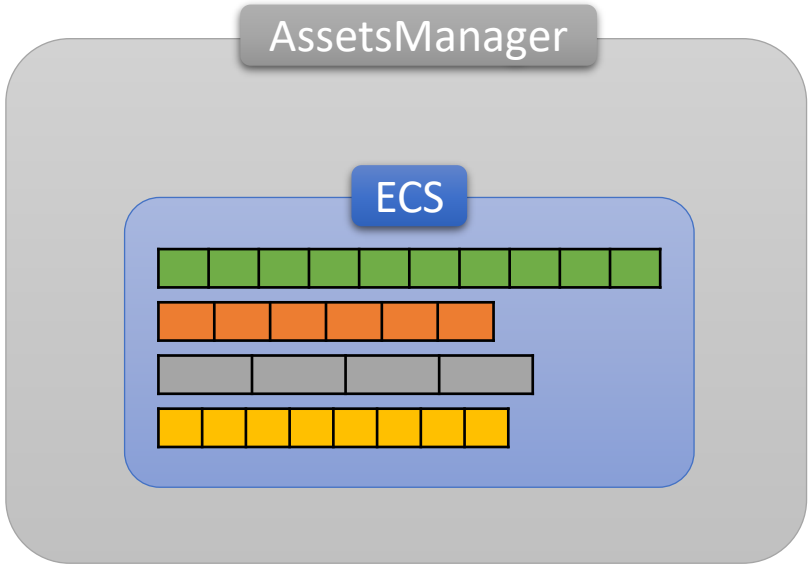
Asset Pipeline of Fools Engine

Work in progress

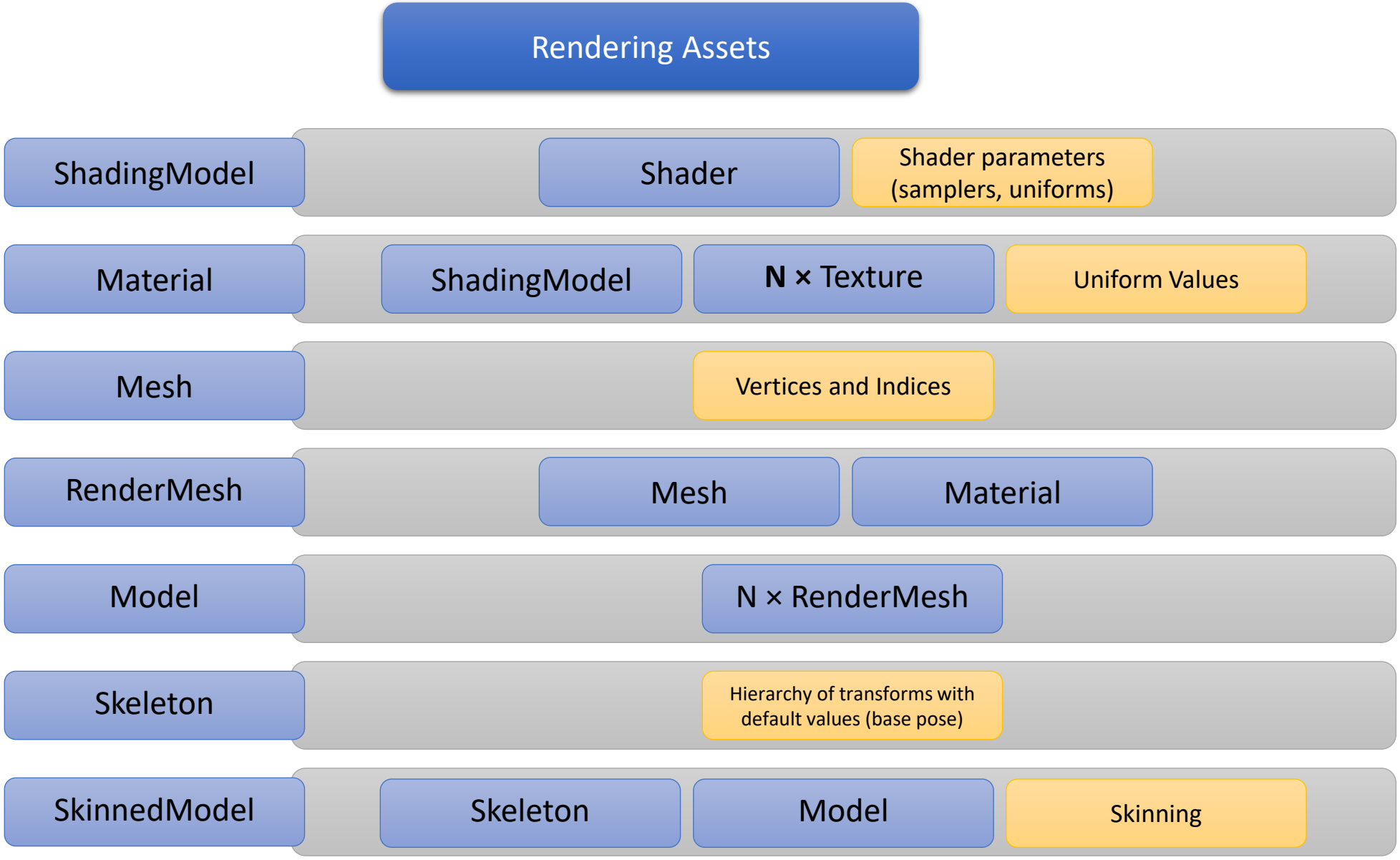
Storage structure
Editor and Game

