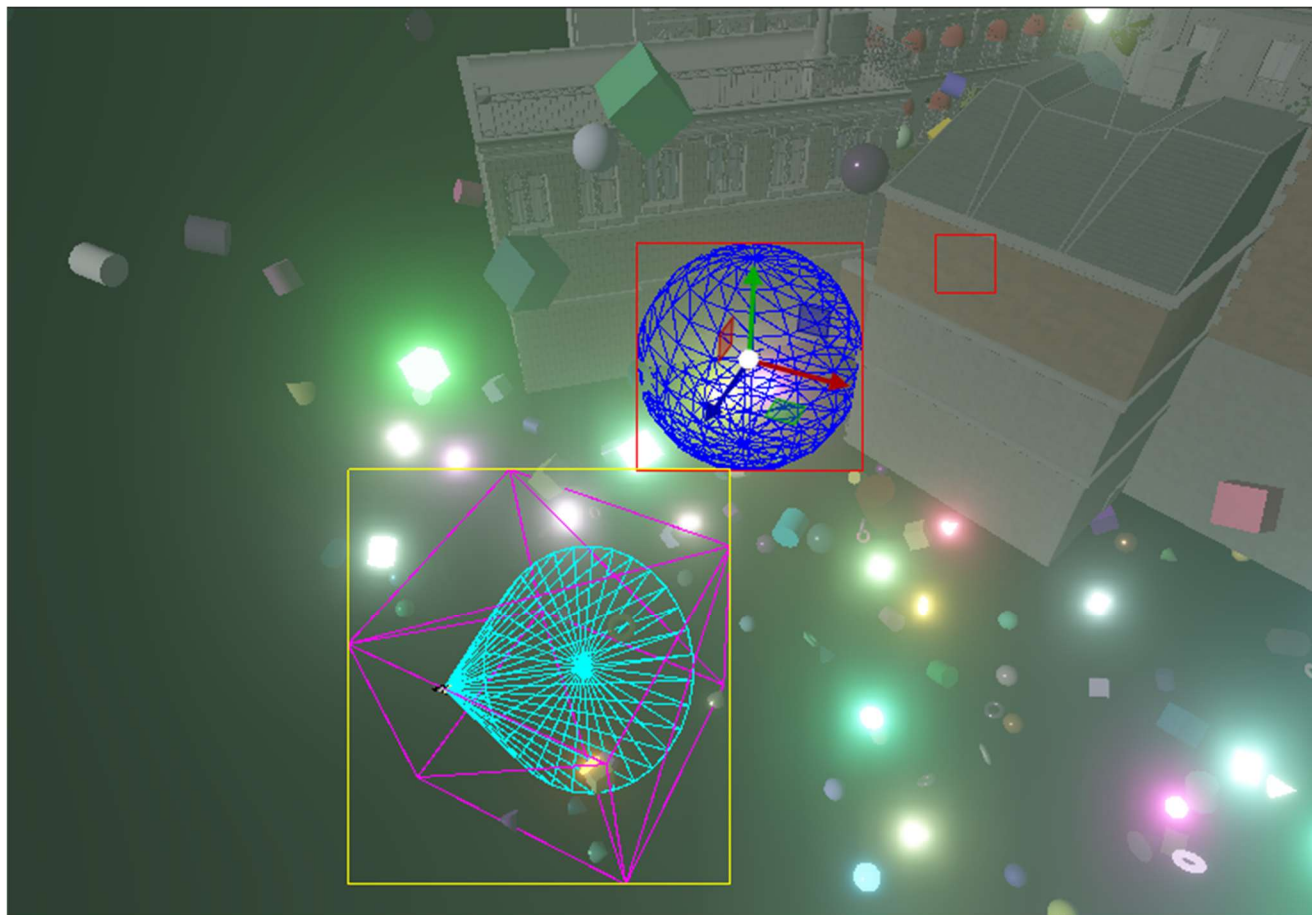
- ▶ Hi-Z is a hierarchical representation of the depth buffer built as a **mip chain**.
- ▶ The highest-resolution level contains the original depth information.
- ▶ Each subsequent mip level represents a larger screen-space region with **lower spatial resolution**.
- ▶ In this implementation, every mip level is generated from a **2×2 region of the previous level**.
- ▶ For a reversed-depth configuration, the **maximum depth value of the 2×2 region** is propagated to the next mip level.
- ▶ This produces a conservative depth hierarchy that allows **large screen-space regions to be tested using a single texture lookup**.
- ▶ The depth hierarchy allows the culling algorithm to choose the mip level that approximately **matches the projected size** of the tested bounding volume.



Culling XXX.

Screen-Space Projection and Mip Selection

- ▶ Occlusion testing is performed in **screen space**.
- ▶ The object's bounding sphere is projected onto the screen using the **Zeux sphere projection method**.
- ▶ The projection provides:
 - ▶ the screen-space center,
 - ▶ the projected X/Y extent,
 - ▶ and the closest depth of the bounding sphere.
- ▶ The **projected center** is used as the **Hi-Z texture coordinate**.
- ▶ The **projected size determines which Hi-Z mip level** should be sampled.
- ▶ The selected mip must fully cover the projected bounding volume.

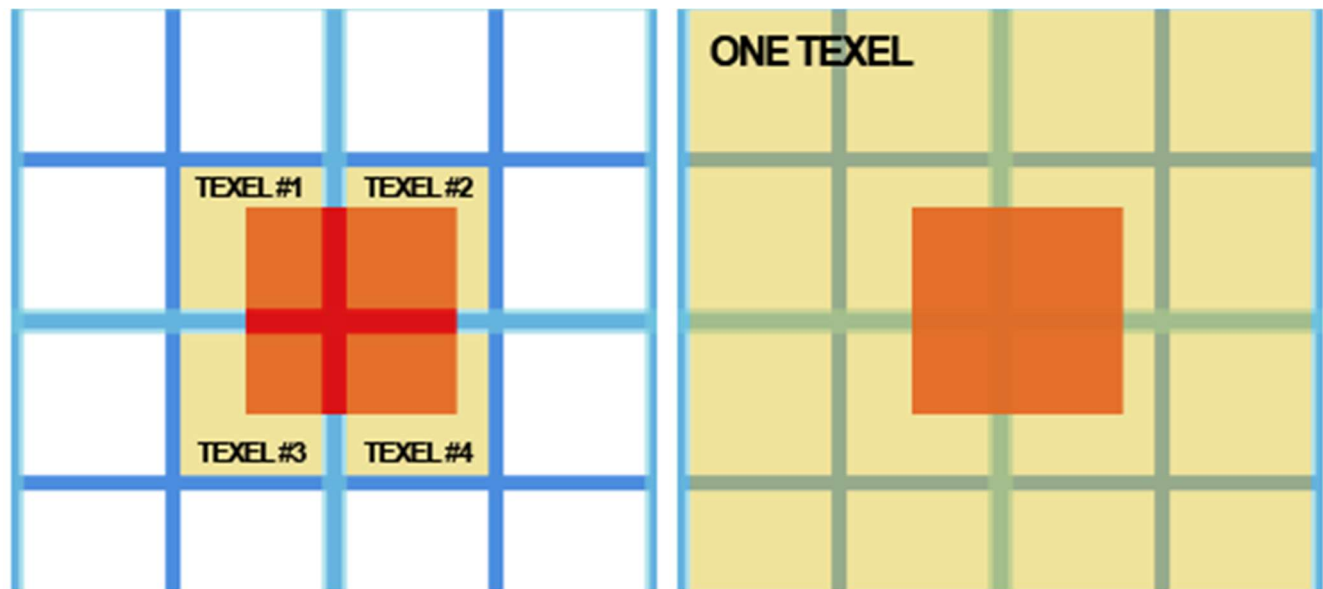


<https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/Shaders/Includes/Utils/Occlusion.glsl>

Culling XXXI.

Meshlet-Level Hi-Z Occlusion

- ▶ The selected Hi-Z mip level must fully cover the projected bounding volume.
- ▶ To guarantee conservative coverage, the projected bounds are sampled at their **four corners**.
- ▶ A **Vulkan max reduction sampler** combines these samples and returns the maximum depth value.
- ▶ This makes the **four-corner depth test efficient** while preserving conservative occlusion results.
- ▶ **Meshlets are particularly well suited for Hi-Z occlusion** because their small bounding volumes allow sampling from much higher-resolution mip levels.
- ▶ As the culling hierarchy progresses from Model → Mesh → Meshlet, the **Hi-Z test becomes increasingly precise and effective**.



<https://www.rastergrid.com/blog/2010/10/hierarchical-z-map-based-occlusion-culling/>

Culling XXXII.

Hi-Z Occlusion Culling Implementation

- ▶ The occlusion test begins by transforming the bounding sphere into **view space**.
- ▶ The **Zeux approximate sphere projection** calculates the projected screen-space bounds of the sphere.
- ▶ From these bounds, the screen-space size is calculated and used to determine the appropriate **Hi-Z mip level**.
- ▶ **Zero-Pixel Culling**: Objects smaller than one pixel can be rejected immediately.
- ▶ The Hi-Z depth is sampled using the projected bounds, and the depth of the sphere's closest point to the camera is calculated.
- ▶ If the closest point of the sphere is still behind the sampled Hi-Z depth, the object is considered **fully occluded** and can be rejected.

```
// Zeux Approximate Sphere Projection
bool ProjectSpherePerspective(vec3 viewCenter, float radius, mat4 proj, float near, out vec4 uvBounds) {
    if (-viewCenter.z - radius < near) return false;
    vec2 cx = vec2(viewCenter.x, -viewCenter.z);
    vec2 vx = vec2(sqrt(dot(cx, cx) - radius * radius), radius);
    vec2 minx = mat2(vx.x, vx.y, -vx.y, vx.x) * cx;
    vec2 maxx = mat2(vx.x, -vx.y, vx.y, vx.x) * cx;
    vec2 cy = vec2(viewCenter.y, -viewCenter.z);
    vec2 vy = vec2(sqrt(dot(cy, cy) - radius * radius), radius);
    vec2 miny = mat2(vy.x, vy.y, -vy.y, vy.x) * cy;
    vec2 maxy = mat2(vy.x, -vy.y, vy.y, vy.x) * cy;
    float p00 = abs(proj[0][0]);
    float p11 = abs(proj[1][1]);

    uvBounds = vec4(minx.x / minx.y * p00, minx.x / miny.y * p11,
                    maxx.x / maxx.y * p00, maxx.x / maxy.y * p11);
    uvBounds = uvBounds * 0.5 + 0.5;

    return true;
}

bool IsSphereOccludedPerspective(vec3 worldCenter, float radius, mat4 view, mat4 proj, float zNear,
float zFar, sampler2D depthPyramid, vec2 screenRes, bool enableDepthOcclusion, out float outScreenSizePixels) {
    vec3 viewCenter = (view * vec4(worldCenter, 1.0)).xyz;
    vec4 uv;
    outScreenSizePixels = INFINITE_SCREEN_SIZE;

    if (ProjectSpherePerspective(viewCenter, radius, proj, zNear, uv)) {
        vec2 sizeInPixels = (vec2(uv.zw) - vec2(uv.xy)) * screenRes;
        outScreenSizePixels = max(sizeInPixels.x, sizeInPixels.y);

        if (outScreenSizePixels < 1.0) return true;

        if (enableDepthOcclusion) {
            float lod = max(0.0, ceil(log2(outScreenSizePixels * 0.5)));
            vec2 centerUV = (uv.xy + uv.zw) * 0.5;
            centerUV.y = 1.0 - centerUV.y;
            float maxDepth = textureLod(depthPyramid, centerUV, lod).r;

            float sphereClosestDepth = -viewCenter.z - radius;
            float normalizedDepth = (sphereClosestDepth - zNear) / (zFar - zNear);

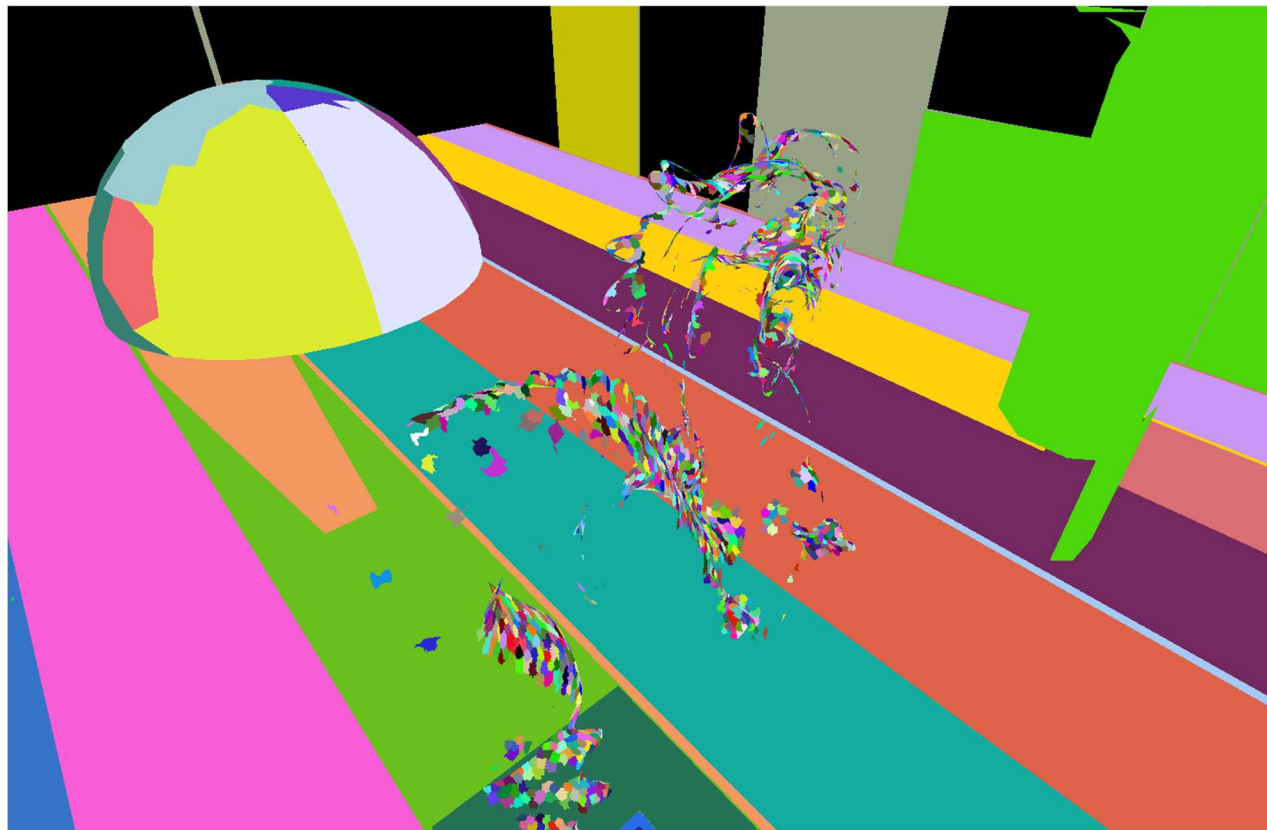
            return normalizedDepth > maxDepth;
        }
    }
    return false;
}
```

<https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/Shaders/Includes/Utils/Occlusion.glsl>

Culling XXXIII.

Meshlet Cone Culling

- ▶ Each meshlet is assigned a **bounding cone**, generated automatically by **meshoptimizer** during meshlet construction.
- ▶ The cone represents the dominant orientation of the triangles contained within the meshlet.
- ▶ Its axis is derived from the overall direction of the meshlet's triangle normals, while the cone cutoff represents their **maximum angular deviation**.
- ▶ During rendering, the direction from the meshlet towards the camera is compared against this cone.
- ▶ If the entire meshlet faces away from the camera, **all of its triangles are effectively back-facing** and the meshlet can be rejected before launching the Mesh Shader.
- ▶ This allows **software-based backface culling** to be performed at the meshlet level.



```
bool TestConeCulling(vec3 apex, vec3 axis, float cutoff, vec3 cameraEye)
{
    vec3 view = normalize(apex - cameraEye);
    return dot(view, axis) >= cutoff;
}
```

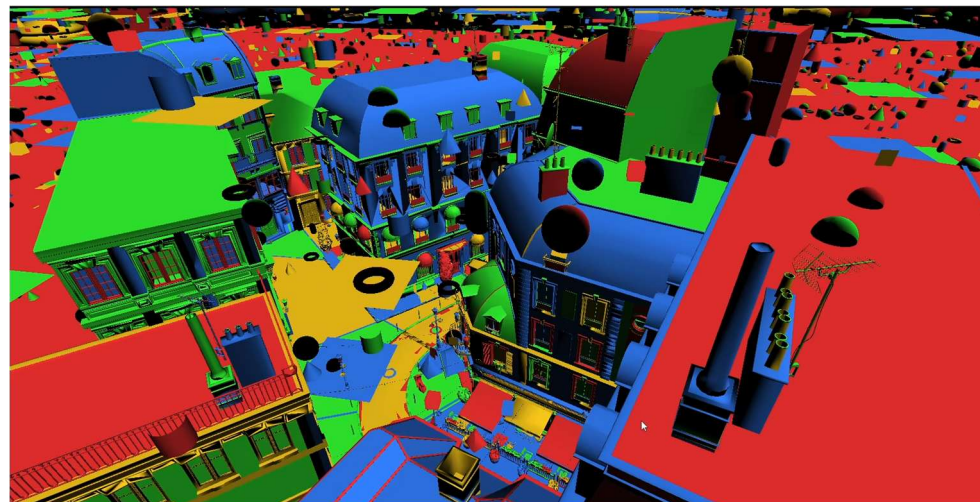
<https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/Shader/Includes/Utils/CullingMath.glsl>

Culling XXXIV.

Dynamic LOD Selection

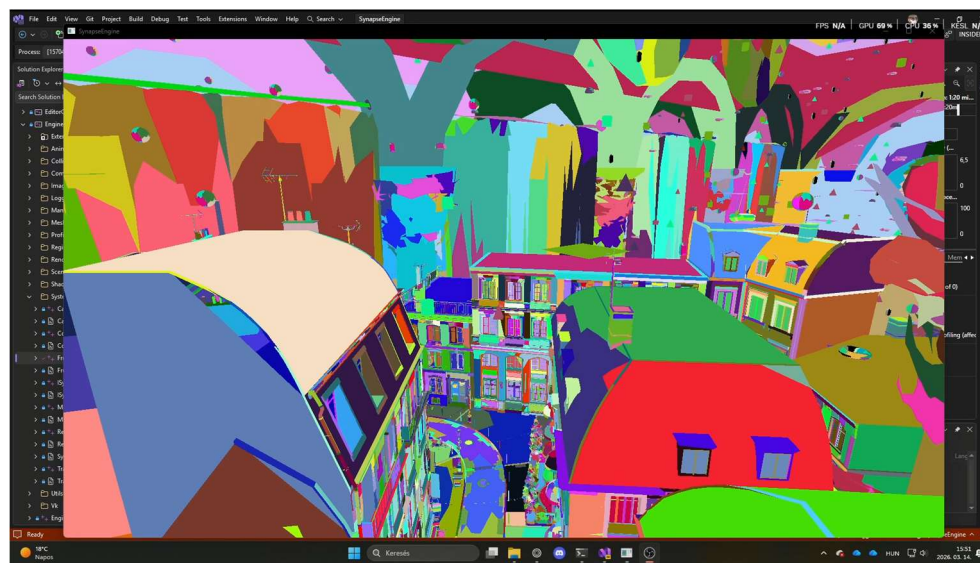
- ▶ The object's **screen-space size** is already determined during Hi-Z occlusion culling.
- ▶ This value is reused for **dynamic LOD selection**.
- ▶ Screen-space size is a **better metric** than distance alone:
 - ▶ Accounts for both **distance and object size**
 - ▶ Large objects can remain highly detailed even when far away
 - ▶ Small objects can use lower detail even when relatively close
- ▶ LOD level is selected based on the projected size:

▶ >512 px:	LOD 0	100% detail
▶ >256 px:	LOD 1	50% detail
▶ >128 px:	LOD 2	25% detail
▶ >64 px:	LOD 3	12.5% detail
- ▶ Reduces geometric workload while **preserving detail where it is visually relevant**.



Random LOD level color visualization.

(Nvidia Bistro is 1 big model but has different LOD on meshes)

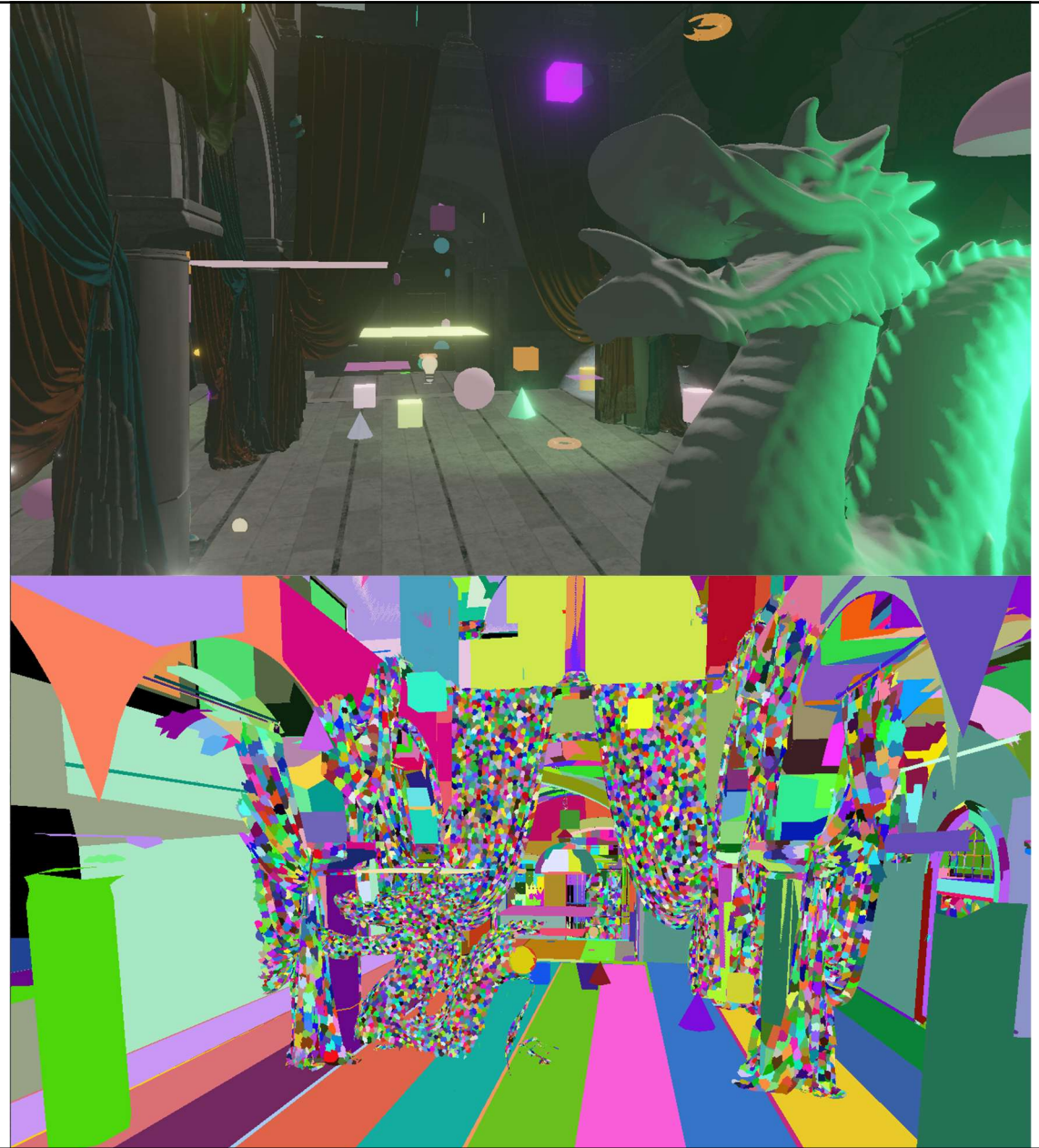


<https://youtu.be/h64PygG19x4>

Culling XXXV.

Final Hierarchical Culling Benchmark

- ▶ Hierarchically, the engine progressively rejects invisible **Models, Meshes, and finally Meshlets**, entirely on the GPU using **Compute and Task Shaders**.
- ▶ The benchmark was performed on a scene containing **1 million entities**:
 - ▶ No Culling: 124.407 ms
 - ▶ + Cone Culling: 94.123 ms
 - ▶ + Frustum Culling: 2.878 ms
 - ▶ + Hi-Z Occlusion: 0.681 ms
 - ▶ + Dynamic LOD: **0.653 ms**
- ▶ **Final result:** the complete GPU-driven hierarchical culling pipeline reduces the rendering cost from **124.407 ms to 0.653 ms**, approximately a **190× reduction** for this 1 million entity scene.



Culling XXXVI.

Animation and Culling

- ▶ The culling pipeline **uniformly handles both static and animated objects**.
- ▶ For each animation, **all frames are precomputed on the CPU** with:
 - ▶ Model AABB and bounding sphere
 - ▶ Mesh AABB and bounding sphere
 - ▶ Meshlet AABB and bounding sphere
- ▶ The culling shader dynamically checks whether an entity has an **animation component**.
- ▶ For animated objects:
 - ▶ The current animation frame is determined.
 - ▶ The corresponding **precomputed colliders** are retrieved.
 - ▶ These colliders are used for the remaining culling stages.
- ▶ Static objects use their **default model colliders**.
- ▶ The rest of the culling pipeline remains **identical for both cases**.
- ▶ No separate static/animated culling shaders are required → **a single unified GPU-driven culling pipeline** handles both.



```
// 4. Resolve Model metadata and addresses
ModelAllocationInfo mAlloc = GET_MODEL_ALLOC(ctx.globalModelAllocationBufferAddr, modelComp.modelIndex);
GpuModelAddresses addr = GET_MODEL_ADDRESSES(ctx.modelAddressBufferAddr, modelComp.modelIndex);
GpuMeshCollider localGlobalCollider = addr.globalCollider;

// Evaluate Animation frame collider if applicable
if(ctx.animationSparseMapBufferAddr != 0) {
    uint animSparseIndex = GET_SPARSE_INDEX(ctx.animationSparseMapBufferAddr, entityId);
    if (animSparseIndex != INVALID_INDEX) {
        AnimationComponent animComp = GET_ANIM_COMP(ctx.animationBufferAddr, animSparseIndex);
        if (animComp.animationIndex != INVALID_INDEX) {
            GpuAnimationAddresses animAddrs = GET_ANIM_ADDRESSES(ctx.animationAddressBufferAddr, animComp.animationIndex);

            if (animAddrs.isReady == 1) {
                localGlobalCollider = GET_MESH_COLLIDER(animAddrs.frameGlobalColliders, animComp.frameIndex);
            }
        }
    }
}

// Transform global collider to world space
GpuMeshCollider worldCollider = TransformCollider(localGlobalCollider, transform.transform);
```

<https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/Animation/Processor/Geometry/AnimationColliderProcessor.cpp>

Synapse Engine: Lighting and Shading

Created By: Péter Tamás

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

2026

Lighting and Shading I.

From Geometry to Lighting

- ▶ So far, we have explored how the engine handles **GPU-driven rendering** and efficiently generates the required draw commands and instance data.
- ▶ We have seen how geometry can be processed through a highly batched rendering pipeline.
- ▶ We also examined the complete **hierarchical GPU-driven culling architecture**.
- ▶ Compute shaders are used to progressively determine which **models, meshes, and meshlets** actually contribute to the final image.
- ▶ **Invisible geometry is rejected** as early as possible, minimizing unnecessary geometry processing and rendering work.
- ▶ At this point, the engine is able to determine **what should be rendered** and render the remaining geometry efficiently.
- ▶ The next question is: **how should the scene be illuminated?**
- ▶ The engine currently supports three main light types: **Directional Lights, Point Lights, and Spot Lights**.
- ▶ Unlike geometry, light sources can potentially affect a large number of visible pixels, making efficient light evaluation equally important.
- ▶ This section explores different **lighting and shading techniques**, from the limitations of traditional forward shading to the implementation of **Deferred Rendering and Forward+ lighting**.

Lighting and Shading II.

Traditional Forward Shading

- ▶ Forward shading is one of the **simplest** and most direct rendering approaches.
- ▶ Geometry is rasterized, and every **generated fragment is shaded** immediately.
- ▶ The Fragment Shader has access to all relevant scene lights.
- ▶ For every visible fragment, the **shader iterates through the available light sources** and accumulates their contribution.
- ▶ The engine supports three primary light types: **Directional, Point and Spot**.
- ▶ Each light evaluates its contribution based on the fragment's material properties, position, and surface normal.
- ▶ After all light contributions have been evaluated, the final color is written directly to the framebuffer.



```
void main()
{
    vec3 finalColor = vec3(0.0);

    for (uint i = 0; i < directionalLightCount; ++i)
        finalColor += CalculateDirectionalLight(...);

    for (uint i = 0; i < pointLightCount; ++i)
        finalColor += CalculatePointLight(...);

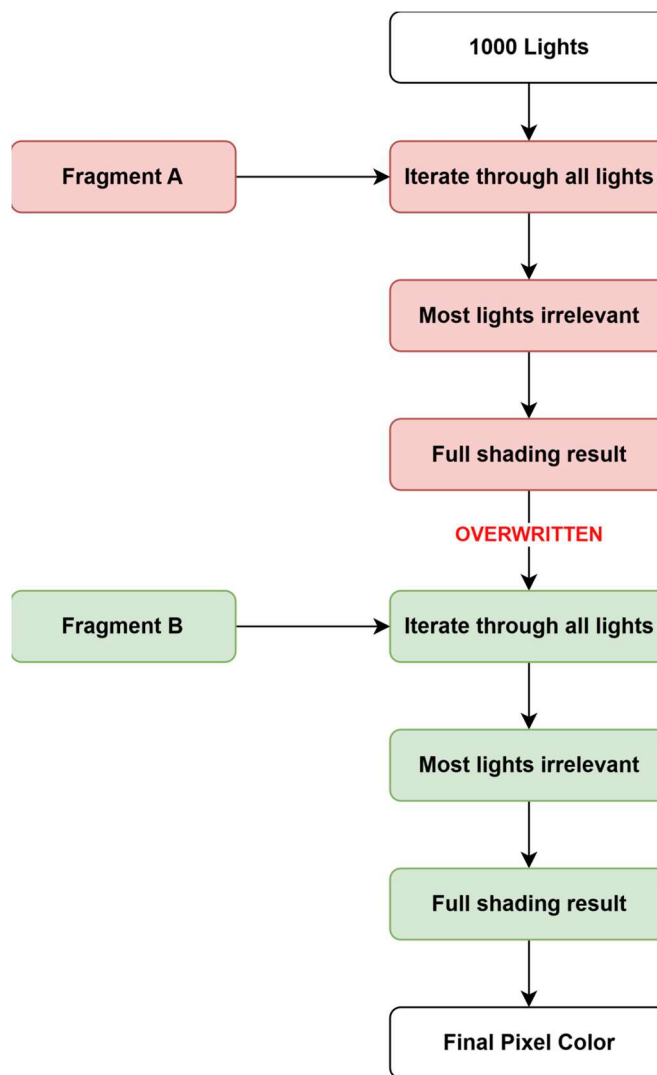
    for (uint i = 0; i < spotLightCount; ++i)
        finalColor += CalculateSpotLight(...);

    outputColor = vec4(finalColor, 1.0);
}
```

Lighting and Shading III.

The Problem with Forward Shading

- ▶ The lighting calculation is performed during geometry rendering.
- ▶ Every fragment potentially iterates through **all available light sources**.
- ▶ Traditional forward shading does not inherently limit the light calculations based on the **screen-space region affected by each light**.
- ▶ A Point Light or Spot Light only affects a **limited volume**, but the shader may still need to iterate over it for every fragment.
- ▶ A software volume or distance check can reject lights that do not affect the fragment, but the **iteration and evaluation overhead still remains**.
- ▶ With hundreds or thousands of lights, iterating over the complete light list for every pixel quickly becomes prohibitively expensive.
- ▶ In addition, multiple fragments can be generated for the same screen-space pixel.
- ▶ A fragment may perform all lighting calculations, **only to be later overwritten by a closer fragment**.
- ▶ This creates two major sources of wasted work: **evaluating irrelevant lights and shading fragments that never contribute to the final image**.



Lighting and Shading IV.

Deferred Shading - GBuffer

- ▶ Deferred Shading **separates geometry processing from lighting calculations.**
- ▶ The rendering pipeline is divided into two main stages: **Geometry Pass, Lighting Pass**
- ▶ During the Geometry Pass, objects are rendered normally, but **lighting is not calculated.**
- ▶ Instead, the Fragment Shader stores the information required for later shading into multiple render targets, collectively called the **G-Buffer.**
- ▶ Typical G-Buffer data includes: **Normal, Color, Roughness/Metallic, Emissive, Depth**
- ▶ Depth testing still determines which fragment is actually visible at every screen-space pixel.
- ▶ Therefore, once the Geometry Pass is complete, the **G-Buffer contains only the data of the fragments that actually contribute** to the final image.
- ▶ The expensive lighting calculations are performed afterwards, using this final visible surface information.
- ▶ This eliminates one of the major inefficiencies of Forward Shading: **lighting calculations are no longer performed for fragments that are later overwritten.**



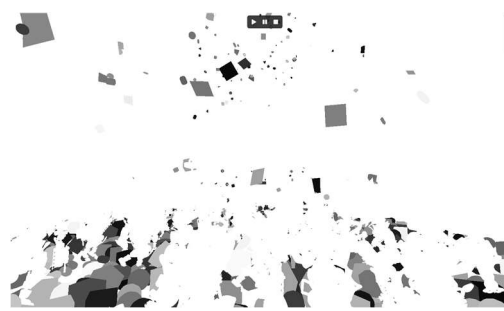
← Final



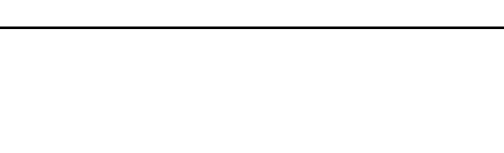
← Color



Normal →



← Metallic



Roughness →



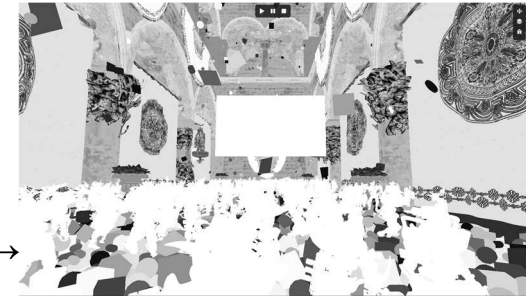
← AO



Depth →



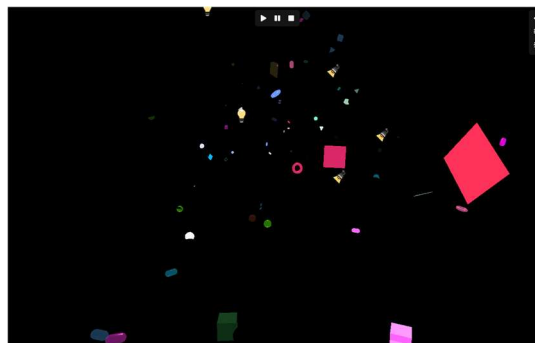
Emissive →



Lighting and Shading V.

Deferred Shading – Additive Light Accumulation

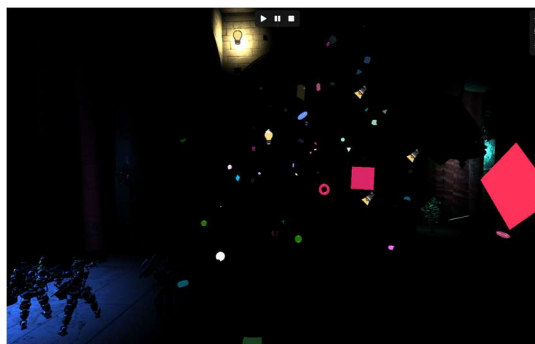
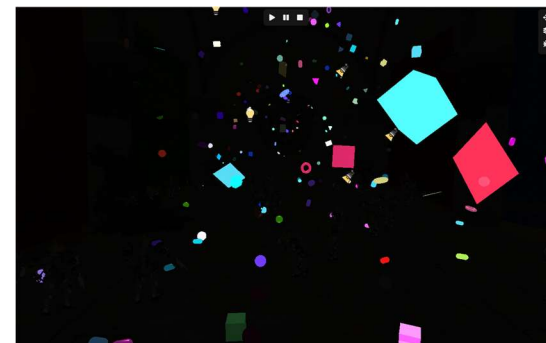
- ▶ The Geometry Pass produces the **G-Buffer**, containing the final visible surface information for every screen-space pixel.
- ▶ The **main lighting texture** target is initially cleared to **black**.
- ▶ Each light is then rendered individually during the Lighting Pass.
- ▶ The Fragment Shader reads the required surface data from the G-Buffer and calculates the contribution of the current light.
- ▶ The resulting light contribution is **additively one-to-one blended** into the main lighting texture.
- ▶ Instead of shading the entire screen for every local light, **geometry representing the light's volume** is rendered.
- ▶ Only the **pixels covered by this volume generate fragments** and evaluate the corresponding light contribution.
- ▶ The final image is **gradually accumulated** as the contributions of Directional, Point, and Spot Lights are rendered.



← Starter
Black Image

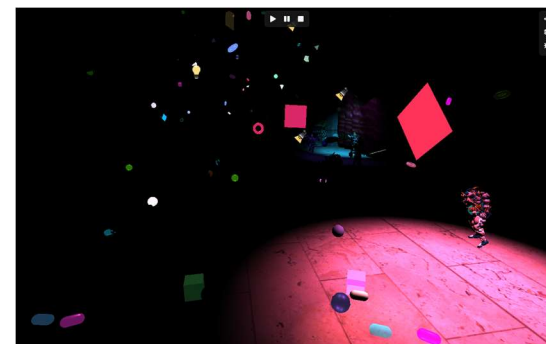
NOTE! This image is originally full black, it already contains the transparent ones here!

Emissive +
AO →



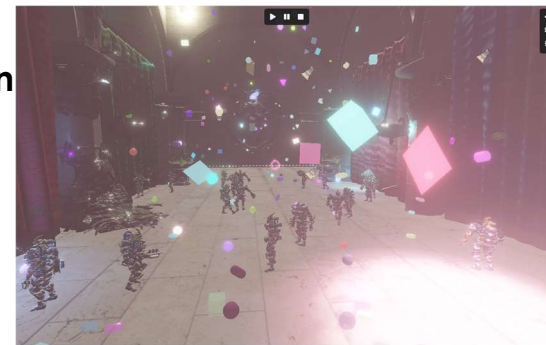
← Point Light
Simulation

Spot Light
Simulation →



← Directional
Light Simulation

Final Image
Addition +
Bloom →



Lighting and Shading VI.

Entity ID and Visibility Buffer

- ▶ In addition to the G-Buffer, the renderer stores per-pixel identification and visibility information in an **R32G32 texture**.
- ▶ The **R32 channel** stores the final rendered **Entity ID**.
- ▶ This enables **pixel-perfect object picking** in the editor: reading a single pixel directly identifies the entity visible at that screen-space position.
- ▶ The **G32 channel** acts as a compact **visibility/debug buffer**.
- ▶ Multiple values are packed into this 32-bit integer using **bit packing**, including information such as: **PipelineType**, **MaterialType**, **TriangleIndex**, **MeshletIndex**, **MeshIndex**
- ▶ This information can later be decoded to create different **debug and visibility visualizations** without requiring additional render targets.

