

Game Engine Architecture

Chapter 7 Resources and the File System

Overview

- File System
- Resource Manager

Data

- Game engines are inherently data management systems
- Handle all forms of input media
 - Textures
 - 3D mesh data
 - Animations
 - Audio clips
 - World layouts
 - Physics data
- Memory is limited, so we need to manage these resources intelligently

File system

- Resource managers make heavy use of the file system
- Often the file system calls are wrapped
 - Provides device independence
 - Provides additional features like file streaming
- Sometimes the calls for accessing different forms of media are distinct - Disk versus a memory card on a console
- Wrapping can remove this consideration

Engine file system

- Game engine file systems usually address the following issues
 - Manipulating filenames and path
 - Opening, closing, reading and writing individual files
 - Scanning the contents of a directory
 - Handling asynchronous file I/O requests (for Streaming)

File names and paths

- Path is a string describing the location of a file or directory
 - volume/directory1/directory2/.../directoryN/file-name
- They consist of an optional volume specifier followed by path components separated with a reserved character (/ or \)
- The root is indicated by a path starting with a volume specifier followed by a single path separator

OS differences

- UNIX and Mac OS X
 - Uses forward slash (/)
 - Supports current working directory, but only one
- Mac OS 8 and 9
 - Uses the colon (:)
- Windows
 - Uses back slash (\) – more recent versions can use either
 - Volumes are specified either as C: or \\some-computer\some-share
 - Supports current working directory per volume and current working volume
- Consoles – often used predefined names for different volumes

Pathing

- Both windows and UNIX support absolute and relative pathing
 - Absolute
 - Windows - C:\Windows\System32
 - Unix - /usr/local/bin/grep
 - Relative
 - Windows – system32 (relative to CWD of c:\Windows)
 - Windows – X:animation\walk\anim (relative to CWD on the X volume)
 - Unix – bin/grep (relative to CWD of /usr/local)

Search path

- Don't confuse path with search path
 - Path refers to a single file
 - Search path is multiple locations separated by a special character
- Search paths are used when trying to locate a file by name only
- Avoid searching for a file as much as possible – it's costly

Path API

- Windows offers an API for dealing with paths and converting them from absolute to relative
 - Called shlwapi.dll
 - <https://docs.microsoft.com/en-us/windows/win32/api/shlwapi/>
- Playstation 3 and 4 have something similar
- Often better to build your own stripped down version

Basic file i/o

- Standard C library has two APIs for file I/O
 - Buffered
 - Manages its own data buffers
 - Acts like streaming bytes of data
 - Unbuffered
 - You manage your own buffers

File operations

Operation	Buffered API	Unbuffered API
Open a file	fopen()	open()
Close a file	fclose()	close()
Read from a file	fread()	read()
Write to a file	fwrite()	write()
Seek an offset	fseek()	seek()
Return current offset	ftell()	tell()
Read a line	fgets()	n/a
Write a line	fputs()	n/a
Read formatted string	fscanf()	n/a
Write a formatted string	fprintf()	n/a
Query file status	fstat()	stat()

File system specific

- On UNIX system, the unbuffered operations are native system calls
- On Windows, there is even a lower level
 - Some people wrap these calls instead of the standard C ones
- Some programmers like to handle their own buffering
 - Gives more control to when data is going to be written

To wrap or not

- Three advantages to wrapping
 - Guarantee identical behavior across all platforms
 - The API can be simplified down to only what is required
 - Extended functionality can be provided
- Disadvantages
 - You have to write the code
 - Still impossible to prevent people from working around your API

Synchronous file i/o

- The standard C file I/O functions are all synchronous

```
bool syncReadFile(const char* filePath, U8* buffer, size_t bufferSize, size_t& rBytesRead){  
    FILE* handle = fopen(filePath, "rb");  
    if(handle){  
        size_t bytesRead = fread(buffer, 1, bufferSize, handle); //blocks until all bytes are read  
        int err = ferror(handle);  
        fclose(handle);  
        if(0==err){  
            rBytesRead = bytesRead;  
            return true;  
        }  
    }  
    return false;  
}
```

Asynchronous I/O

- Often it is better to make a read call and set a callback function when data become available
- This involves spawning a thread to manage the reading, buffering, and notification
- Some APIs allow the programmer to ask for estimates of the operations durations
- They also allow external control

Priorities

- It is important to remember that certain data is more important than others
 - If you are streaming audio or video then you cannot lag
- You should assign priorities to the operations and allow lower priority ones to be suspended

Best practices

- Asynchronous file I/O should operate in its own thread
- When the main thread requests an operation, the request is placed on a queue (could be priority queue)
- The file I/O handles one request at a time
- Virtually *any* synchronous operation you can imagine can be transformed into an asynchronous operation by moving the code into a separate thread—or by running it on a physically separate processor, such as on one of the CPU cores on the PlayStation 4.

Resource manager

- All good game engine have a resource manager
- Every resource manager has two components
 - Offline tools for integrating resource into engine ready form
 - Runtime resource management

Off-line resource management

- Revision control - can be managed using a shared drive or a complex system like SVN or Perforce
- Cautions about data size
 - Remember that code is small compared to images or video
 - May not want many copies lying around (they all need to be backed up)
 - Some places deal with this by using symlinking

