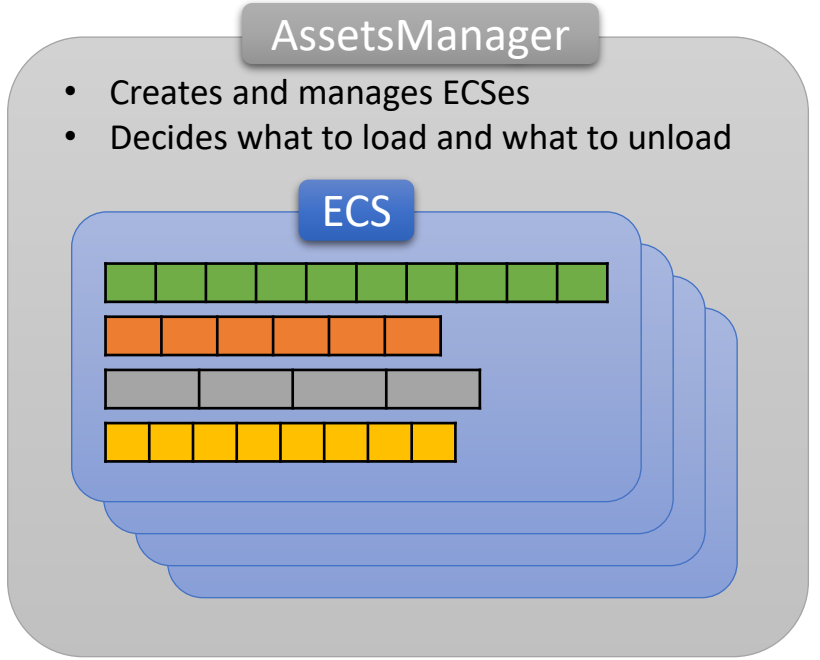
Asset Pipeline of Fools Engine

Work in progress

Runtime Management
Editor and Game



Storage structure:

Assets are broken down into components and stored in ECS (each asset type has its own - runtime ID may be identical for multiple assets as long, as they are different types).
This gives flexibility of allowing easy creation of asset subtypes (e.g. standalone mesh and mesh that is part of a model), connections and dependencies between assets without redesigning whole asset system or fighting with poorly predicted inheritance hierarchy. Lack of complex interactions between assets (only loading dependencies and similar) makes ECS perfect fit.

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 those scores
3. Unloads assets with score 0
4. Load as many assets as much memory is available

```
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

Asset

- Generic interface to ECS

- Concrete Asset
- Public only methods specific to its asset type
- Can be conceptualized as an asset itself, but actually is just an interface to ECS

- Texture : Asset
- Shader : Asset
- Material : Asset
- Audio : Asset

AssetHandle<tnAsset>

AssetID + AssetLoadingPriority

.Use()

.Observe()

AssetUser<> : tnAsset

- Like unique_ptr
- Like lock guard

AssetObserver<> : tnAsset

- Like shared_ptr
- Like lock guard
- Always const

Synchronization

Mutex A

Mutex B

Atomic Observers Count

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

Synchronized Runtime Access
Examples

Use() and Observe() are chainable for convenience.
Synchronization is inline (beware of the cost!).

```
auto miObserver = render_mesh_component.MaterialInstance.Observe();

auto shaderID = miObserver.GetMaterial().Observe().GetShaderID();
{
    auto shaderUser = AssetHandle<Shader>(shaderID, AssetLoadingPriority::None).Use();

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

render_mesh_component.Mesh.Use().Draw(miObserver);
```

Dedicated scope to destroy shaderUser and release locks when its no longer needed

AssetHandle is constructible on the fly.

AssetHandles with AssetLoadingPriority::None are not globally counted – no hidden cost.