Transparent Object Handling

- ▶ Transparent objects are also written into the ID texture. For picking purposes, they are rendered as **opaque** into this buffer.
- ▶ This ensures that depth testing selects the **closest transparent entity** at every pixel.



← Entity ID



Pipeline Type →



← LOD



Mesh Index→

← Meshlet Index

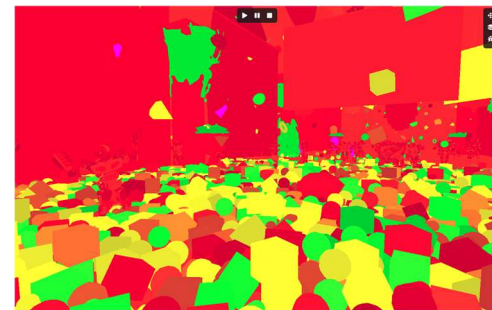
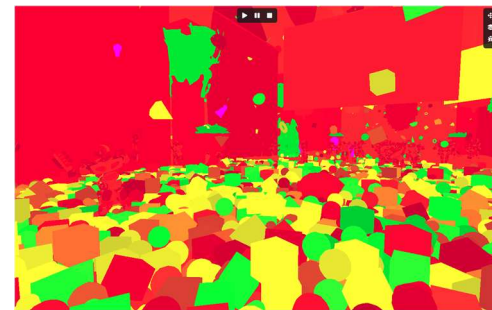
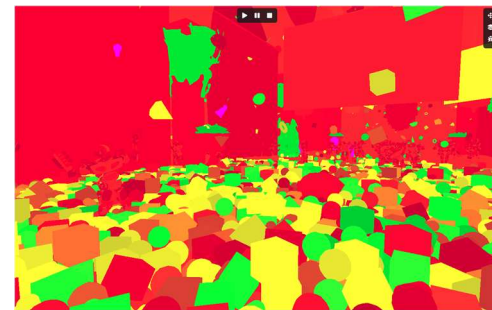
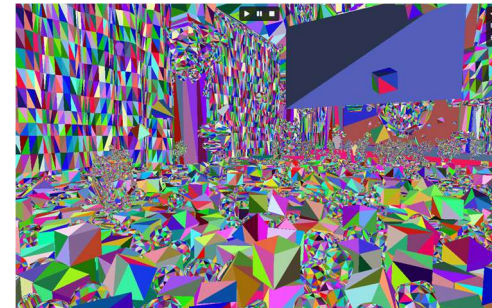
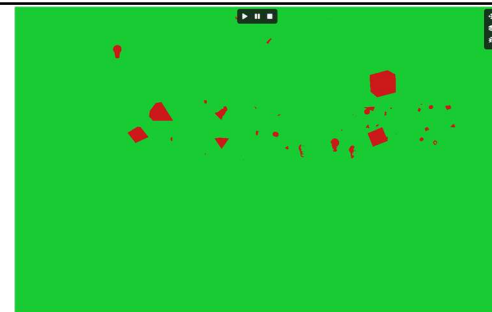


Triangle Index→

← Pipelin/LOD/
Meshlet/
Combined



Material Type→



Lighting and Shading VII.

Deferred Shading – Directional Light Shader

- ▶ A Directional Light has no local position or finite influence volume, therefore its lighting contribution must be evaluated across the **entire screen**.
- ▶ A **full-screen triangle** is rendered to generate fragments for every visible screen-space pixel.
- ▶ The shader reconstructs the required surface data from the G-Buffer, including **world position, albedo, normal, metallic, and roughness**.
- ▶ The light direction is retrieved from the Directional Light buffer and combined with the reconstructed surface and view direction.
- ▶ The complete directional lighting calculation is then performed using the material properties stored in the G-Buffer.
- ▶ The resulting radiance is **additively blended** into the main lighting texture.

```
void main()
{
    outUV = vec2((gl_VertexIndex << 1) & 2, gl_VertexIndex & 2);
    gl_Position = vec4(outUV * 2.0 - 1.0, 0.0, 1.0);

    uint entityId = GET_DIRECTION_VISIBLE_LIGHT(ctx.directionLightVisibleIndexBufferAddr, gl_InstanceIndex);
    uint lightDenseIndex = GET_SPARSE_INDEX(ctx.directionLightSparseMapBufferAddr, entityId);
    uint cameraIndex = GET_SPARSE_INDEX(ctx.cameraSparseMapBufferAddr, ctx.activeCameraEntity);

    outLightDenseIndex = lightDenseIndex;
    outEntityLightIndex = entityId;
    outCameraIndex = cameraIndex;
}

void main()
{
    vec3 position = ReconstructWorldPosition(
        inUV,
        depth,
        camera.viewProjVulkanInv
    );

    vec4 colorMetallic = texture(colorMetallicTexture, inUV);
    vec4 normalRoughness = texture(normalRoughnessTexture, inUV);

    vec3 albedo = colorMetallic.rgb;
    float metallic = colorMetallic.a;
    vec3 normal = normalize(normalRoughness.xyz);
    float roughness = clamp(normalRoughness.a, 0.04, 1.0);

    vec3 dirLightDirection = GET_DIRECTION_LIGHT(ctx.directionLightDataBufferAddr,
        inLightDenseIndex).direction.xyz;

    vec3 lightDir = normalize(-dirLightDirection);

    vec3 viewDir = normalize(camera.eye.xyz - position);

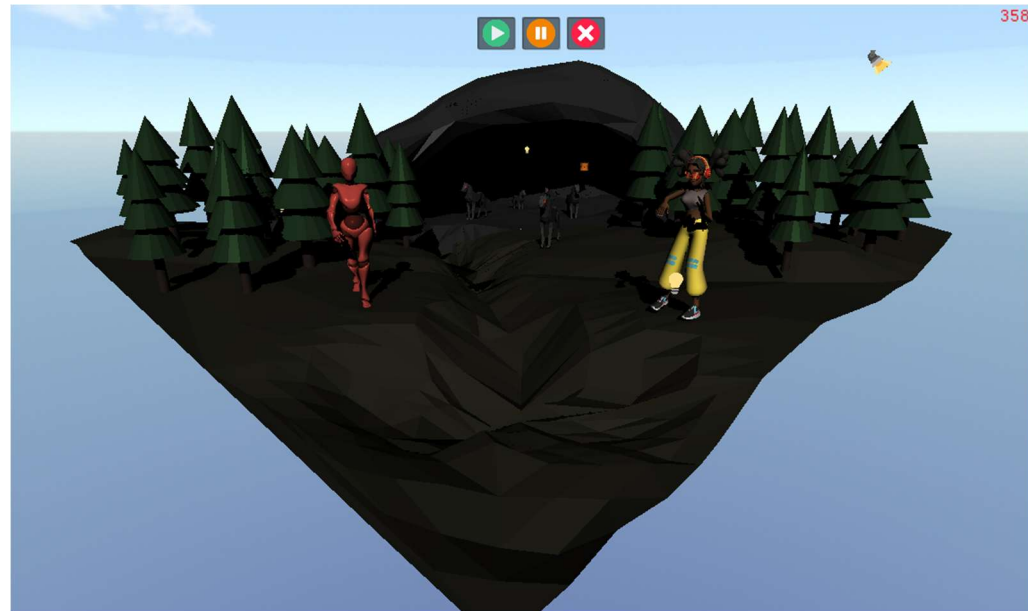
    vec3 radiance = SimulateDirectionalLight(
        ctx.directionLightDataBufferAddr,
        inLightDenseIndex,
        albedo,
        normal,
        viewDir,
        roughness,
        metallic,
        ...
    );
}
```

<https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/Shader/Passes/Shading/Deferred/Lighting/DeferredDirectionalLight.frag>

Lighting and Shading VIII.

Deferred Shading – Directional Light Result

- ▶ The lighting buffer initially contains only the accumulated contribution of the previous lighting passes (AO).
- ▶ The Directional Light shader reads the final **visible surface information directly from the G-Buffer**.
- ▶ The lighting calculation is performed **independently for every screen-space pixel**.
- ▶ Surface orientation and material properties determine how strongly each pixel responds to the incoming light direction.
- ▶ The calculated radiance is then **additively accumulated** into the main lighting texture.

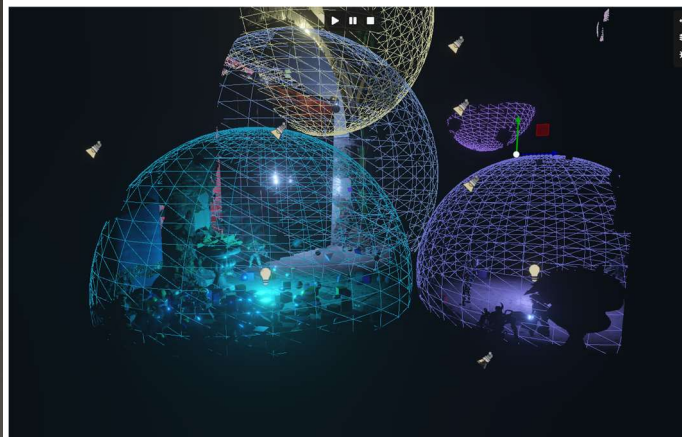


After Dir
Light
Simulation

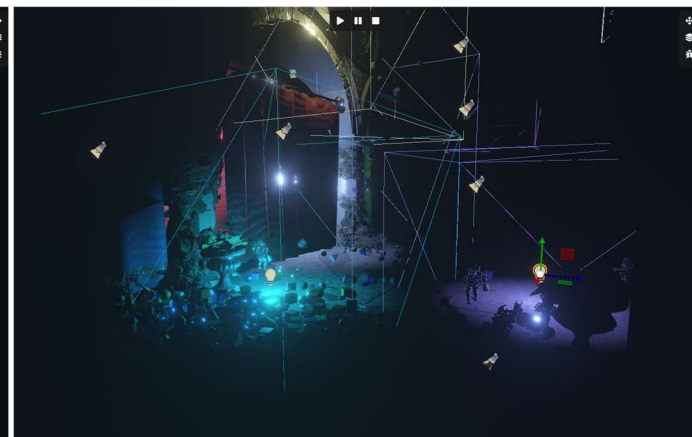
Lighting and Shading IX.

Point and Spot Volumes

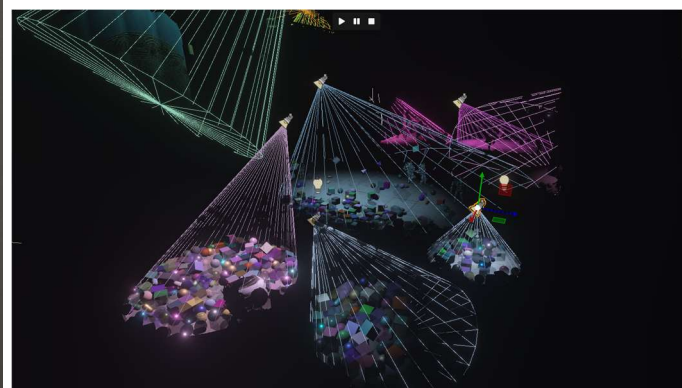
- ▶ Unlike Directional Lights, Point and Spot Lights have a **well-defined spatial influence volume**.
- ▶ A Point Light affects a **spherical region** defined by its position and radius, while a Spot Light affects a **cone-shaped volume**.
- ▶ Instead of rendering a full-screen triangle, geometry representing the light volume is rasterized, **generating fragments only within the affected screen-space region**.
- ▶ The actual volume is not rendered as a high-resolution sphere or cone, since this would introduce unnecessary geometric complexity and approximation artifacts.
- ▶ Point Lights use a simple **bounding cube** enclosing the spherical volume, while Spot Lights use a **pyramid enclosing the conical volume**.
- ▶ **These proxy volumes require only a small number of vertices** and provide a resolution-independent approximation; fragments outside the actual light volume are rejected dynamically in the shader.
- ▶ This slightly **increases the number of generated fragments**, but significantly reduces geometric complexity while maintaining accurate lighting boundaries.



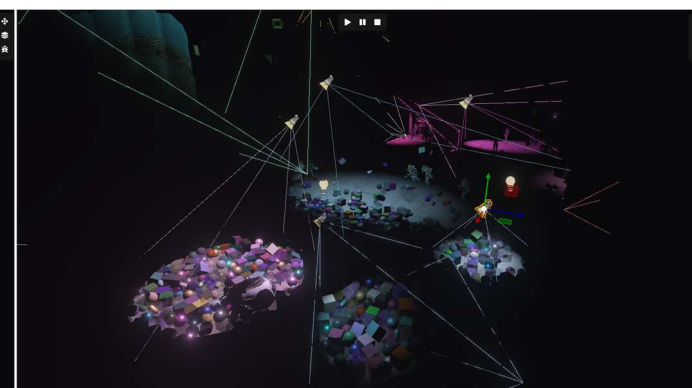
Point Light Sphere Volume



Point Light Proxy Cube



Spot Light Cone Volume

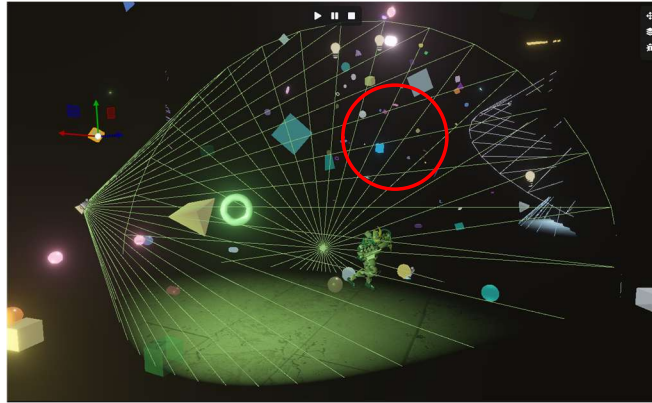


Spot Light Proxy Pyramid

Lighting and Shading X.

Light Volume Rendering

- ▶ Each Point Light and Spot Light is represented by a lightweight **proxy volume geometry**.
- ▶ The proxy geometry is also rendered using **instancing**, allowing many lights of the same type to be processed efficiently in a single draw.
- ▶ A Point Light uses a **cube proxy** that conservatively encloses its spherical influence volume.
- ▶ A Spot Light uses a **pyramid proxy** that conservatively encloses its conical influence volume.
- ▶ The fragment shader performs the **exact light-volume test** and discards fragments outside the actual sphere or cone.
- ▶ **Front-face culling** is used when rendering the volumes. This ensures that the lighting volume remains visible even when the camera is located **inside the light volume**.
- ▶ The next step is **to determine which light volumes are actually relevant** before they are rendered.



```
//Spot Light Shader
void main()
{
    //...

    // 1. Distance-based Culling
    float distToLight = distance(position, light.position);
    if (distToLight > light.range) {
        discard;
    }

    // 2. Angle-based Culling (Spot Light Cone)
    vec3 lightDirToFrag = normalize(position - light.position);
    vec3 spotDirection = normalize(light.direction);
    float theta = dot(lightDirToFrag, spotDirection);

    if (theta < light.outerCosAngle) {
        discard;
    }

    //...
}
```



```
//Point Light Shader
void main()
{
    //...

    // 3. Spherical Culling
    float distToLight = distance(position, light.position);
    if (distToLight > light.radius) {
        discard;
    }

    //...
}
```

<https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/Shader/Passes/Shading/Deferred/Lighting/DeferredSpotLight.frag>

Lighting and Shading XI.

Light Volume Culling

- ▶ The same **CPU-driven and GPU-driven culling architecture** used for scene geometry can also be applied **to Point and Spot Lights**.
- ▶ Since light volumes are rendered as instances, the culling pass determines which lights are visible and updates the corresponding **indirect draw commands and instance index buffers**.
- ▶ **Point Lights** can be tested directly using their **spherical** influence volume.
- ▶ **Spot Lights** use conservative bounding volumes for culling, combining a **bounding sphere and AABB** around the conical light volume
- ▶ In the GPU-driven pipeline, Compute Shaders perform the same **Frustum and Hi-Z Occlusion Culling** techniques introduced earlier.
- ▶ GPU-driven culling is particularly beneficial for lights because **fully occluded light volumes can be discarded before rasterization**, preventing unnecessary fragment generation and lighting calculations.
- ▶ A CPU-driven implementation is also available, but **it cannot take advantage of the GPU-side Hi-Z depth hierarchy** in the same way.
- ▶ The final lighting pass **uses indirect instanced draws**, rendering only the visible Point and Spot Light volumes.

```
void main() {
    FrameGlobalContext ctx = GET_FRAME_CONTEXT(pc.frameGlobalContextBufferAddr);

    uint globalId = gl_GlobalInvocationID.x;
    if (globalId >= ctx.pointLightCount) return;

    PointLightColliderGPU collider = GET_POINT_LIGHT_COLLIDER(ctx.pointLightColliderBufferAddr, globalId);

    uint tagSparseIndex = GET_SPARSE_INDEX(ctx.tagSparseMapBufferAddr, collider.entityIndex);
    if (tagSparseIndex == INVALID_INDEX) return;
    uint entityFlags = GET_TAG_DATA(ctx.tagDataBufferAddr, tagSparseIndex);

    bool isActive = (entityFlags & (1u << 0)) != 0;
    if (!isActive) return;

    uint cameraIdx = GET_SPARSE_INDEX(ctx.cameraSparseMapBufferAddr, ctx.mainCameraEntity);
    CameraComponent camera = GET_CAMERA(ctx.cameraBufferAddr, cameraIdx);

    // 1. Frustum Culling (Sphere)
    uint visibility = INTERSECTION_INTERSECT;
    if (ctx.enablePointLightFrustumCulling == 1)
    {
        uint visibility = TestSphereFrustum(collider.center, collider.radius, camera);
        if (visibility == INTERSECTION_OUTSIDE) return;
    }

    // 2. Occlusion és 0-pixel Culling
    float screenSizePixels = 0.0;
    vec2 screenRes = vec2(ctx.screenWidth, ctx.screenHeight);
    bool enableDepthOcclusion = (ctx.enablePointLightOcclusionCulling == 1);

    if (IsSphereOccludedPerspective(collider.center, collider.radius, camera.view, camera.projVulkan,
        camera.params.x, camera.params.y, depthPyramid, screenRes, enableDepthOcclusion, screenSizePixels)) {
        return;
    }

    uint needsWrite = 1;
    uint subgroupWriteCount = subgroupAdd(needsWrite);
    uint localOffset = subgroupExclusiveAdd(needsWrite);
    uint baseOffset = 0;

    if (subgroupElect()) {
        baseOffset = atomicAdd(GET_VK_DRAW_CMD(ctx.pointLightIndirectCommandBufferAddr, 0).instanceCount, subgroupWriteCount);
    }

    baseOffset = subgroupBroadcastFirst(baseOffset);

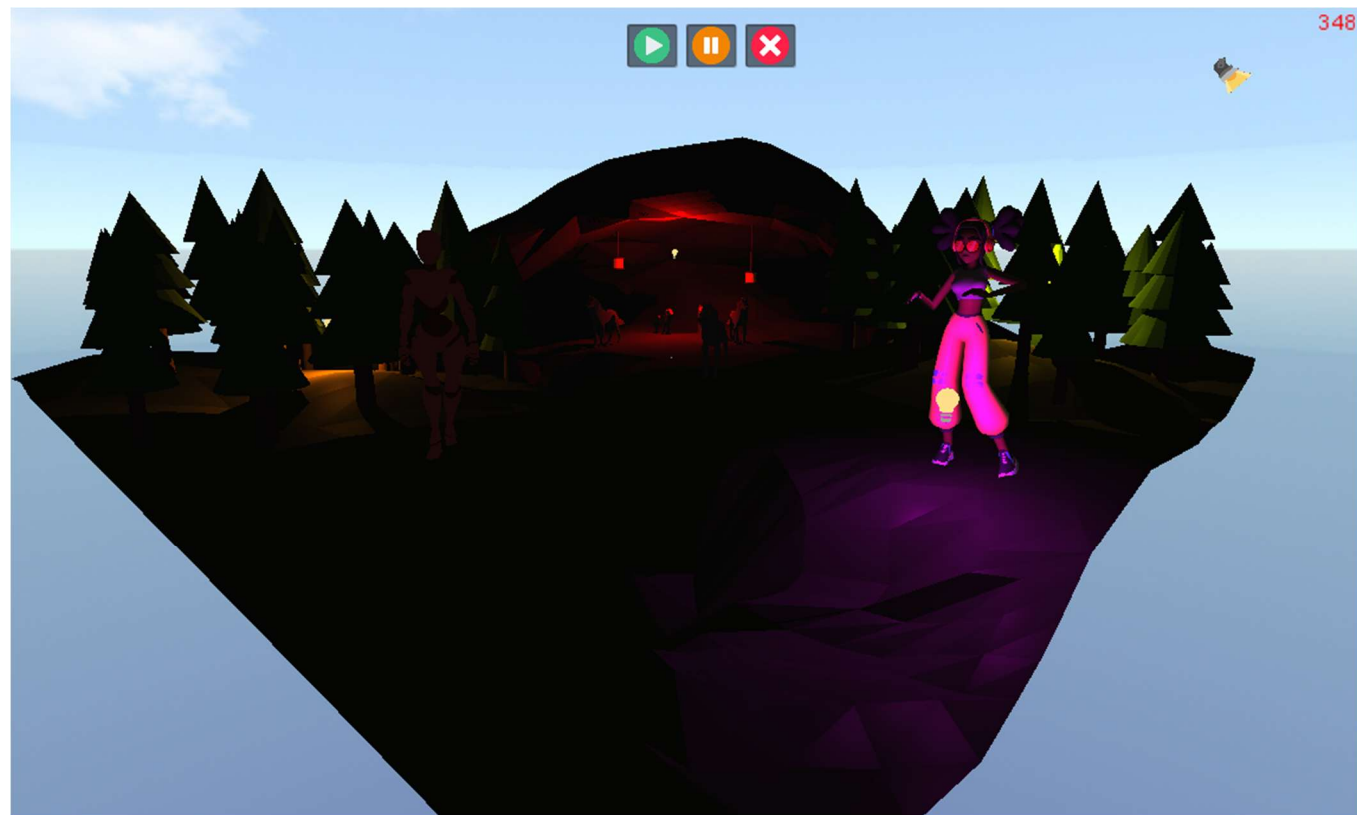
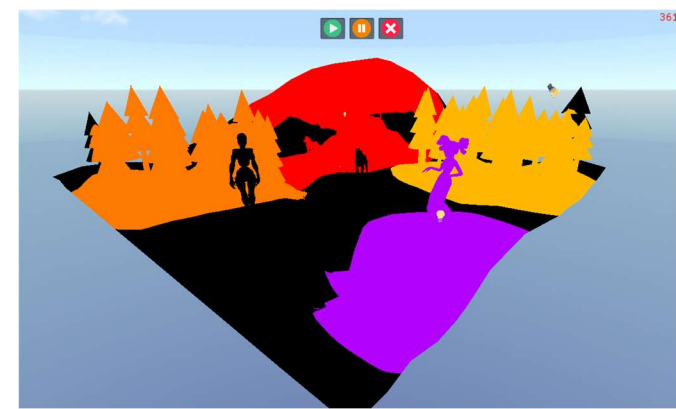
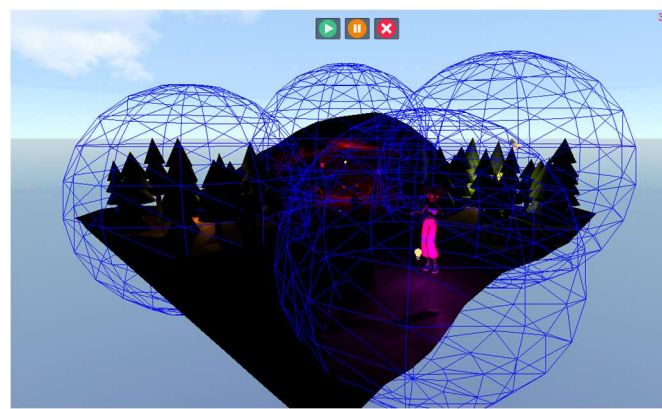
    uint finalIndex = baseOffset + localOffset;
    GET_POINT_VISIBLE_LIGHT(ctx.pointLightVisibleIndexBufferAddr, finalIndex) = collider.entityIndex;
}
```

<https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/Shaders/Passes/Culling/PointLight/PointLightCulling.comp>

Lighting and Shading XII.

Point Light Volume Simulation

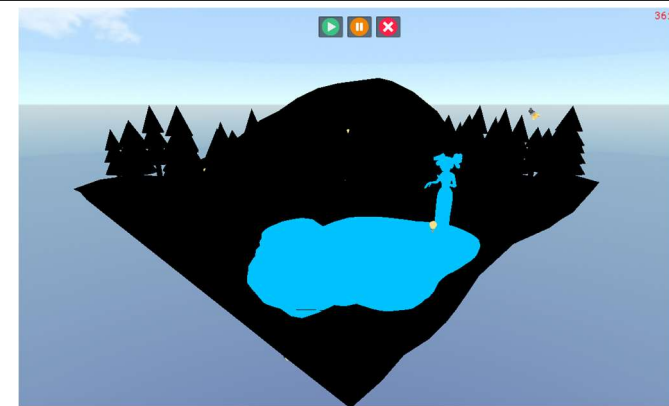
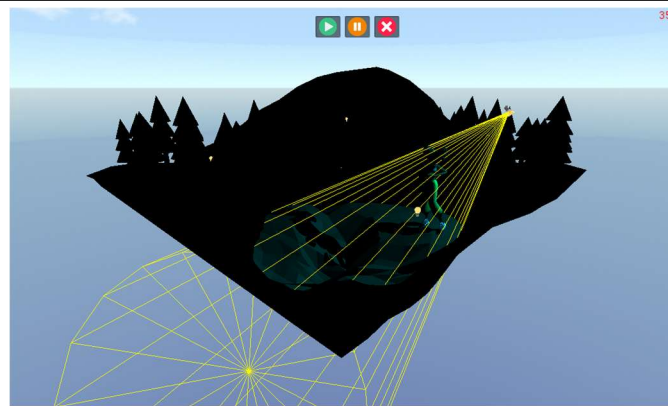
- ▶ Each rasterized fragment reconstructs its corresponding surface data from the **G-Buffer**.
- ▶ The Point Light position and radius are used to **calculate the distance and attenuation**.
- ▶ Fragments outside the analytical spherical influence range are **discarded**.
- ▶ The light direction is calculated from the shaded surface toward the light source.
- ▶ The final lighting contribution is evaluated using the material properties and **added to the lighting buffer**.



Lighting and Shading XIII.

Spot Light Volume Simulation

- ▶ Surface and material data are reconstructed from the **G-Buffer** in the same way as for Point Lights.
- ▶ The light direction is evaluated relative to the Spot Light position and orientation.
- ▶ The **spotlight cone angle** **determines** whether the fragment lies inside the illuminated region.
- ▶ **Angular falloff** is applied to create a smooth transition near the edge of the cone.
- ▶ The final lighting contribution **combines distance attenuation, angular attenuation**, and the material response.



Lighting and Shading XIV.

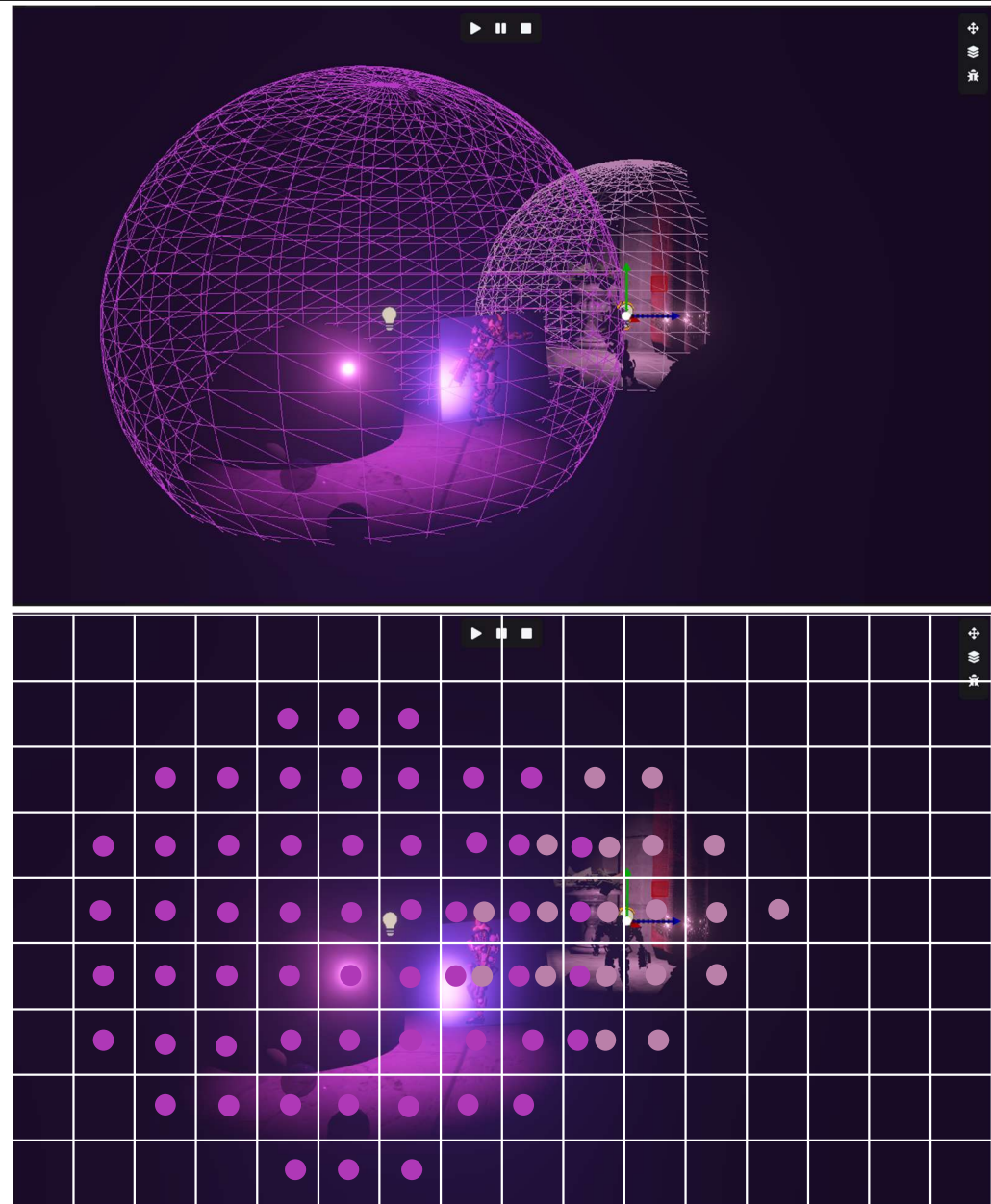
Deferred Shading – Limitations and the Path to Forward+

- ▶ **Deferred Shading solves the two major problems of traditional Forward Shading:** unnecessary lighting work caused by fragment overwrites and evaluating lights outside their actual influence volumes.
- ▶ Geometry is processed first, ensuring that lighting is evaluated only for the **final visible surfaces**.
- ▶ Local light volumes further restrict shading to pixels that can actually be affected by each light.
- ▶ However, every light volume still executes its own **fragment shader and repeatedly reads the same surface data from the G-Buffer**.
- ▶ If a single pixel is affected by **50 lights**, the G-Buffer data may be fetched and reconstructed up to **50 times**.
- ▶ This creates significant **texture bandwidth and memory access overhead**, especially with many overlapping lights.
- ▶ The lighting computation is distributed across separate light passes instead of processing all relevant lights together for a pixel.
- ▶ Forward+ addresses this by preserving efficient light culling while processing the G-Buffer/surface information only once per pixel.
- ▶ The goal is to combine the strengths of both approaches: **visibility-aware lighting, spatial light culling, and a single coherent shading pass**.

Lighting and Shading XV.

Tiled Forward Shading

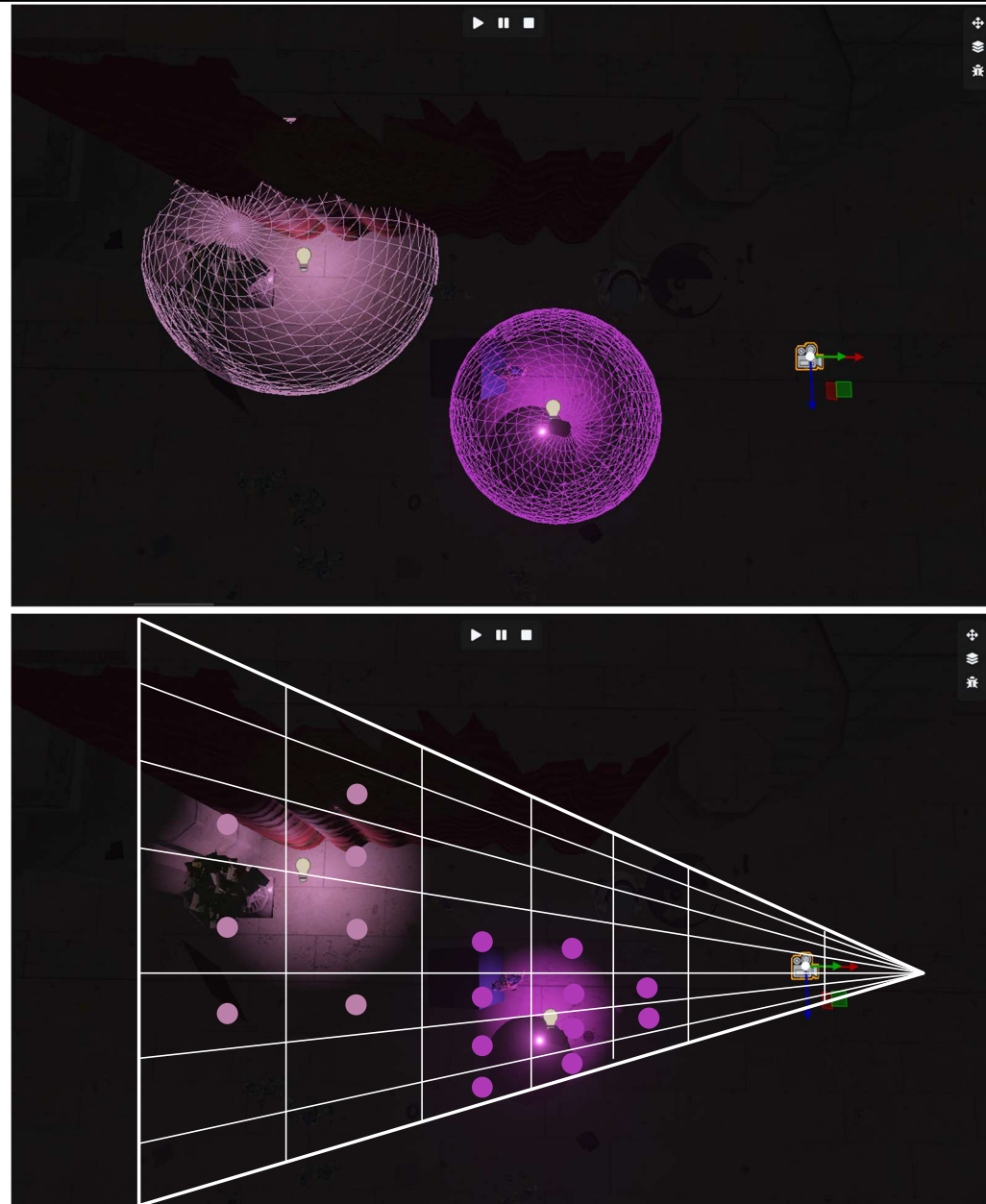
- ▶ The screen is divided into a **two-dimensional grid of fixed-size tiles**.
- ▶ For each tile, a **Compute Shader determines which lights potentially affect that screen-space region**.
- ▶ The resulting **light indices are stored in a compact list** and reused during Forward shading.
- ▶ Each pixel only **evaluates the lights assigned to its corresponding tile** instead of iterating over every light in the scene.
- ▶ This significantly reduces the number of unnecessary lighting calculations compared to traditional Forward Shading.
- ▶ However, the **partitioning exists only in screen-space X/Y dimensions**. All geometry at different depths within the same tile shares the same light list.
- ▶ As a result, lights may still be assigned to fragments that are spatially far away along the **Z axis**.



Lighting and Shading XVI.

Clustered Forward Shading

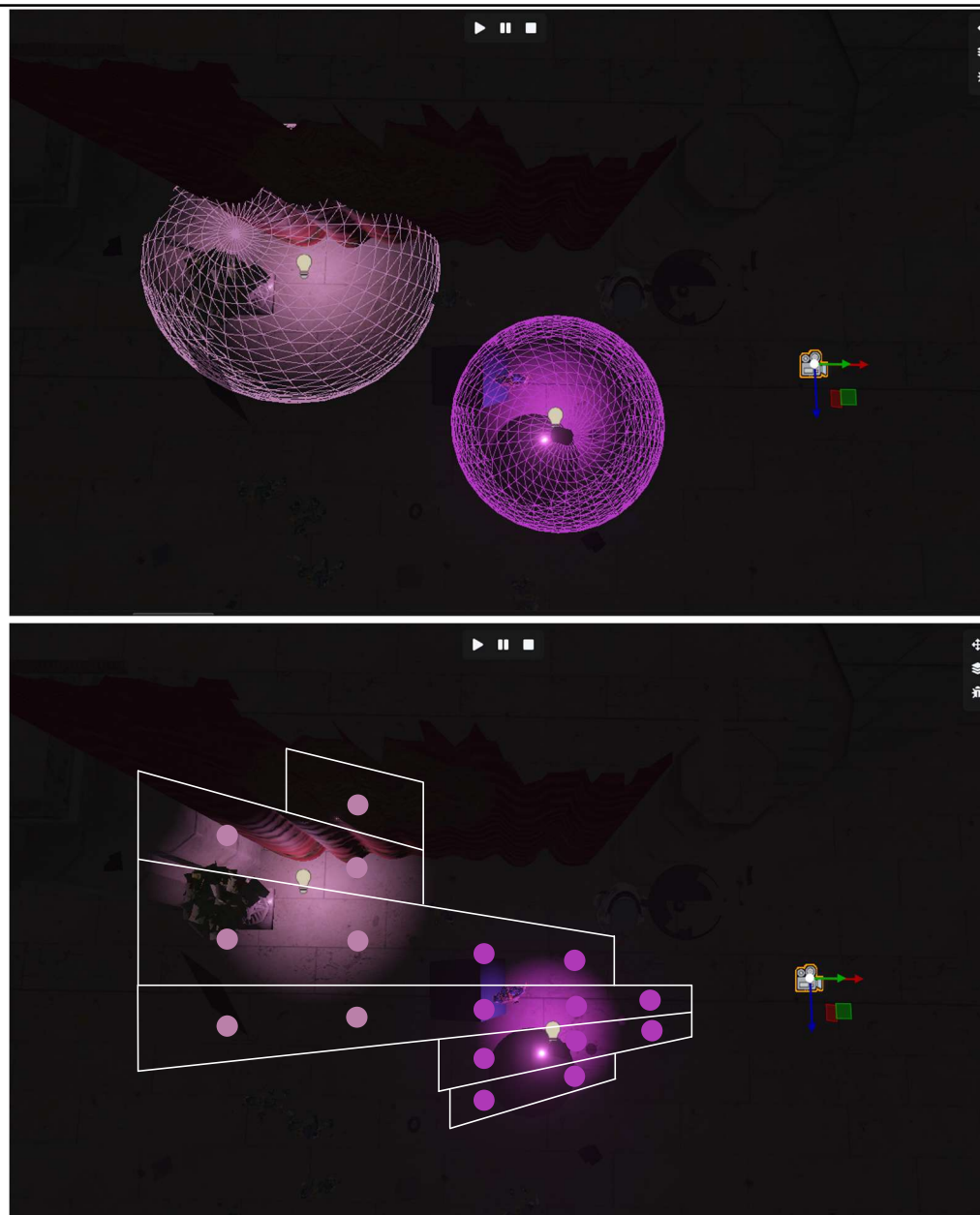
- ▶ Clustered Forward Shading extends tiled lighting into a **three-dimensional spatial hierarchy**.
- ▶ The screen is divided into X/Y tiles, while the view space is additionally partitioned along the **Z axis**.
- ▶ Each cluster therefore **represents a smaller three-dimensional region** instead of an entire depth range.
- ▶ The Z dimension is typically distributed non-linearly, allowing more detailed partitioning near the camera and larger cells at greater distances.
- ▶ **Lights are assigned only to the clusters intersecting** their spatial influence volumes.
- ▶ During shading, each fragment **determines its corresponding 3D cluster** and evaluates only the lights stored in that cluster.
- ▶ This **provides significantly better light rejection** than purely two-dimensional tiled approaches.
- ▶ However, the cluster grid is usually **predefined**, meaning that **many clusters may exist in empty regions containing no visible geometry**.