Resource database

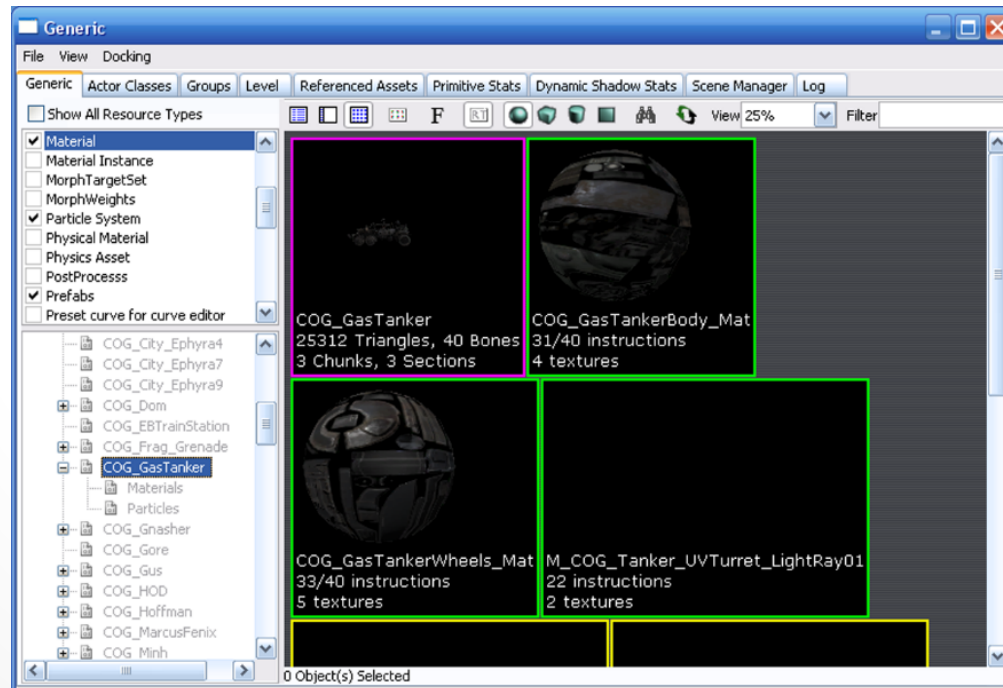
- The resource database contains information about how an asset needs to be conditioned to be useful in a game
- For example, an image may need to be flipped along its x-axis, some images should be scaled down
- This is particularly true when many people are adding assets

Resource data

- The ability to deal with multiple types of resources
- The ability to create new resources
- The ability to delete resources
- The ability to inspect and modify resources
- The ability to move the resources source file to another location
- The ability for a resource to cross-reference another resource
- The ability to maintain referential integrity
- The ability to maintain revision history
- Searches and queries

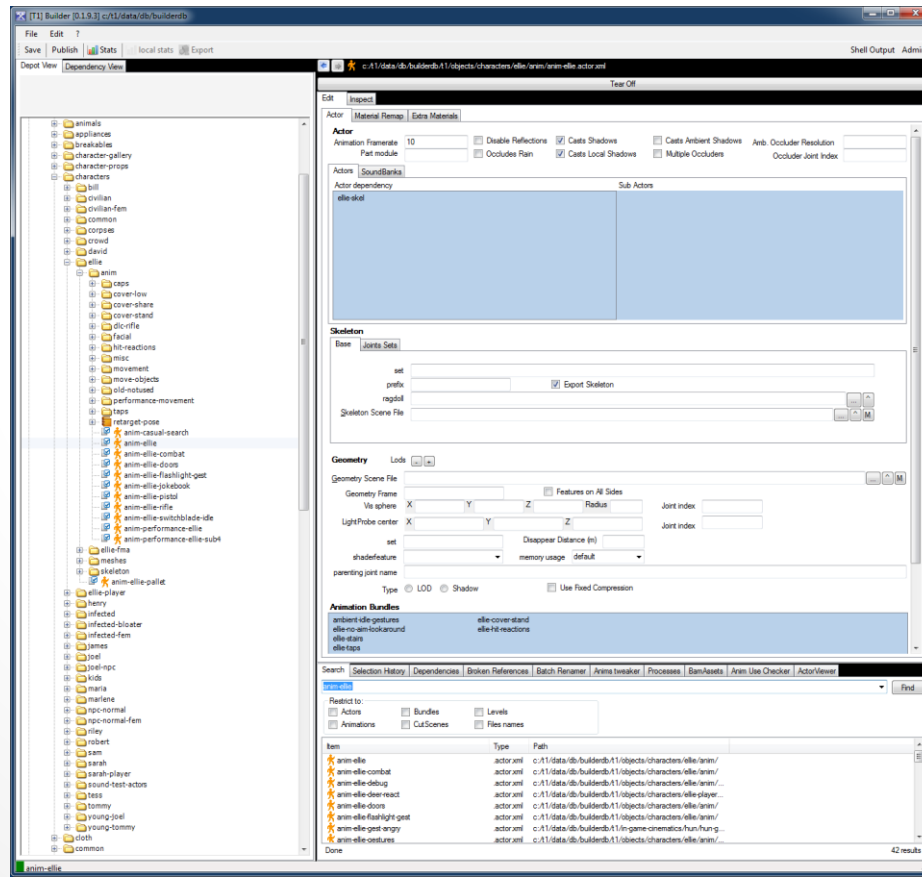
Some successful designs

- UT4
 - All managed using UnrealEd
 - Has some serious advantages, but is subject to problems during multiple simultaneous updates
 - Stores everything in large binary files – impossible for SVN



Another example

- Uncharted/Last of Us
 - Uses a MySQL database – later changed to XML using Perforce
 - Asset conditioning done using offline command prompt tools



Still others...

