Runtime types:

1. Individual texture
2. TexturePool
3. Geometry
4. GeoPool
5. Sound effect
6. Music
7. Entity Info
8. Level
9. Animations

Components:

1. FileImporter library
2. Pipeline tool

Intermediate formats:

1. DDS for textures
2. WAV for sound effect
3. TBD for music
4. Name/Value Pairs for Entity Info
5. N/A for Level
6. Custom simple format for Geometry

File source formats:

1. TXT for entity info
2. DDS and all WIC for textures
3. DDS for texture packs (mini-pools)
4. WAV for sound effect
5. OBJ for static geometry

Next steps:

1. Create FileImporter lib
2. Move image & sound effect code there
3. Add geometry load there
4. Fully implement Content object
5. Create entity type system basics with testcube type
6. Create basic spawn method on Gameplay that creates a testcube

# Pipeline Architecture:

File Importers will load source assets into intermediate content objects. These are C++ types that wrap and allow manipulation of the raw data. They support being saved out to a final destination in the designated format for the resource type (eg. TextureContent will save out as a simplified DDS).

Content Processors take in intermediate types (and can demand load additional assets using the file importers), process the data into a game usable resource, and then invoke the appropriate save code paths after determining where (and which) data should be saved.

The content objects’ save methods are agnostic to specifics of files and paths. There is an OutputManager that controls where data gets saved, whether it be individual files and folders, or the equivalent of lumps in a packed file.

To control which assets need to be processed into resources for runtime use, and to prevent redundant processing, there will be an asset manifest. This is just a text file that describes the source assets, what processing should be done, and the identifier we wish to use to reference them with at runtime. Processing starts with this file, and forks out into sub processing units to handle the described payload. Timestamp checks are done on the outputs to ensure the same files are reprocessed unnecessarily. An output manifest is created which acts as the directory or table of contents for the runtime’s asset loading system. This describes which asset identifiers are valid, and what they map to, including strongly typed resource types (such as Texture, Geometry, or StaticLevel).

The runtime asset loader will get an overhaul, and will take in identifier names. These identifiers are looked up in the output manifest and the source data loaded. There will be a means for dev override, most likely by looking up the type of the data and using a standard set of optional extensions to look for files with the identifier name + one of those extensions.

The runtime asset loader will be much more efficient and intelligent with how it loads things. The methods will be custom tailored to optimize the individual resource loads that we support. For example, the texture load should be able to optimally load a single mip directly into an existing texture if needed. All disk formats will be optimized for fastest/simplest possible runtime overhead in loading, and to support the types of load patterns we’ll have.