Lighting and Shading XVII.

Depth-Bounded Tiled Forward Shading

- ▶ Depth-Bounded Tiled Forward Shading improves tiled lighting by incorporating the **actual depth range of visible geometry**.
- ▶ Each screen-space **tile collaboratively reads its depth values in compute shader** and determines the local minimum and maximum depth.
- ▶ After synchronization, this depth range is used to construct a **tighter view-space volume** for the tile.
- ▶ Light culling is therefore performed **against the occupied depth range** instead of the entire camera frustum depth.
- ▶ This can significantly reduce false-positive light assignments compared to traditional Tiled Forward Shading.
- ▶ However, the approach still depends on a **single depth interval per tile**.
- ▶ A tile containing both **very near and very distant geometry can still produce a large depth range** containing significant empty space.
- ▶ Additionally, relying only on the opaque depth buffer creates complications when the same light-culling structure must also support **transparent geometry**.



Lighting and Shading XVIII.

Rethinking Forward+ Lighting

- ▶ While exploring existing Forward+ techniques, **none of the available approaches fully matched the requirements** of the engine.
- ▶ **Tiled Forward Shading** provides efficient screen-space partitioning but **completely ignores depth**.
- ▶ **Clustered Forward Shading** introduces three-dimensional light culling but relies on a predefined spatial grid that may contain **large numbers of empty clusters**.
- ▶ **Depth-Bounded Tiled Forward Shading** uses actual scene depth but represents each tile with only a **single depth range** and **does not naturally extend to transparent geometry**.
- ▶ Computing depth ranges directly from the depth buffer also becomes increasingly expensive as the tile resolution grows.

- ▶ A larger tile size, such as **64×64 or 256×256 pixels**, would require significantly more depth samples and synchronization if the **Min/Max range were computed directly by the workgroup**.
- ▶ The key idea was therefore to combine the strengths of existing approaches while removing their fixed spatial assumptions.

The resulting design goals

- ▶ Use **hierarchical depth information** instead of repeatedly reducing raw depth samples.
- ▶ Adapt the **depth partitioning** dynamically to the occupied depth range of each tile.
- ▶ **Avoid generating clusters for empty regions** of the camera frustum.
- ▶ **Support both opaque and transparent geometry** within the same light-culling structure.
- ▶ Allow **flexible screen-space tile resolutions** without scaling the depth-range reduction cost with the number of pixels inside the tile.
- ▶ **The result is a new Forward+ approach designed specifically around the existing GPU-driven architecture of the engine**

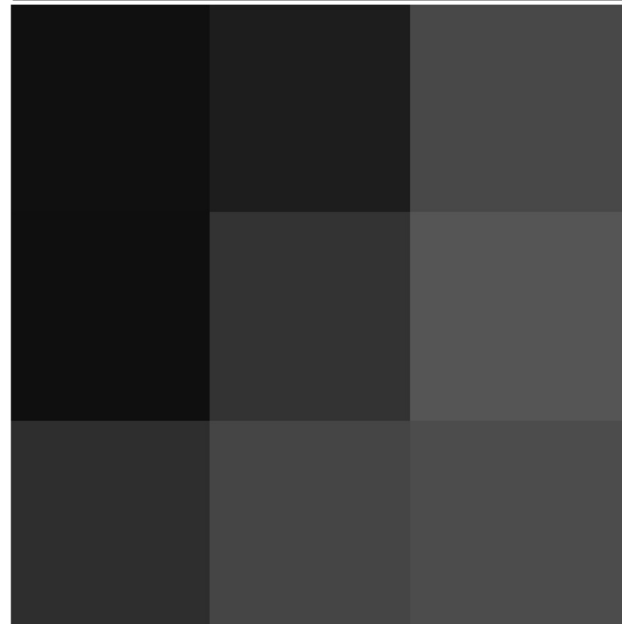
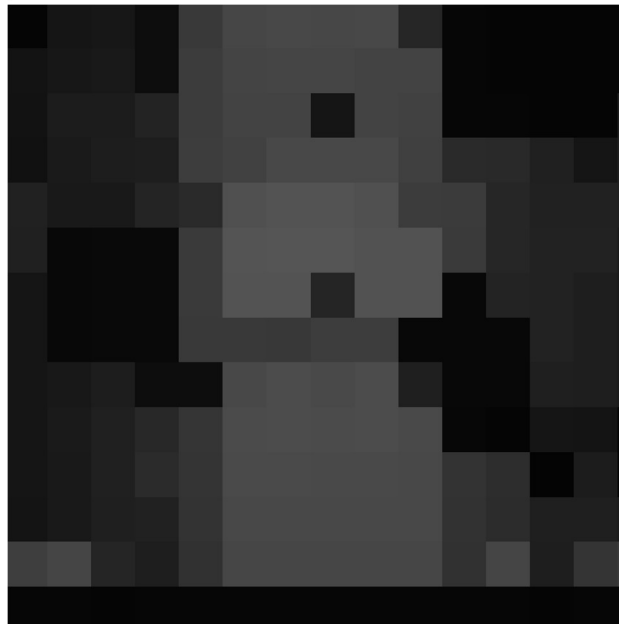
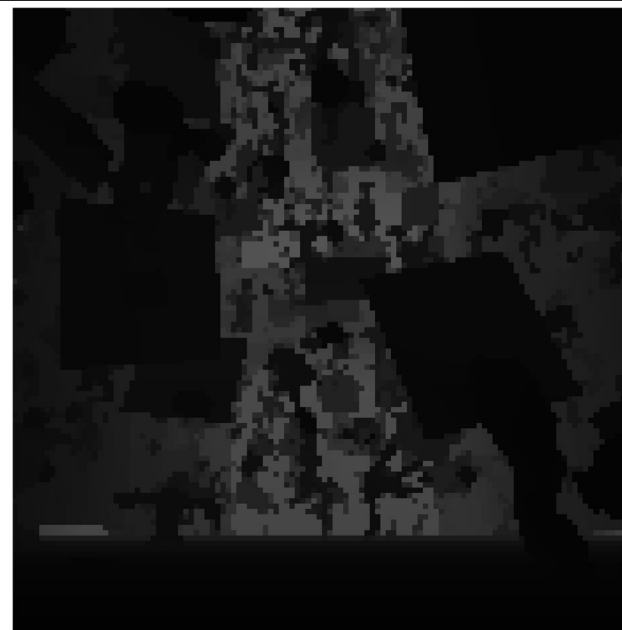
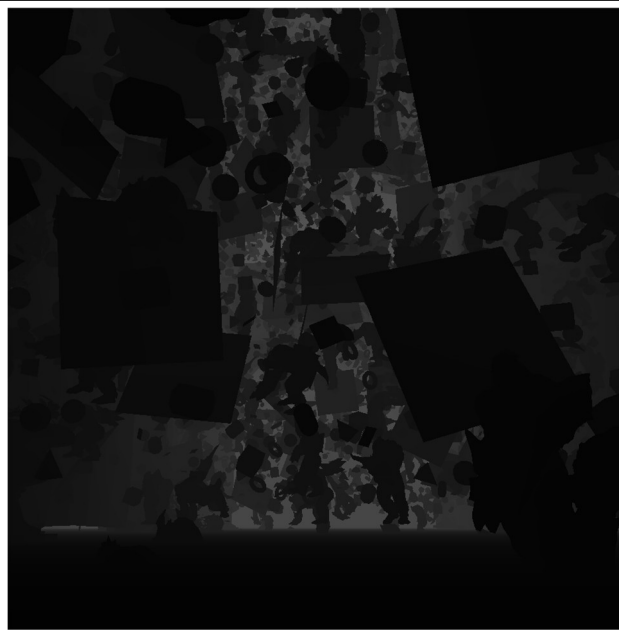
Dual-HiZ Adaptive Clustered Forward Shading

An Original Rendering Technique by Péter Tamás

Lighting and Shading XIX.

The Origin of Dual-Hiz

- ▶ The key idea originated from two systems that were already present in the engine.
- ▶ The first is the **Max-Reduction Hi-Z Depth Pyramid**, originally built from the opaque depth buffer for GPU-driven occlusion culling.
- ▶ This hierarchy already stores depth information from full screen resolution down to a **1×1 mip level**.
- ▶ Therefore, the **maximum depth of an arbitrary screen-space region** can be obtained directly from the appropriate **Hi-Z mip level**.
- ▶ Unlike traditional **Depth-Bounded Tiled Forward Shading**, no Compute Shader reduction over every pixel of a **fixed 16×16 tile** is required.
- ▶ The tile resolution is therefore no longer fixed: larger regions such as **32×32, 64×64, or even larger tiles can be queried** with the same hierarchical approach.

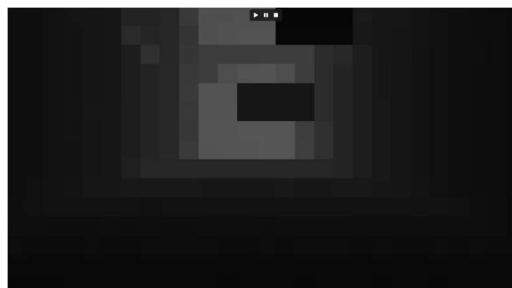


Lighting and Shading XX.

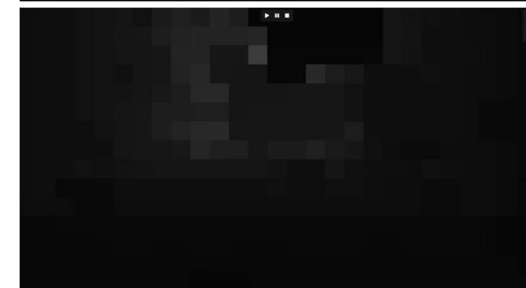
Form Depth Buffers to Dual-Hiz

- ▶ A second depth buffer already exists in the engine for **pixel-accurate transparent object picking**.
- ▶ It starts as a copy of the opaque depth buffer, after which **transparent geometry is rendered into it as if it were opaque**.
- ▶ This ensures that the **nearest transparent surface is preserved** while retaining the depth information of opaque geometry.
- ▶ Building a **Min-Reduction Hi-Z pyramid** from this depth representation provides the opposite depth boundary of the occupied screen-space region.
- ▶ Together, the **Max-Reduction Opaque Hi-Z** and **Min-Reduction Opaque + Transparent Hi-Z** define a **depth interval containing the relevant scene geometry**.
- ▶ This information is obtained through hierarchical texture sampling and works for **both opaque and transparent objects**.
- ▶ **Two Hi-Z hierarchies provide the depth boundaries required to construct a cluster** without reducing individual depth samples inside a fixed Compute Shader tile.

Opaque Max
HiZ



Opaque +
Transparent
Min HiZ



Mip0:
1x1 Tile

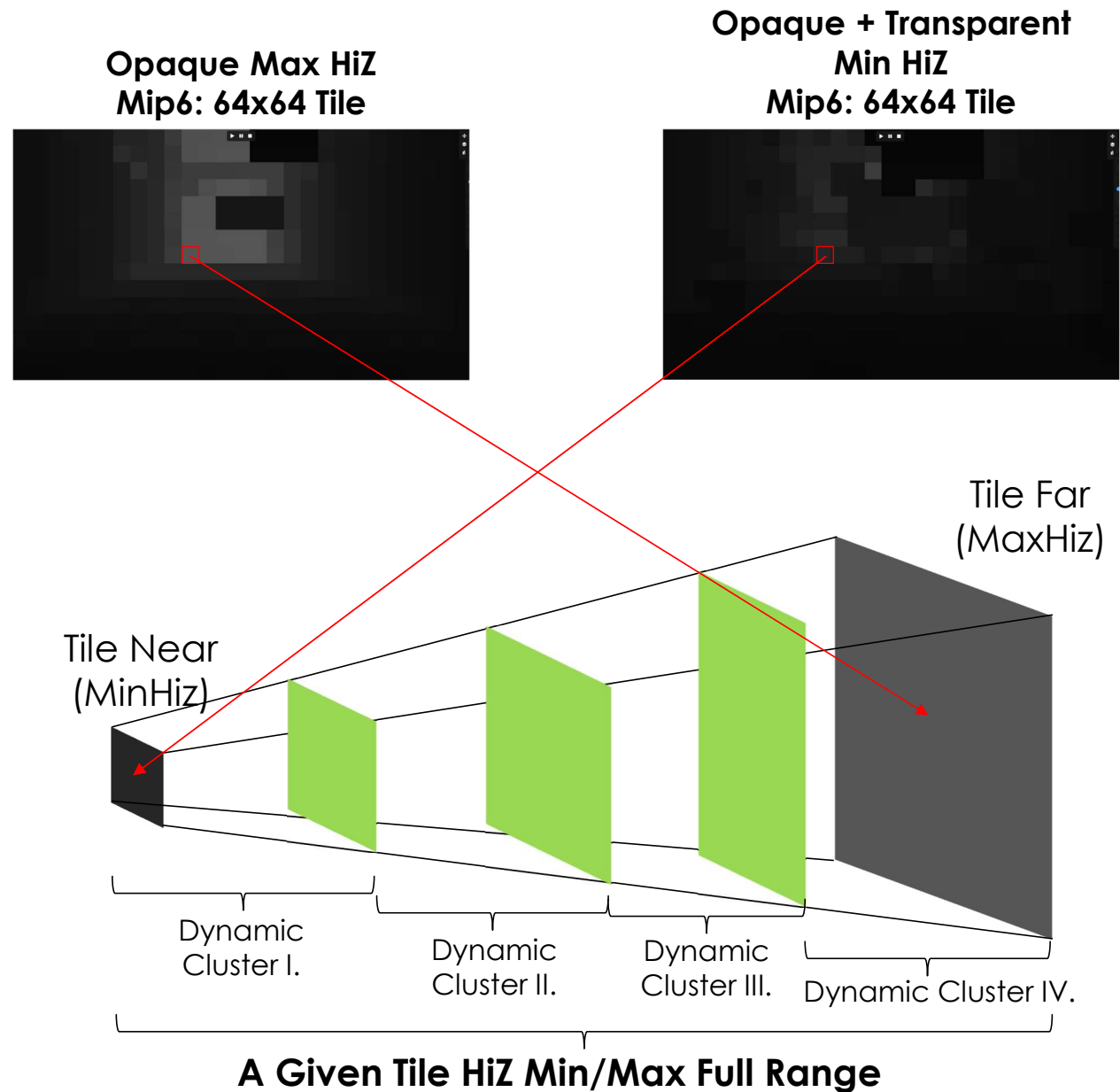
Mip4:
16x16 Tile

Mip6:
64x64 Tile

Lighting and Shading XXI.

Dual-HiZ Adaptive Clustering

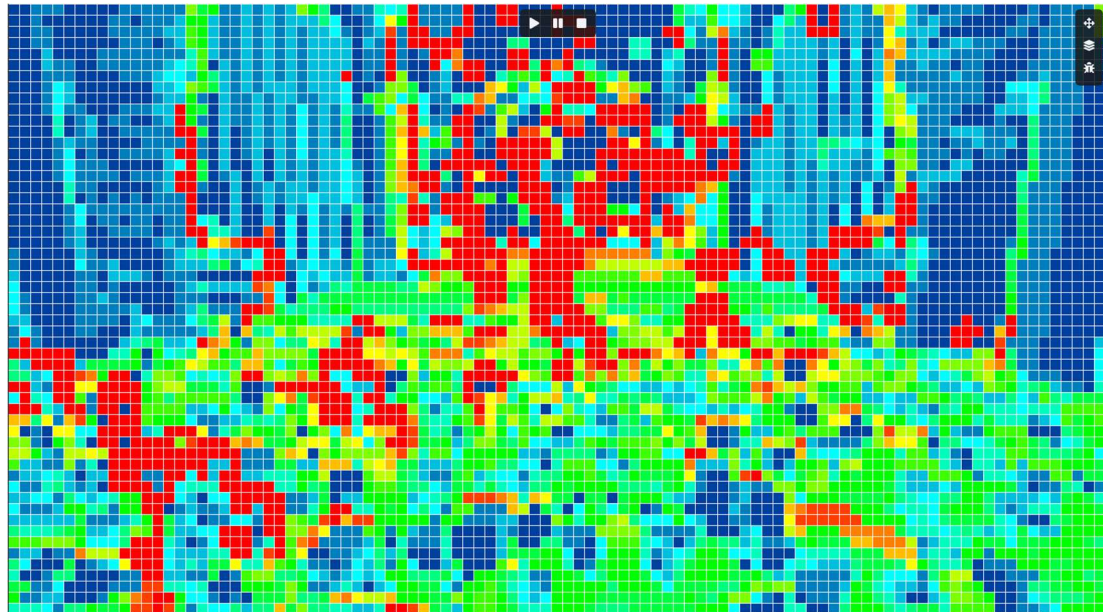
- ▶ The two Hi-Z hierarchies provide the depth boundaries of each screen-space tile using only hierarchical texture sampling.
- ▶ The **Max-Reduction Opaque Hi-Z** provides one boundary of the occupied depth range.
- ▶ The **Min-Reduction Opaque + Transparent Hi-Z** provides the opposite boundary while also accounting for transparent geometry.
- ▶ The resulting depth range can then be **partitioned dynamically instead of using a globally predefined 3D cluster grid**.
- ▶ Each tile is subdivided along the Z axis using a **logarithmic depth distribution**.
- ▶ The number and extent of the resulting clusters are determined by the depth range represented by that tile, **up to a configurable maximum such as 16 clusters**.
- ▶ **Empty regions** outside the Dual-HiZ depth range **do not need to be represented** by physical clusters.
- ▶ The resulting clusters are then used for Point Light and Spot Light culling, producing compact per-cluster light index ranges.



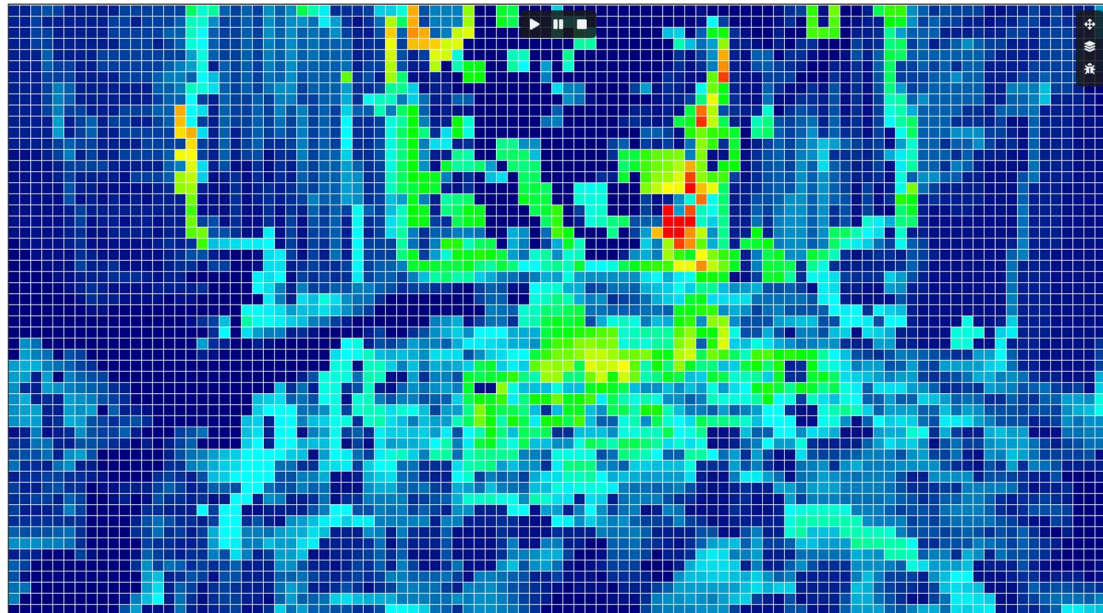
Lighting and Shading XXII.

Cluster Light Assignment and Shading

- ▶ Each adaptive cluster is represented by its own **Sphere and AABB collider**, allowing efficient intersection tests against Point Lights and Spot Lights.
- ▶ During the light culling stage, every cluster is tested against the potentially visible lights.
- ▶ Each cluster owns a dedicated **Point Light and Spot Light index range**, where the indices of intersecting lights are stored.
- ▶ This creates a **compact light list containing only the lights** that can affect the corresponding cluster.
- ▶ During Forward+ shading, the fragment's **screen-space XY position identifies its tile**, while its **depth determines the corresponding adaptive depth cluster**.
- ▶ **Once the cluster is reconstructed, the shader retrieves its light index range and iterates only over the lights assigned to that cluster.**



Cluster
Adaptive
Chunk
Count
Visualization

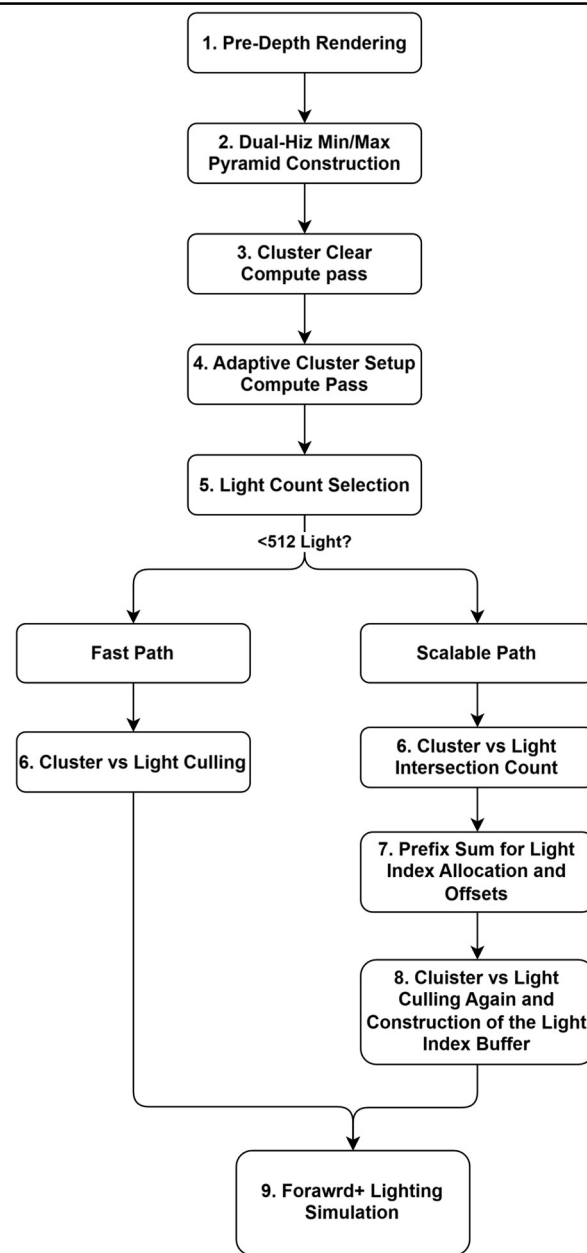


Cluster
Light Count
Visualization

Lighting and Shading XXIII.

Dual-Hiz Adaptive Clustered Forward Shading – Implementation Pipeline

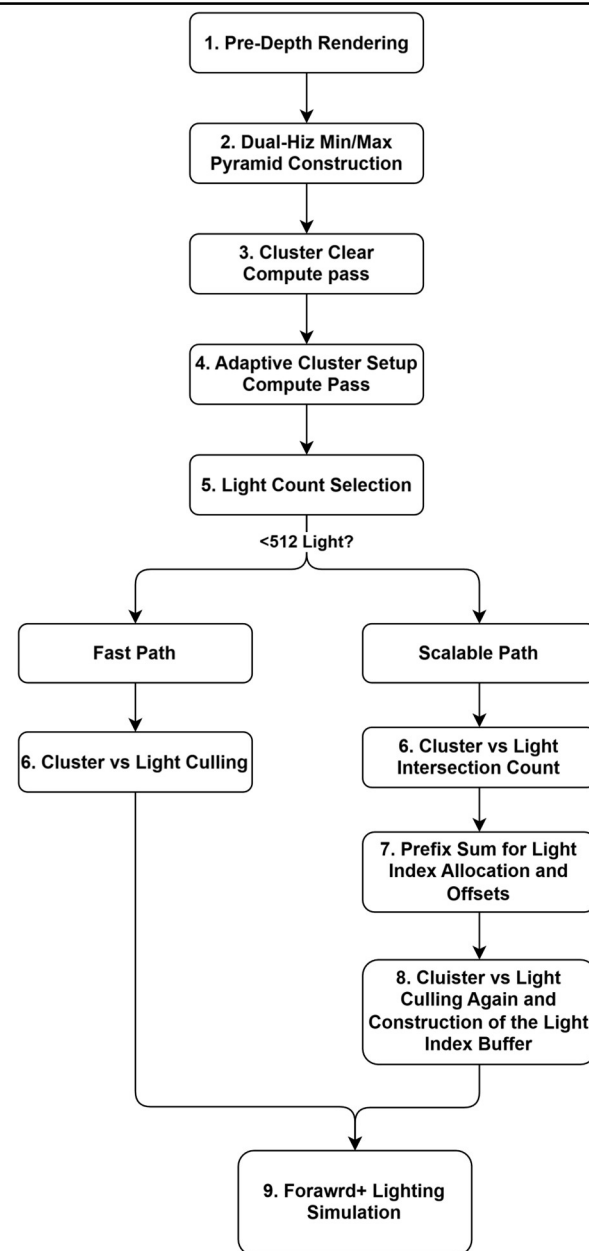
- ▶ **1. Pre-Depth Rendering:** Opaque geometry is rendered first to generate the primary scene depth buffer. Transparent geometry is then rendered into the secondary depth representation used by the Dual-HiZ system.
- ▶ **2. Dual-HiZ Min/Max Pyramid Construction:** The two depth representations are converted into hierarchical Min/Max depth pyramids, providing the occupied depth range for arbitrary screen-space tile sizes.
- ▶ **3. Cluster Buffer Clear:** All cluster counters, index ranges, and temporary GPU buffers are reset before the adaptive cluster generation begins.



Lighting and Shading XXIV.

Dual-Hiz Adaptive Clustered Forward Shading – Implementation Pipeline

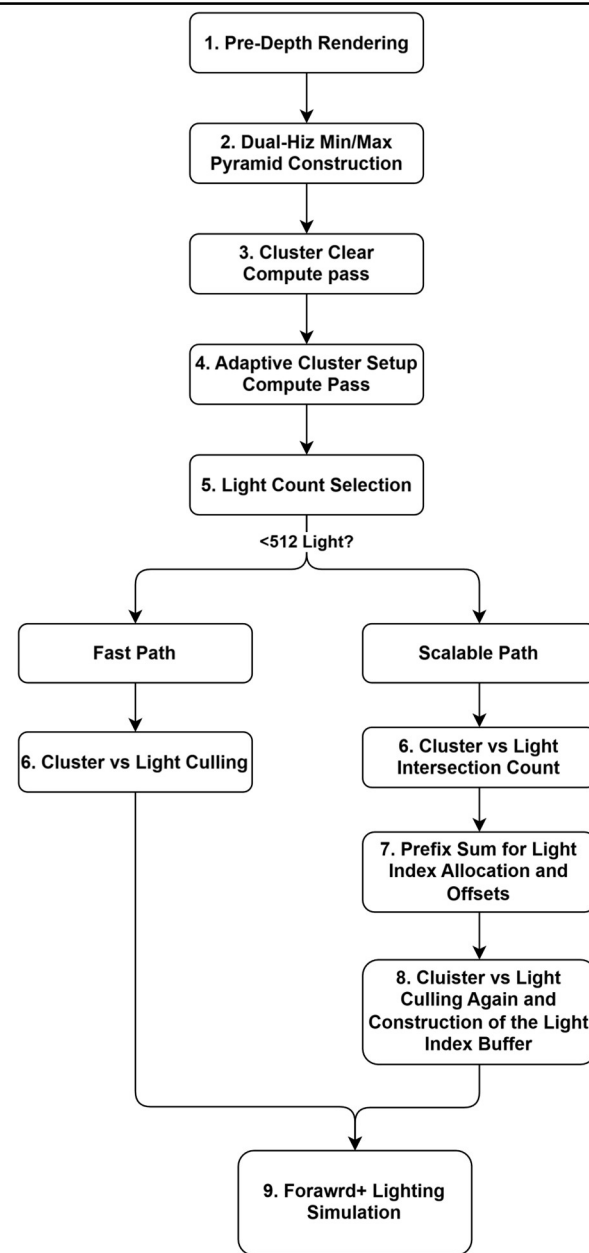
- ▶ **4. Adaptive Cluster Setup:** Each screen-space tile reads its Dual-HiZ depth range and dynamically partitions the occupied depth interval into clusters. Sphere and AABB colliders are generated for every cluster.
- ▶ **5. Light Count Selection:** The implementation selects between two GPU execution paths depending on the number of potentially visible local lights:
 - ▶ **Fast Path:** fewer than **512 lights**, local shared-memory allocation.
 - ▶ **Scalable Path:** more than **512 lights**, counting and prefix-sum allocation.
- ▶ **6. Fast Path - Cluster vs. Light Culling:** Clusters are tested against Point Lights and Spot Lights, and the resulting light indices are directly collected into their corresponding cluster ranges.



Lighting and Shading XXV.

Dual-Hiz Adaptive Clustered Forward Shading – Implementation Pipeline

- ▶ **6. Scalable Path - Light Intersection Count:** For larger light counts, the first pass only counts how many Point Lights and Spot Lights intersect each cluster.
- ▶ **7. Prefix Sum - Light Index Allocation:** A prefix sum converts the per-cluster light counts into compact offsets and index ranges within the global light index buffer.
- ▶ **8. Scalable Path - Light Index Buffer Construction:** The cluster-versus-light intersection tests are executed again, this time writing the intersecting Point Light and Spot Light indices into their preallocated ranges.
- ▶ **9. Forward+ Lighting:** During shading, each fragment reconstructs its **Tile** → **Adaptive Cluster**, retrieves the corresponding light index range, and evaluates only the lights assigned to that cluster.



Lighting and Shading XXVI.

Dual-Hiz Depth Range and Adaptive Cluster Count

- ▶ Each screen-space tile is processed by a **16-thread compute workgroup**.
- ▶ The corresponding **Dual-HiZ texel** is read to obtain the occupied minimum and maximum depth range.
- ▶ **Only Thread 0 performs the Hi-Z lookup**, after which the result is shared with the entire workgroup.
- ▶ Tiles containing only **background pixels are immediately discarded** without generating any clusters.
- ▶ The occupied depth range defines the actual spatial region that must be partitioned.
- ▶ Instead of using a fixed number of depth slices, the **number of clusters is calculated dynamically**.
- ▶ Larger occupied depth ranges can generate more clusters, while small ranges require fewer.
- ▶ This allows the cluster structure to adapt to the actual depth distribution of each screen-space tile.

```
layout(local_size_x = 16, local_size_y = 1, local_size_z = 1) in;

void main()
{
    uint threadId = gl_LocalInvocationID.x;
    ivec2 tileId = ivec2(gl_WorkGroupID.xy);
    uint flattenTileIdx = tileId.y * ctx.tileCountX + tileId.x;

    if (threadId == 0) {

        ivec2 mipSize = textureSize(hizDepth, int(ctx.hizMipLevel));
        ivec2 texelCoord = min(tileId, mipSize - 1);

        vec2 hizValue =
            texelFetch(
                hizDepth,
                texelCoord,
                int(ctx.hizMipLevel)
            ).xy;

        float zMax = near + hizValue.x * (far - near);

        float zMin = near + hizValue.y * (far - near);

        // Background -> Skip
        if (hizValue.y >= 0.9999) {
            s_sliceCount = 0;
        } else {
            s_minZ = max(zMin, near);
            s_maxZ = min(zMax, far);

            s_sliceCount =
                CalculateClusterSliceCount(
                    s_minZ,
                    s_maxZ,
                    ctx.sliceScaleFactor
                );
        }
    }

    //...
}
```

<https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/Shaders/Passes/Shading/ForwardPlus/Clustering/ClusterSetup.comp>

Lighting and Shading XXVII.

Dynamic Cluster Allocation and Logarithmic Slicing

- ▶ Each active thread generates one depth cluster, resulting in **up to 16 dynamically generated clusters per tile**.
- ▶ The occupied depth range is partitioned **logarithmically** between the minimum and maximum depth.
- ▶ This provides finer depth resolution near the camera and larger depth intervals farther away.
- ▶ Because each tile can generate a different number of clusters, the global cluster buffer cannot use a fixed layout.
- ▶ **Subgroup operations** first determine how many threads are actively generating clusters.
- ▶ A single **atomic allocation** reserves one contiguous region in the global cluster buffer. Each active thread receives a local offset within this allocated region.
- ▶ The final **base offset + slice count** allows **every tile to efficiently reference** its dynamically generated cluster range.

```
void main()
{
    //...

    barrier();

    uint needsWrite = (threadId < s_sliceCount) ? 1 : 0;
    uint subgroupWriteCount = subgroupAdd(needsWrite);
    uint localOffset = subgroupExclusiveAdd(needsWrite);
    uint baseOffset = 0;

    if (subgroupElect() && subgroupWriteCount > 0) {
        baseOffset = atomicAdd(GET_CLUSTER_COUNT(ctx.forwardPlusClusterCountBufferAddr), subgroupWriteCount);
    }

    baseOffset = subgroupBroadcastFirst(baseOffset);

    if (threadId == 0) {
        TileData tileData;
        tileData.minZ = s_minZ;
        tileData.maxZ = s_maxZ;
        tileData.sliceCount = s_sliceCount;
        tileData.clusterBaseOffset = baseOffset;
        GET_TILE_DATA(ctx.forwardPlusTileGridListBufferAddr, flattenTileIdx) = tileData;
    }

    if (needsWrite == 1) {
        float sliceMinZ, sliceMaxZ;

        if (s_sliceCount == 1) {
            sliceMinZ = s_minZ;
            sliceMaxZ = s_maxZ;
        }
        else {
            float ratio = s_maxZ / s_minZ;
            sliceMinZ = s_minZ * pow(ratio, float(threadId) / float(s_sliceCount));
            sliceMaxZ = s_minZ * pow(ratio, float(threadId + 1) / float(s_sliceCount));
        }

        uint finalIndex = baseOffset + localOffset;

        ClusterData clusterData;
        clusterData.tileIndex = flattenTileIdx;
        clusterData.sliceIndex = threadId;
        clusterData.minZ = sliceMinZ;
        clusterData.maxZ = sliceMaxZ;
        clusterData.pointLightCount = 0;
        clusterData.pointLightOffset = 0;
        clusterData.spotLightCount = 0;
        clusterData.spotLightOffset = 0;

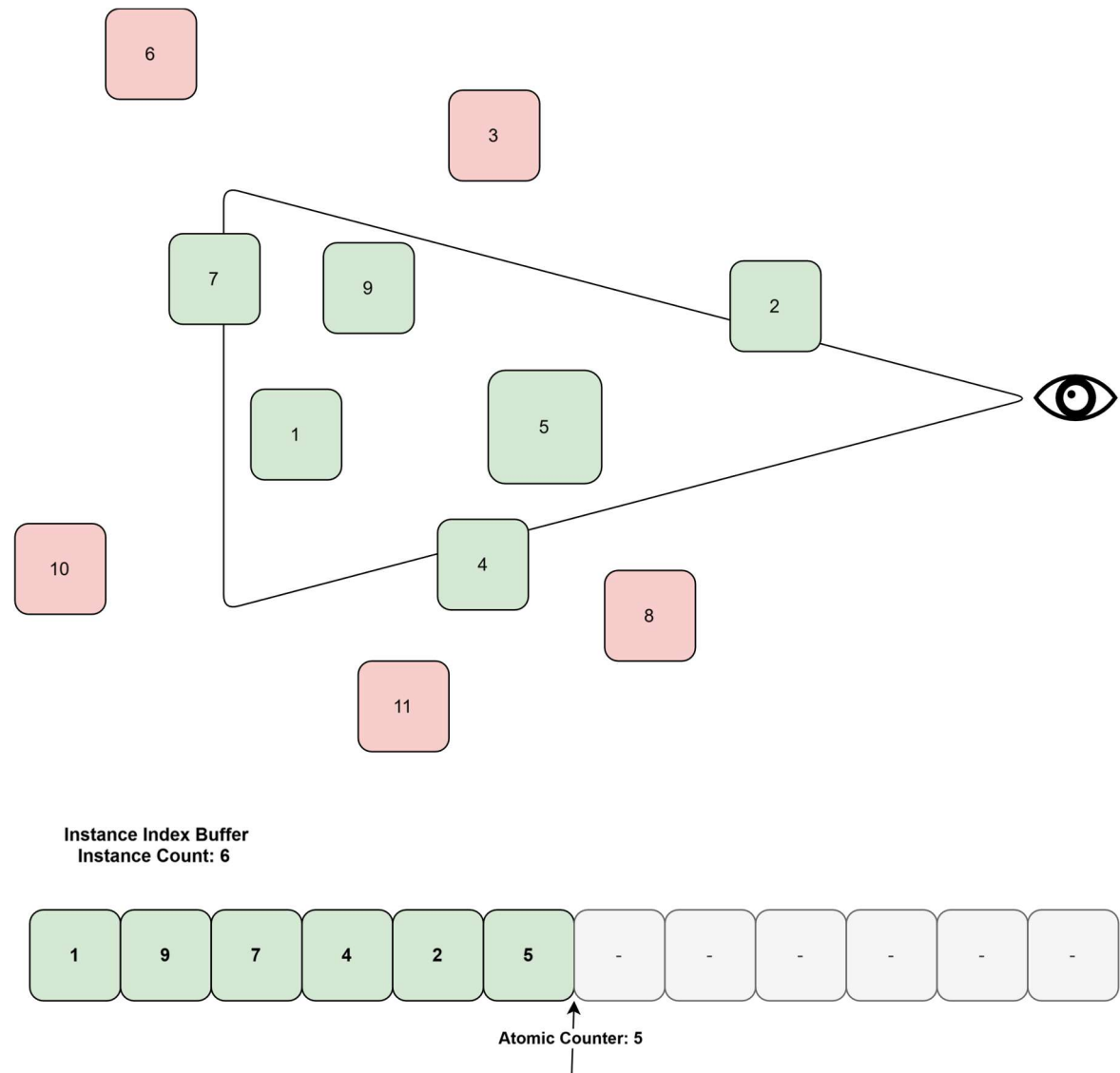
        GET_CLUSTER_DATA(ctx.forwardPlusClusterListBufferAddr, finalIndex) = clusterData;
    }
}
```

<https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/Shaders/Passes/Shading/ForwardPlus/Clustering/ClusterSetup.comp>

Lighting and Shading XXVIII.