- Ogre
 - Runtime only resource management
- XNA
 - A plugin to VS IDE called Game Studio Express

Asset conditioning

- Assets are produced from many source file types
- They need to be converted to a single format for ease of management
- There are three processing stages
 - Exporters – These get the data out into a useful format
 - Resource compilers – pre-calculation can be done on the files to make them easier to use
 - Resource linker – multiple assets can be brought together in one file

Resource dependencies

- You have to be careful to manage the interdependencies in a game
- Assets often rely on one another and that needs to be documented
- Documentation can take written form or automated into a script
- *make* is a good tool for explicitly specifying the dependencies

Runtime resource management

- Ensure that only *one* copy of an asset is in memory
- Manages the *lifetime* of the object
- Handles loading of *composite resources*
- Maintains *referential integrity*
- Manages *memory usage*
- Permits *custom processing*
- Provides a *unified interface*
- Handles *streaming*

Resource files

- Games can manage assets by placing them loosely in directory structures
- Or use a zip file (Better)
 - Open format
 - Virtual file remember their relative position
 - They may be compressed
 - They are modular
- UT3 uses a proprietary format called pak (for package)

Resource file formats

- Assets of the same type may come in many formats
 - Think images (BMP, TIFF, GIF, PNG...)
- Some conditioning pipelines standardize the set
- Other make up there own container formats
- Having your unique format makes it easier to control the layout in memory

Resource guides

- You will need a way to uniquely identify assets in your game
- Come up with a naming scheme
 - Often involves more than just the file path and name
- In UT files are named using
 - *package.folder.file*

Resource registry

- In order to only have an asset loaded once, you have to have a registry
- Usually done as a giant hashmap (keyed on GUID)
- Resources loading can be done on the fly, but that is usually a bad idea
 - Done in-between levels
 - In the background

Resource lifetime

- Some resources are LSR (load-and-stay resident)
 - Character mesh
 - HUD
 - Core Animations
- Some are level specific
- Some are very temporary – a cut-scene in a level
- Other are streaming
- Lifetime is often defined by use, sometimes by reference counting

Memory management

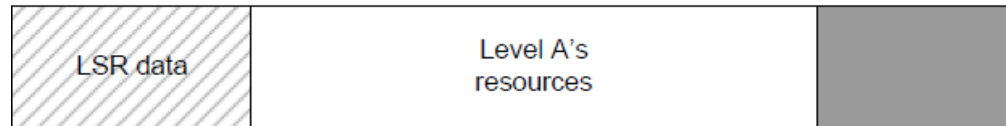
- Very closely tied to general memory management
- Heap allocation – allow the OS to handle it
 - Like malloc() or new
 - Works fine on a PC, not so much on a memory limited console
- Stack allocation
 - Can be used if
 - The game is linear and level centric
 - Each level fits in memory

Stack allocation

Load LSR data, then obtain marker.



Load level A.



Unload level A, free back to marker.

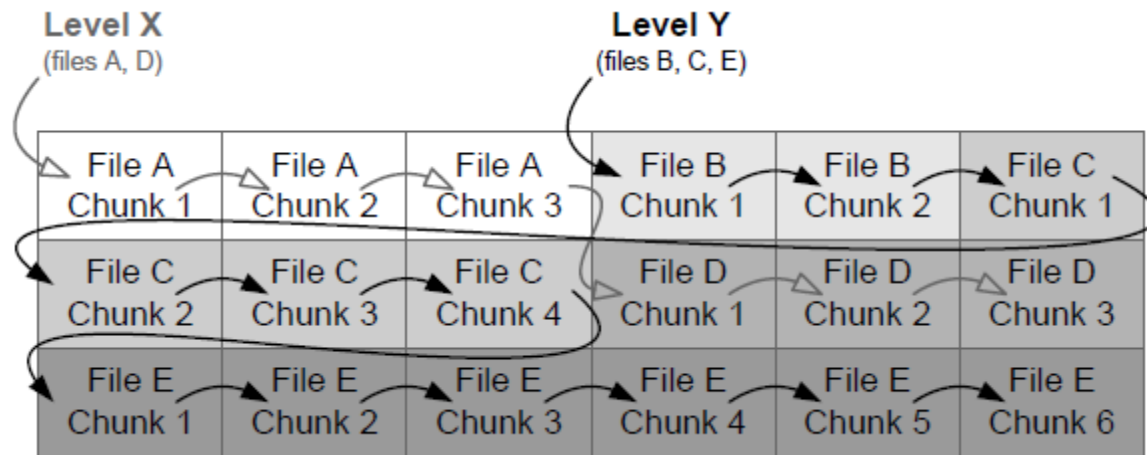


Load level B.



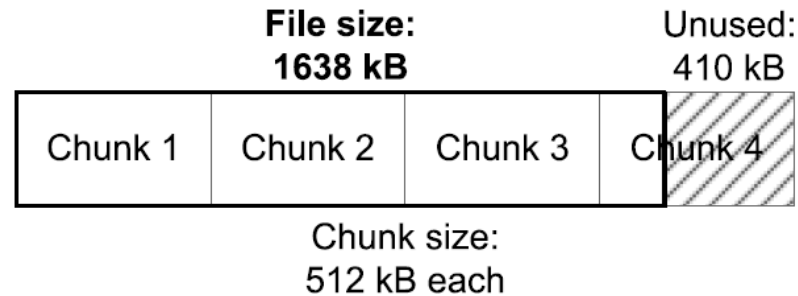
Pool allocation

- Load the data in equal size chunks
 - Requires resource to be laid out to permit chunking
- Each chunk is associated with a level