Assets are broken down into components and stored in ECS.

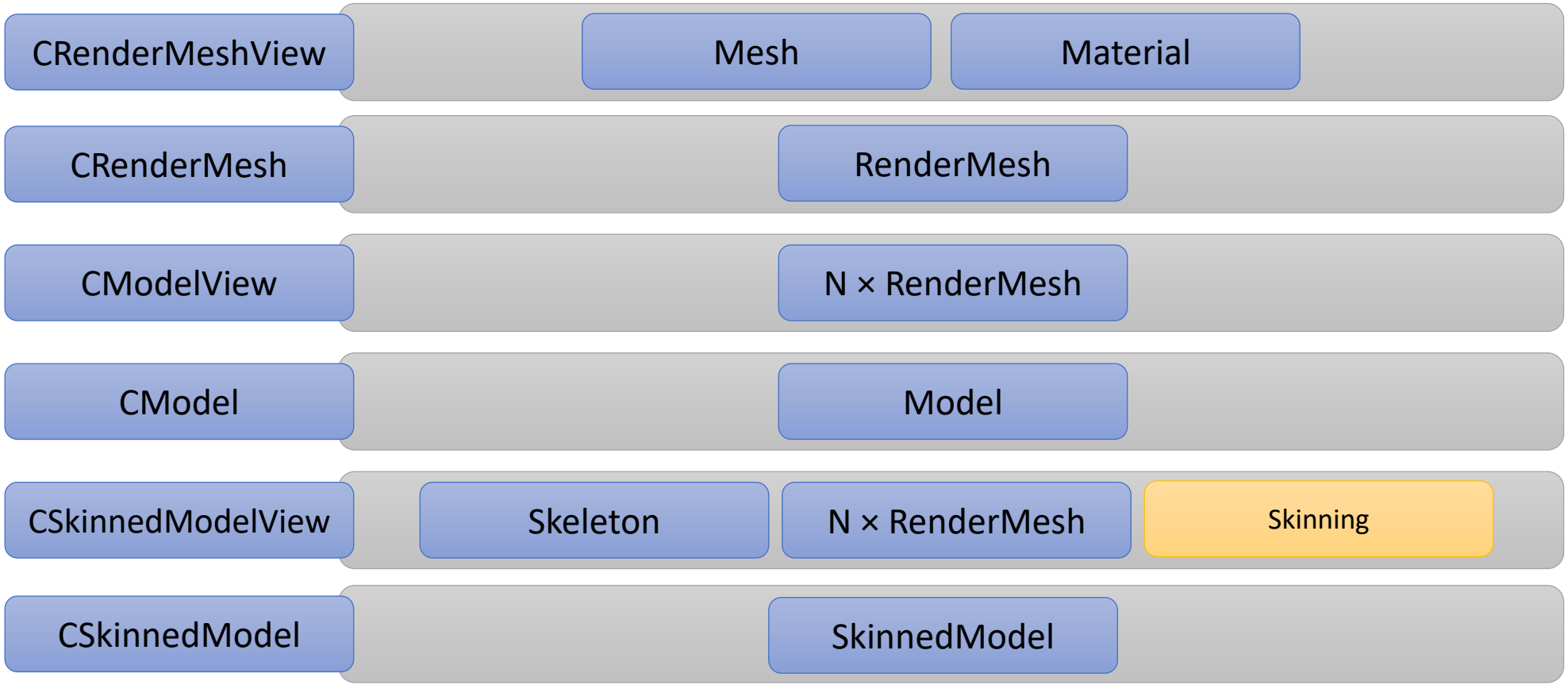
AssetManager creates assets and decides what to load and what to unload.