Hierarchical Light Culling

- ▶ The GPU-driven light culling system introduced for Deferred Shading is reused by the Forward+ pipeline.
- ▶ Point Lights and Spot Lights are first tested against the **camera frustum**.
- ▶ Lights **outside the visible camera region are discarded** before any Forward+ cluster processing begins.
- ▶ The remaining visible light indices are already stored in compact GPU buffers.
- ▶ Forward+ cluster culling therefore does not iterate over every light in the scene. Instead, each **cluster is tested only against the previously visible lights**.
- ▶ This creates a hierarchical two-stage light culling pipeline:
Scene Lights → **Camera-Visible Lights** → **Cluster-Visible Lights**
- ▶ The same GPU-driven infrastructure, indirect commands, and compact visible-light index buffers can therefore be reused directly.



<https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/Shader/Passes/Culling/PointLight/PointLightCulling.comp>

Lighting and Shading XXIX.

Fast Point Light Cluster Culling

- ▶ Each compute workgroup processes one active cluster **collaboratively using 32 parallel threads**.
- ▶ The **cluster's view-space AABB is reconstructed once** and shared by the entire workgroup.
- ▶ Only the previously visible Point Lights are processed, reducing the number of light-cluster intersection tests.
- ▶ Each thread iterates over Point Lights and performs a **Sphere vs AABB intersection test**.
- ▶ **Subgroup ballot operations** identify all intersecting lights within the current iteration.
- ▶ A single subgroup atomic operation reserves local positions for the intersecting lights.
- ▶ The Point Light entity indices are written into a fixed cluster-specific index region.
- ▶ This fast path supports up to **512 Point Lights per cluster**, avoiding the additional prefix-sum allocation pipeline.

```
void main()
{
    //...

    uint totalLights = GET_VK_DRAW_CMD(ctx.pointLightIndirectCommandBufferAddr, 0).instanceCount;
    uint globalOffset = clusterIdx * MAX_LIGHTS_PER_CLUSTER;

    for (uint i = threadId; i < totalLights; i += gl_WorkGroupSize.x) {
        uint entityIdx = GET_POINT_VISIBLE_LIGHT(ctx.pointLightVisibleIndexBufferAddr, i);
        uint denseIdx = GET_SPARSE_INDEX(ctx.pointLightSparseMapBufferAddr, entityIdx);
        PointLightColliderGPU collider = GET_POINT_LIGHT_COLLIDER(ctx.pointLightColliderBufferAddr, denseIdx);

        vec3 viewSpaceCenter = (camera.view * vec4(collider.center, 1.0)).xyz;
        bool isIntersecting = TestSphereAABB(viewSpaceCenter, collider.radius, s_clusterAABBMin, s_clusterAABBMax);

        uvec4 ballot = subgroupBallot(isIntersecting);
        uint activeCount = subgroupBallotBitCount(ballot);

        uint baseOffset = 0;
        if (subgroupElect() && activeCount > 0) {
            baseOffset = atomicAdd(s_localPointLightCount, activeCount);
        }
        baseOffset = subgroupBroadcastFirst(baseOffset);

        uint isIntersectingInt = isIntersecting ? 1 : 0;
        uint threadOffset = subgroupExclusiveAdd(isIntersectingInt);

        if (isIntersecting) {
            uint localIdx = baseOffset + threadOffset;

            if (localIdx < MAX_LIGHTS_PER_CLUSTER) {
                uint finalIdx = globalOffset + localIdx;
                GET_LIGHT_INDEX(ctx.forwardPlusPointLightIndexListBufferAddr, finalIdx) = entityIdx;
            }
        }
    }

    //...
}
```

<https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/Shaders/Passes/Shading/ForwardPlus/Clustering/ClusterPointLightSingle.comp>

Lighting and Shading XXX.

Fast Spot Light Cluster Culling

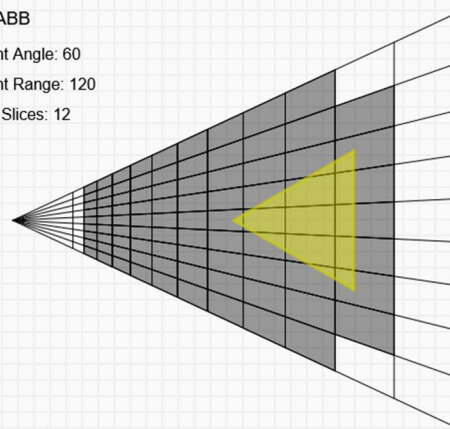
- ▶ The **same hierarchical GPU-driven approach** used for Point Lights is applied to Spot Lights.
- ▶ However, **the cluster intersection test requires a different strategy due to the conical shape of the Spot Light volume.**
- ▶ Testing a conservative Spot Light **bounding sphere or AABB against the cluster AABB** would produce a large number of **false-positive** intersections.
- ▶ Many clusters could intersect the bounding volume while lying completely outside the actual light cone.
- ▶ Instead, the representation is reversed: the **Spot Light cone is used directly** for the intersection test. And each cluster is approximated by a **conservative bounding sphere**.
- ▶ A **Cone vs Sphere intersection test** therefore follows the actual Spot Light volume much more closely.
- ▶ This **significantly reduces the number of clusters assigned** to each Spot Light and avoids unnecessary lighting calculations during Forward+ shading.

Method: Simple Sphere vs AABB

Spotlight Angle: 60
Spotlight Range: 120
Cluster Slices: 12

☒ Free light
☐ Debug View

Mouse wheel to rotate light

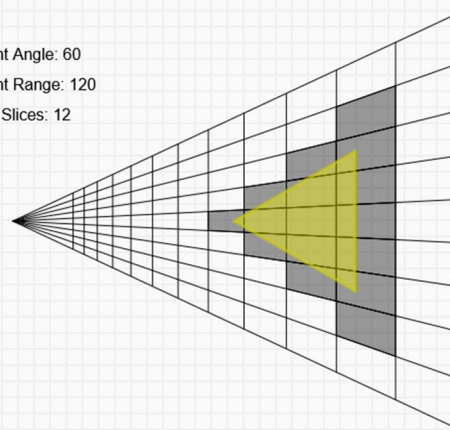


Method: Cone vs Sphere

Spotlight Angle: 60
Spotlight Range: 120
Cluster Slices: 12

☒ Free light
☐ Debug View

Mouse wheel to rotate light



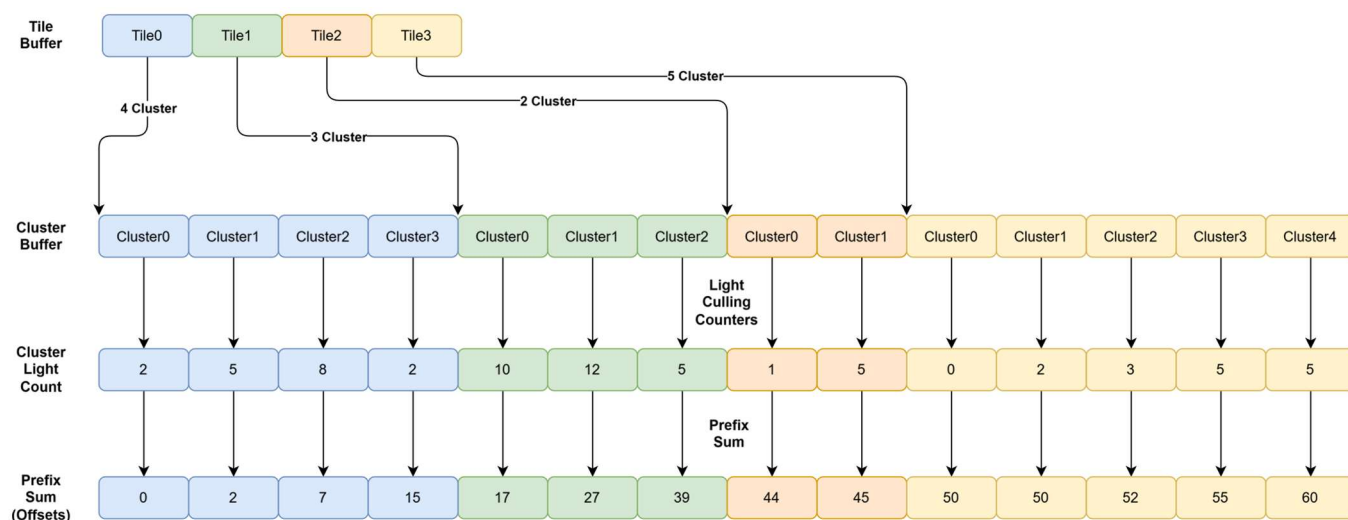
```
bool SpotlightVsAABB(Spotlight spotlight, AABB aabb)
{
    float sphereRadius = dot(aabb.extents, aabb.extents);
    float3 v = aabb.venter - spotlight.position;
    float lenSq = dot(v, v);
    float v1len = dot(v, direction);
    float distanceClosestPoint = cos(spotlight.angle) * sqrt(lenSq - v1len * v1len) - v1len * sin(spotlight.angle);
    bool angleCull = distanceClosestPoint > sphereRadius;
    bool frontCull = v1len > sphereRadius + spotlight.range;
    bool backCull = v1len < -sphereRadius;
    return !(angleCull || frontCull || backCull);
}
```

<https://simoncoenen.com/blog/programming/graphics/SpotlightCulling>

Lighting and Shading XXI.

Scalable Cluster Light Allocation

- ▶ The fast path uses a fixed-size light index region for every cluster, which is efficient when the number of lights is limited.
- ▶ With **thousands of lights, preallocating a fixed maximum number of indices for every possible cluster** would cause the light index buffer size to grow excessively.
- ▶ **The worst-case memory requirement** depends on the number of screen-space tiles, the maximum number of depth clusters per tile, and the maximum number of potentially intersecting lights.
- ▶ Instead, the scalable path allocates the light index buffer dynamically using a **count** → **prefix sum** → **write** pipeline.
- ▶ **First, the same Point Light and Spot Light intersection tests** are performed, but only the number of intersecting lights is stored for each cluster.
- ▶ A **parallel prefix sum** is then executed over all cluster light counts.
- ▶ The resulting **prefix value defines the exact offset** of each cluster inside the global light index buffer.
- ▶ A final culling pass runs again and writes the intersecting light indices directly into their compactly allocated ranges.



<https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/Shaders/Passes/Shading/ForwardPlus/Clustering/ClusterPointLightCount.comp>

Lighting and Shading XXXII.

Subgroup-Optimized Parallel Prefix Sum

- ▶ A **parallel prefix sum** is used to convert the per-cluster Point Light and Spot Light counts into compact global buffer offsets.
- ▶ Although algorithms such as the **Blelloch scan** could be used, this implementation takes advantage of modern GPU **subgroup operations**.
- ▶ Each compute workgroup processes up to **512 clusters** at a time.
- ▶ **Hardware subgroup operations perform the exclusive prefix** sum independently inside each subgroup. The total count of each subgroup is stored in shared memory.
- ▶ A lightweight scan combines the subgroup totals to determine the offset of every subgroup within the 512-element block.
- ▶ The final cluster offset is obtained by combining the global block carry, the subgroup offset, and the local subgroup prefix sum.
- ▶ The accumulated block total becomes the **carry value for the next 512-cluster block**.

```
// Single workgroup loops over all clusters. Size 512 is a safe maximum for all GPU vendors.
layout(local_size_x = 512, local_size_y = 1, local_size_z = 1) in;

void main() {
    //...

    // Loop through all active clusters in blocks of gl_WorkGroupSize.x (512)
    for (uint baseIdx = 0; baseIdx < totalClusters; baseIdx += gl_WorkGroupSize.x)
    {
        uint globalIdx = baseIdx + threadId;
        uint pCount = 0;
        uint sCount = 0;

        if (globalIdx < totalClusters) {
            ClusterData cluster = GET_CLUSTER_DATA(ctx.forwardPlusClusterListBufferAddr, globalIdx);
            pCount = cluster.pointLightCount;
            sCount = cluster.spotLightCount;
        }

        // 1. Hardware subgroup scan (Fast prefix sum within 32 threads)
        uint pSubOffset = subgroupExclusiveAdd(pCount);
        uint sSubOffset = subgroupExclusiveAdd(sCount);

        // 2. Total count within the subgroup
        uint pSubTotal = subgroupAdd(pCount);
        uint sSubTotal = subgroupAdd(sCount);

        // 3. Store subgroup totals in shared memory for cross-subgroup scanning
        if (subgroupElect()) {
            s_pointSubgroupSums[gl_SubgroupId] = pSubTotal;
            s_spotSubgroupSums[gl_SubgroupId] = sSubTotal;
        }

        barrier();

        // 4. Thread 0 sequentially scans the subgroup totals
        if (threadId == 0)
        {
            uint pSum = 0;
            uint sSum = 0;

            for (uint i = 0; i < gl_NumSubgroups; ++i) {
                uint pVal = s_pointSubgroupSums[i];
                uint sVal = s_spotSubgroupSums[i];

                s_pointSubgroupSums[i] = pSum;
                s_spotSubgroupSums[i] = sSum;

                pSum += pVal;
                sSum += sVal;
            }

            s_pointSubgroupTotal = pSum;
            s_spotSubgroupTotal = sSum;
        }

        barrier();

        // 5. Calculate final global offsets and write directly to VRAM
        if (globalIdx < totalClusters)
        {
            uint finalPOffset = s_globalPointCarry + s_pointSubgroupSums[gl_SubgroupId] + pSubOffset;
            uint finalSOffset = s_globalSpotCarry + s_spotSubgroupSums[gl_SubgroupId] + sSubOffset;

            GET_CLUSTER_DATA(ctx.forwardPlusClusterListBufferAddr, globalIdx).pointLightOffset = finalPOffset;
            GET_CLUSTER_DATA(ctx.forwardPlusClusterListBufferAddr, globalIdx).spotLightOffset = finalSOffset;
        }

        // 6. Accumulate the carry for the next 512-element block
        if (threadId == 0) {
            s_globalPointCarry += s_pointSubgroupTotal;
            s_globalSpotCarry += s_spotSubgroupTotal;
        }

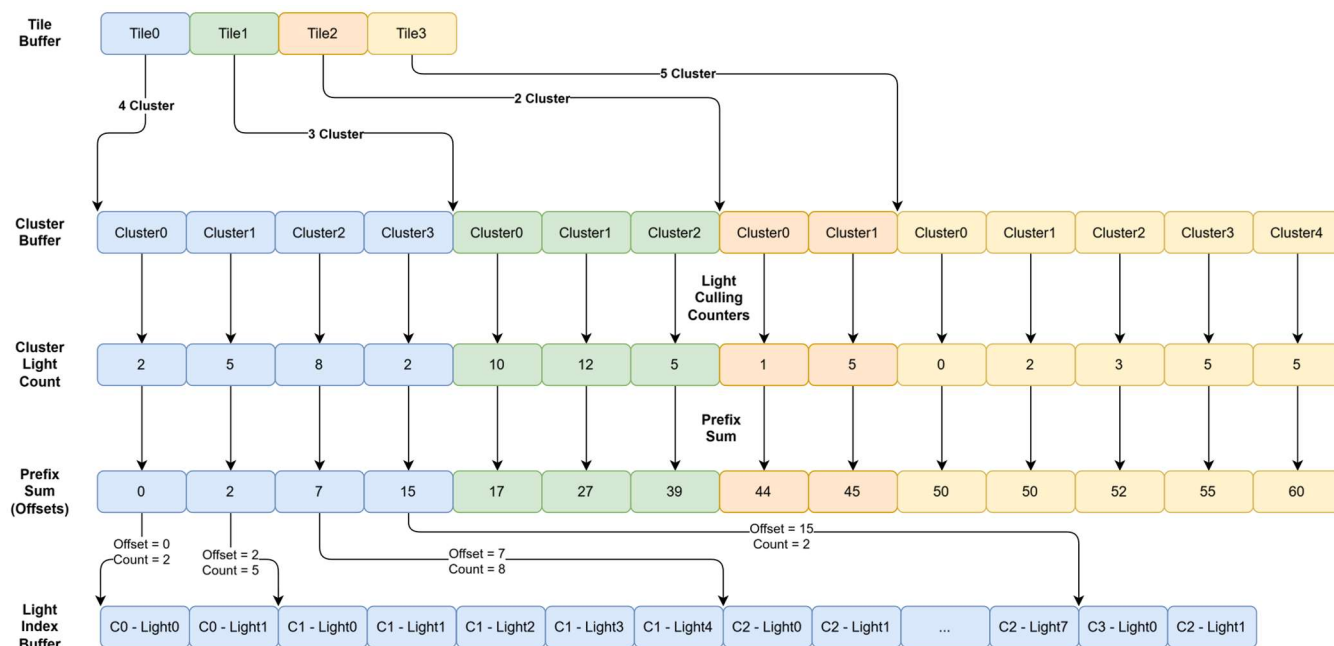
        barrier();
    }
}
```

<https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/Shaders/Passes/Shading/ForwardPlus/Clustering/ClusterPrefixSum.comp>

Lighting and Shading XXXIII.

Scalable Light Index Buffer Construction

- ▶ The prefix sum provides the **exact offset** of every cluster and the previous counting pass already provides the **number of intersecting Point Lights and Spot Lights**.
- ▶ However, the **actual light indices belonging to each cluster are still unknown**. Therefore, the Point Light and Spot Light **intersection tests are executed a second time**.
- ▶ This time, every successful intersection directly writes the corresponding light entity index into the cluster's allocated region of the global light index buffer.
- ▶ The previously calculated **offset and count** guarantee that every cluster writes into its own compact memory range.
- ▶ This scalable path can handle clusters intersecting with **thousands of lights**, without requiring a fixed per-cluster memory allocation.
- ▶ The trade-off is that the light intersection tests must be performed twice: **once for counting and once for writing**.



<https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/Shaders/Passes/Shading/ForwardPlus/Clustering/ClusterPointLightWrite.comp>

Lighting and Shading XXXIV.

Forward+ Light Simulation

- ▶ The **visible geometry is rendered once again** after the Forward+ clusters and light lists have been constructed.
- ▶ However, the scene has already completed the **pre-depth rendering pass**, meaning that the final visible surface for every pixel is already known.
- ▶ The Forward+ shading pass therefore uses **Depth Equal**, ensuring that only fragments matching the previously generated depth buffer are processed.
- ▶ As a result, **every visible pixel generates only a single fragment**, avoiding the repeated material and G-Buffer texture reads required by the Deferred Shading approach.
- ▶ The fragment shader evaluates the complete material only once.
- ▶ The screen-space fragment coordinates directly determine which **XY tile** the fragment belongs to.
- ▶ The fragment depth is then used together with the tile's dynamically generated minimum and maximum depth range to determine the corresponding **adaptive cluster slice**.
- ▶ The shader **then iterates only over the compact light index ranges** assigned to the current cluster.

```
void main()
{
    // 1. Fetch Material
    Material mat = GET_MATERIAL(ctx.materialBufferAddr, materialId);
    vec2 finalUV = inUV * mat.uvScale;

    // 2. Evaluate Albedo & Alpha
    vec4 albedoAlpha = EvaluateAlbedoAlpha(ctx.textureMetadataBufferAddr, mat, finalUV);

    // 4. Evaluate Metalness & Roughness
    vec2 metalRough = EvaluateMetallicRoughness(ctx.textureMetadataBufferAddr, mat, finalUV);
    float finalMetalness = metalRough.x;
    float finalRoughness = clamp(metalRough.y, 0.04, 1.0);

    // Material Evaluation (Normal, Clearcoat, IOR, Specular, Emissive, etc.) ...

    uint tileX = uint(gl_FragCoord.x) / ctx.tileSize;
    uint tileY = uint(gl_FragCoord.y) / ctx.tileSize;
    uint tileIndex = tileY * ctx.tileCountX + tileX;

    TileData tile = GET_TILE_DATA(ctx.forwardPlusTileGridListBufferAddr, tileIndex);
    uint sliceIdx = GetClustersSliceIndex(viewDepth, tile.minZ, tile.maxZ, tile.sliceCount);

    uint clusterIndex = tile.clusterBaseOffset + sliceIdx;
    ClusterData cluster = GET_CLUSTER_DATA(ctx.forwardPlusClusterListBufferAddr, clusterIndex);

    // 7. Simulate Lighting
    vec3 totalRadiance = vec3(0.0);

    for(uint i = 0; i < ctx.activeDirectionalLightCount && ctx.enableForwardPlusDirectionalLights == 1; ++i) {
        uint entityIdx = GET_DIRECTION_VISIBLE_LIGHT(ctx.directionalLightVisibleIndexBufferAddr, i);
        uint lightDenseIndex = GET_SPARSE_INDEX(ctx.directionalLightSparseMapBufferAddr, entityIdx);

        vec3 dirLightDirection = GET_DIRECTION_LIGHT(ctx.directionalLightDataBufferAddr, lightDenseIndex).direction.xyz;
        vec3 lightDir = normalize(-dirLightDirection);

        vec3 lightContribution = SimulateDirectionalLight(
            ctx.directionalLightDataBufferAddr, lightDenseIndex,
            albedoAlpha.rgb, finalNormal, viewDir, finalRoughness, finalMetalness,
            ior, specularFactor, specularColor, clearcoatFactor, clearcoatRoughness, clearcoatNormal
        );

        totalRadiance += lightContribution * shadowFactor;
    }

    for(uint i = 0; i < cluster.pointLightCount && ctx.enableForwardPlusPointLights == 1; ++i) {
        uint globalLightIndex = cluster.pointLightOffset + i;
        uint lightEntityIndex = GET_LIGHT_INDEX(ctx.forwardPlusPointLightIndexListBufferAddr, globalLightIndex);
        uint lightDenseIndex = GET_SPARSE_INDEX(ctx.pointLightSparseMapBufferAddr, lightEntityIndex);

        vec3 pointLightPosition = GET_POINT_LIGHT(ctx.pointLightDataBufferAddr, lightDenseIndex).position.xyz;

        vec3 lightContribution = SimulatePointLight(
            ctx.pointLightDataBufferAddr, lightDenseIndex, worldPos,
            albedoAlpha.rgb, finalNormal, viewDir, finalRoughness, finalMetalness,
            ior, specularFactor, specularColor, clearcoatFactor, clearcoatRoughness, clearcoatNormal
        );

        totalRadiance += shadowFactor * lightContribution;
    }

    for(uint i = 0; i < cluster.spotLightCount && ctx.enableForwardPlusSpotLights == 1; ++i) {
        uint globalLightIndex = cluster.spotLightOffset + i;
        uint lightEntityIndex = GET_LIGHT_INDEX(ctx.forwardPlusSpotLightIndexListBufferAddr, globalLightIndex);
        uint lightDenseIndex = GET_SPARSE_INDEX(ctx.spotLightSparseMapBufferAddr, lightEntityIndex);

        vec3 spotLightPosition = GET_SPOT_LIGHT(ctx.spotLightDataBufferAddr, lightDenseIndex).position.xyz;
        vec3 lightDir = normalize(spotLightPosition - worldPos);

        vec3 lightContribution = SimulateSpotLight(
            ctx.spotLightDataBufferAddr, lightDenseIndex, worldPos,
            albedoAlpha.rgb, finalNormal, viewDir, finalRoughness, finalMetalness,
            ior, specularFactor, specularColor, clearcoatFactor, clearcoatRoughness, clearcoatNormal
        );

        totalRadiance += shadowFactor * lightContribution;
    }
}
```

Material

Calculate TileIndex,
and Adaptive
Cluster Index

Directional Light

Point Light

Spot Light

<https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/Shaders/Passes/Shading/ForwardPlus/Lighting/OpaqueForward.frag>

Lighting and Shading XXXV.

Performance Evaluation

- Both rendering approaches successfully handle **thousands of dynamic Point and Spot Lights**, demonstrating that the Deferred and Forward+ implementations remain fully scalable in terms of scene complexity.
- At lower light counts, Forward+ introduces additional GPU work for **cluster construction, light culling, and index-buffer generation**, resulting in a relatively small performance overhead.
- As the number of lights increases, however, the cost of **Deferred Shading grows significantly** because each light requires a separate lighting evaluation and repeated G-Buffer accesses.
- The effect becomes especially visible with **8192 Point Lights + 8192 Spot Lights**, where Deferred Shading reaches **8.652 ms**, while the complete Forward+ pipeline requires only **2.873 ms**.
- This corresponds to approximately a **3× reduction in total GPU time**, despite Forward+ including the complete clustering and hierarchical light-culling process.
- The results demonstrate that **Forward+ scales substantially better with increasing local-light density**, because material data is evaluated once per visible fragment and only the lights assigned to the corresponding cluster are simulated.

Table 7: Forward+ pipeline performance breakdown.

Stage	1024+1024 ms	4096+4096 ms	8192+8192 ms
Cluster Setup	0.011	0.010	0.010
Point Light Culling	0.048	0.096	0.144
Spot Light Culling	0.111	0.306	0.477
Prefix Sum	0.021	0.017	0.016
Point Light Index Write	0.063	0.122	0.172
Spot Light Index Write	0.138	0.358	0.551
Forward Geometry + Lighting	0.428	0.660	1.502
Total	0.820	1.569	2.873

Table 8: Deferred shading performance breakdown.

Stage	1024+1024 ms	4096+4096 ms	8192+8192 ms
G-Buffer Rendering	0.006	0.009	0.008
Emissive + AO	0.057	0.062	0.062
Directional Lights	0.063	0.072	0.070
Point Lights	0.091	0.372	0.802
Spot Lights	0.895	3.393	7.469
Total	1.328	4.146	8.652

Table 6: Forward+ and Deferred Shading performance comparison.

Lights (Point + Spot)	Forward+ (ms)	Deferred (ms)	Speedup
1024 + 1024	0.820	1.328	1.62×
4096 + 4096	1.569	4.146	2.64×
8192 + 8192	2.873	8.652	3.01×

Lighting and Shading XXXVI.

Adaptive Tile Size Performance

- ▶ The Dual-HiZ hierarchy allows the Forward+ system to operate with **different screen-space tile sizes without rebuilding the depth bounds using a dedicated compute reduction pass.**
- ▶ Each tile directly accesses the corresponding Hi-Z mip level, providing its occupied **minimum and maximum depth range** through texture fetches.
- ▶ Very small tiles create a large number of clusters, significantly increasing **cluster setup, light intersection testing, and index generation costs.**
- ▶ Very large tiles reduce the number of clusters, but each cluster covers a much larger spatial region and therefore contains **more potentially intersecting lights.**
- ▶ With **4096 Point Lights and 4096 Spot Lights**, the best performance was achieved with a **64×64 tile size**, requiring only **2.362 ms.**
- ▶ This is approximately **2.06× faster than Deferred Shading** in the same scene, which required **4.866 ms.**

Table 9: *Forward+ performance with different adaptive tile sizes using 4096 Point Lights and 4096 Spot Lights.*

Tile Size	Total GPU Time (ms)
8 × 8	20.412
16 × 16	7.168
32 × 32	3.274
64 × 64	2.362
128 × 128	2.921
256 × 256	4.009
512 × 512	9.033
Deferred Shading	4.866

Lighting and Shading XXXVII.

Dual-HiZ Adaptive Clustered Forward Shading Summary

- ▶ **Two complementary Hi-Z hierarchies** provide the minimum and maximum occupied depth range of every screen-space tile, combining opaque and transparent scene information.
- ▶ The occupied depth interval of each tile is **adaptively partitioned into logarithmic clusters**, avoiding unnecessary fixed-depth cells in empty regions.
- ▶ Each generated cluster receives its own **spatial bounds and compact Point Light and Spot Light index ranges**, allowing the Forward+ shader to evaluate only the lights relevant to the current fragment.
- ▶ The system uses **hierarchical light culling**, first reducing the light set against the camera frustum and then performing precise cluster-level intersection tests.
- ▶ The scalable path supports **thousands of local lights without a fixed upper bound**, using counting, prefix sums, and compact global light index buffers.

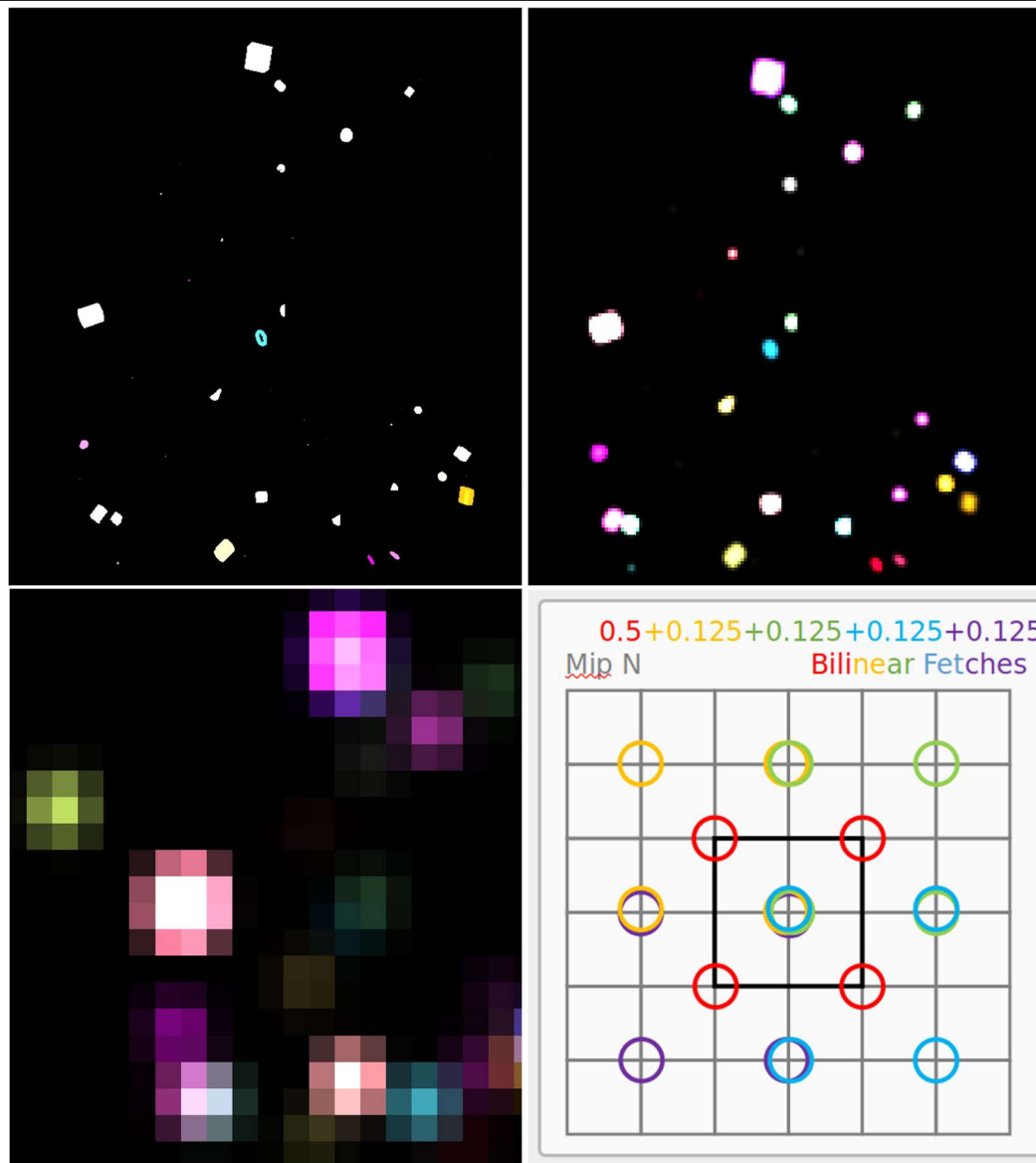
Runtime Adaptive Tile Resolution

- ▶ **Tile resolution is not fixed** and can be changed dynamically at runtime without modifying the Forward+ clustering algorithm.
- ▶ Each tile resolution directly corresponds to a different **Hi-Z mip level**, allowing resolutions such as **32×32, 64×64, 128×128, or 256×256** to be selected.
- ▶ This removes the dependency on a fixed **16×16 compute-based depth reduction**, while still providing the required minimum and maximum depth range for every tile.
- ▶ Different tile resolutions provide different performance characteristics: **smaller tiles improve light culling precision**, while **larger tiles reduce the number of generated clusters and culling operations**.
- ▶ The resolution can therefore be **adaptively adjusted during runtime**, allowing **light-heavy scene regions to use smaller tiles**, such as **32×32 or 64×64**, while **sparsely lit regions can use significantly larger tiles**, such as **128×128 or 256×256!!!**

Lighting and Shading XXXVIII.

Bloom Simulation - Downsampling

- ▶ The lighting pipeline produces an **HDR scene texture**, where pixel values are not limited to the $[0,1]$ range.
- ▶ Bloom is generated from pixels whose luminance exceeds a configurable **brightness threshold**.
- ▶ **Bright regions are extracted into a separate bloom texture**, while low-intensity pixels are suppressed toward black.
- ▶ The extracted image is progressively **downsampled through multiple mip levels**, reducing the resolution at each step.
- ▶ Each downsample level **averages neighboring bright regions**, allowing intense light to spread over increasingly larger screen-space areas.
- ▶ The resulting mip hierarchy represents the same high-intensity lighting information at multiple spatial scales.

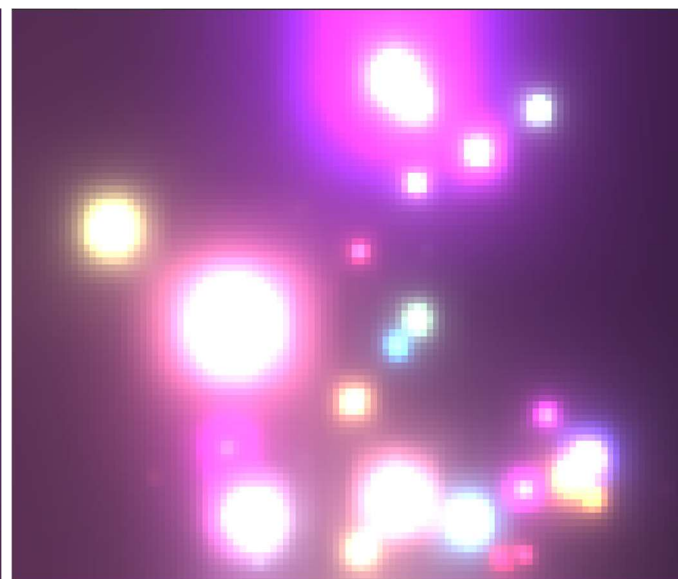
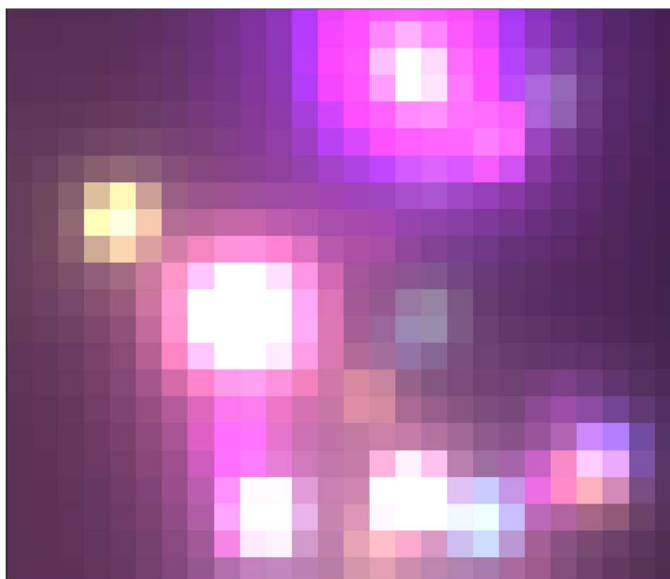


<https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/Shaders/Passes/PostProcess/Bloom/BloomDownsample.comp>

Lighting and Shading XXXIX.

Bloom Simulation – Upsampling and Reconstruction

- ▶ After reaching the smallest mip level, the Bloom hierarchy is processed in the opposite direction.
- ▶ Each lower-resolution level is **upsampled and blended into the previous higher-resolution level**.
- ▶ During upsampling, filtered sampling creates a **smooth spatial distribution** instead of preserving sharp pixel boundaries.
- ▶ The accumulated result gradually reconstructs a large-scale, **blurred representation of the original bright regions**.
- ▶ The final **Bloom texture is added back to the HDR** lighting result before tone mapping.
- ▶ This creates the characteristic glow around intense light sources and high-luminance surfaces.

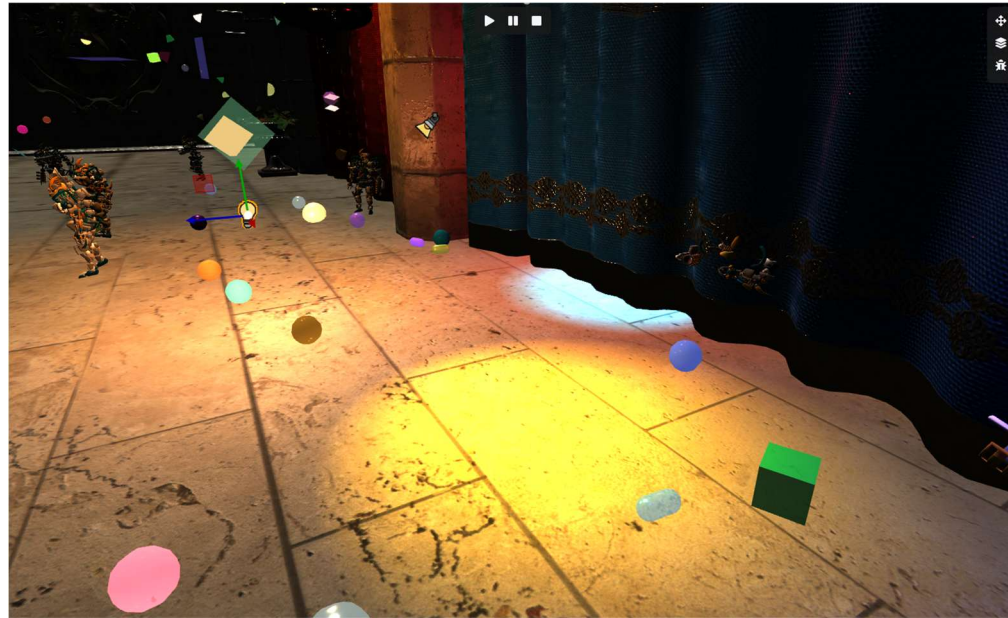


<https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/Shaders/Passes/PostProcess/Bloom/BloomUpsample.comp>

Lighting and Shading XL.

Beyond Emissive Materials

- ▶ **Bloom is not exclusively associated with emissive materials.**
- ▶ Any **lighting calculation can produce HDR values** above the Bloom threshold.
- ▶ Highly intense Point Lights, Spot Lights, and Directional Lights can therefore generate Bloom naturally.
- ▶ Bright emissive surfaces contribute in the same way, as their radiance is accumulated into the final HDR scene texture.
- ▶ The Bloom pass operates on the **final lighting intensity**, rather than on the material type that originally produced it.
- ▶ After tone mapping, the Bloom contribution creates a more natural perception of extremely bright light.



No Bloom



With Bloom

Synapse Engine: Shadows

Created By: Péter Tamás

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

2026

Shadow I.

GPU-Driven Shadow Rendering

- ▶ The GPU-driven rendering pipeline has already been established, including **material bucketing**, allowing all models, meshes, and LODs to be rendered through a unified draw pipeline.
- ▶ The same **hierarchical GPU-driven culling** is reused for shadow rendering, with visibility evaluated through the **light volumes** instead of the camera frustum.
- ▶ The existing **chunk → model → mesh → meshlet** culling hierarchy is therefore directly applicable to shadow casters.
- ▶ The same **instancing and GPU-generated instance buffers** are also reused to construct the shadow rendering workload.
- ▶ Shadow mapping is used to generate the shadow information, but the visibility problem becomes significantly more complex because **many light sources may require shadows simultaneously**.