Pool allocation

- Chunks can be wasteful



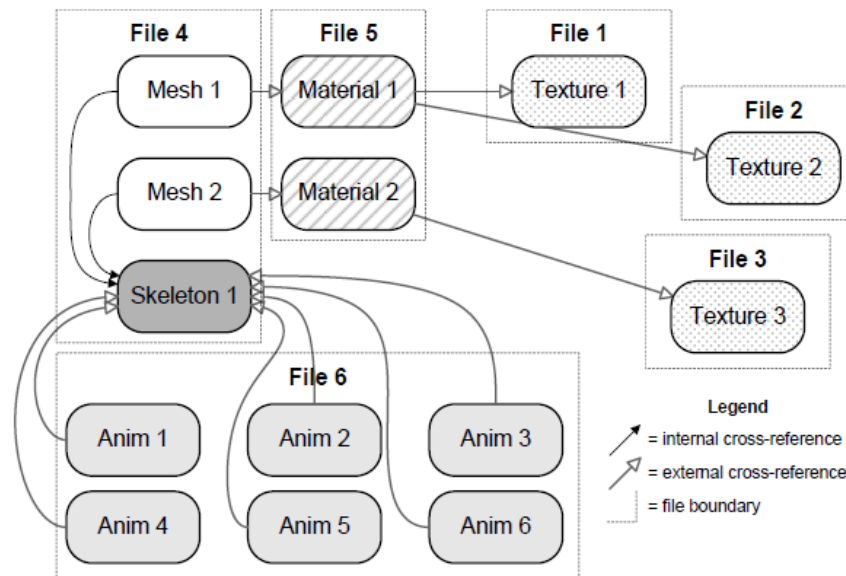
- Choose the chunk size carefully
 - Consider using the OS I/O buffer size as a guide

Resource chunk allocator

- You can reclaim unused areas of chunks
- Use a linked list of all chunks with unused memory along with the size
- Works great if the original chunk owner doesn't free it
- Can be mitigated by considering lifetimes
 - Only allocated unused parts to short lifetime objects

Composite resources

- Resource database contains multiple *resource files* each with one or more *data objects*
- Data objects can cross-reference each other in arbitrary ways
- This can be represented as a directed graph



Composite resources

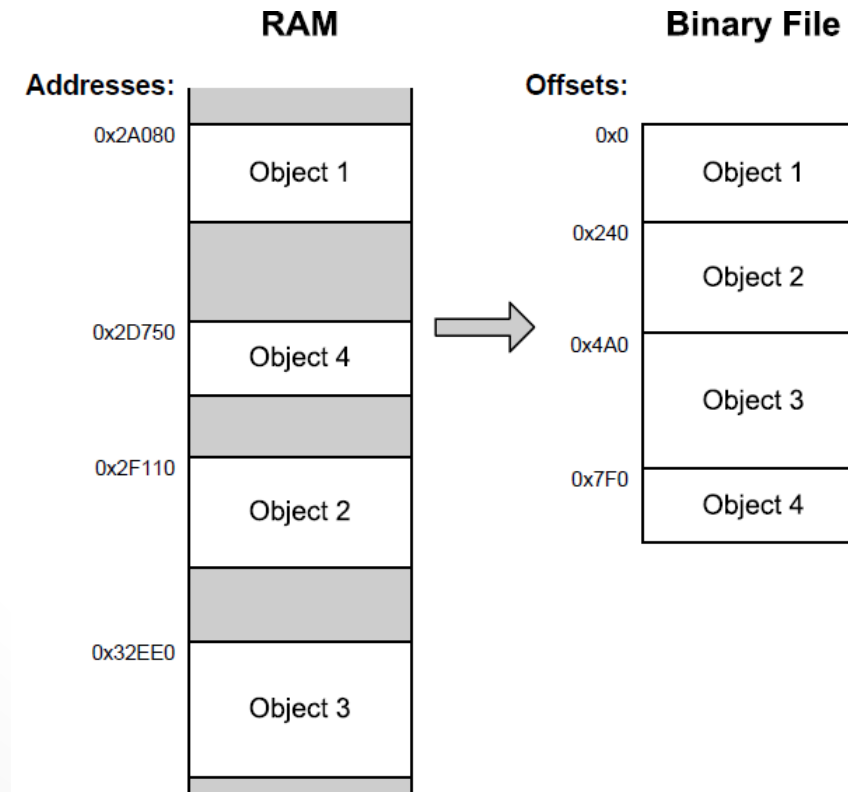
- A cluster of interdependent resources is referred to as a composite resource
- For example, a *model* consists of
 - One or more triangle meshes
 - Optional skeleton
 - Optional animations
 - Each mesh is mapped with a material
 - Each material refers to one or more textures

Handling cross-references

- Have to ensure that referential integrity is maintained
 - Can't rely on a pointer because they are meaningless in a file
- One approach is to use GUIDs
 - When a resource is loaded the GUID is stored in a hashmap along with a reference to it