Asset Types:
- Texture,
- Texture2D,
- Shader,
- ShadingModel,
- Material,
- Mesh,
- RenderMesh,
- Model,
- Skeleton,
- SkinnedModel,
- Animation,
- Scene,
- Prefab,
- Audio


Rendering Assets
ShadingModel
Shader
Shader parameters (samplers, uniforms)
Material
ShadingModel
N × Texture
Uniform Values
Mesh
Vertices and Indices
RenderMesh
Mesh
Material
Model
N × RenderMesh
Skeleton
Hierarchy of transforms with default values (base pose)
SkinnedModel
Skeleton
Model
Skinning

Scene Components for Mesh Assets



Runtime Management
Editor and Game

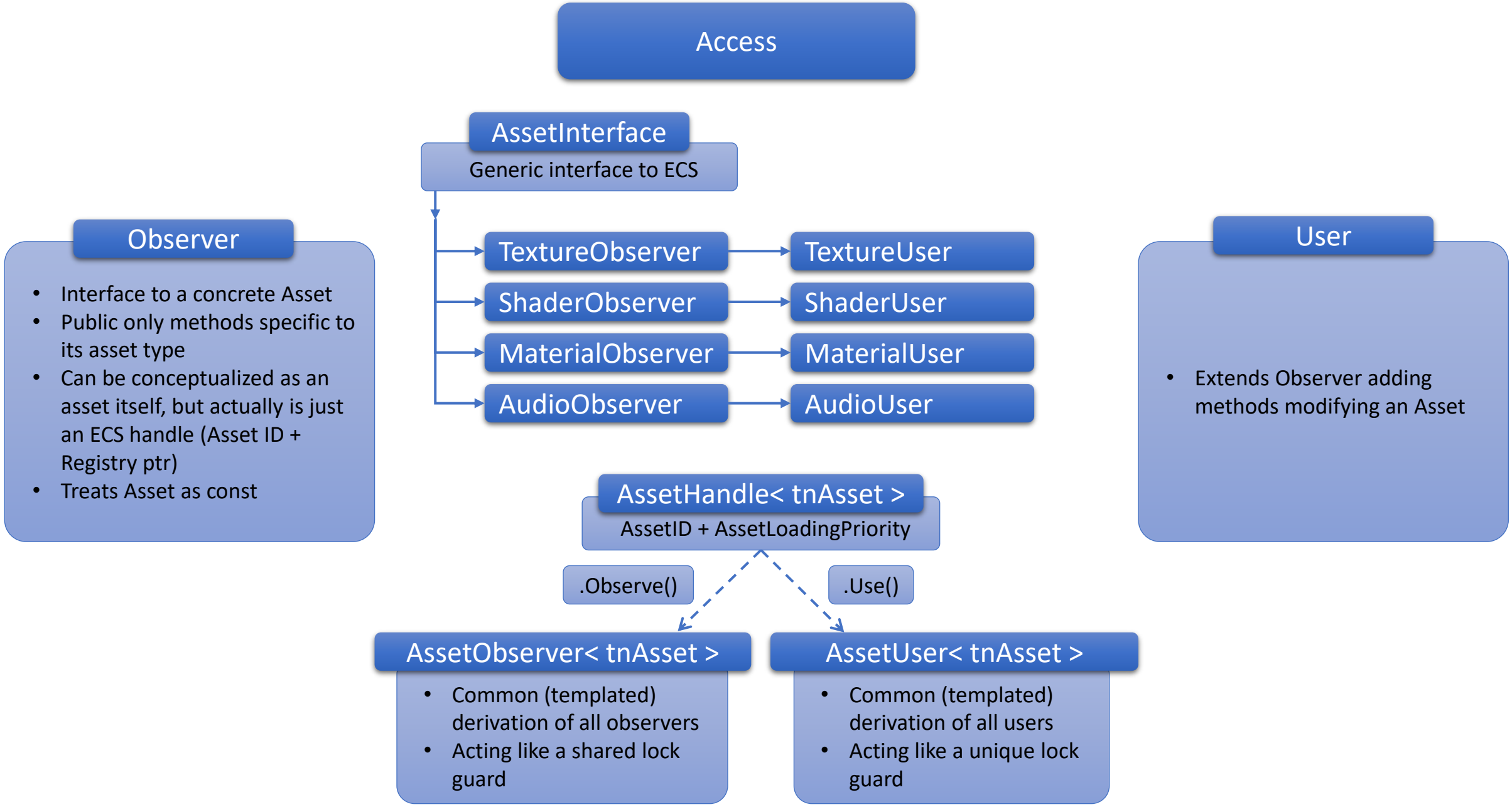
Centralized streaming:

You can never assume an asset is loaded and there is no explicit asset loading request possibility. Instead, a handle to the asset has a loading priority value - enum. Asset handles of each loading priority are atomically counted in a component of that asset (reference counting). Asset Manager uses those counts to decide centrally which assets to load and unload:

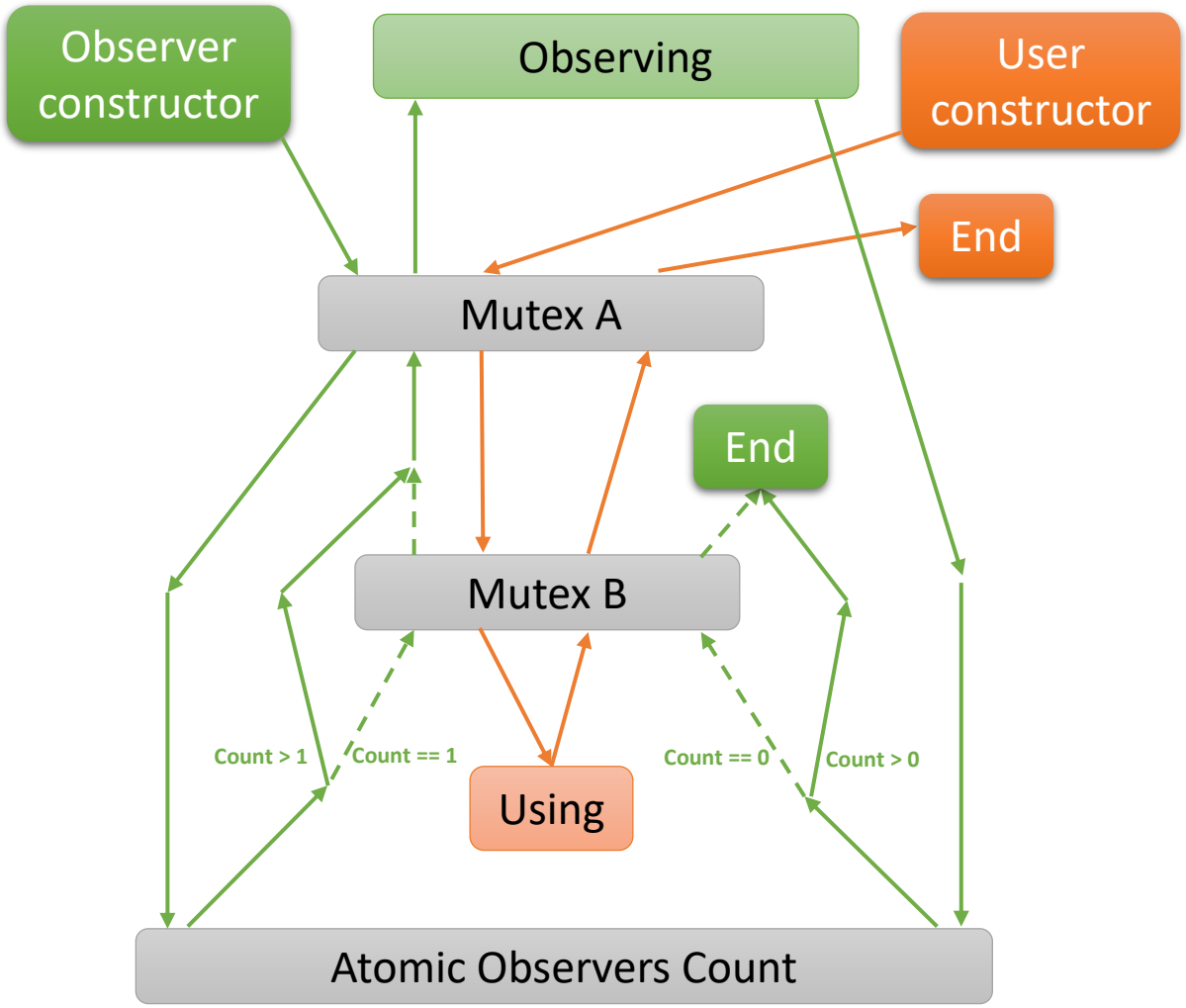
1. Calculates score of each asset as ∑ of count*priority
2. Sorts assets based on their's scores:
3. Load as many assets as much memory is available going from the highest score down (and unloading going from lowest score up).

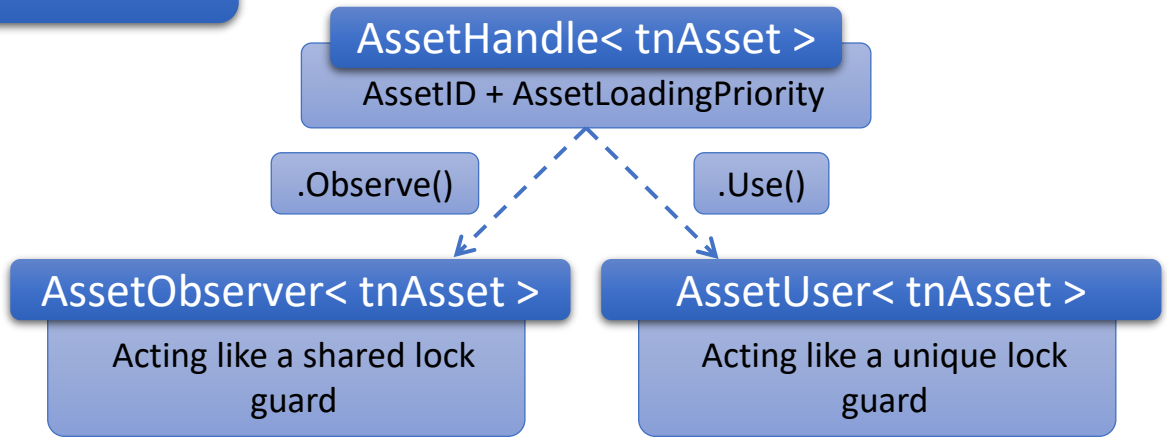
```
enum AssetLoadingPriority : uint32_t
{
    None = 0,
    Low = 1,
    Standard = 10,
    High = 100,
    Critical = -1
};
```

This makes dynamic adaptive asset loading management much easier and safer (e.g. preloading assets for next level simply by setting their handles from None to Low). Asset loading/unloading and score calculation/sorting runs in an asynchronous loop, but in a foreground (structured concurrency).