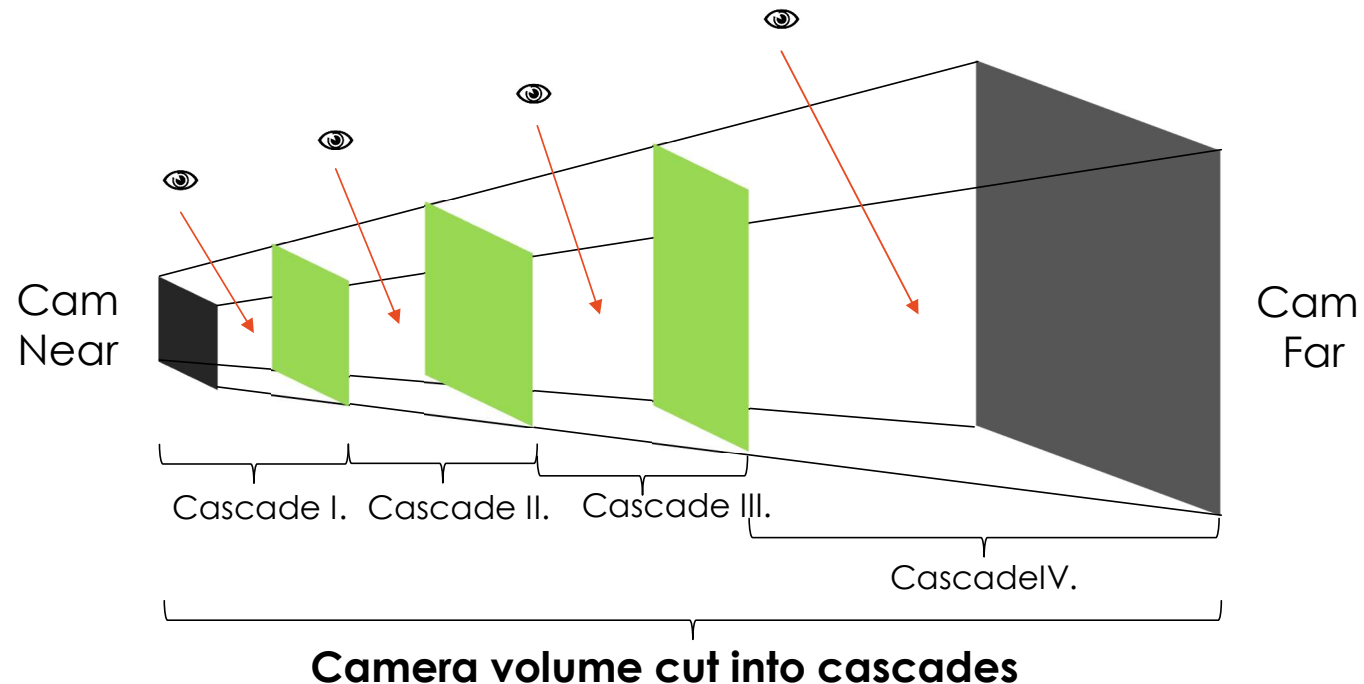
Virtualized Shadow Maps

- ▶ A conventional **shadow-mapping setup would require a separate depth texture** for each light, which conflicts with the unified GPU-driven rendering pipeline.
- ▶ To preserve the single-pass rendering architecture, all shadow maps are stored inside **one large depth texture**.
- ▶ This texture is treated as a **virtual atlas**, divided into independently allocated regions for individual light sources.
- ▶ The available atlas resolution is distributed dynamically: **nearby lights receive larger regions, while distant lights receive smaller regions**.
- ▶ The result is a fully GPU-driven, **virtualized and LOD-aware shadow-mapping system**, supporting both traditional vertex rendering and **task → mesh shader pipelines**.
- ▶ The technique will now be examined separately for **Directional, Point, and Spot Lights**.

Shadow II.

Directional Light Shadow Volume

- ▶ Directional lights have no finite position, therefore the light needs to cover the **entire camera frustum**.
- ▶ Using a single shadow volume for the complete frustum would result in a **very large covered area**.
- ▶ With a fixed shadow-map resolution, this would significantly reduce the available **shadow detail**, especially close to the camera.
- ▶ To solve this, the camera frustum is divided into multiple **depth ranges called cascades**.
- ▶ Each cascade covers only a part of the camera frustum, allowing the available **shadow resolution to be distributed more efficiently**.



Shadow III.

Cascaded Shadow Map Volumes

- ▶ The **near cascades cover a smaller spatial region**, allowing significantly higher shadow detail where it is most visible.
- ▶ The **far cascades cover progressively larger regions**, where lower shadow detail is acceptable due to the distance from the camera.
- ▶ Each **cascade therefore defines its own shadow volume**, fitted around the corresponding part of the camera frustum.
- ▶ For GPU-driven rendering, these cascade volumes replace the camera frustum as the **culling volume**.
- ▶ The existing **hierarchical culling pipeline** can therefore be reused to determine the visible chunks, models, meshes and meshlets for every cascade.



Cascaded
Shadow
Mapping

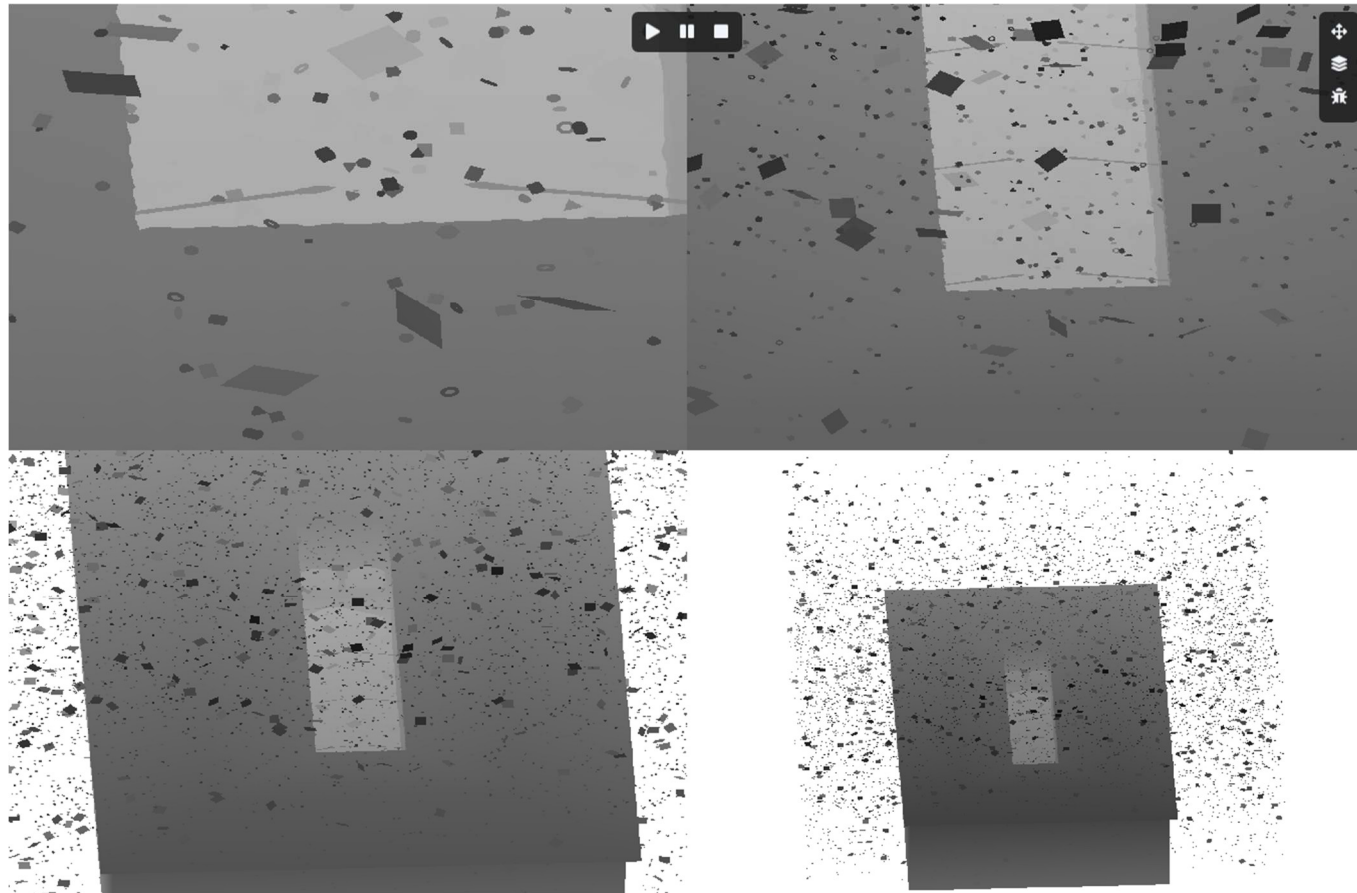


Cascade
Region Color
Visualization

Shadow IV.

Shadow Atlas Partitioning

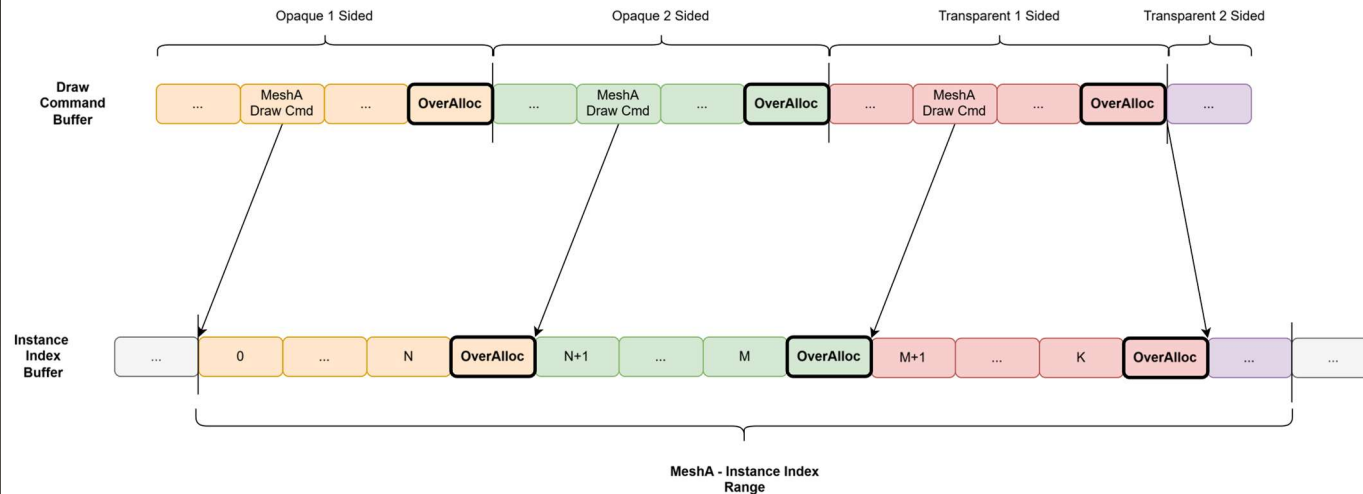
- ▶ The shadow atlas is a single **4K depth texture**, divided into multiple rectangular regions.
- ▶ Each region is defined by a **start point and end point**, forming the shadow-map area assigned to a cascade.
- ▶ All cascades of all directional lights are therefore managed **within the same shadow atlas** and during rendering, only this **single depth texture needs to be bound**.
- ▶ The final shadow projection is transformed in the vertex shader into the **corresponding atlas region** of the active cascade.
- ▶ This allows the complete shadow rendering pipeline to remain **GPU-driven without requiring separate depth textures for individual cascades**.



Shadow V.

Draw Buffer Allocation of Dirlights

- ▶ The existing **batched draw command buffer** generated by the RenderSystem can be reused **1:1 for shadow rendering**, together with its batched instance index buffer and offset/count allocation.
- ▶ The main difference is that the instance buffer no longer stores only an entity, but an **(entity, directional light, cascade)** tuple. (Bitpacked!)
- ▶ Therefore, each **draw command contains the instances of all directional lights and all of their cascades**, allowing them to be rendered together through the same GPU-driven pipeline.
- ▶ The required instance buffer **offsets and counts scale with the number of directional lights × 4 cascades**, while the underlying bucket structure remains unchanged.
- ▶ During vertex processing, the stored light and cascade information determines the **corresponding shadow-atlas region** into which the geometry has to be projected.
- ▶ To prevent geometry from spilling into neighboring atlas regions, the projection uses the built-in **gl_ClipDistance** to clip fragments outside the active cascade rectangle.



```
void main() {
    FrameGlobalContext ctx = GET_FRAME_CONTEXT(pc.frameGlobalContextBufferAddr);

    // 1. Fetch Draw Descriptor
    MeshDrawDescriptor desc = GET_DRAW_DESCRIPTOR(ctx.globalIndirectCommandDescriptorBufferAddr, pc.baseDescriptorOffset + gl_DrawIDARB);

    // 2. Fetch Instance and Entity ID
    uint shadowInstanceOffset = (desc.instanceOffset * ctx.directionLightShadowMultiplier) + gl_InstanceIndex;
    uint payload = GET_INSTANCE(ctx.directionLightShadowInstanceBufferAddr, shadowInstanceOffset);

    uint entityId = payload & 0x3FFFFFFu;
    uint cascadeIdx = (payload >> 26) & 0x3u;
    uint lightIdx = (payload >> 28) & 0x7u;

    //...

    // 9. Calculate Final World Position and Outputs
    vec4 clipPos = viewProj * transform.transform * finalModelMat * vec4(v.position, 1.0);

    gl_ClipDistance[0] = clipPos.w + clipPos.x;
    gl_ClipDistance[1] = clipPos.w - clipPos.x;
    gl_ClipDistance[2] = clipPos.w + clipPos.y;
    gl_ClipDistance[3] = clipPos.w - clipPos.y;

    vec4 rect = GET_DIRECTION_LIGHT_SHADOW(ctx.directionLightShadowDataBufferAddr, lightShadowDenseIndex).cascadeAtlasRects[cascadeIdx];
    vec2 scale = rect.zw;
    vec2 offset = rect.xy * 2.0 + rect.zw - 1.0;

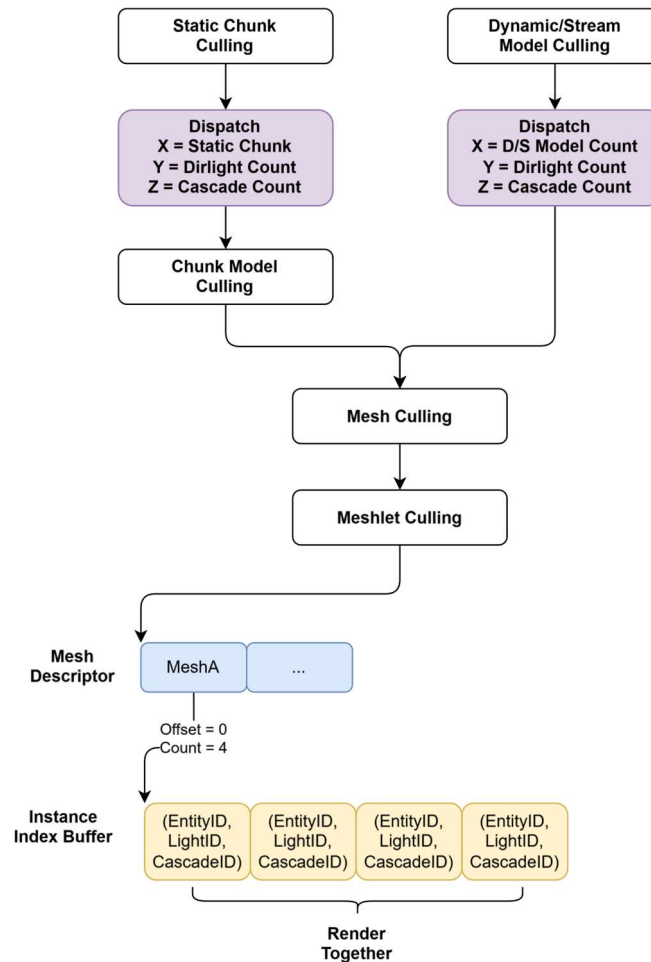
    clipPos.xy = clipPos.xy * scale + offset * clipPos.w;

    // Atlas Positioning
    gl_Position = clipPos;
}
```


Shadow VI.

Compute Culling Shaders for Cascades

- ▶ The existing **chunk** → **model** → **mesh** → **meshlet** hierarchical culling pipeline is reused to determine the geometry visible in each directional-light cascade.
- ▶ The compute dispatch is extended into a **3D grid**, where **X = static chunks**, **Y = directional lights**, and **Z = cascades**.
- ▶ The culling process itself remains unchanged: the additional dimensions simply ensure that every chunk/model/mesh is tested against **every light and every cascade**.
- ▶ The engine provides both **GPU-driven compute culling** and a CPU-driven implementation, allowing the rendering path to be switched at runtime.
- ▶ The instance payload compactly stores the **entity ID, light index, cascade index, and frustum-intersection state** in a single uint32.
[31: Inside Frustum] [30:28: Light Index] [27:26: Cascade Index] [25:0: Entity ID]



```
// 9. Encode Payload: [Bit 31: Inside Frustum] [Bits 28-30: LightIdx] [Bits 26-27: CascadeIdx] [Bits 0-25: EntityID]
uint payload = (entityId & 0x3FFFFFFu);
payload |= ((cascadeIdx & 0x3u) << 26);
payload |= ((lightIdx & 0x7u) << 28);
payload = SET_BIT_TO(payload, 31, visibility == INTERSECTION_INSIDE);
```

Shadow VII.

GPU-Driven Directional Light Shadow Rendering

- ▶ After **cascade-specific GPU culling**, the generated draw commands and (Entity, Light, Cascade) data are rendered using the same GPU-driven architecture.
- ▶ The instance payload identifies the **entity, directional light and cascade**, allowing multiple lights and cascades to share a single batched draw command.
- ▶ The vertex/mesh shader resolves the corresponding **cascade view-projection matrix** and transforms the instance into shadow space.
- ▶ The geometry is projected directly into its assigned **shadow atlas region** using the stored atlas rectangle.
- ▶ The result is a fully **GPU-driven shadow pass**, from hierarchical culling to indirect rendering and atlas projection.
- ▶ A single bucket can therefore contain geometry for **multiple directional lights × cascades**, with each instance projected to its correct atlas region.

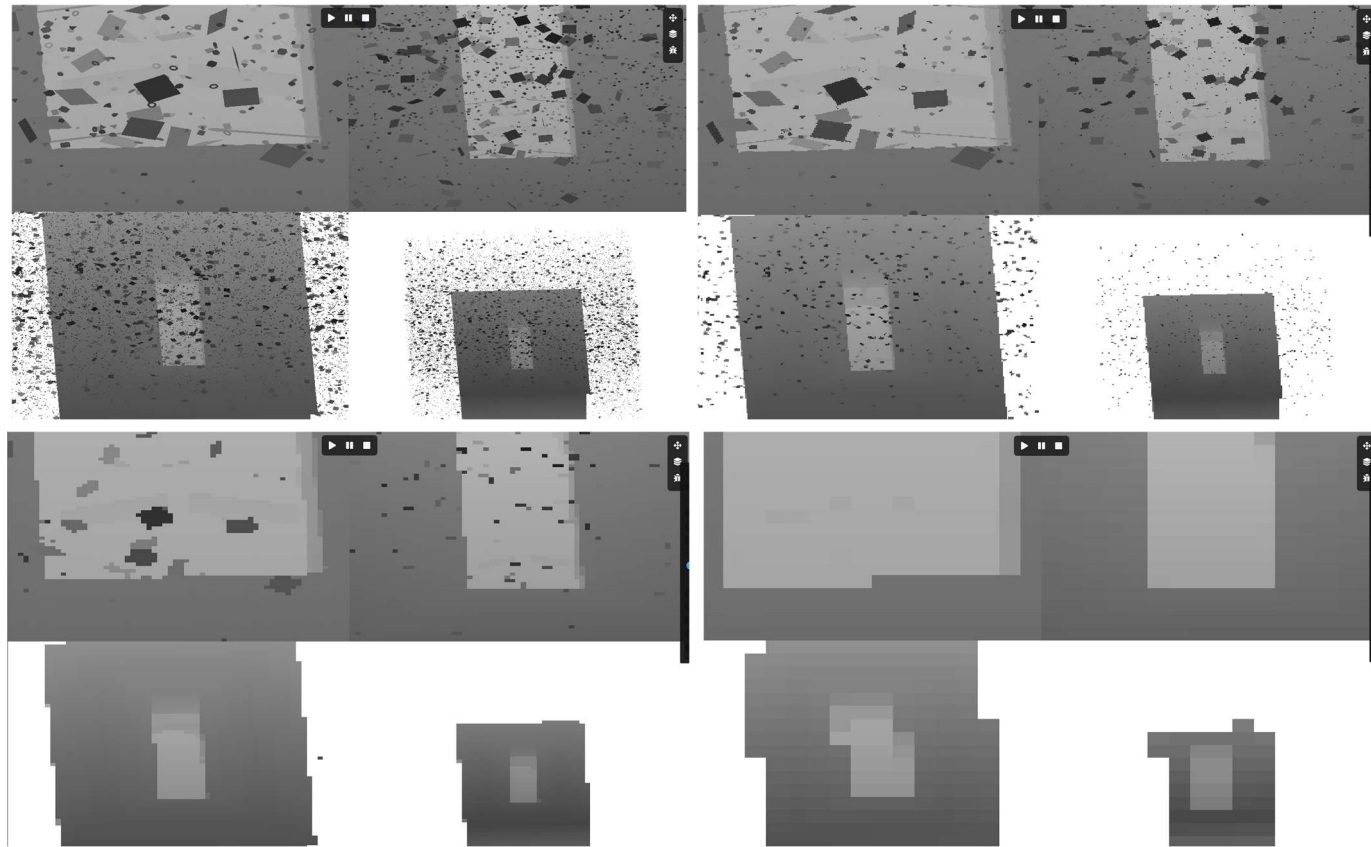


<https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/Shaders/Passes/Shadow/DirectionLight/DirectionLightShadowTraditional.vert>

Shadow VIII.

Direction Light HiZ Occlusion Culling

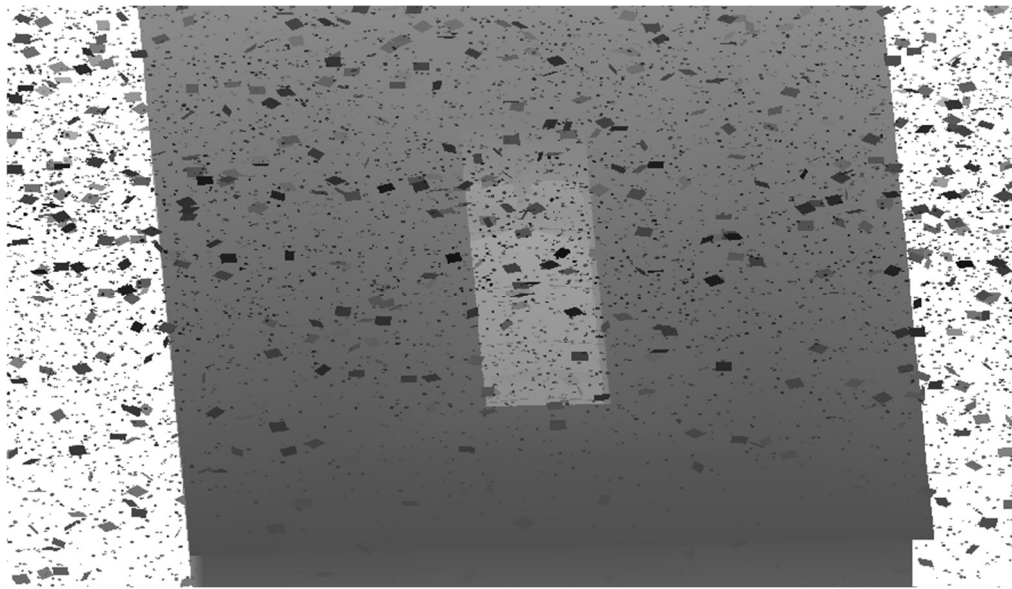
- ▶ The existing **Hi-Z occlusion culling** can be reused for directional-light shadow rendering.
- ▶ A Hi-Z pyramid is generated from the **4K shadow atlas**, allowing occlusion tests to be performed directly against the shadow-map depth.
- ▶ Each cascade can use **min/max depth reduction** to determine which geometry is potentially visible from the light.
- ▶ Fully occluded chunks, models, meshes and meshlets can therefore be **rejected before shadow rendering**.
- ▶ The Hi-Z hierarchy also provides the foundation for **cascade bounds tightening**, reducing the effective shadow volume where possible.
- ▶ The additional Hi-Z construction **introduces a cost due to the large 4K atlas resolution**, but can become beneficial when a cascade contains large amounts of occluded geometry.



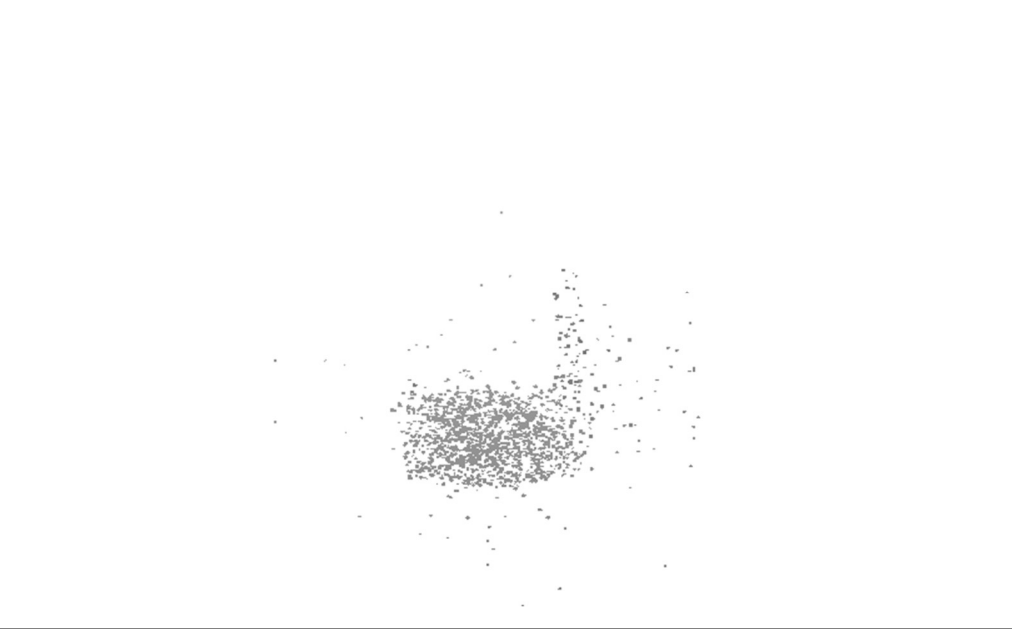
Shadow IX.

Static Shadow Rendering & Baking

- ▶ Directional-light cascades can cover **significantly more geometry than the main camera**, making shadow rendering expensive even with efficient GPU-driven culling.
- ▶ When both the **light and geometry are static**, their shadow contribution does not need to be rendered every frame.
- ▶ **Static geometry** can therefore be **baked into the shadow atlas**, removing it from subsequent shadow-rendering passes.
- ▶ Only **dynamic entities have to be rendered again** each frame and composited with the baked shadow contribution.
- ▶ The existing **batched draw-command and material-bucket system** can be reused for this process, keeping the rendering GPU-driven.
- ▶ This significantly reduces the amount of geometry and **draw work required per frame**, especially for large static scenes.



Static Baked
Workload

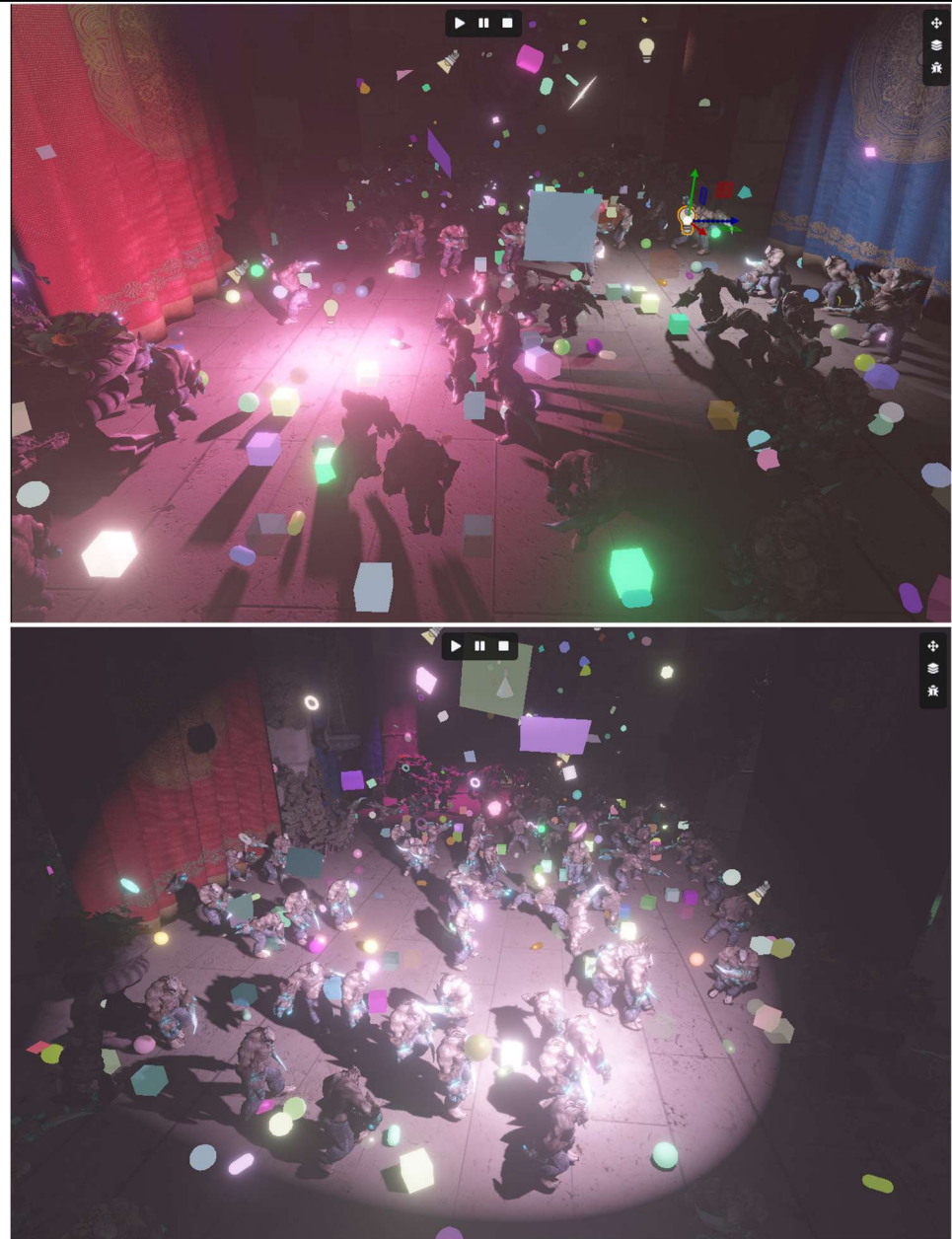


Dynamic
Per-Frame
Workload

Shadow X.

Point/Spot Light Shadow Rendering

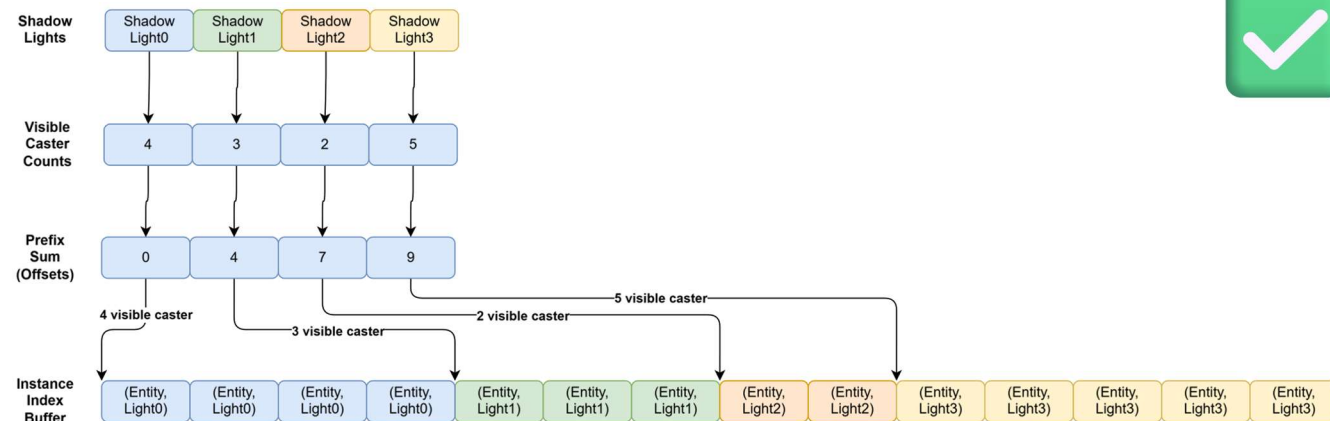
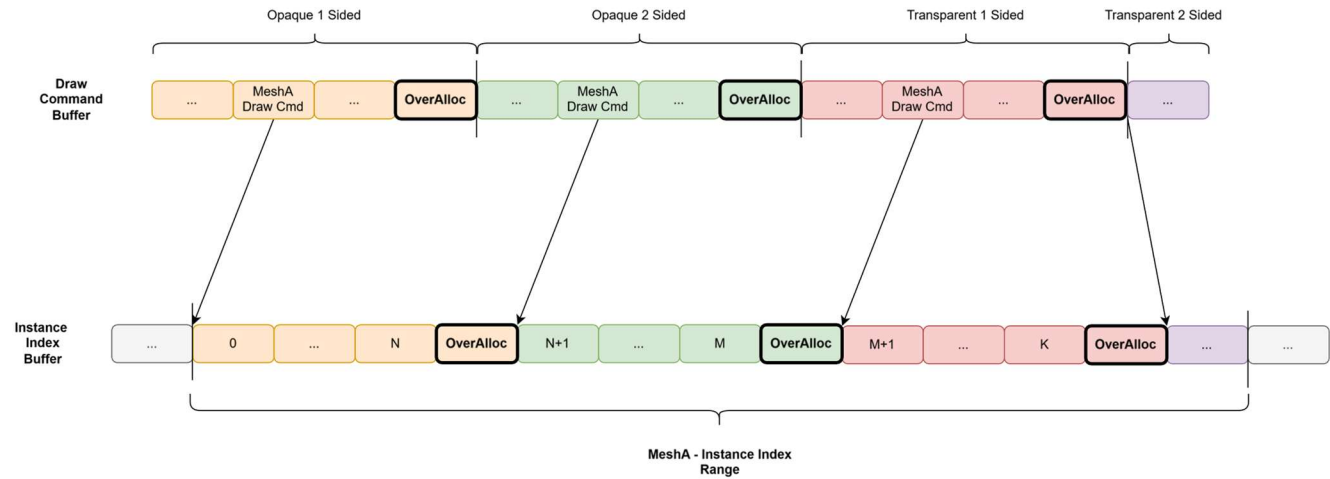
- ▶ The **same GPU-driven shadow rendering architecture** introduced for directional lights is reused for point and spot lights.
- ▶ A **single large shadow atlas** remains bound as the depth target, with individual light shadow views dynamically assigned to regions of the atlas.
- ▶ The existing **batched material buckets and draw-command architecture** is preserved: one mesh rendered as a caster for 15 different spot lights can still be handled by **a single draw command**.
- ▶ Each instance carries the information required to project its geometry into the **appropriate light and atlas region**.
- ▶ The complete **Chunk → Model → Mesh → Meshlet** hierarchical culling pipeline is retained.
- ▶ The key difference is the culling volume: instead of the camera frustum or directional-light cascades, objects are culled against the **individual point and spot light volumes**.



Shadow XI.

Volume Light Shadow Instance Buffer Allocation

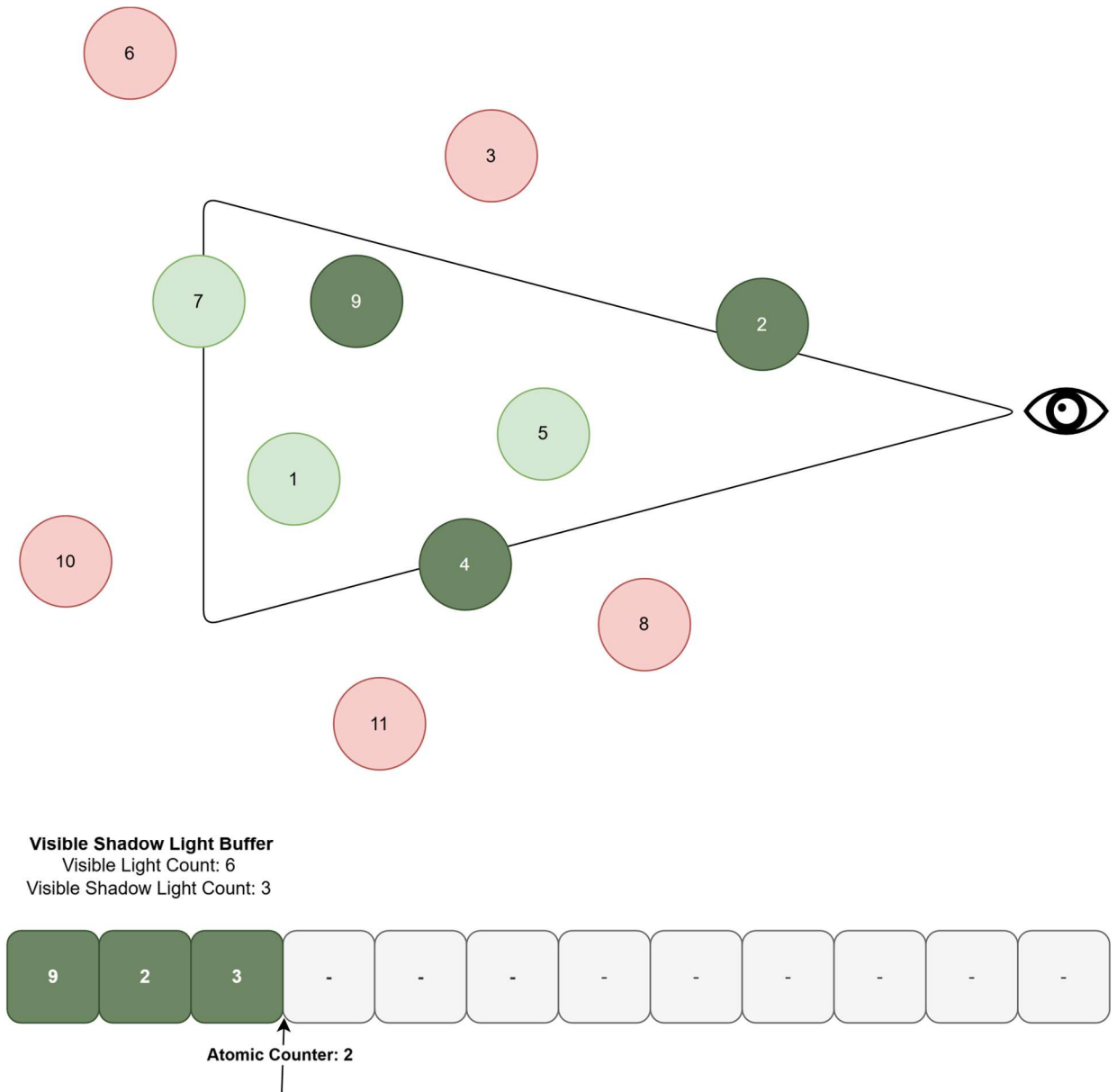
- ▶ **Directional lights are relatively few** typically only 1–2 per scene, making fixed instance-buffer regions practical even with their large shadow volumes.
- ▶ **Point and spot lights can be much more numerous**, potentially reaching hundreds of lights, making pre-allocated per-light instance ranges inefficient.
- ▶ Their shadow volumes are also **far more spatially constrained** than directional-light cascades, so significantly fewer objects are usually visible for each light.
- ▶ A fixed allocation would therefore either **waste large amounts of instance-buffer memory** or require conservative upper bounds that do not scale well.
- ▶ The solution follows the same principle as the **scalable Forward+ path**: visibility results are first **compactly collected** rather than written into predefined regions.
- ▶ The collected (entity, light) visibility pairs are then **radix-sorted**, allowing the engine to generate the final **per-light offsets and counts** required for GPU-driven shadow rendering.