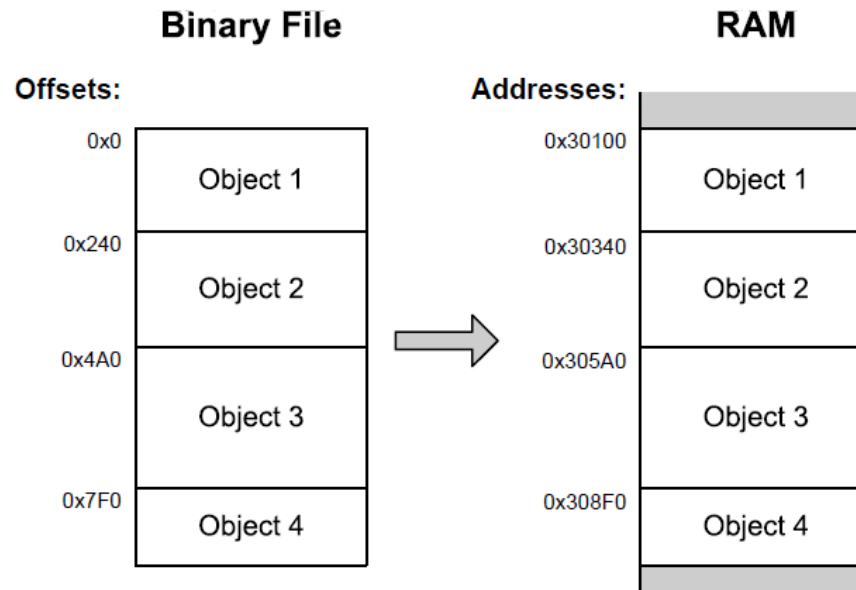
Pointer fix-up tables

- Another approach is to convert pointers to file offsets



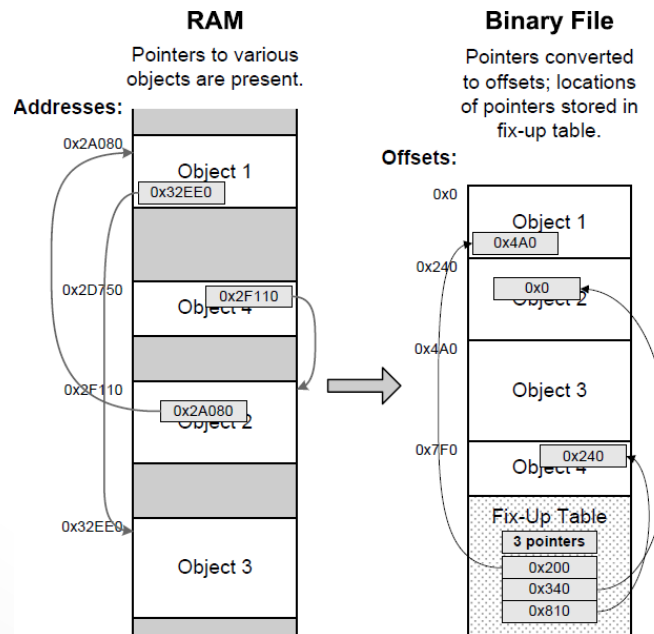
Pointer fix-up tables

- During file writing all references are converted from pointers to the offset location in the file
 - Works because offsets are smaller than pointers
- During reading, we convert offsets back to pointers
 - Known as *pointer fix-ups*
 - Easy to do because now the file is contiguous in memory



Pointer fix-up tables

- Also need to remember the location of all pointers that need fixing
- This is done by creating a table during file writing
 - Known as a *pointer fix-up table*



Constructors

- When dealing with storing C++ objects make sure you call the object constructors
- You can save the location of the objects and use *placement new* syntax to call the constructor

```
void* pObject = convertOffsetToPointer(objectOffset, pAddressOfFileImage);  
::new(pObject) ClassName;
```

Handling external references

- Externally referenced objects have to be handled differently
- Store the path along with the GUID or offset
- Load each file first then fix the references in a second pass

Post-load initialization

- Cannot always load in a ready-to-go state
 - Unavoidable – need to move vertex data to the video card
 - Avoidable, but convenient – calculating spline data during development
- In C++ using virtual functions like `init()` and `destroy()` may be the simplest strategy

Unity Assets Management

- Everything beneath the project's Asset folder

What are Assets

- Unity-Native Types (Assets/Create Menu)
 - Scenes
 - Prefabs
 - Scriptable Objects
 - Sprites
- External Data Types (Added to Assets Folder)
 - Models
 - Images
 - Audio
 - Scripts
 - Folders

Asset Handling and Pipeline

- By default assets are in binary format
 - This makes very difficult for version control system to work with
 - Unity handles this with “text serialization” in Yaml format (human readable)

Unity Basic Asset LifeCycle

1. Assign a GUID

1. Globally Unique Identifier
2. 32 hex digits: 128 bit
3. New Asset → New GUID
4. Unity uses this number to store and track the assets
5. Stores the asset in a meta file

2. Generate a Metafile

1. SomeScript.cs
2. SomeScript.cs.meta

3. Process data to Library

Meta file example

- `fileFormatVersion: 2`
- `guid: 38e917c52987daa41af6a0305d9813c6`
- `MonoImporter:`
 - `externalObjects: {}`
 - `serializedVersion: 2`
 - `defaultReferences: []`
 - `executionOrder: 0`
 - `icon: {instanceID: 0}`
 - `userData:`
 - `assetBundleName:`
 - `assetBundleVariant:`

Prefab Exercise

MonoBehaviour:

```
m_ObjectHideFlags: 0
m_CorrespondingSourceObject:
{fileID: 0}
m_PrefabInstance: {fileID: 0}
m_PrefabAsset: {fileID: 0}
m_GameObject: {fileID:
11564155490070968}
m_Enabled: 1
m_EditorHideFlags: 0
m_Script: {fileID: 11500000,
guid: 38e917c52987daa41af6a0305d9813c
6, type: 3}
m_Name:
m_EditorClassIdentifier:
projectilePrefab: {fileID: 0}
```

- Create a script
- Look at its meta file
- Copy its GUID number
- Create a game object
- Attach the script to it
- Make a prefab
- Open the .prefab using visual studio
- And search for the GUID that you copied
- You should see it at the bottom of the YAML file in Monobehavior
- Notice, there is no file name!!!
- Notice, there is also "file id"

Sound Asset Example

- Open the prototype 3
- Go to Assets/Course Library/Sound/Music
- Select one of the sound
- Look at the “Load In Background” checkbox in the inspector → it’s unchecked
- Open the folder in the File Explorer
- Open the sound “meta” file in the Visual Studio
- An change the “Load In Background” value to 1
- Save and check out the inspector
- Meta file stores all the “importer settings”

Modify Sound meta file

```
fileFormatVersion: 2
guid: 834bc99524a174a378e982623582c06d
AudioImporter:
  externalObjects: {}
  serializedVersion: 6
  defaultSettings:
    loadType: 0
    sampleRateSetting: 0
    sampleRateOverride: 44100
    compressionFormat: 1
    quality: 1
    conversionMode: 0
  platformSettingOverrides: {}
  forceToMono: 0
  normalize: 1
  preloadAudioData: 1
  loadInBackground: 1
  ambisonic: 0
  3D: 1
  userData:
  assetBundleName:
  assetBundleVariant:
```

Library

- Library is the Unity's data closet
- Go to explorer and look at "Library/Artifacts" or "Library/Import Data"
- There are 256 folders. Each folder is a combination of two hexa digits
- For example: our sound meta file, we had:
 - guid: 834bc99524a174a378e982623582c06d
 - Right Click on a "sound" file in the "project" panel, and select the "View In Import Activity Window"
- You can find the binary associated to that file!

