Asset Validation Framework

1. Overview

This page describes the Asset Validation Framework implemented for the project. The framework verifies the technical and aesthetic quality of assets before final usage in Unreal Engine 5 cinematics. The approach involves checks both outside Unreal Engine (to validate FBX files, textures, and metadata) and inside Unreal Engine (to confirm correct import settings, naming conventions, and performance criteria).

2. Asset Types

2.1 3D Models / Skeletal Meshes

- Created in Blender or Character Creator
- Exported as FBX files
- Include rigged characters or static props

2.2 Textures & Materials

- Created in Substance Painter, Blender, or other tools
- Imported as image files (PNG, TGA)
- Linked to materials in Unreal Engine

2.3 Niagara Effects

- Particle systems that enhance visual fidelity
- Configurable for GPU or CPU calculations

2.4 Motion Graphics & UMG Widgets

- UI elements inside Unreal Engine
- May contain PNG textures, fonts, and layout files

2.5 Levels (Maps)

- Contain static meshes, post process volumes, lights, and cameras
- Include settings for advanced rendering features

3. Validation Approach

This framework relies on two validation layers:

1. External Validation: Occurs after assets leave Blender (or similar tools) and before they are committed to GitLab. It ensures FBX files, textures, and other resources meet fundamental criteria (e.g., polycount, texture resolution).

2. In-Engine Validation: Occurs once assets are imported into Unreal Engine. It confirms correct naming conventions, material setups, and engine-specific configurations (e.g., texture compression, post process settings).

4. Implementation

4.1 External Checks (Outside Unreal Engine)

- Python Scripts analyze FBX files for polycount, rig integrity, and pivot alignment.
- Texture Checks confirm file format, resolution, and color profile.
- GitLab CI Integration runs these scripts on commit or merge request events.
- Reports appear as pipeline artifacts, indicating pass, fail, or warnings.

4.2 In-Engine Checks (Within Unreal Engine)

- Editor Utility Widgets (EUW) provide a UI for scanning imported assets.
- Unreal Python API runs automated scripts that detect issues such as incorrect texture compression, missing material instances, or unusual post process volumes.
- Performance Boundaries (particle count, polycount, lighting complexity) are verified through these tools.

5. Technical & Quality Criteria

5.1 Geometry & Rigging

- Polycount remains within project-approved limits.
- Rig integrity includes required bones and naming.
- Transform scale matches the standard (1 Unreal Unit = 1 cm).

5.2 Textures & Materials

- Textures use 2K resolution unless exceptions exist.
- Normal, Roughness, Diffuse, Metallic, and AO maps are present if required.
- Compression settings match map types (e.g., Normal Map compression for normals).
- Materials reference valid textures, and naming conventions follow the project format.

5.3 Niagara Effects

- Particle systems remain within performance budgets (particle count, GPU usage).
- Naming conventions use a prefix (e.g., FX_).

5.4 Motion Graphics (UMG)

- File naming follows UI_ prefix.
- Layout scales correctly at 1920x1080.
- Fonts and textures link to existing, valid resources.

5.5 Levels & Post Process

- Scenes include correct post process volumes (bloom, exposure, color grading).
- Lights do not exceed performance constraints.
- Cameras and Sequencer references match naming conventions (MAP_, CAM_, etc.).

6. Quarantine vs. Warnings

The pipeline flags assets as Critical Fail or Warning:

- Critical Fail (Quarantine): Assets are moved to a holding area for manual review. This status applies when essential checks fail (e.g., missing rig bones, extreme polycount, broken textures).

- Warning (Allow): Assets pass into the main branch or folder but carry a warning. Examples include minor naming deviations or optional map omissions. These issues appear in logs for future refinement.

7. Example Workflow

1. Asset Creation: An artist exports a model from Blender as an FBX file.
2. External Validation: A local or GitLab CI script checks the file for geometry, rig, and naming.
3. Commit to GitLab: The asset passes or enters Quarantine if critical issues appear.
4. Unreal Engine Import: The user imports the asset. Editor Utility Widgets perform in-engine checks for texture settings, materials, and naming.
5. Report & Resolution: Assets with minor warnings remain available. Quarantined assets are fixed and re-validated.
6. Approved Assets: Once no critical issues remain, assets proceed to final cinematic use.

8. Next Steps & Notes

- A dedicated naming convention document is under review, ensuring consistent prefixes (CHAR_, PROP_, MAP_, FX_, UI_).
- A more detailed GitLab CI configuration is planned, adding automated triggers for newly updated FBX and texture files.
- Periodic batch scans in Unreal Editor are under consideration for deeper coverage of older assets.