Shadow XII.

Camera-Space Light Culling

- ▶ As already demonstrated for **Deferred and Forward+ lighting**, point and spot lights are **first tested against the main camera frustum**.
- ▶ The existing GPU-driven light culling pipeline is reused, including **frustum, occlusion and 0-pixel culling**.
- ▶ This ensures that lights completely outside the camera view are rejected **before entering the shadow pipeline**.
- ▶ The same **sphere + AABB visibility tests** are used to efficiently determine whether a light can contribute to the current frame.
- ▶ The resulting visible lights are already stored in the engine's **GPU-driven visible light index buffer**.
- ▶ This provides the first filtering stage before the much more expensive **light-volume shadow culling**.



Shadow XIII.

Shadow-Casting Light Extraction

- ▶ The existing light culling shader is extended with a **second indirect draw command and visible-index buffer** dedicated to shadow-casting lights.
- ▶ After passing the camera visibility tests, each light is checked for a valid **shadow configuration**. Only lights that are both **camera-visible and shadow-enabled** are written into the shadow-visible light buffer.
- ▶ This creates a compact list containing only the lights that can actually require **shadow atlas rendering** in the current frame.
- ▶ The shadow pipeline can therefore operate directly on this GPU-generated list instead of processing every point and spot light in the scene.
- ▶ The resulting light list becomes the input for **shadow atlas allocation, light-volume culling and GPU-driven shadow rendering**.

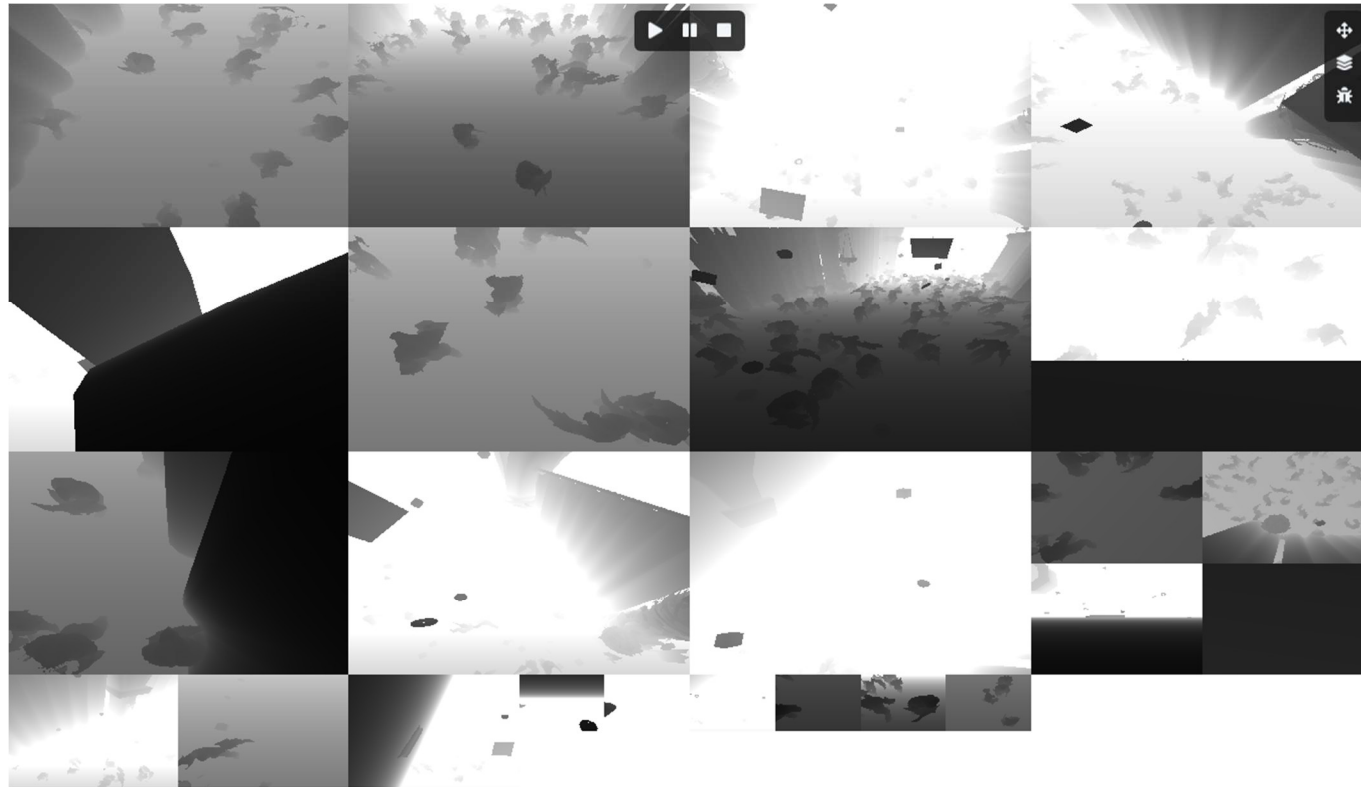
```
void main() {  
    // 0. Identify light and get collider  
    // 1. Frustum Culling (Sphere + AABB) ...  
    // 2. Occlusion and 0-pixel Culling ...  
    // 3. Subgroup atomic add and light index buffer creation ...  
  
    uint shadowSparseIdx = GET_SPARSE_INDEX(ctx.spotLightShadowSparseMapBufferAddr, collider.entityIndex);  
    if (shadowSparseIdx != INVALID_INDEX)  
    {  
        float screenSizePixels = CalculateSphereScreenSizePerspective(collider.center, collider.radius, camera.view,  
            camera.projVulkan, camera.params.x, screenRes);  
  
        uint blockSizePx = 64;  
        if (screenSizePixels > 1024.0) blockSizePx = 1024;  
        else if (screenSizePixels > 512.0) blockSizePx = 512;  
        else if (screenSizePixels > 256.0) blockSizePx = 256;  
        else if (screenSizePixels > 128.0) blockSizePx = 128;  
        uint blocksRequired = blockSizePx / 64;  
  
        uint invertedSize = 0xFFu - blocksRequired;  
        uint sortKey = (invertedSize << 24) | (shadowSparseIdx & 0xFFFFFu);  
  
        uint needsShadowWrite = 1;  
        uint shadowSubgroupWriteCount = subgroupAdd(needsShadowWrite);  
        uint shadowLocalOffset = subgroupExclusiveAdd(needsShadowWrite);  
        uint shadowBaseOffset = 0;  
  
        if (subgroupElect()) {  
            shadowBaseOffset = atomicAdd(GET_SPOT_VISIBLE_COUNT_DATA(ctx.spotLightShadowVisibleCountBufferAddr), shadowSubgroupWriteCount);  
        }  
  
        shadowBaseOffset = subgroupBroadcastFirst(shadowBaseOffset);  
        uint finalShadowIndex = shadowBaseOffset + shadowLocalOffset;  
        GET_SPOT_VISIBLE_SHADOW_LIGHT(ctx.spotLightVisibleShadowIndexBufferAddr, finalShadowIndex) = collider.entityIndex;  
  
        GET_SPOT_ATLAS_SORT_KEY(ctx.spotLightShadowAtlasSortKeyBufferAddr, finalShadowIndex) = sortKey;  
        GET_SPOT_ATLAS_SORT_VALUE(ctx.spotLightShadowAtlasSortValueBufferAddr, finalShadowIndex) = collider.entityIndex;  
    }  
}
```

<https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/Shaders/Passes/Culling/SpotLight/SpotLightCulling.comp>

Shadow XIV.

GPU-Generated Shadow Atlas Allocation

- ▶ The **shadow atlas layout is generated directly on the GPU**, using the same allocation logic already implemented on the CPU side.
- ▶ The previous light-culling pass produces a compact buffer containing each **shadow-casting light index and its required atlas size**.
- ▶ These entries are sorted using a **GPU radix sort**, ordering the lights from the largest required regions to the smallest.
- ▶ The atlas allocator then processes the sorted list and assigns each light a **contiguous atlas region** according to its required resolution.
- ▶ The current implementation uses a **single GPU thread for the actual allocation**, rather than a fully parallel bin-packing algorithm.
- ▶ The result is a **size-aware virtual shadow atlas**, where large, close light sources receive more resolution and distant lights occupy smaller regions.

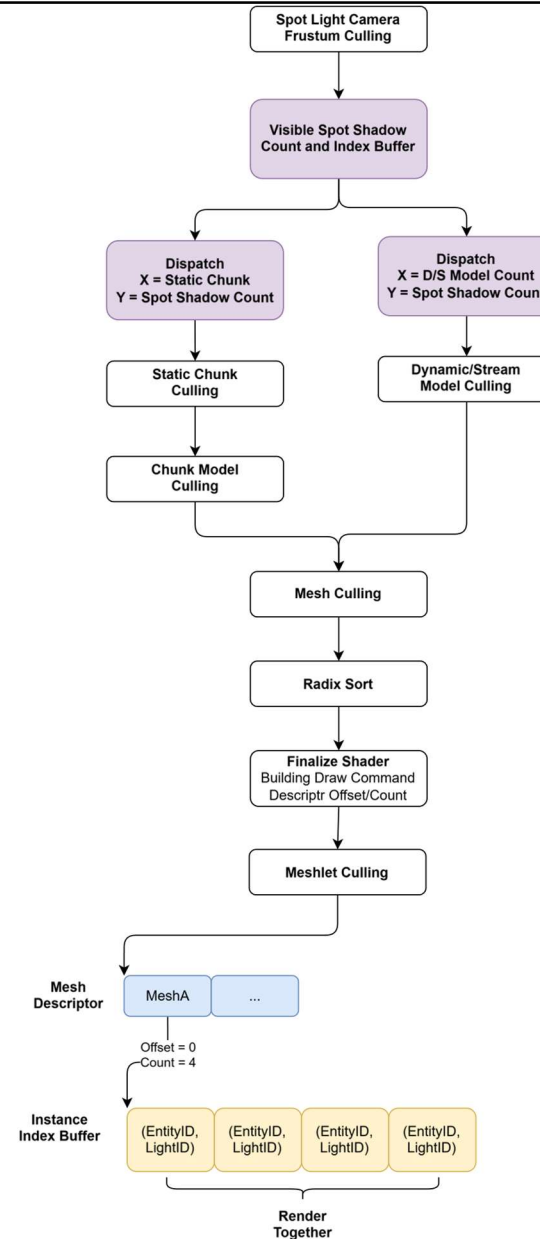


<https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/Shader/Passes/Culling/SpotLight/SpotLightShadowAtlasAllocator.comp>

Shadow XV.

GPU-Driven Spot Light Shadow Culling

- ▶ The same **hierarchical GPU-driven culling pipeline** is reused for spot light shadows, from **Chunk** → **Model** → **Mesh** → **Meshlet**.
- ▶ Unlike directional lights, culling is performed against the **visible shadow-casting spot lights**, rather than against cascades.
- ▶ The indirect dispatch is extended into a **2D work grid**:
 - ▶ X = visible chunks / models
 - ▶ Y = visible shadow-casting lights
- ▶ During mesh and meshlet culling, the resulting visibility information is stored as a **bit-packed (Entity, Light) pair**, identifying both the shadow caster and the spot light it belongs to.
- ▶ This produces the GPU-driven **draw command and instance data** required to render all visible spot light shadow casters into the previously allocated shadow atlas regions.



Shadow XVI.

Compact Pair Sorting & Draw Command Finalization

- ▶ The hierarchical culling pass does **not directly generate the final draw commands**. Instead, it compactly collects the visible **(Entity, Light) pairs** produced by the mesh and meshlet culling stages.
- ▶ Each visible pair is associated with the **indirect draw command index** that will eventually render it. This index becomes the **radix-sort key**, while the pair itself remains the associated value.
- ▶ A **GPU radix sort groups all visible instances belonging to the same draw command**, producing a contiguous region for every command in the instance buffer.
- ▶ A dedicated **Finalize Compute Shader** then processes the sorted entries and reconstructs the required **offset and count** for every indirect draw command.
- ▶ The shader uses **atomic increments** to accumulate the instance count of each draw command while traversing the sorted pairs.
- ▶ The **first occurrence of a new draw-command index** identifies the beginning of that command's region its position in the sorted buffer directly becomes the **instance offset**.
- ▶ The result is a compact, GPU-generated **indirect draw + instance buffer**, ready to render all spot-light shadow casters into their assigned atlas regions.

```
layout(local_size_x = 256, local_size_y = 1, local_size_z = 1) in;

void main() {
    FrameGlobalContext ctx = GET_FRAME_CONTEXT(pc.frameGlobalContextBufferAddr);

    uint visibleCount = GET_SPOT_VISIBLE_COUNT_DATA(ctx.spotLightShadowVisibleMeshCountBufferAddr);
    uint globalId = gl_GlobalInvocationID.x;

    if (globalId >= visibleCount) return;

    // Fetch sorted key and original unsorted slot index
    uint currentKey = GET_SPOT_DRAW_CALL_KEY_DATA(ctx.spotLightShadowDrawCallKeyBufferAddr, globalId);
    uint originalIndex = GET_SPOT_SORTED_VALUE(ctx.spotLightShadowSortValuesBufferAddr, globalId);

    // Gather from unsorted and scatter into final compact instance buffer
    uvec2 payload = GET_SPOT_SHADOW_INSTANCE_UNSORTED(ctx.spotLightShadowUnsortedInstanceBufferAddr, originalIndex);
    GET_SPOT_SHADOW_INSTANCE(ctx.spotLightShadowInstanceBufferAddr, globalId) = payload;

    // Detect draw call boundaries
    bool isFirst = (globalId == 0);
    uint prevKey = isFirst ? 0xFFFFFFFF : GET_SPOT_DRAW_CALL_KEY_DATA(ctx.spotLightShadowDrawCallKeyBufferAddr, globalId - 1);

    uint indirectIdx = currentKey & 0x7FFFFFFF;
    bool isMeshlet = (currentKey >> 31) != 0;

    uint descIdx = isMeshlet ? (indirectIdx + ctx.globalTraditionalCommandsCount) : indirectIdx;

    // If this is the start of a new draw command, save the compact offset
    if (currentKey != prevKey) {
        GET_DRAW_DESCRIPTOR(ctx.spotLightDrawDescriptorBufferAddr, descIdx).instanceOffset = globalId;
    }

    // Append instance to the appropriate indirect draw command using atomic additions
    if (isMeshlet) {
        uint64_t meshletBaseAddr = ctx.spotLightShadowIndirectGeometryCommandBufferAddr + (ctx.globalTraditionalCommandsCount * 16ul);
        atomicAdd(GET_MESHLET_CMD(meshletBaseAddr, indirectIdx).groupCountX, 1);
    } else {
        atomicAdd(GET_TRADITIONAL_CMD(ctx.spotLightShadowIndirectGeometryCommandBufferAddr, indirectIdx).instanceCount, 1);
    }
}
```

<https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/Shader/Passes/Culling/SpotLight/SpotLightShadowFinalize.comp>

Shadow XVII.

GPU-Driven Spot Light Shadow Rendering

- ▶ Once the **draw commands and (Entity, Light) instance data** are generated, the same GPU-driven rendering architecture is used.
- ▶ The instance payload identifies the **entity and spot light**, allowing multiple lights to share a single batched draw command.
- ▶ The vertex/mesh shader resolves the corresponding **spot light shadow view-projection matrix** and transforms the instance into shadow space.
- ▶ The geometry is projected directly into the light's assigned **region of the shared 4K shadow atlas**.
- ▶ A single bucket can therefore contain geometry for **many spot lights**, with each instance dynamically projected to its corresponding atlas region.



<https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/Shaders/Passes/Shadow/SpotLight/SpotLightShadowTraditional.vert>

Shadow XVIII.

Point Light Shadow Atlas

- ▶ The point light follows **the same GPU-driven shadow architecture introduced for spot lights**.
- ▶ Unlike a spot light, a point light emits light in **all directions**, requiring a complete **cubemap representation**.
- ▶ The six cubemap faces are laid out directly into the shadow atlas as a **2×3 region**.
- ▶ Each point light therefore receives one atlas region, which is further subdivided into **six individual shadow faces**.
- ▶ The same shared atlas and GPU-driven rendering pipeline can then handle **multiple point lights within a single depth texture**.



<https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/Shader/Passes/Culling/PointLight/PointLightShadowAtlasAllocator.comp>

Shadow XIX.

Point Light Face Culling

- ▶ Point light shadow culling starts with a **Point Light Sphere vs. Mesh AABB/Sphere** intersection test.
- ▶ During culling, we additionally **determine which of the six cubemap faces** the geometry intersects.
- ▶ Geometry that does not intersect a particular face is **not rendered into that face**, avoiding unnecessary shadow rendering.
- ▶ The culling output therefore stores the **light index + cubemap face index**, which is later used by the radix-sort based buffer allocation and rendering shaders.
- ▶ A single geometry instance can therefore **appear in multiple faces, but only where it is actually required**.
- ▶ In the worst case, one point light may require the same geometry to be rendered **up to 6 times**, once for each cubemap face.

```
uint GetPointLightFaceVisibilityMask(vec3 modelCenter, float modelRadius, vec3 lightCenter) {
    vec3 relMin = (modelCenter - vec3(modelRadius)) - lightCenter;
    vec3 relMax = (modelCenter + vec3(modelRadius)) - lightCenter;

    float minAbsX = GetMinAbs(relMin.x, relMax.x);
    float minAbsY = GetMinAbs(relMin.y, relMax.y);
    float minAbsZ = GetMinAbs(relMin.z, relMax.z);

    uint faceMask = 0;
    if (relMax.x > 0.0 && relMax.x >= minAbsY && relMax.x >= minAbsZ) faceMask |= (1u << 0); // +X
    if (relMin.x < 0.0 && -relMin.x >= minAbsY && -relMin.x >= minAbsZ) faceMask |= (1u << 1); // -X
    if (relMax.y > 0.0 && relMax.y >= minAbsX && relMax.y >= minAbsZ) faceMask |= (1u << 2); // +Y
    if (relMin.y < 0.0 && -relMin.y >= minAbsX && -relMin.y >= minAbsZ) faceMask |= (1u << 3); // -Y
    if (relMax.z > 0.0 && relMax.z >= minAbsX && relMax.z >= minAbsY) faceMask |= (1u << 4); // +Z
    if (relMin.z < 0.0 && -relMin.z >= minAbsX && -relMin.z >= minAbsY) faceMask |= (1u << 5); // -Z

    return faceMask;
}

void main()
{
    // Boundary check ...
    // Resolve Point/Transform/Camera/Model/Animation Data ...

    //Collaborative Loop: Process all meshes
    for (uint m = localThreadId; m < addrs.meshCount; m += gl_WorkGroupSize.x) {

        // Lod, Frustum, Occlusion Culling ...

        // Face Visibility (6-way Bitmask)
        uint faceMask = GetPointLightFaceVisibilityMask(worldCollider.center, worldCollider.radius, lightCollider.center);
        if (faceMask == 0u) continue;

        // 10. Emit to Radix Sort Buffers
        MeshAllocationInfo meshAlloc = GET_MESH_ALLOC(ctx.globalMeshAllocationBufferAddr, mAlloc.meshAllocationOffset + (m * 4) + lod);
        uint pipeIdx = GET_PIPELINE_INDEX(ctx.pipelineLookupBufferAddr, modelComp.pipelineOffset + m);

        if (meshAlloc.activeTypes[pipeIdx][matType] == 1)
        {
            uint indirectIdx = meshAlloc.indirectIndices[pipeIdx][matType];
            uint isMeshlet = (pipeIdx == PIPELINE_MESHLET) ? 1u : 0u;

            uint drawCallKey = (isMeshlet << 31) | (indirectIdx & 0x7FFFFFFF);
            uint entityData = pureEntityId;
            entityData = SET_BIT_TO(entityData, 31, visibility == INTERSECTION_INSIDE);

            for (uint f = 0; f < 6; ++f) {
                if ((faceMask & (1u << f)) != 0u) {
                    uint lightDataPacked = (f << 29) | (lightIdx & 0x1FFFFFFF);
                    uvec2 gpuPayload = uvec2(entityData, lightDataPacked);

                    uint sortSlot = atomicAdd(GET_POINT_VISIBLE_COUNT_DATA(ctx.pointLightShadowVisibleMeshCountBufferAddr), 1);
                    GET_POINT_DRAW_CALL_KEY_DATA(ctx.pointLightShadowDrawCallKeyBufferAddr, sortSlot) = drawCallKey;
                    GET_POINT_SORTED_VALUE(ctx.pointLightShadowSortValuesBufferAddr, sortSlot) = sortSlot;
                    GET_POINT_SHADOW_INSTANCE_UNSORTED(ctx.pointLightShadowUnsortedInstanceBufferAddr, sortSlot) = gpuPayload;
                }
            }
        }
    }
}
```

<https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/Shaders/Passes/Culling/PointLight/PointLightShadowMeshCulling.comp>

Shadow XX.

GPU-Driven Point Light Shadow Rendering

- ▶ Once the **draw commands and (Entity, Light, Face) instance data** are generated, point light shadows use the same GPU-driven rendering architecture.
- ▶ The instance payload identifies the **point light and cubemap face**, allowing all required faces to share the same batched draw structure.
- ▶ The vertex shader resolves the light's shadow data and selects the **atlas sub-region corresponding to the active face**.
- ▶ The geometry is projected directly into the selected **cubemap face region** of the shared shadow atlas.
- ▶ No additional rendering architecture is required per face the existing **GPU-driven bucket, indirect draw and instance system** already supports this structure.
- ▶ A single draw command can therefore render geometry for **multiple point lights and their active faces**, completing the point light shadow map generation.



<https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/Shaders/Passes/Shadow/PointLight/PointLightShadowTraditional.vert>

Shadow XXI.

16-Bucket Rendering & Alpha-Test Separation

- ▶ The **16-bucket material system** integrates naturally with GPU-driven shadow rendering, since each bucket can be rendered through a single draw command.
- ▶ **Alpha-tested and opaque geometry are separated into dedicated buckets**, allowing their shaders to follow different execution paths.
- ▶ Opaque shadow casters require **no material lookup or alpha evaluation**, keeping the shader path minimal.
- ▶ The opaque path can therefore fully benefit from **early fragment testing**, avoiding unnecessary fragment processing.
- ▶ Alpha-tested materials use their dedicated shader variant, performing the **material lookup, texture sampling and alpha discard** only where required.
- ▶ This separation keeps the shadow pass **GPU-driven while avoiding the cost of alpha testing for the vast majority of opaque shadow casters**.

```
#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
}
```

<https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/Shader/Passes/Shadow/DirectionLight/DirectionLightShadow.frag>

Shadow XXII.

Shadow Sampling & Depth Representation

- ▶ The shadow atlas stores the standard **non-linear Vulkan depth**, rather than a linear distance representation.
- ▶ The world-space position is first projected into the light's **shadow view-projection space**, producing the corresponding NDC depth.
- ▶ Because the stored depth is non-linear, the **shadow bias** is calculated in **linear light-space distance** and then converted back to Vulkan's non-linear depth.
- ▶ A **shadow sampler with PCF** is used to perform filtered depth comparisons, producing soft shadow boundaries instead of a binary result.
- ▶ The sampling coordinates are transformed into the light's assigned **shadow atlas region**, with half-texel clamping preventing samples from leaking into neighboring regions.
- ▶ The final shadow factor is obtained from the filtered depth comparisons and directly modulates the light contribution.

```
vec3 CalculateSpotLightShadow(const uint64_t spotLightShadowDataBufferAddr, const uint64_t spotLightShadowSparseMapBufferAddr,
uint lightEntityIndex, vec3 worldPos, vec3 normal, vec3 lightDir, sampler2DShadow shadowAtlas
) {
    if (spotLightShadowSparseMapBufferAddr == 0) return vec3(1.0);

    uint shadowDenseIndex = GET_SPARSE_INDEX(spotLightShadowSparseMapBufferAddr, lightEntityIndex);
    if (shadowDenseIndex == INVALID_INDEX) return vec3(1.0);

    SpotLightShadowComponent shadowComp = GET_SPOT_LIGHT_SHADOW(spotLightShadowDataBufferAddr, shadowDenseIndex);

    // 1. Project to clip space
    vec4 clipPos = shadowComp.viewProj * vec4(worldPos, 1.0);
    if (clipPos.w <= 0.0) return vec3(1.0);

    vec3 ndc = clipPos.xyz / clipPos.w;
    if (ndc.z < 0.0 || ndc.z > 1.0 || ndc.x < -1.0 || ndc.x > 1.0 || ndc.y < -1.0 || ndc.y > 1.0) {
        return vec3(1.0);
    }

    float near = shadowComp.planes.x;
    float far = shadowComp.planes.y;
    float linearDist = clipPos.w;

    float NoL = clamp(dot(normal, lightDir), 0.0, 1.0);
    float tanTheta = sqrt(1.0 - NoL * NoL) / (NoL + 0.0001);

    float constantBias = 0.1;
    float slopeBias = 0.2;
    float linearBias = constantBias + slopeBias * tanTheta;
    float biasedDist = max(linearDist - linearBias, near + 0.001);

    // Convert biased linear distance back to non-linear Vulkan depth [0, 1]
    float currentDepth = (far / (far - near)) - (far * near) / ((far - near) * biasedDist);

    // 3. NDC to Atlas UV mapping
    vec2 uv = ndc.xy * 0.5 + 0.5;
    vec4 rect = shadowComp.atlasRect;
    uv = uv * rect.zw + rect.xy;

    // Half-texel clamp to prevent atlas bleeding
    vec2 texelSize = 1.0 / vec2(textureSize(shadowAtlas, 0));
    vec2 minUV = rect.xy + (texelSize * 0.5);
    vec2 maxUV = rect.xy + rect.zw - (texelSize * 0.5);
    uv = clamp(uv, minUV, maxUV);

    // 4. 3x3 PCF filter
    float opaqueShadow = 0.0;

    for (int x = -1; x <= 1; ++x) {
        for (int y = -1; y <= 1; ++y) {
            vec2 offset = vec2(x, y) * texelSize;
            vec2 sampleUV = clamp(uv + offset, minUV, maxUV);

            // Opaque Shadow
            opaqueShadow += texture(shadowAtlas, vec3(sampleUV, currentDepth));
        }
    }

    return vec3(opaqueShadow / 9.0);
}
```

<https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/Shaders/includes/Utils/ShadowMath.glsf>

Shadow XXIII.

Shadow Intergration into Forward+ and Deferred Shading

- ▶ Shadow evaluation is integrated directly into both the **Forward+ and Deferred lighting pipelines**, reusing the same shadow atlas and shadow data.
- ▶ For each visible light, the shader resolves whether a corresponding **shadow map exists** through the shadow sparse map.
- ▶ The resulting **shadow factor multiplies the light contribution**, requiring no separate shadow-rendering path during lighting.
- ▶ In Forward+, this is performed for the lights assigned to the fragment's cluster, while Deferred shading applies the same mechanism during deferred light evaluation.

```
for (uint i = 0; i < cluster.spotLightCount && ctx.enableForwardPlusSpotLights == 1; ++i) {
    uint globalLightIndex = cluster.spotLightOffset + i;
    uint lightEntityIndex = GET_LIGHT_INDEX(ctx.forwardPlusSpotLightIndexListBufferAddr, globalLightIndex);
    uint lightDenseIndex = GET_SPARSE_INDEX(ctx.spotLightSparseMapBufferAddr, lightEntityIndex);

    vec3 spotLightPosition = GET_SPOT_LIGHT(ctx.spotLightDataBufferAddr, lightDenseIndex).position.xyz;
    vec3 lightDir = normalize(spotLightPosition - worldPos);

    vec3 shadowFactor = CalculateSpotLightShadow(
        ctx.spotLightShadowDataBufferAddr,
        ctx.spotLightShadowSparseMapBufferAddr,
        lightEntityIndex,
        worldPos,
        finalNormal,
        lightDir,
        spotLightShadowAtlas
    );

    vec3 lightContribution = SimulateSpotLight(
        ctx.spotLightDataBufferAddr, lightDenseIndex, worldPos,
        albedoAlpha.rgb, finalNormal, viewDir, finalRoughness, finalMetalness,
        ior, specularFactor, specularColor, clearcoatFactor, clearcoatRoughness, clearcoatNormal
    );

    totalRadiance += shadowFactor * lightContribution;
}
```

<https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/Shaders/Passes/Shading/ForwardPlus/Lighting/OpaqueForward.frag>

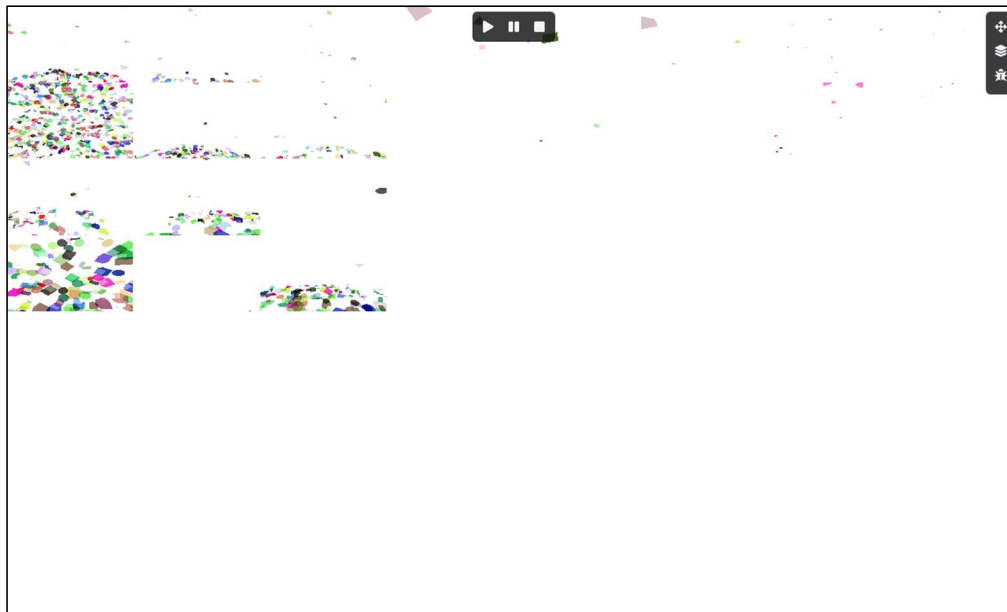
Shadow XXIV.

Colored Transparent Shadow Maps

- ▶ The existing shadow system currently handles **opaque shadow casters**, while the engine's bucket architecture already supports transparent materials.
- ▶ To **extend shadowing to transparent** objects, the same **GPU-driven rendering architecture** can be reused without changing the atlas allocation or culling pipeline.
- ▶ Instead of a D32 depth atlas, transparent shadows use an **R16G16B16A16 shadow atlas** to store the transmitted shadow color.
- ▶ The existing **draw commands, instance index buffers and bucket structure** are reused to render transparent shadow casters into their assigned atlas regions.
- ▶ This adds **64 bits per texel** compared with the original D32 atlas, while extending the shadow system to transparent materials.



Direction Light
Transparent
Shadow Color
Atlas

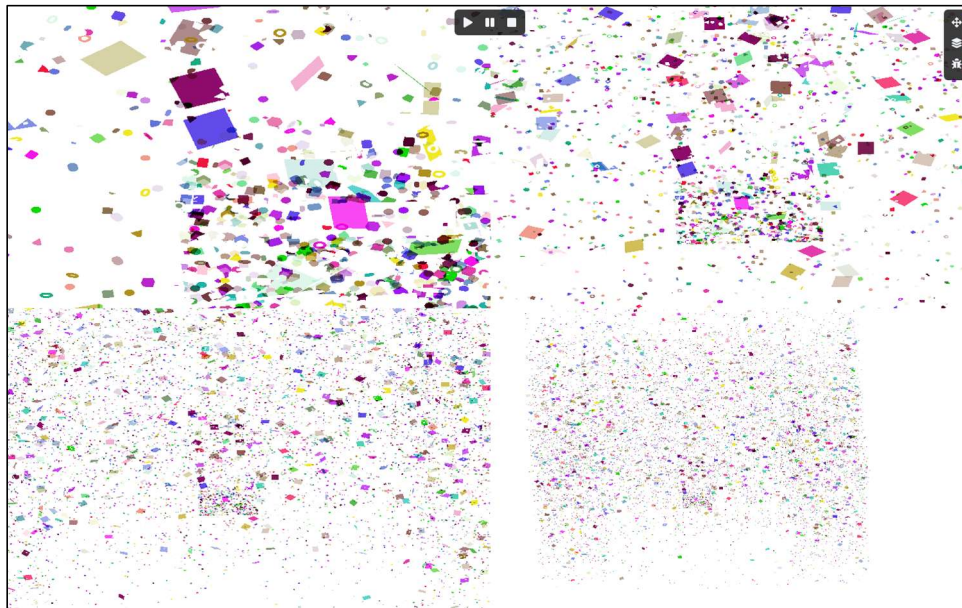


Point Light
Transparent
Shadow
Color Atlas

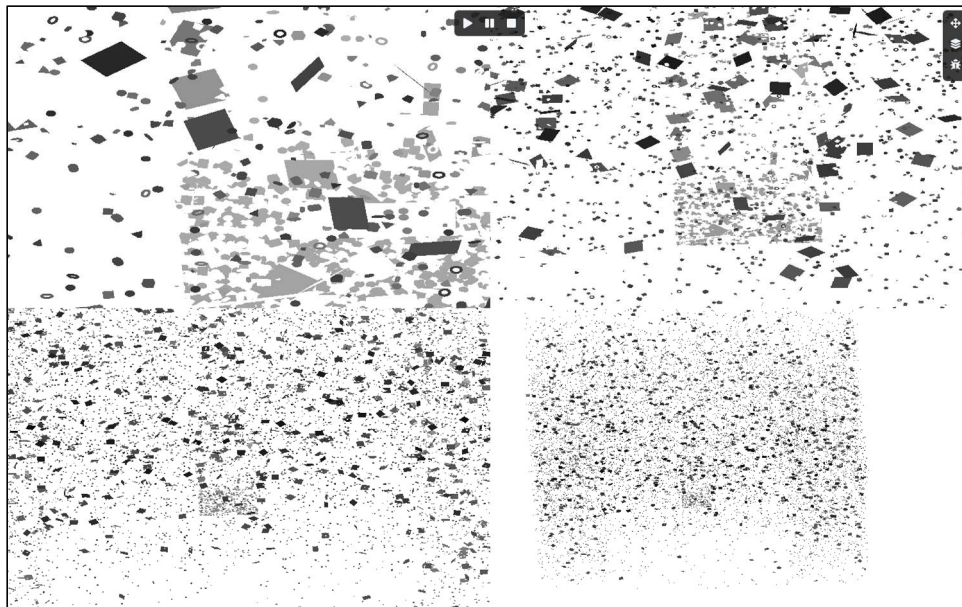
Shadow XXV.

Alpha Channel as Transparent Shadow Depth

- ▶ The **A16 channel stores the depth** of the closest transparent layer rendered into the shadow atlas.
- ▶ This depth is required to prevent a transparent object from immediately **casting its own transparent shadow onto itself** during shadow evaluation.
- ▶ The RGB channels therefore represent the transmitted shadow color, while **A stores the depth needed for visibility testing**.
- ▶ Vulkan allows the RGB and Alpha channels to use **independent blend factors and blend operations**, making this possible within a single render target.
- ▶ The alpha channel uses **VK_BLEND_OP_MIN**, so overlapping transparent fragments retain the **nearest depth value**.
- ▶ This effectively combines **colored accumulation and depth tracking in one R16G16B16A16 atlas**, without requiring a separate transparent-depth texture.



RGB Channels

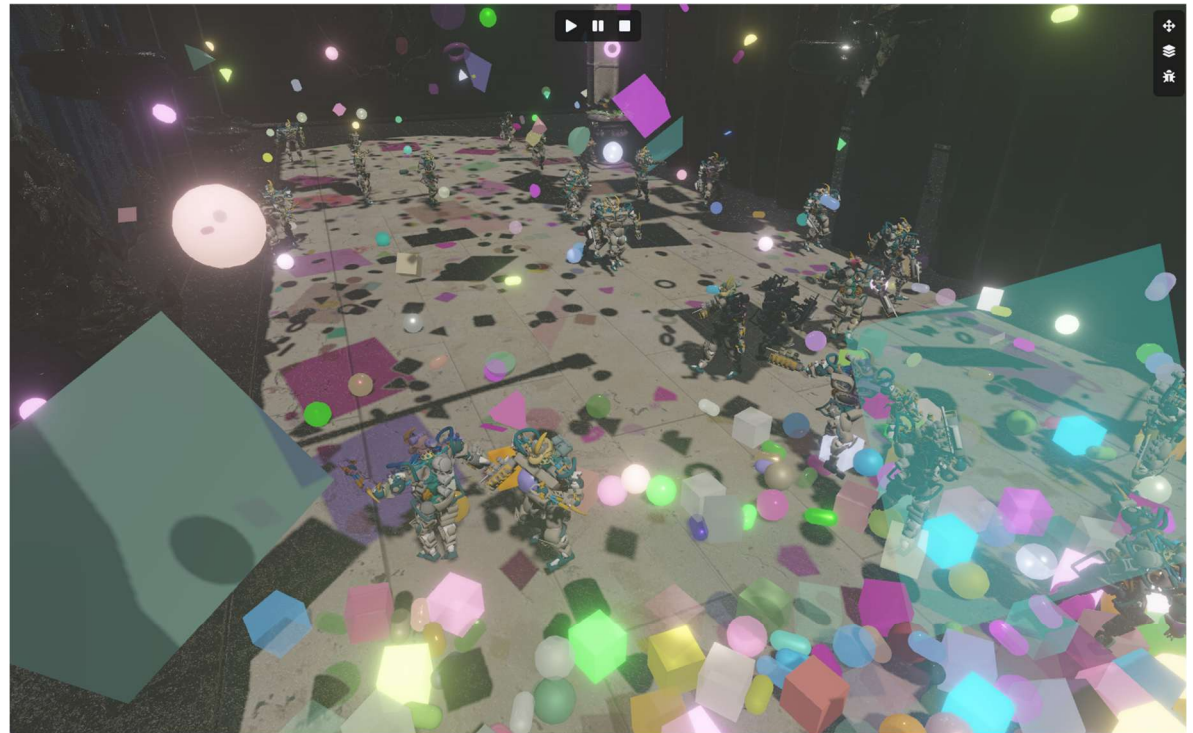


A Channel
(Depth)

Shadow XXVI.