Import Activity

Import Activity

Show OverviewOptions

Asset	Last Import	Duration (ms)
cron_audio_8-bit_modern01.ogg	5 hours ago	1,103
cm.en.txt	5 hours ago	2
plastic-gui.en.txt	5 hours ago	2
guihelp.zh-Hant.txt	5 hours ago	2
semantic.zh-Hant.txt	5 hours ago	2
guihelp.ko.txt	5 hours ago	2
mergetool.es.txt	5 hours ago	2
cm-help.es.txt	5 hours ago	2
semantic.zh-Hans.txt	5 hours ago	2
cm-help.ja.txt	5 hours ago	2
guihelp.zh-Hans.txt	5 hours ago	2
basecommands.en.txt	5 hours ago	2
guihelp.en.txt	5 hours ago	2
guihelp.ja.txt	5 hours ago	2
cm.es.txt	5 hours ago	2
semantic.en.txt	5 hours ago	2
configurehelper.en.txt	5 hours ago	3
clientcommon.es.txt	5 hours ago	2
cm-help.zh-Hans.txt	5 hours ago	2
clientcommon.en.txt	5 hours ago	2
semantic.ko.txt	5 hours ago	2
cm-help.zh-Hant.txt	5 hours ago	3
package.json	5 hours ago	2
CHANGELOG.md	5 hours ago	2
package.json	5 hours ago	3
CHANGELOG.md	5 hours ago	3
semantic.ja.txt	5 hours ago	2
mergetool.en.txt	5 hours ago	2
CHANGELOG.md	5 hours ago	2
commontypes.en.txt	5 hours ago	2
basecommands.es.txt	5 hours ago	2
package.json	5 hours ago	2
cm-help.ko.txt	5 hours ago	3
guihelp.es.txt	5 hours ago	2
plastic-gui.es.txt	5 hours ago	2
ValidationExceptions.json	5 hours ago	4
configurehelper.es.txt	5 hours ago	3
commontypes.es.txt	5 hours ago	2
cm-help.en.txt	5 hours ago	7
ProjectUnlinkBuildWarning.cs	5 hours ago	2
ActionDelegator.cs	5 hours ago	2

cron_audio_8-bit_modern01.ogg

Assetcron_audio_8-bit_modern01

GUID834bc99524a174a378e982623582c06d

Asset Size623.7 KB

PathAssets/.../cron_audio_8-bit_modern01.ogg

Editor Revision2022.1.20f1 (01d83b40d570)

Timestamp16-10-2022 11:08:57

Duration1,103 ms (+10%)

Reason for Import

Reason

the .meta file 'Assets/Course Library/Sound/Music/cron_audio_8-bit_modern01.ogg.meta' was changed

Produced Files/Artifacts (2)571.3 KB

File Library Path	Extension	Size
Library/Artifacts/93c38bf2a3fadb4b61113b1a98482890		38 KB
Library/Artifacts/67f671d3a712cab778f406513eda15f11e	.resource	533.3 KB

Dependencies (9)

Dependency Name	Dependency Value
SourceAsset/MetaFileHash/834bc99524a174a378e98262358	53435b0c6ebadc27383a629ac2a68435
SourceAsset/HashOfSourceAssetByGUID/834bc99524a174a3	0f880d7960abf4f6c0f0f6fb01e34566
ImporterRegistry/PostProcessorVersionHash/AudioPostproces	1909f56bfc062723c751e8b465ee728b
ImporterRegistry/ImporterVersion/AudioImporter	8
ImportParameter/NameOfAsset	cron_audio_8-bit_modern01.ogg
ImportParameter/ImporterType	AudioImporter
ImportParameter/BuildTargetPlatformGroup	Windows, Mac, Linux
Global/ArtifactFormatVersion	2395151755
Global/AllImporterVersion	1

BinaryToText

- Unity has a tool that converts binary to text
- Make sure that you change the highlighted version
- C:\Program Files\Unity\Hub\Editor\2022.1.20f1\Editor\Data\Tools
- Go to command line:
 - C:\Program Files\Unity\Hub\Editor\2022.1.20f1\Editor\Data\Tools>binary2text
 - Usage: binary2text inputbinaryfile [outputtextfile] [-detailed] [-largebinariyhashonly] [-hexfloat]
 - For example, Go to one of the script, find the binary file associated to it
 - C:\Program Files\Unity\Hub\Editor\2022.1.20f1\Editor\Data\Tools>binary2text "C:\Hooman\GBC\GAME3121\Unity\Create With Code\Prototype 3 - 1\Library\Artifacts\c2\c22670af92870d224e96663f8fcb2a0a"
 - It will create c22670af92870d224e96663f8fcb2a0a.txt under the same artifact folder
 - Open it up