Access
AssetInterface
Generic interface to ECS
TextureObserver
TextureUser
ShaderObserver
ShaderUser
MaterialObserver
MaterialUser
AudioObserver
AudioUser
Observer
Interface to a concrete Asset
Public only methods specific to its asset type
Can be conceptualized as an asset itself, but actually is just an ECS handle (Asset ID + Registry ptr)
Treats Asset as const
User
Extends Observer adding methods modifying an Asset
AssetHandle< tnAsset >
AssetID + AssetLoadingPriority
.Observe()
.Use()
AssetObserver< tnAsset >
Common (templated) derivation of all observers
Acting like a shared lock guard
AssetUser< tnAsset >
Common (templated) derivation of all users
Acting like a unique lock guard

Synchronization





Observer()	User()
1. Lock Mutex A 2. Observers Count ++ 3. If 2. was 0->1 • Lock Mutex B 4. Unlock Mutex A	1. Lock Mutex A 2. Lock Mutex B
~Observer()	**~User()**
1. Observers Count -- 2. If 1. was 1->0 • Unlock Mutex B	1. Unlock Mutex B 2. Unlock Mutex A

Synchronized Runtime Access
Examples

```
void Renderer::Draw(
    const Ref<VertexBuffer>& vertexBuffer,
    const AssetObserver<Material>& materialObserver,
    const glm::mat4& transform,
    const glm::mat4& VPMatrix)
{
    FE_PROFILER_FUNC();

    auto& material_core = materialObserver.GetCoreComponent();
    auto sm_observer = AssetObserver<ShadingModel>(material_core.ShadingModelID);
    auto& sm_core = sm_observer.GetCoreComponent();
    auto shaderUser = AssetUser<Shader>(sm_core.ShaderID);

    shaderUser.Bind(s_ActiveGDI);

    shaderUser.UploadUniform(
        s_ActiveGDI,
        Uniform("u_ViewProjection", ShaderData::Type::Mat4),
        (void*)glm::value_ptr(VPMatrix)
    );
    shaderUser.UploadUniform(
        s_ActiveGDI,
        Uniform("u_Transform", ShaderData::Type::Mat4),
        (void*)glm::value_ptr(transform)
    );

    for (const auto& uniform : sm_core.Uniforms)
    {
        auto dataPointer = materialObserver.GetUniformValuePtr(material_core, uniform);
        shaderUser.UploadUniform(s_ActiveGDI, uniform, dataPointer);
    }
```

```
auto render_mesh_observer = render_mesh_component.RenderMesh.Observe();

auto& render_mesh_core = render_mesh_observer.GetCoreComponent();
auto material_observer = AssetObserver<Material>(render_mesh_core.MaterialID);
auto shaderID = AssetObserver<ShadingModel>(material_observer.GetCoreComponent().ShadingModelID).GetCoreComponent().ShaderID;
{
    auto shader_observer = AssetObserver<Shader>(shaderID);

    shader_observer.Bind(GDI);
    shader_observer.UploadUniform(GDI, Uniform("u_ViewProjection", ShaderData::Type::Mat4), VPmatrixPtr);
    shader_observer.UploadUniform(GDI, Uniform("u_ModelTransform", ShaderData::Type::Mat4), modelTransformPtr);
    shader_observer.UploadUniform(GDI, Uniform("u_EntityID", ShaderData::Type::UInt), &ID);
}

AssetObserver<Mesh>(render_mesh_core.MeshID).Draw(material_observer);
```