Colored Transparent Shadow Simulation

- ▶ The shadow evaluation follows the same **projection, atlas lookup and PCF sampling** pipeline introduced for opaque shadows.
- ▶ For each PCF sample, the **opaque shadow atlas** is sampled normally using the current non-linear depth.
- ▶ The transparent shadow atlas is sampled separately, providing both the **accumulated RGB shadow color and the stored transparent depth**.
- ▶ The A16 depth is used to determine whether the transparent layer is **behind the current fragment**, only then is its RGB contribution applied.
- ▶ The **final shadow factor is returned as a vec3**, combining opaque attenuation with the colored transparent shadow contribution.
- ▶ This allows the same shadow system to produce **colored, partially transmitted shadows** while retaining the existing GPU-driven atlas architecture.



```
for (int x = -1; x <= 1; ++x) {
    for (int y = -1; y <= 1; ++y) {
        vec2 offset = vec2(x, y) * texelSize;
        vec2 sampleUV = clamp(uv + offset, minUV, maxUV);

        // Opaque Shadow
        opaqueShadow += texture(shadowAtlas, vec3(sampleUV, currentDepth));

        // Transparent Shadow
        vec4 transData = textureLod(shadowColorAtlas, sampleUV, 0.0);
        if (currentDepth > transData.a) {
            transparentShadow += transData.rgb;
        } else {
            transparentShadow += vec3(1.0);
        }
    }
}

return vec3(opaqueShadow / 9.0) * (transparentShadow / 9.0);
```

Synapse Engine: Beyond Rendering and Architecture

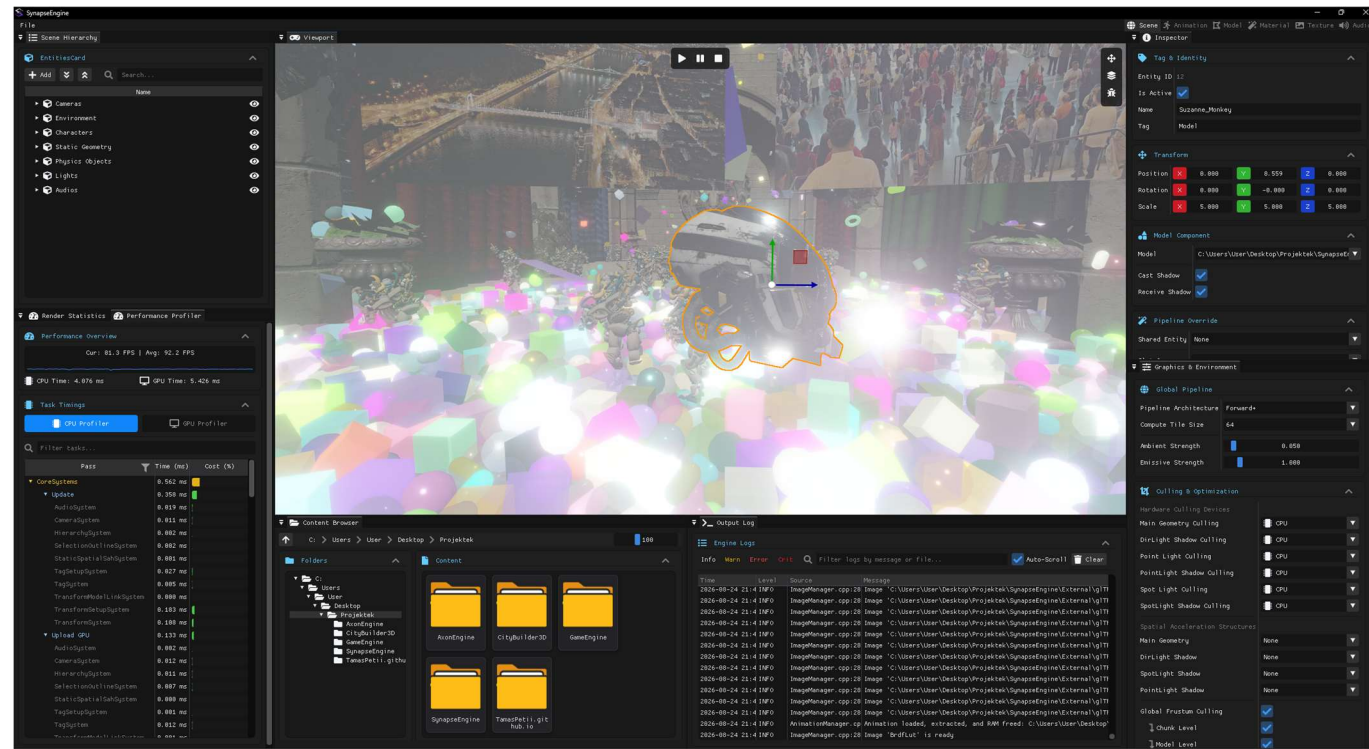
Created By: Péter Tamás
Github: <https://github.com/TamasPetii/SynapseEngine>

2026

Beyond Rendering and Architecture I.

Scene Workspace

- ▶ Provides the central workspace for **building, inspecting and controlling scenes**.
- ▶ The **Entity Hierarchy** allows entities to be created, removed and organized, while the Inspector exposes their components and properties.
- ▶ The **Inspector** allows components to be added, removed and individually customized, including properties such as **Transform, Model, Materials and rendering settings**.
- ▶ The integrated **Performance Profiler** provides a detailed CPU/GPU breakdown of the running engine and individual pipeline stages.
- ▶ Global scene and rendering settings expose **CPU/GPU-driven execution, culling and rendering configuration** directly from the editor.
- ▶ The workspace integrates a **Content Browser and filesystem view** for managing engine assets and project resources.
- ▶ An integrated **Engine Logger** provides runtime diagnostics and allows engine activity to be monitored directly alongside the scene.

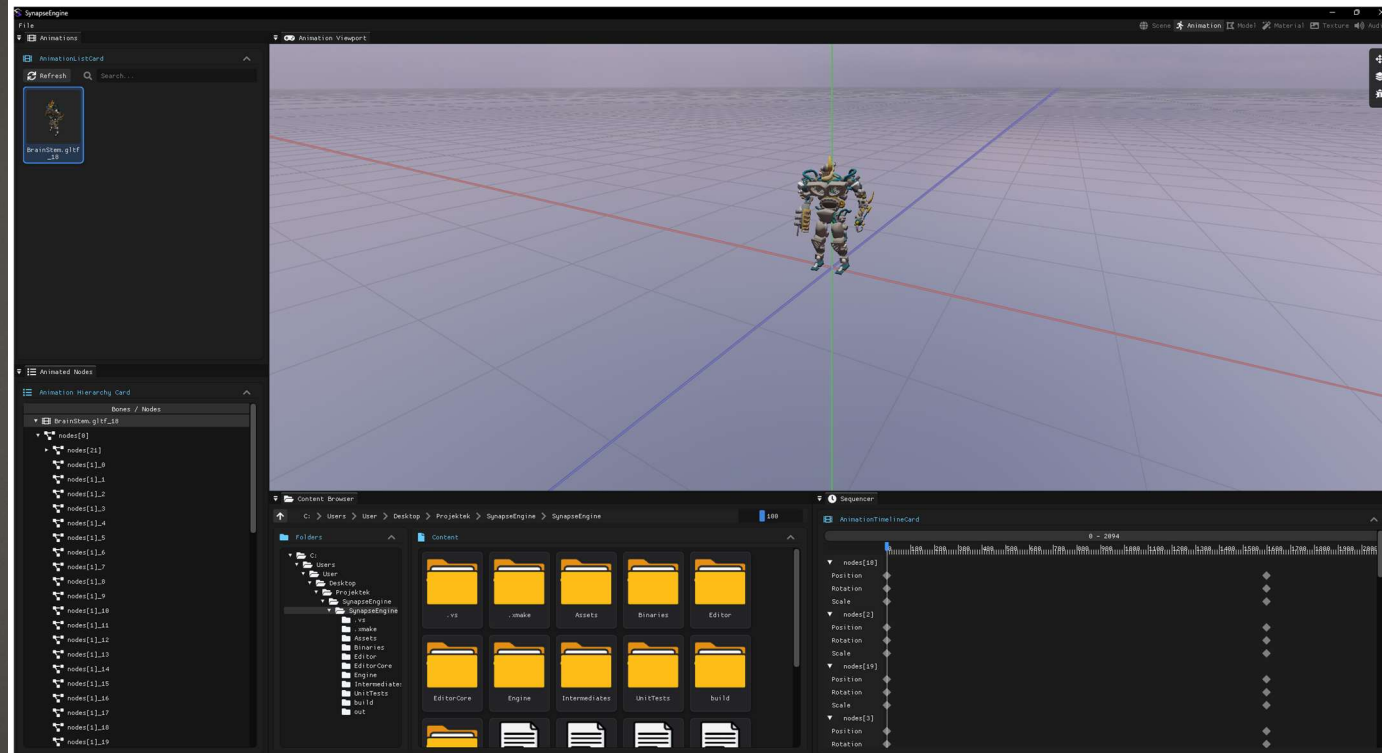


<https://github.com/TamasPetii/SynapseEngine/tree/main/SynapseEngine/Editor/Workspace/SceneWorkspace>

Beyond Rendering and Architecture II.

Animation Workspace

- ▶ Provides a dedicated workspace for **visualizing and previewing skeletal animations**.
- ▶ Displays animations as **visual preview cards**, with the animated character shown directly inside each panel for quick inspection.
- ▶ Includes a preliminary **animation timeline** for playback, navigation and inspecting animation duration.
- ▶ The current implementation is intentionally **basic and focused on visualization**: animation editing is not yet supported.
- ▶ **Skeleton wireframe visualization, per-bone keyframe editing and IK controls** are planned for future development.
- ▶ The long-term goal is to evolve the workspace into a complete **animation authoring environment** for creating and editing animations directly in the engine.

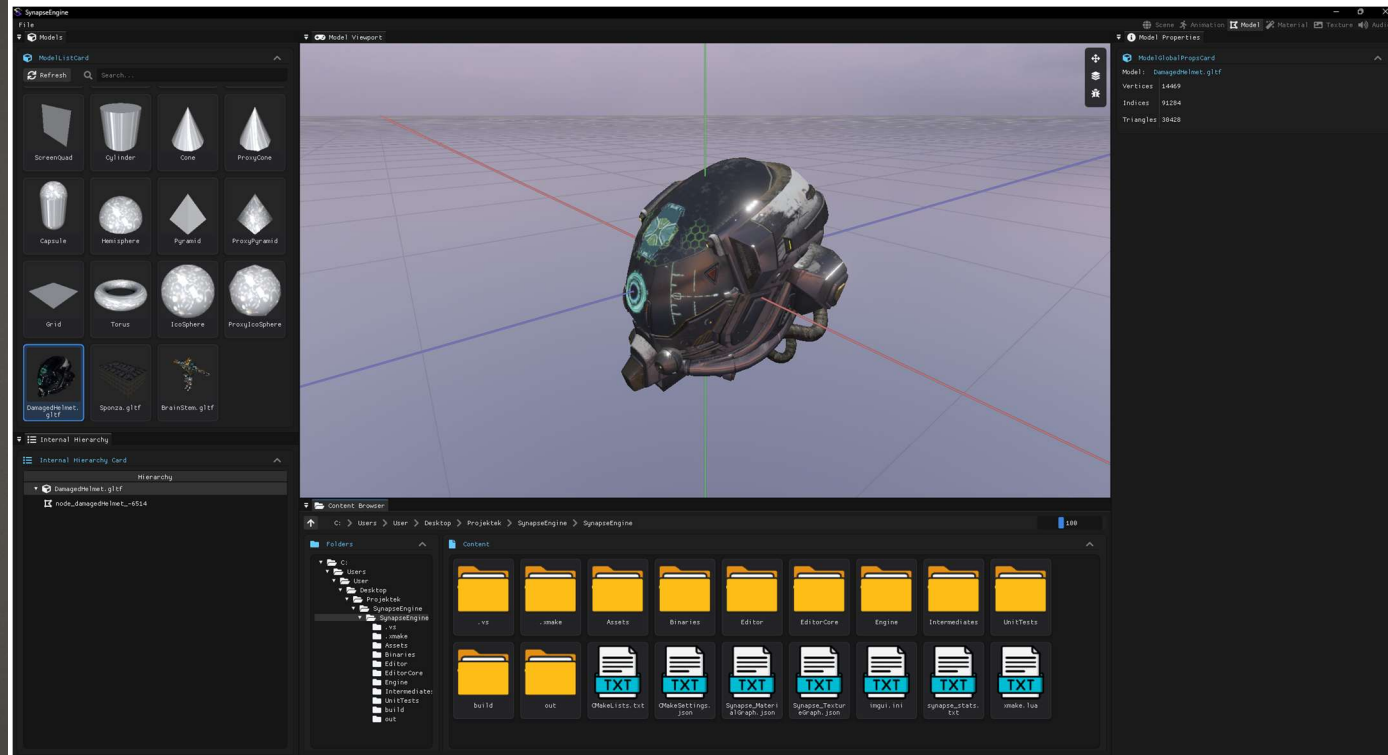


<https://github.com/TamasPetii/SynapseEngine/tree/main/SynapseEngine/Editor/Workspace/AnimationWorkspace>

Beyond Rendering and Architecture III.

Model Workspace

- ▶ Provides a dedicated workspace for **inspecting and previewing imported 3D models**.
- ▶ Displays the model's internal **node hierarchy**, making its structure and relationships directly inspectable.
- ▶ Exposes model-level and node-level **parameters and metadata** for detailed asset inspection.
- ▶ Includes an interactive **3D model preview** with a reference grid for spatial orientation.
- ▶ Provides **preview textures** and visual asset information directly within the workspace.
- ▶ Designed as the foundation for future **model editing and asset preparation workflows** directly inside the engine.

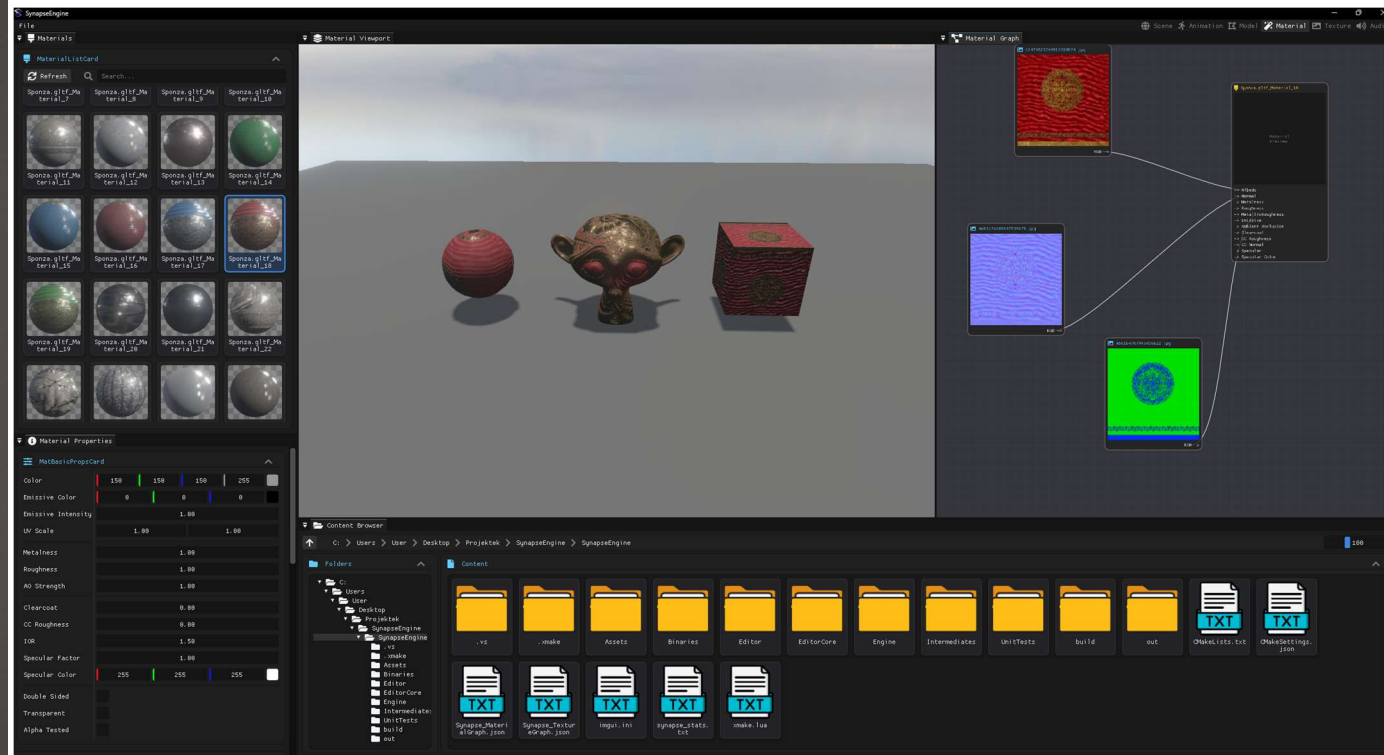


<https://github.com/TamasPetii/SynapseEngine/tree/main/SynapseEngine/Editor/Workspace/ModelWorkspace>

Beyond Rendering and Architecture IV.

Material Workspace

- ▶ Provides a dedicated workspace for **inspecting and configuring globally loaded materials**.
- ▶ Each material is displayed on a **high-quality sphere preview**, including specular response, reflections and bloom for a more realistic representation.
- ▶ Exposes the complete set of **material parameters, textures, samplers and color properties** for direct editing.
- ▶ A dedicated preview scene contains multiple geometries, allowing the material to be evaluated under **different shapes and viewing conditions**.
- ▶ **Changes are reflected immediately** in both the material preview and the surrounding **3D scene**.
- ▶ Includes an initial **material node editor**, providing the foundation for a future node-based material authoring workflow.

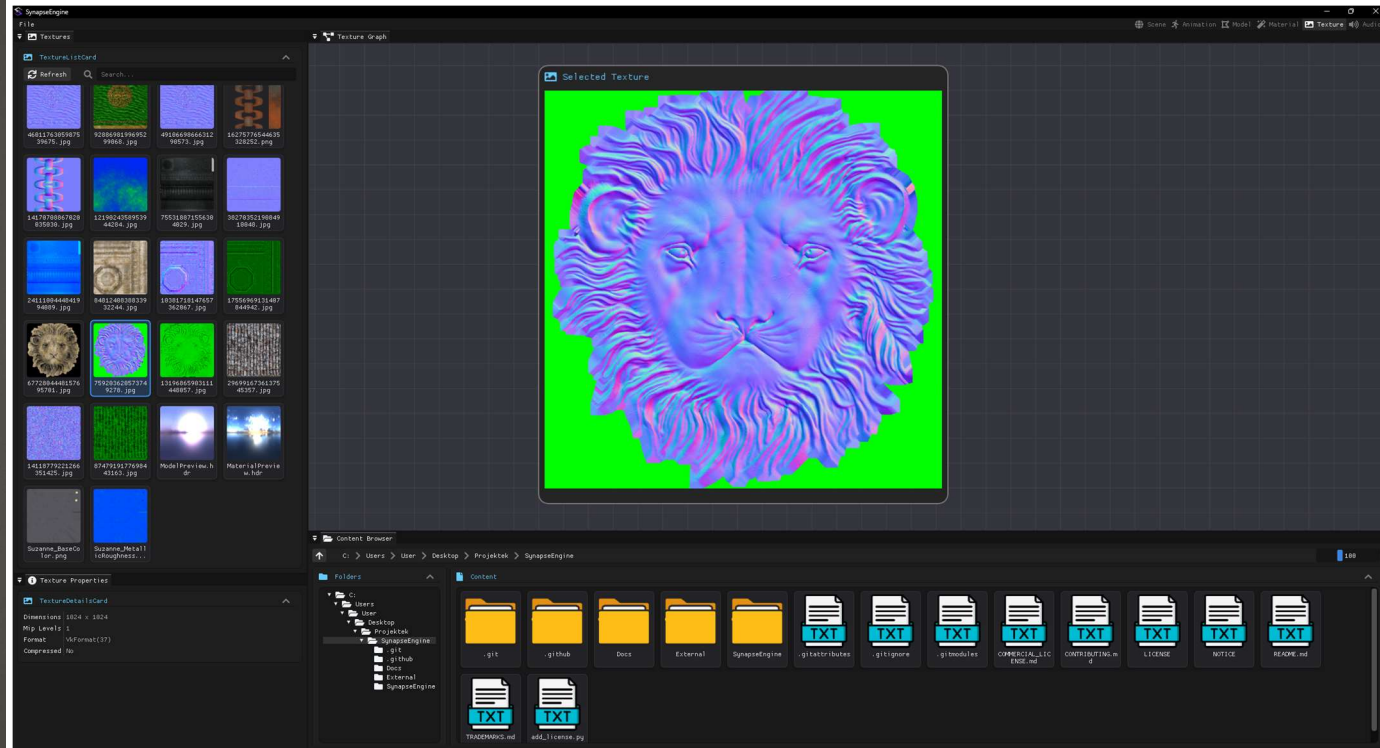


<https://github.com/TamasPetii/SynapseEngine/tree/main/SynapseEngine/Editor/Workspace/MaterialWorkspace>

Beyond Rendering and Architecture V.

Texture Workspace

- ▶ Provides a dedicated workspace for **inspecting all loaded and procedurally generated textures**.
- ▶ Displays available textures together with their **resource parameters and metadata**.
- ▶ The selected texture is visualized at **high resolution** directly inside the workspace.
- ▶ Includes an initial **texture node editor** for inspecting the texture processing and generation workflow.
- ▶ The current implementation is still **in an early development stage**, focused primarily on texture visualization and inspection.
- ▶ Future development will enable **node-based texture editing and procedural texture generation** directly within the engine.

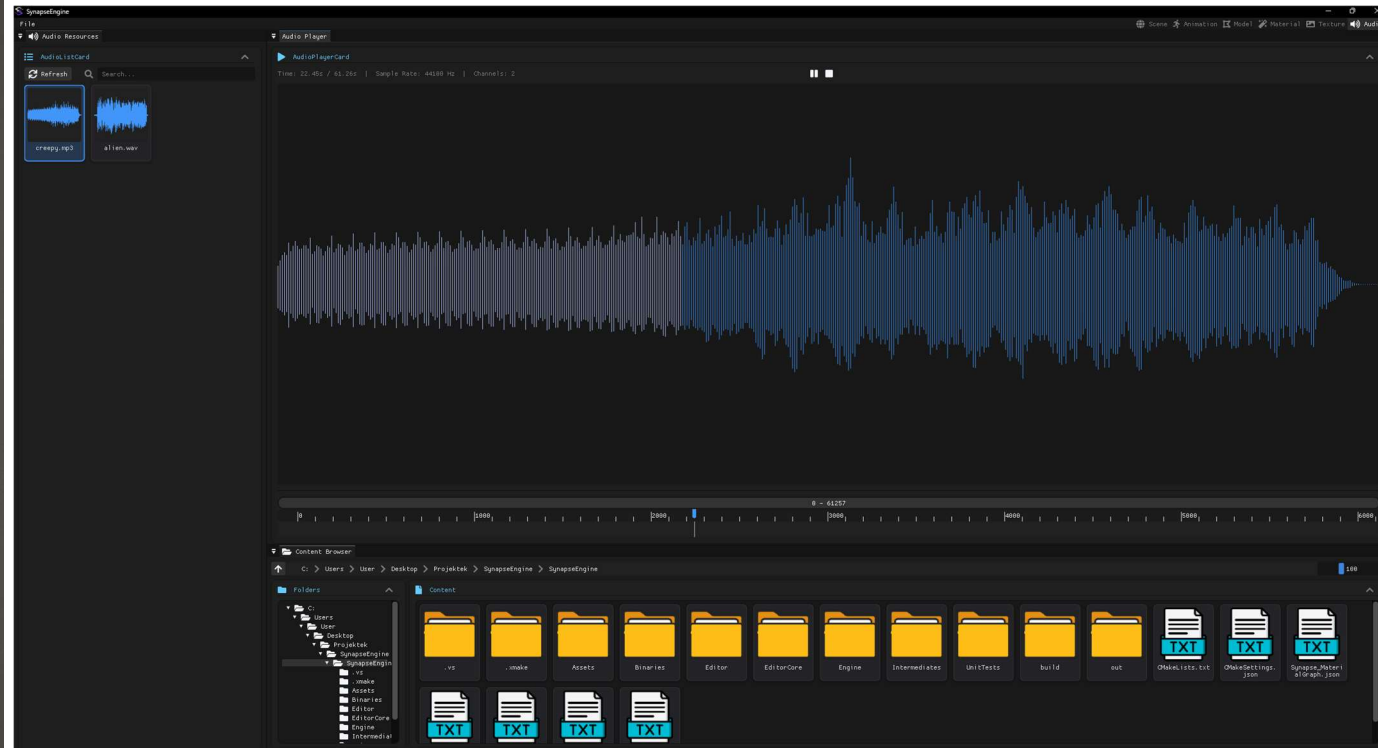


<https://github.com/TamasPetii/SynapseEngine/tree/main/SynapseEngine/Editor/Workspace/TextureWorkspace>

Beyond Rendering and Architecture VI.

Audio Workspace

- ▶ Provides a dedicated workspace for **inspecting all loaded audio assets**.
- ▶ Displays the **waveform** of the selected sound for visual analysis of its audio data.
- ▶ Supports **audio playback and timeline scrubbing** directly within the workspace.
- ▶ Allows sounds to be quickly previewed without leaving the engine editor.
- ▶ The current implementation is primarily focused on **audio visualization and playback**.
- ▶ Future development will extend the workspace with **direct audio editing and processing capabilities**.
- ▶ A similar dedicated workspace is planned for **video assets**, following the same inspection and playback-oriented workflow. The video workspace is **not yet implemented** and remains part of the planned editor functionality.

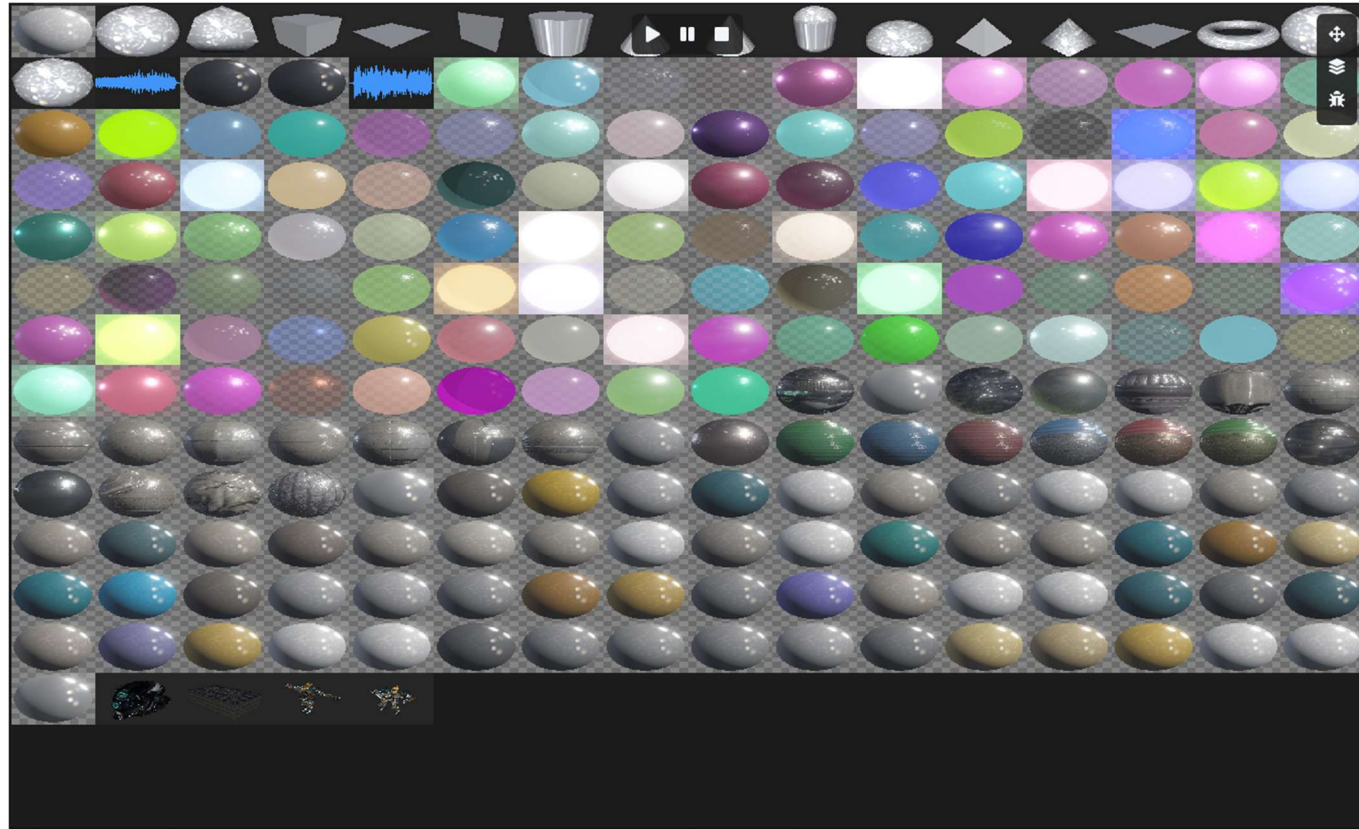


<https://github.com/TamasPetii/SynapseEngine/tree/main/SynapseEngine/Editor/Workspace/AudioWorkspace>

Beyond Rendering and Architecture VII.

Preview Rendering

- ▶ Uses a single **2K preview texture atlas**, divided into **128×128 regions** for engine-generated previews.
- ▶ Material, model, animation and other editor previews are rendered directly into dedicated atlas regions.
- ▶ Instead of allocating separate textures for every preview, the engine maintains a **single shared GPU resource**.
- ▶ ImGui receives the corresponding **atlas region** and displays only the required preview area.
- ▶ This provides a unified and efficient way to manage the large number of small **engine-generated preview images**.
- ▶ The same system can be extended to additional preview types without introducing separate rendering resources.

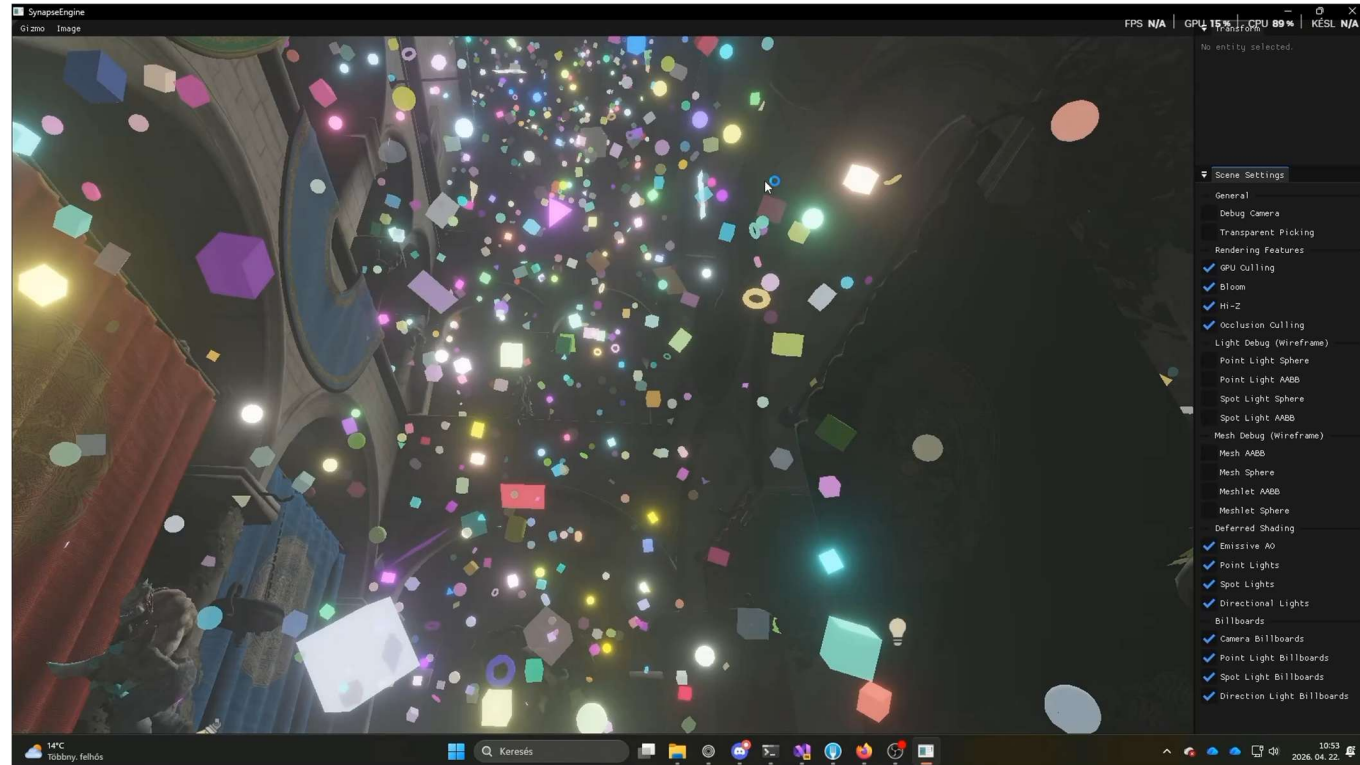


<https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/Render/Passes/Preview/MaterialPreviewPass.cpp>

Beyond Rendering and Architecture VIII.

Physics Simulation

- ▶ The engine currently uses **Jolt Physics** as its primary physics backend, selected for its data-oriented architecture and efficient parallel execution.
- ▶ **Box3D integration** is also available, providing an alternative physics backend within the engine architecture.
- ▶ The current implementation primarily focuses on **rigid-body simulation** and collision handling.
- ▶ Supported shape colliders currently include **convex, mesh, box, sphere and capsule** geometries.
- ▶ Physics objects are integrated with the engine's **entity/component architecture**, allowing simulation data to participate directly in the engine workflow.
- ▶ The physics system is still under active development, with **additional simulation and collision features planned for future iterations**.



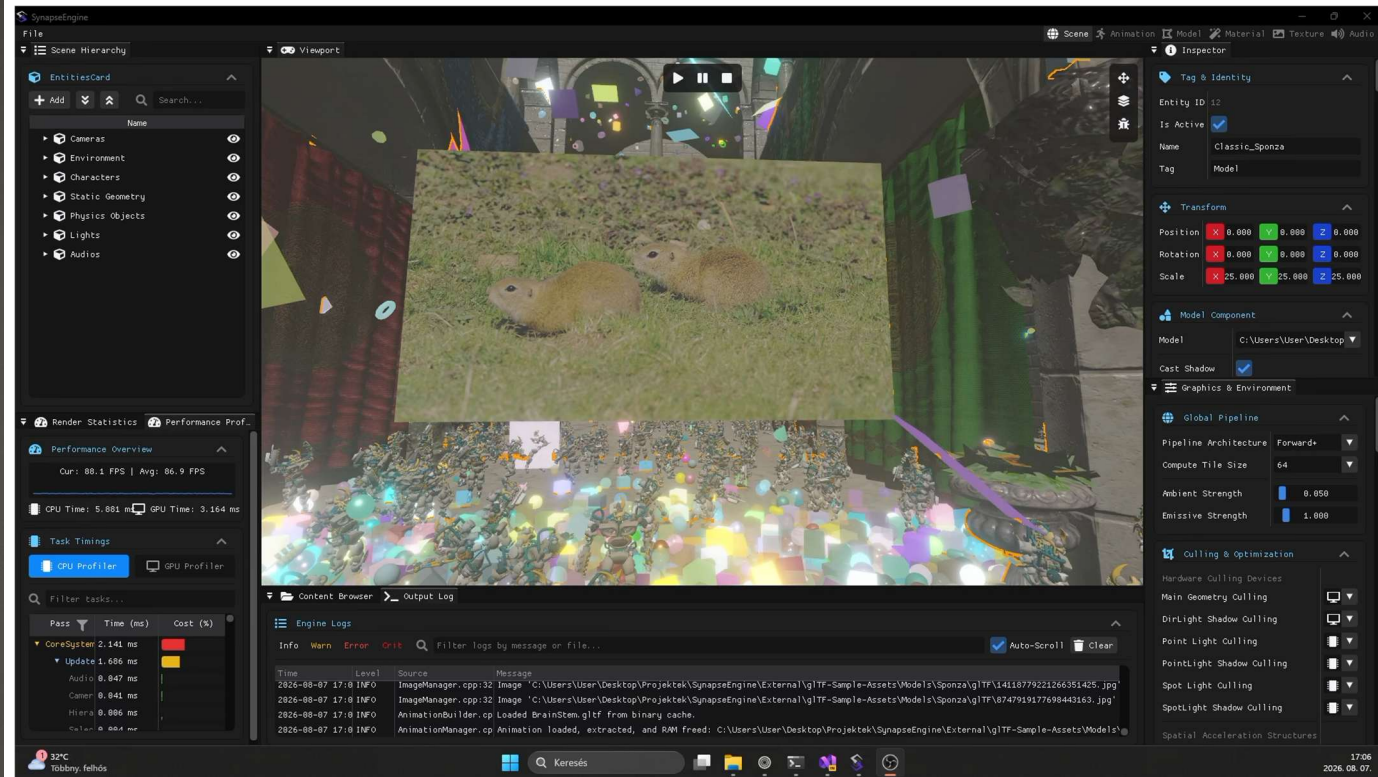
<https://youtu.be/BfT6ExKP9Ik>

<https://github.com/TamasPetii/SynapseEngine/tree/main/SynapseEngine/Engine/Physics/Jolt>

Beyond Rendering and Architecture IX.

Video Streaming

- ▶ Supports **CPU-side video streaming through FFmpeg**, providing a flexible software decoding path.
- ▶ Also implements **native Vulkan Video decoding**, enabling video frames to be decoded directly through the GPU pipeline.
- ▶ Video streams are exposed through the same **texture descriptor abstraction** used by regular 2D textures. From the material and rendering systems' perspective, a streamed video therefore behaves like a **normal texture resource**.
- ▶ Decoded frames are continuously uploaded or decoded into the texture while the rendering pipeline remains unchanged.
- ▶ Video streaming and texture updates run **independently from the rendering path**, allowing the next frame to be streamed in the background without requiring special material or shader handling.



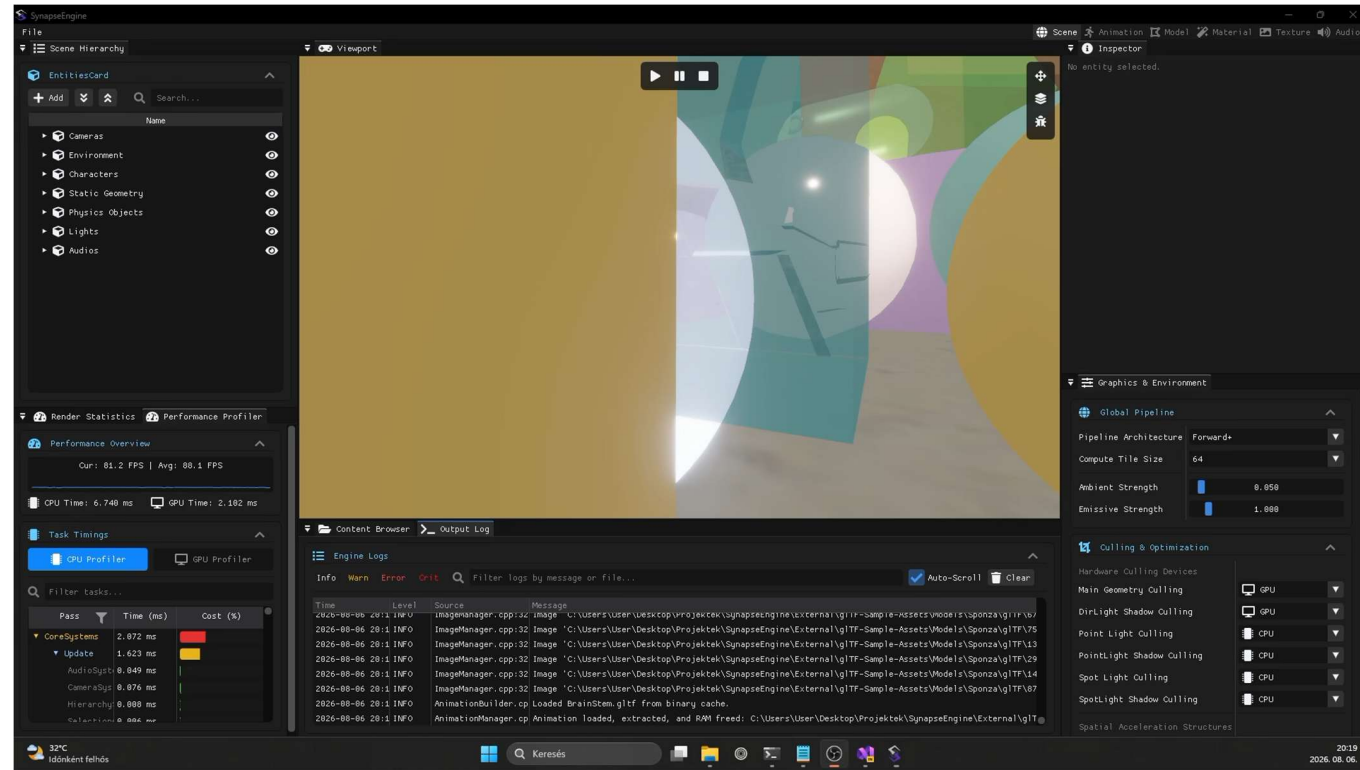
https://youtu.be/9ZgdC-Fxi_w

<https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/Video/Uploader/VulkanGpuVideoUploader.cpp>

Beyond Rendering and Architecture X.