Convert an audio binary file to text

- Exercise: Try the something you did for the script for the audio file
 - Right Click on a “sound” file in the “project” panel, and select the “View In Import Activity Window”
 - Find the binary file
 - Right click show in the explorer
 - Go to command line and run
 - C:\Program Files\Unity\Hub\Editor\2022.1.20f1\Editor\Data\Tools>binary2text "C:\Hooman\GBC\GAME3121\Unity\Create With Code\Prototype 3 - 1\Library\Artifacts\93\93c38bf2a3fadb4b61113b1a98482890"
 - Open up the audio text version

m_Resource vs. m_EditorResource

- How Unity runs the audio in the “Editor” might not be the same format as “The build target”
- So if you change your target setting to iOS, you might have a different format!

m_Resource (StreamedResource)

m_Source

"VirtualArtifacts/Primary/834bc99524a174a378e982623582c06d.resource" (string)

m_Offset 0 (FileSize)

m_Size 546112 (UInt64)

m_CompressionFormat 1 (int)

m_EditorResource (StreamedResource)

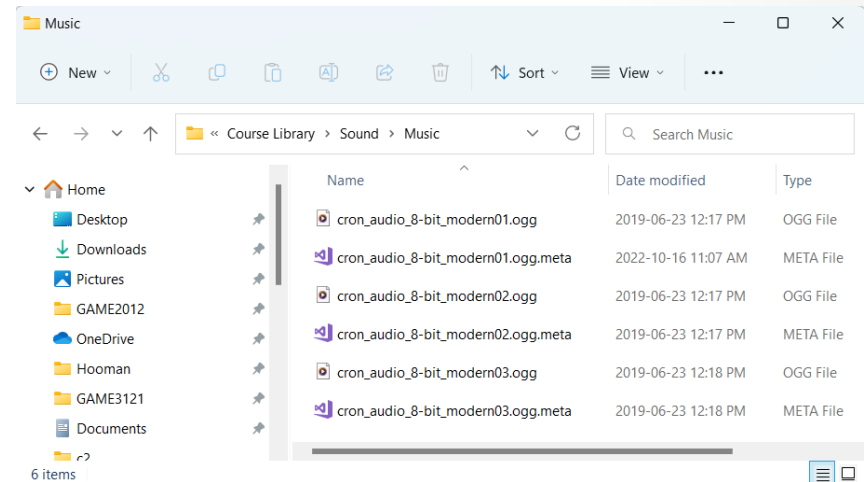
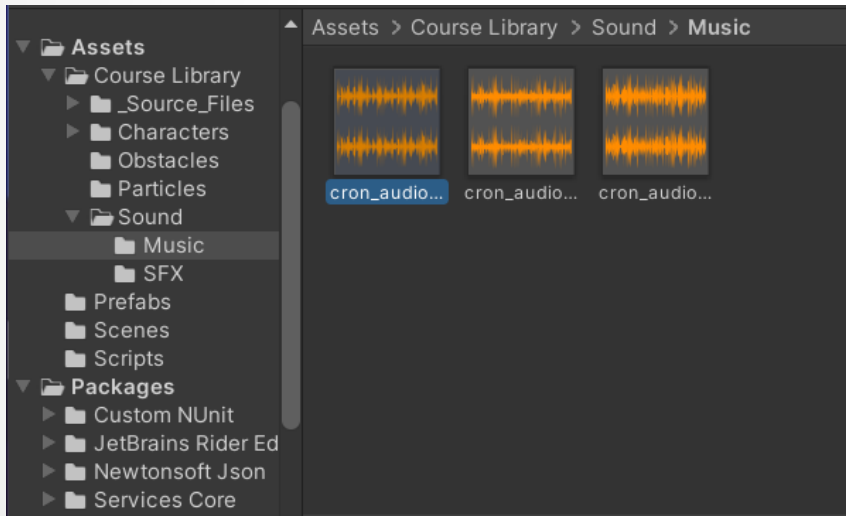
m_Source

"VirtualArtifacts/Primary/834bc99524a174a378e982623582c06d.resource" (string)

m_Offset 0 (FileSize)

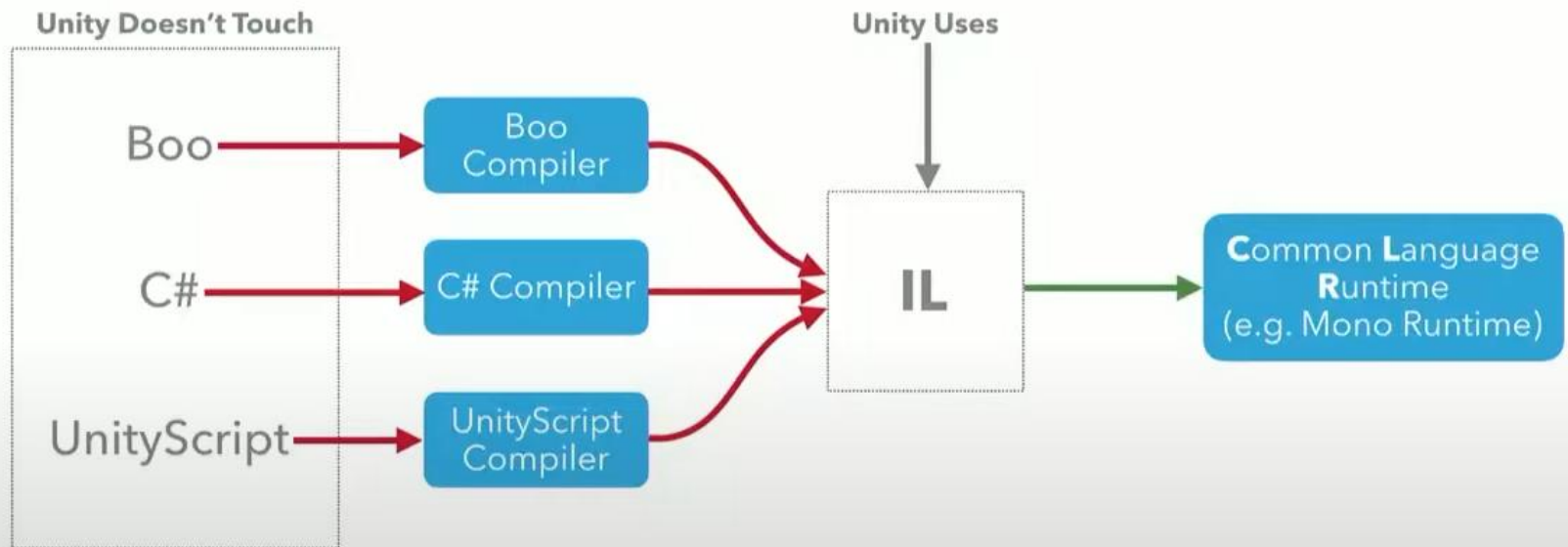
m_Size 546112 (UInt64)

Project View is not a view of the File System



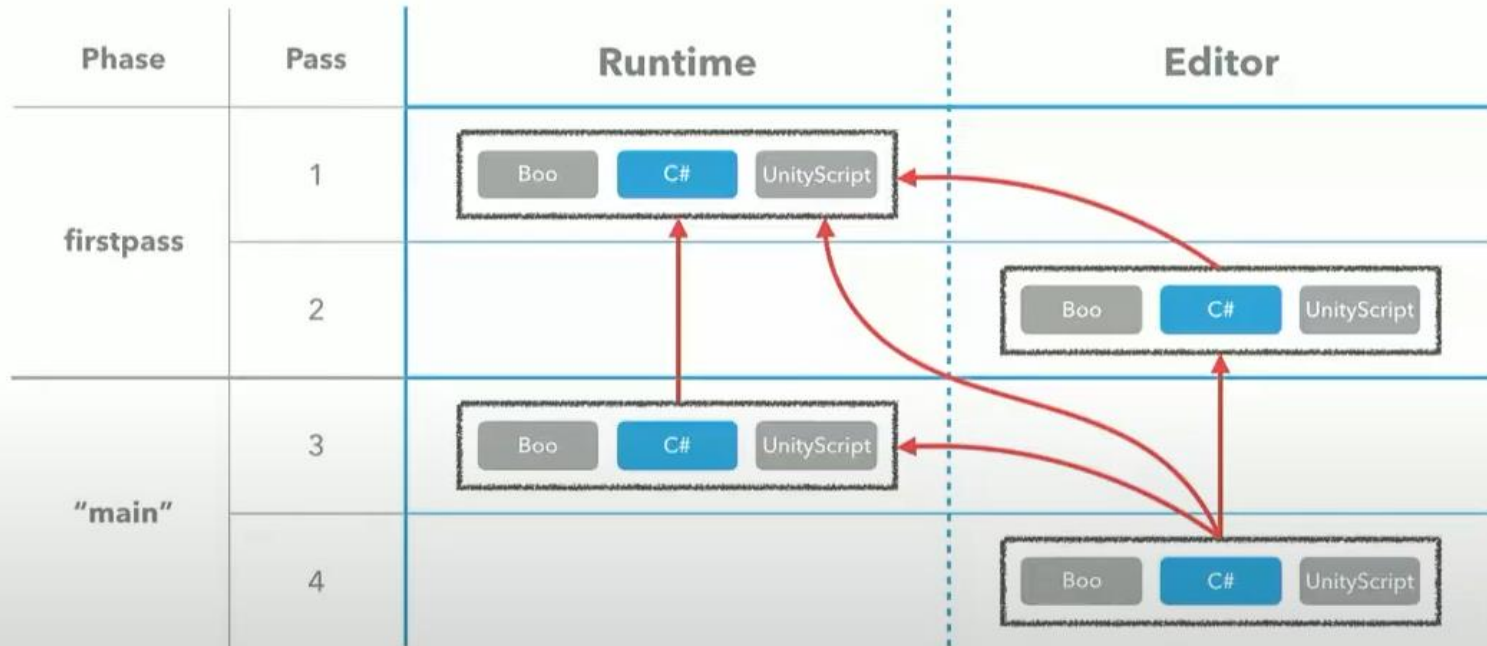
Script Compilation

UNITY RUNS ON IL

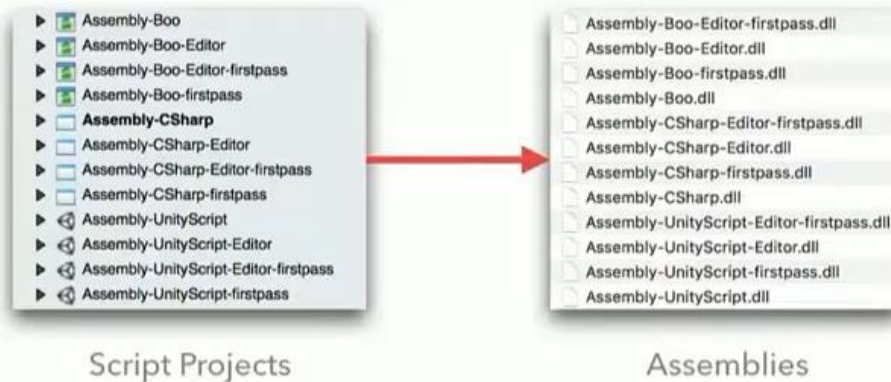


Script Compilation in 4 passes

BUILT-IN PHASES AND PASSES



Assembly Definition Files



Assembly Definition Files - JSON-formatted files that allow you to specify your own compilation pass (and output assembly).