Audio Engine

- ▶ Uses **miniaudio** as the underlying audio backend for lightweight and cross-platform audio playback.
- ▶ Supports both **2D audio playback** for conventional sounds and **spatial 3D audio** for immersive scenes.
- ▶ The **AudioSource component** defines and controls audio emitters attached to entities.
- ▶ The **AudioListener component** represents the listener's position and orientation within the scene.
- ▶ Spatial audio parameters can be evaluated from the relationship between AudioSource and AudioListener entities.
- ▶ The audio system is integrated directly into the engine's **entity/component architecture**, allowing audio to participate naturally in the scene workflow.



<https://youtu.be/5Rdv-zo7o9I>

<https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/Audio/Engine/MiniAudioEngine.cpp>

Beyond Rendering and Architecture XI.

Billboard Rendering

- ▶ Provides **editor billboards** for visualizing non-geometric scene entities such as lights, cameras and audio sources.
- ▶ Billboard icons are rendered directly in the **3D scene**, making the position of these entities immediately visible during editing.
- ▶ **Billboards are interactive**, allowing the corresponding entity to be selected directly by clicking its representation.
- ▶ Billboard geometry is generated in the **vertex shader**, avoiding the overhead of a geometry-shader-based approach.
- ▶ This provides a lightweight visualization layer for **editor-only scene objects** without affecting the underlying scene geometry.

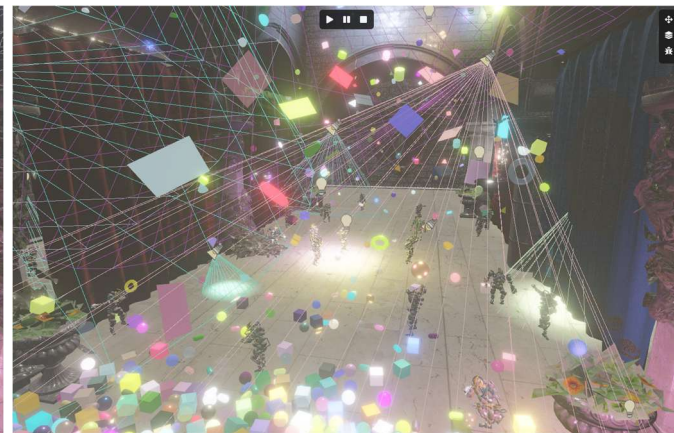
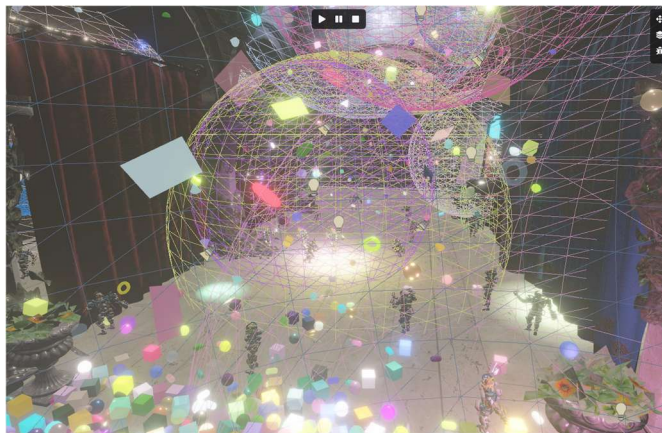
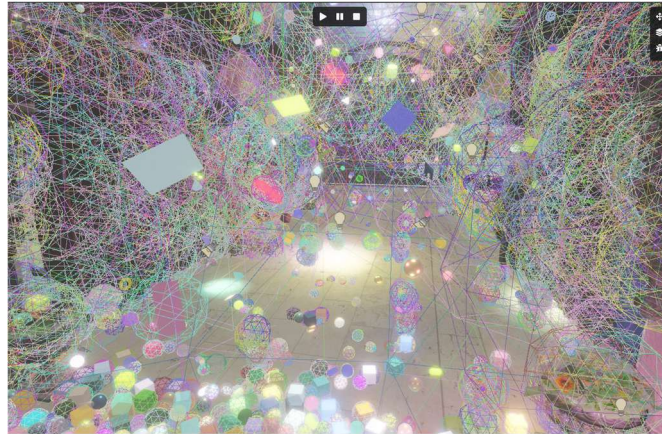


<https://github.com/TamasPetii/SynapseEngine/blob/main/SynapseEngine/Engine/Shader/Passes/Billboard/Billboard.vert>

Beyond Rendering and Architecture XII.

Wireframe Debug Visualization

- ▶ **Light and collider volumes** can be visualized directly in the editor, making spatial relationships and intersections easy to inspect.
- ▶ Wireframe visualization can be **enabled or disabled per debug view**, allowing the same scene to be inspected without affecting the normal rendering.
- ▶ **Meshlet colliders** can also be visualized, including their **AABB, sphere, and cone bounds** used by the GPU-driven culling system.
- ▶ Instead of generating the original meshlet geometry, the **mesh shader generates the corresponding collider geometry** directly on the GPU.
- ▶ The generated geometry is kept within the **256-vertex mesh shader limit**, while still providing an efficient representation for debugging.



Beyond Rendering and Architecture XIII.

Vulkan Extensions

- ▶ The engine is built around a **modern Vulkan extension stack**, avoiding legacy rendering abstractions wherever possible.
- ▶ **Shader Objects** eliminate the need for monolithic, hard-coded graphics pipelines, enabling a much more flexible and modular **shader-driven rendering architecture**.
- ▶ **Dynamic State** is used extensively, minimizing pipeline-state permutations and allowing rendering state to be configured directly at draw time.
- ▶ **Dynamic Rendering** replaces the traditional render-pass/framebuffer model, simplifying render-pass management and making rendering passes significantly more flexible.
- ▶ **Descriptor Buffer** provides a low-level, GPU-friendly descriptor management model, reducing the overhead and complexity of traditional descriptor-set management.
- ▶ **Descriptor Indexing** enables bindless resource access, allowing shaders to dynamically index large arrays of textures, buffers and other GPU resources without per-resource binding overhead.

```
static std::vector<const char*> GetRequiredExtensions() {  
    return {  
        VK_KHR_SWAPCHAIN_EXTENSION_NAME,  
        VK_EXT_SWAPCHAIN_MAINTENANCE_1_EXTENSION_NAME,  
        VK_KHR_PUSH_DESCRIPTOR_EXTENSION_NAME,  
        VK_EXT_MESH_SHADER_EXTENSION_NAME,  
        VK_EXT_SHADER_OBJECT_EXTENSION_NAME,  
        VK_EXT_DESCRIPTOR_BUFFER_EXTENSION_NAME,  
        VK_EXT_DESCRIPTOR_INDEXING_EXTENSION_NAME,  
        VK_EXT_EXTENDED_DYNAMIC_STATE_EXTENSION_NAME,  
        VK_EXT_EXTENDED_DYNAMIC_STATE_2_EXTENSION_NAME,  
        VK_EXT_EXTENDED_DYNAMIC_STATE_3_EXTENSION_NAME,  
        VK_EXT_MEMORY_BUDGET_EXTENSION_NAME,  
        VK_KHR_DYNAMIC_RENDERING_EXTENSION_NAME,  
        VK_KHR_SYNCHRONIZATION_2_EXTENSION_NAME,  
        VK_KHR_BUFFER_DEVICE_ADDRESS_EXTENSION_NAME,  
        VK_EXT_SAMPLER_FILTER_MINMAX_EXTENSION_NAME,  
        VK_EXT_MUTABLE_DESCRIPTOR_TYPE_EXTENSION_NAME,  
        VK_KHR_VULKAN_MEMORY_MODEL_EXTENSION_NAME,  
        VK_EXT_SUBGROUP_SIZE_CONTROL_EXTENSION_NAME,  
        VK_EXT_DESCRIPTOR_HEAP_EXTENSION_NAME,  
        VK_EXT_DEVICE_GENERATED_COMMANDS_EXTENSION_NAME,  
        VK_KHR_VIDEO_QUEUE_EXTENSION_NAME,  
        VK_KHR_VIDEO_DECODE_QUEUE_EXTENSION_NAME,  
        VK_KHR_VIDEO_DECODE_H264_EXTENSION_NAME  
        /*, VK_AMD_SHADER_ENQUEUE_EXTENSION_NAME */  
    };  
}
```

Synapse Engine: Future Directions

Created By: Péter Tamás

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

2026

Future Directions I.

Scripting

- ▶ The engine will introduce a dedicated **scripting layer**, with the exact architecture still under active investigation.
- ▶ The primary low-level approach is **native C++ DLL scripting**, allowing scripts to interact directly with the engine's native systems.
- ▶ A higher-level scripting framework such as **AngelScript** could provide a faster and more flexible development workflow on top of the native layer.
- ▶ This could enable a **dual scripting workflow**: native C++ for maximum performance and engine-level access, with a managed scripting layer for faster iteration during development.
- ▶ The scripting architecture is still in the **research and exploration phase**, with the final execution model and API yet to be determined.

```
#ifdef EXPORT_SCRIPT_DLL
#define SCRIPT_API __declspec(dllexport)
#else
#define SCRIPT_API __declspec(dllimport)
#endif

// Engine Core Interface
class ENGINE_API Script {
public:
    virtual void OnStart() {}
    virtual void OnUpdate(float deltaTime) {}
    virtual void OnTriggerEnter(Entity collidedEntity) {}
};

// Native DLL Implementation
class SCRIPT_API PlayerScript : public Script {
public:
    PlayerScript(std::shared_ptr<Registry> reg, Entity ent) : Script(reg, ent) {}

    void OnUpdate(float deltaTime) override {
        // Maximum performance native logic
    }
};

// C-Linkage DLL Export Factory
extern "C" SCRIPT_API Script* CreateScript(std::shared_ptr<Registry> reg, Entity ent) {
    return new PlayerScript(reg, ent);
}
```

Future Directions II.

DirectX12 & Metal

- ▶ A major future direction is expanding SynapseEngine beyond Vulkan with support for **DirectX 12 and Metal**.
- ▶ Neither API has been used extensively yet, making this both a development goal and an opportunity to deepen the engine's **cross-platform graphics knowledge**.
- ▶ A major architectural step will be introducing a **custom Rendering Hardware Interface (RHI)** layer instead of relying on an existing solution such as **NVRHI**.
- ▶ **DirectX 12** is particularly interesting because of its modern GPU-driven features, especially **Work Graphs**, which could provide new approaches to GPU-driven execution.
- ▶ The RHI will ultimately allow the same high-level rendering architecture to target **multiple graphics APIs**, while keeping their unique capabilities accessible.



Future Directions III.

Slang Shader Support

- ▶ The engine will introduce full **Slang support alongside GLSL**, providing a modern alternative for shader development.
- ▶ After extensive work with GLSL, the goal is to explore a **more expressive and modern shader language** and broaden the engine's shader architecture knowledge.
- ▶ Slang is particularly interesting because it can target **multiple graphics backends**, including DirectX 12, making it a strong candidate for the future RHI.
- ▶ The migration will require rewriting **hundreds of existing shaders**, making this one of the larger future development tasks.



Future Directions IV.

Hardware Ray Tracing

- ▶ The engine will introduce **hardware-accelerated ray tracing** as an additional rendering path alongside the existing rasterization pipeline.
- ▶ Full support for **ray tracing primitives and acceleration structures** will provide the foundation for integrating ray-traced workloads into the engine.
- ▶ Ray tracing will initially complement the existing GPU-driven rasterization pipeline rather than replace it, enabling **hybrid rendering approaches**.
- ▶ This foundation will allow further exploration of techniques such as **ray-traced shadows, reflections, ambient occlusion and global illumination**.
- ▶ The long-term goal is to investigate how **hardware ray tracing can be integrated with the existing GPU-driven architecture** and selectively used where it provides the greatest benefit.

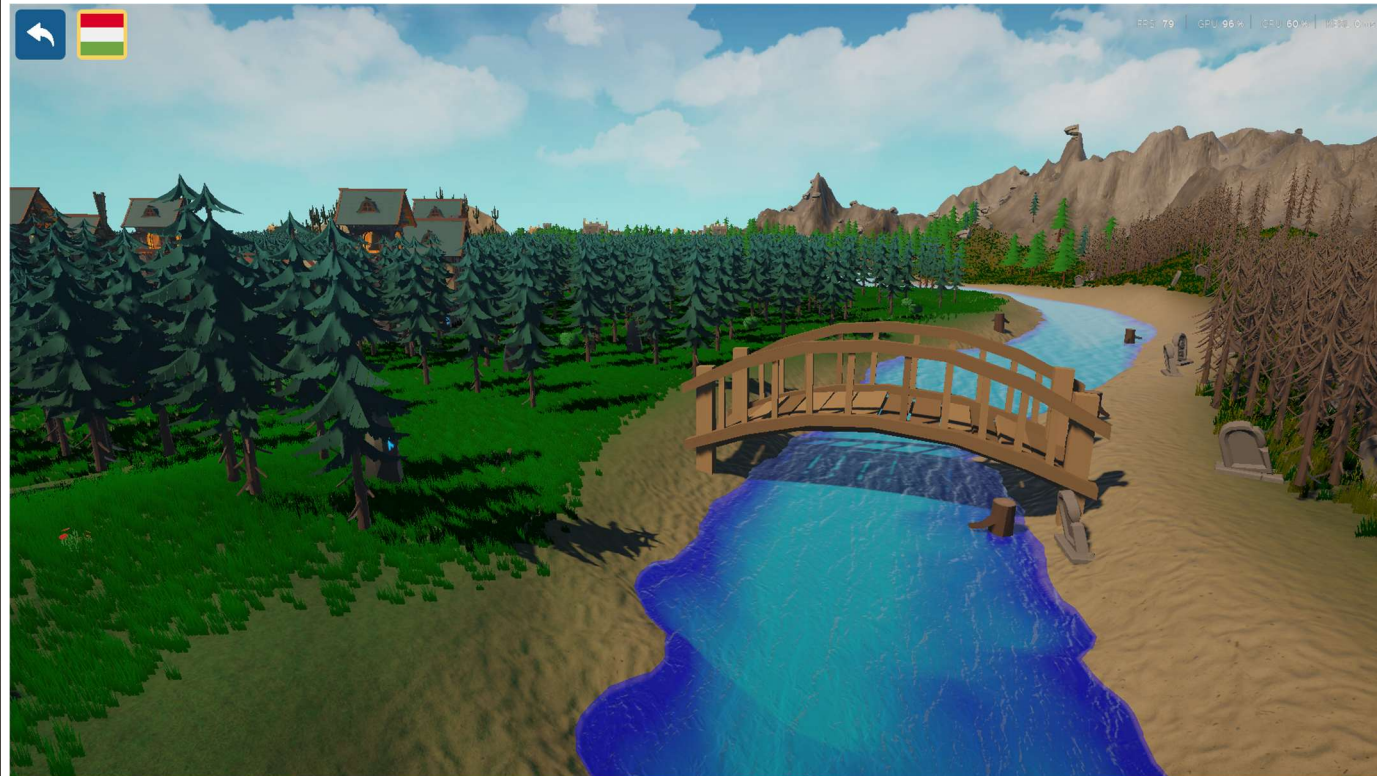


<https://www.youtube.com/watch?v=IMSuGoYcT3s>

Future Directions V.

Terrain & Vegetation

- ▶ A future research direction is **GPU-driven terrain and vegetation rendering**, extending the engine's existing GPU-driven architecture into large-scale environments.
- ▶ Terrain generation and rendering will be explored using **Task and Mesh Shaders**, with geometry generated and selected directly on the GPU.
- ▶ **Dynamic LOD selection** will adapt terrain detail to screen-space importance, allowing large environments to remain efficiently rendered.
- ▶ Vegetation rendering will focus on highly efficient **grass, bushes and other densely distributed objects**, where instance count can become extremely large.
- ▶ The exact techniques and data structures are still **largely unexplored**, so this area will require substantial research before implementation.
- ▶ The long-term goal is to develop a **scalable, GPU-driven environment rendering system** that integrates naturally with the existing SynapseEngine architecture.



Future Directions VI.

Nanite & Virtualized Geometry

- ▶ The current **GPU-driven architecture is approaching the point of diminishing returns**, making a fundamentally different rendering approach increasingly interesting.
- ▶ **Nanite and virtualized geometry** represent a completely different way of thinking about geometry management, visibility and rendering at massive geometric scales.
- ▶ The goal is not simply to reproduce Nanite, but to understand the **underlying mental model and architectural decisions** behind virtualized geometry.
- ▶ Understanding this fundamentally different approach may reveal **new techniques applicable to other parts of the engine**, just as the GPU-driven architecture led to the development of **Dual-HiZ Adaptive Clustered Forward Shading**.
- ▶ Nanite therefore represents both a **future rendering technology to investigate** and an opportunity to expand the architectural thinking behind SynapseEngine.



<https://youtu.be/eviSykqSUUw>

Future Directions VII.

Virtual Textures

- ▶ Explore **virtual texturing** to efficiently manage extremely large textures without keeping the complete texture resident in GPU memory.
- ▶ Large textures are divided into smaller **tiles/pages**, allowing only the visible and required regions to be streamed into physical memory.
- ▶ **MIP levels** provide progressively lower-detail representations, allowing texture resolution to adapt dynamically to distance and screen-space importance.
- ▶ Vulkan provides **sparse resource mechanisms** that can serve as a foundation for implementing virtualized texture memory and tile residency.
- ▶ The goal is to investigate how **virtualized resources and GPU-driven streaming** could extend SynapseEngine's resource management architecture.

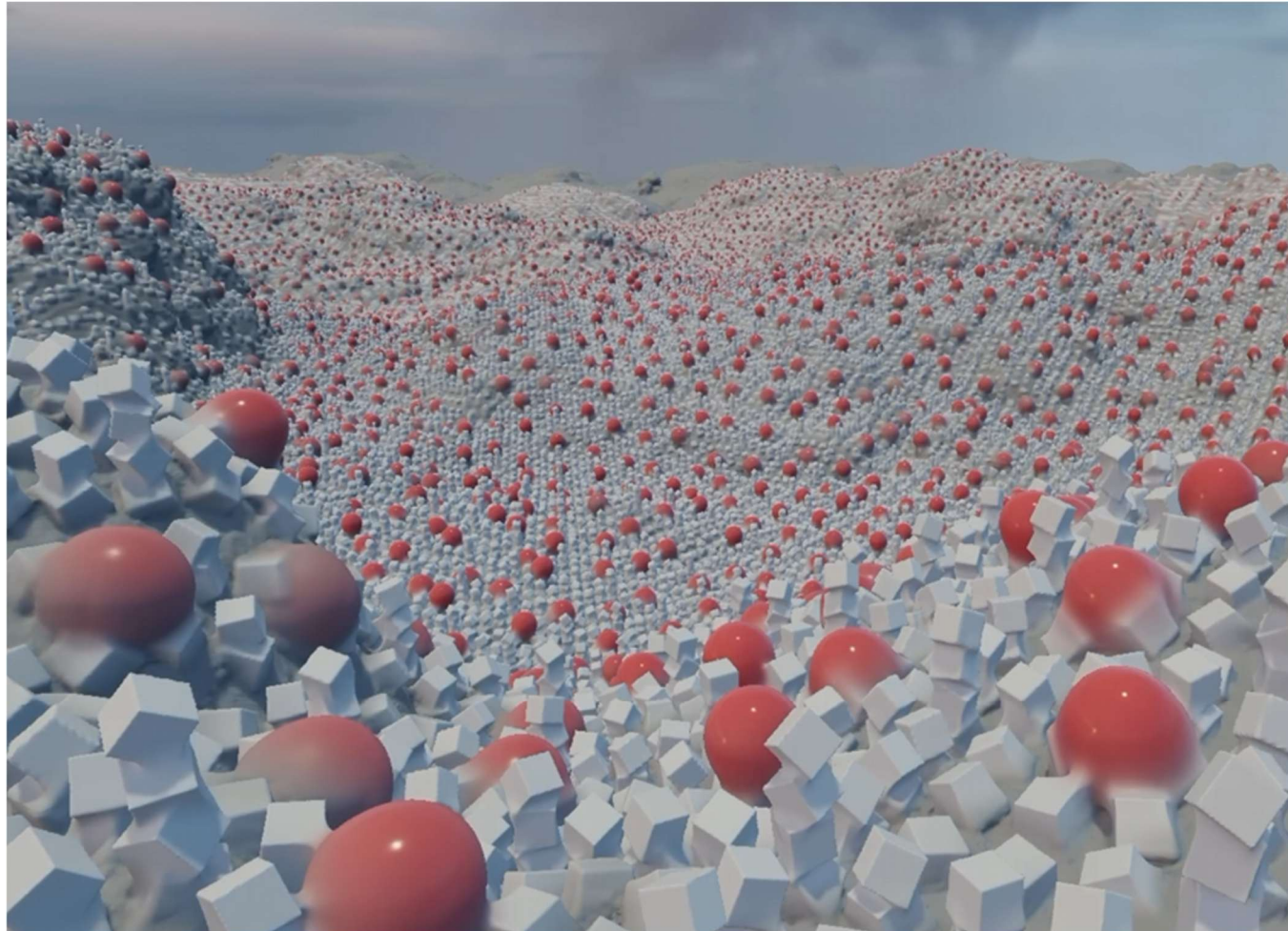


<https://turanszkij.wordpress.com/2024/06/02/texture-streaming/>

Future Directions VIII.

Dynamic SDFs & Ray Marching

- ▶ Explore **Signed Distance Fields (SDFs)** and ray marching as a fundamentally different approach to representing and rendering geometry.
- ▶ Investigate **dynamic SDF-based worlds**, where geometry can be added, removed or modified with high fidelity directly during runtime.
- ▶ Explore **cached and sparse SDF representations**, such as brick-based storage, to overcome the performance and memory limitations of naïve ray marching.
- ▶ Investigate how SDF geometry could integrate with **physics, terrain and large-scale environments**, following approaches such as Jolt-based collision handling and SDF terrain.
- ▶ Build a small **SDF/ray-marching framework** to experiment with these techniques and understand the different architectural and rendering mental model they provide.

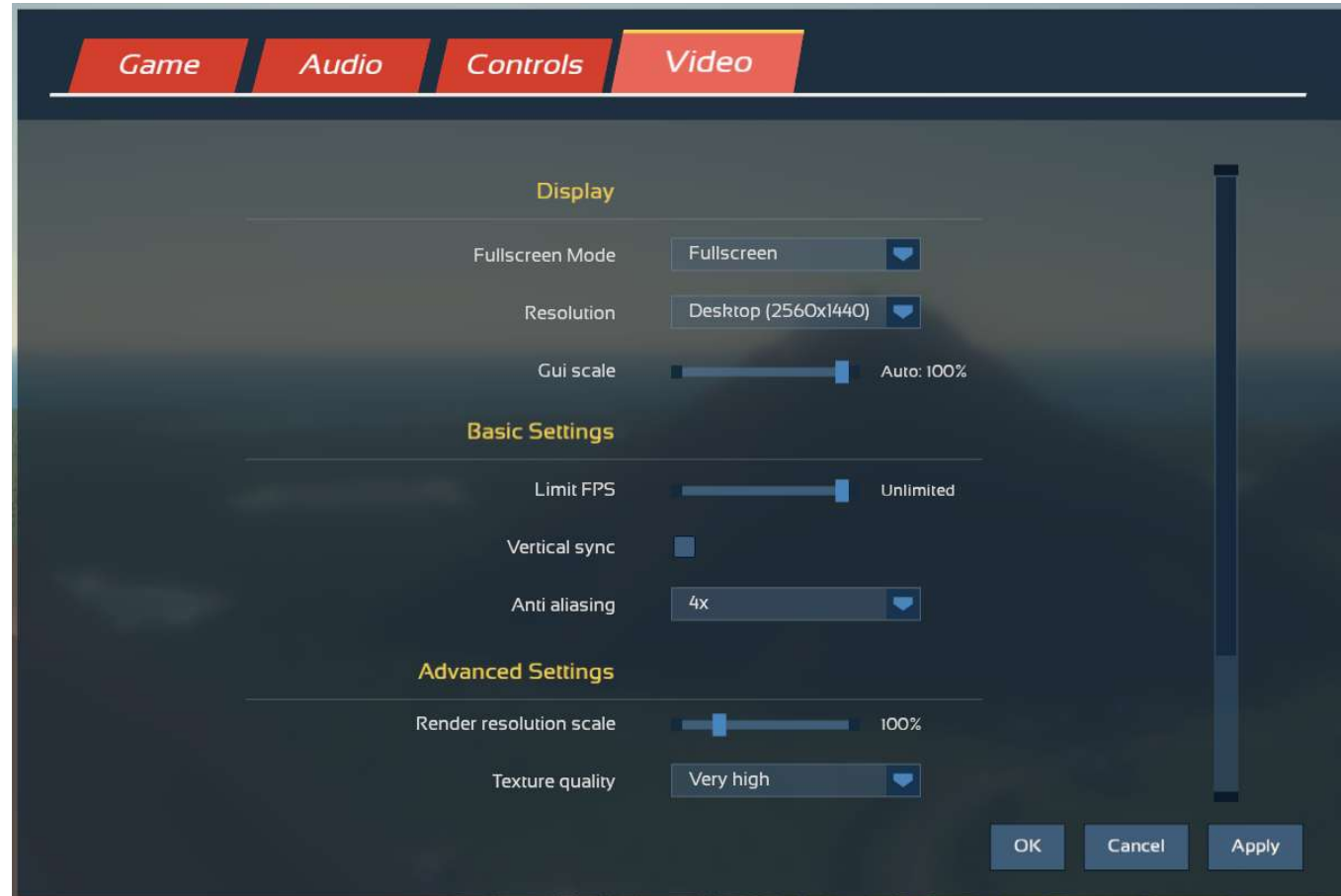


<https://youtu.be/il-TXbn5iMA>

Future Directions IX.

UI Scripting & UI System

- ▶ Explore a dedicated **2D and 3D UI system** for runtime interfaces and interactive content.
- ▶ Primarily investigate **RmlUi** as the foundation for the runtime UI architecture, while keeping ImGui focused on editor tooling.
- ▶ Support **UI scripting** to allow interfaces and interactions to be defined and controlled directly from scripts.
- ▶ Investigate how the UI system could integrate with the engine's **rendering, input and scripting architectures**.
- ▶ The exact architecture is still **under investigation**, with RmlUi currently being the primary direction to explore.



<https://github.com/mikke89/RmlUi>

Future Directions X.

GPU-Driven Voxel & Block World

- ▶ Explore **voxel- and block-based world representations** as a fundamentally different approach to large-scale environment rendering.
- ▶ Investigate **GPU-driven voxel rendering**, using task/mesh shaders, indirect rendering and compact GPU data structures.
- ▶ Explore techniques such as **greedy meshing, face culling, occlusion culling and hierarchical LOD** to efficiently render massive block-based worlds.
- ▶ Investigate **chunk-based world organization and streaming**, allowing extremely large environments to remain efficiently manageable.
- ▶ Use this direction to explore how the existing **GPU-driven architecture of SynapseEngine** could be extended toward Minecraft-like procedural and highly scalable worlds.

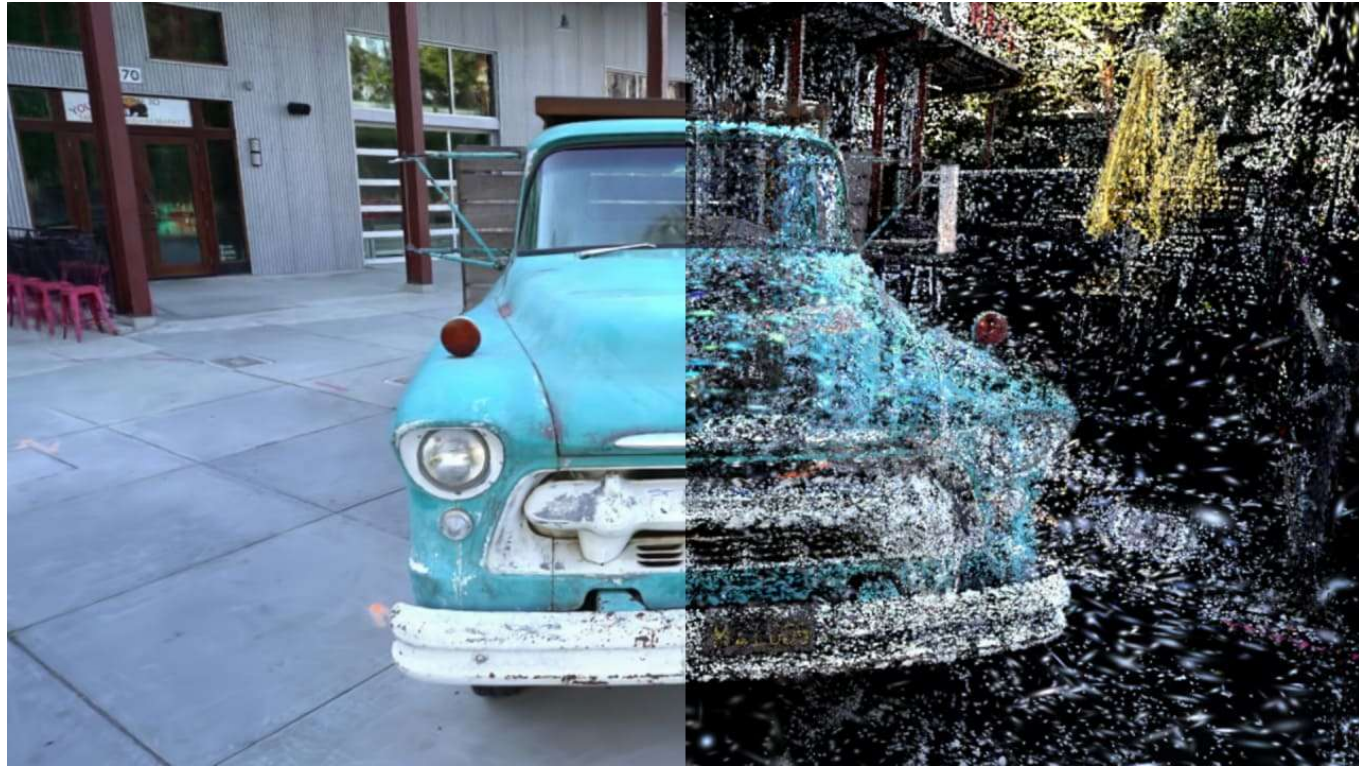


<https://youtu.be/40JzyaOYJeY>

Future Directions XI.

Gaussian Splatting

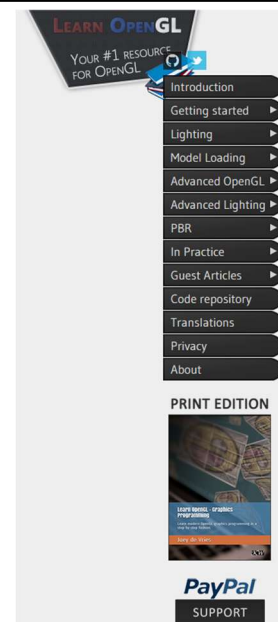
- ▶ **Explore 3D Gaussian Splatting** as a cutting-edge approach to scene representation and novel view synthesis.
- ▶ A primary goal is to **build the entire pipeline completely from scratch**, demystifying the underlying mathematics, sorting algorithms, and rasterization techniques.
- ▶ This will require investigating how to integrate a fundamentally different, non-polygonal rendering paradigm into the engine's existing graphics architecture.
- ▶ **Mastering and natively supporting Gaussian Splatting** has been a long-standing personal ambition, making it an ideal candidate for deep technical exploration and future development.



Future Directions XII.

Educational Development & Technical Devlogs

- ▶ Create **modern, high-performance graphics and game-engine tutorials** inspired by resources such as LearnOpenGL and Learn Vulkan.
- ▶ Focus on topics that are often difficult to find in an accessible form: **GPU-driven rendering, engine architecture, modern Vulkan features and advanced rendering techniques.**
- ▶ Move beyond outdated approaches and document **current-generation techniques and architectural patterns** as they are actually used in modern engines.
- ▶ Build a combination of **educational articles, technical devlogs and practical implementations**, where concepts are explained through real engine development rather than isolated examples.
- ▶ **Education is a personal priority of the project** sharing knowledge, making difficult graphics topics accessible, and helping others understand why these systems work is an important part of SynapseEngine's future.



Welcome to OpenGL

Welcome to the online book for learning OpenGL! Whether you are trying to learn OpenGL for academic purposes, to pursue a career or simply looking for a hobby, this book will teach you the basics, the intermediate, and all the advanced knowledge using **modern** (core-profile) OpenGL. The aim of LearnOpenGL is to show you all there is to modern OpenGL in an easy-to-understand fashion with clear examples, while also providing a useful reference for later studies.

So why read these chapters?

Throughout the internet there are thousands of documents, books, and resources on learning OpenGL, however, most of these resources are only focused on OpenGL's immediate mode (commonly referred to as the old OpenGL), are incomplete, lack proper documentation, or are not suited for your learning preferences. Therefore, my aim is to provide a platform that is both complete and easy to understand.

If you enjoy reading content that provides step-by-step instructions, clear examples, and that won't throw you in the deep with millions of details, this book is probably for you. The chapters aim to be understandable for people without any graphics programming experience, but are still interesting to read for the more experienced users. We also discuss practical concepts that, with some added creativity, could turn your ideas into real 3D applications. If all of the previous sounds like someone that could be you, then by all means, please continue.



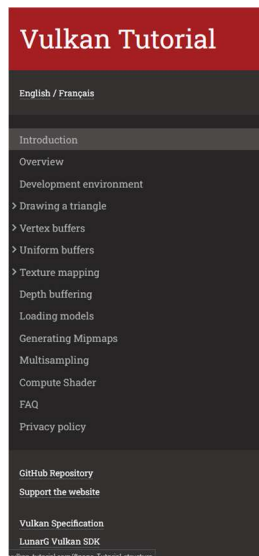
What will you learn?

The focus of these chapters are on Modern OpenGL. Learning (and using) modern OpenGL requires a strong knowledge of graphics programming and how OpenGL operates under the hood to really get the best of your experience. So we will start by discussing core graphics aspects, how OpenGL actually draws pixels to your screen, and how we can leverage that knowledge to create some funky looking effects.

On top of the core knowledge we will discuss many useful techniques that you can use for your applications, like: traversing a scene, create beautiful lighting, load custom-made objects from a modelling program, do cool post-processing techniques, and much more. We also feature a walkthrough series where we actually create a small game based on our obtained OpenGL knowledge, so you will really get a feel of what it's like to actually do graphics programming.

Where to start

Learn OpenGL is free, and will always be free, for anyone who wants to start with graphics programming. All content is available here at the menu to your left. Simply hit the Introduction button and you're ready to start your journey!



Introduction

- Read before following this tutorial
- About
- E-book
- Tutorial structure
- License

Read before following this tutorial

This tutorial was written shortly after Vulkan was initially released, back in 2016. A lot has changed since then and this tutorial no longer reflects the best way to use Vulkan today.

Instead of reading this website, I recommend to follow the guide or one of the tutorials linked here: <https://vulkan.org/learn>

About

This tutorial will teach you the basics of using the **Vulkan** graphics and compute API. Vulkan is a new API by the Khronos group (known for OpenGL) that provides a much better abstraction of modern graphics cards. This new interface allows you to better describe what your application intends to do, which can lead to better performance and less surprising driver behavior compared to existing APIs like OpenGL and DirectX. The ideas behind Vulkan are similar to those of DirectX 12 and Metal, but Vulkan has the advantage of being fully cross-platform and allows you to develop for Windows, Linux and Android at the same time.

However, the price you pay for these benefits is that you have to work with a significantly more verbose API. Every detail related to the graphics API needs to be set up from scratch by your application, including initial frame buffer creation and memory management for objects like buffers and texture images. The graphics driver will do a lot less hand holding, which means that you will have to do more work in your application to ensure correct behavior.

The takeaway message here is that Vulkan is not for everyone. It is targeted at programmers who are enthusiastic about high performance computer graphics, and are willing to put some work in. If you are more interested in game development, rather than computer graphics, then you may wish to stick to OpenGL or DirectX, which will not be deprecated in favor of Vulkan anytime soon. Another alternative is to use an engine like **Unreal Engine** or **Unity**, which will be able to use Vulkan while exposing a much higher level API to you.

With that out of the way, let's cover some prerequisites for following this tutorial:

Support the Project

- ▶ **I strongly believe that knowledge should be completely free and accessible to everyone.** Synapse Engine is entirely open-source, and all tutorials, devblogs, and educational materials will always remain free.
- ▶ Creating this 250-slide presentation and developing an enterprise-grade rendering architecture from scratch requires immense time and dedication.
- ▶ **As I am currently dedicating my time to this project** without a steady income, community support plays a crucial role in making this work possible.
- ▶ If you **enjoyed this presentation, learned something new**, or believe in the project's mission, you can support its continued development through GitHub Sponsors.
- ▶ **Sponsoring is entirely optional, but every single contribution is deeply appreciated** and directly helps fund the future of open-source graphics!

Fund the Future of Open-Source Graphics

The Synapse Engine is a research-oriented, high-performance rendering framework built with C++23 and modern Vulkan. My mission is to translate complex, enterprise-grade architecture into accessible, high-quality, and completely free educational content.

By sponsoring this project, you are funding the research, the devblogs, and the time required to develop this engine full-time. Knowledge should be free. You can help keep it that way.

Monthly Sponsorship Tiers

HIGHLY APPRECIATED The Meshlet Supporter \$4 / month A foundational contribution. Your support directly assists in the continuous research and publication of free educational materials. Select Tier	The ECS Backer \$12 / month A significant contribution. This helps fund the infrastructure for benchmarking advanced data-oriented architectures. Select Tier	The Forward+ Sponsor \$29 / month A major investment. Your support allows me to dedicate the required hours to author comprehensive technical tutorials. Select Tier	The GPU-Driven Patron \$99 / month For organizations or dedicated professionals ensuring the long-term, full-time development of the engine. Select Tier
---	---	--	--

<https://synapseengine.dev/sponsors/>