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

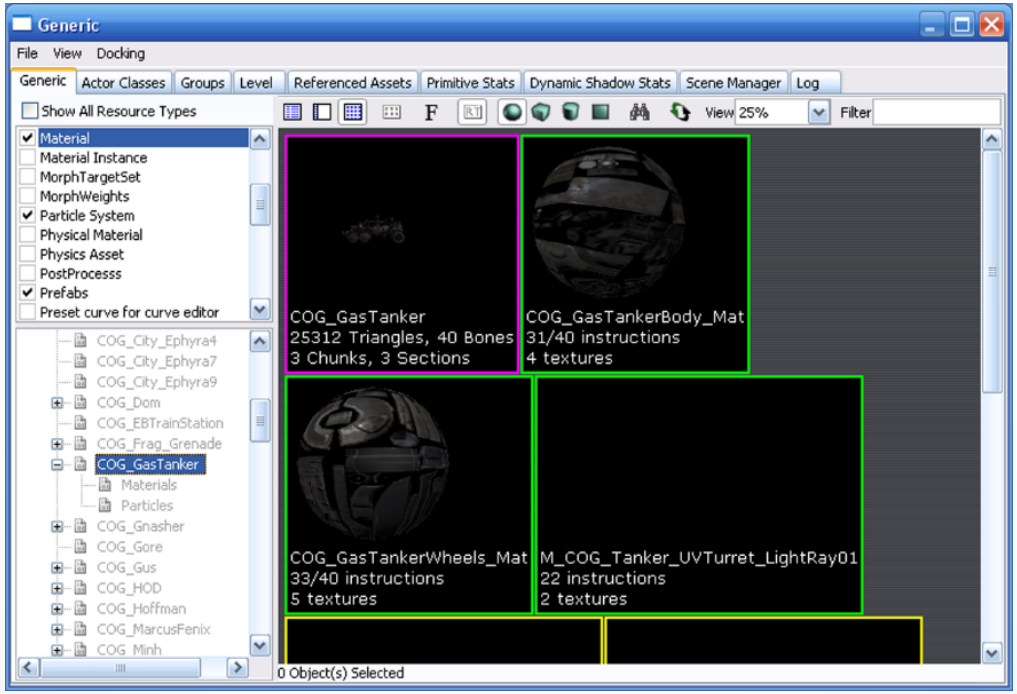
- The resource database contains information about how an asset needs to be conditioned to be useful in a game
- For example, an image may need to be flipped along its x-axis, some images should be scaled down
- This is particularly true when many people are adding assets

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

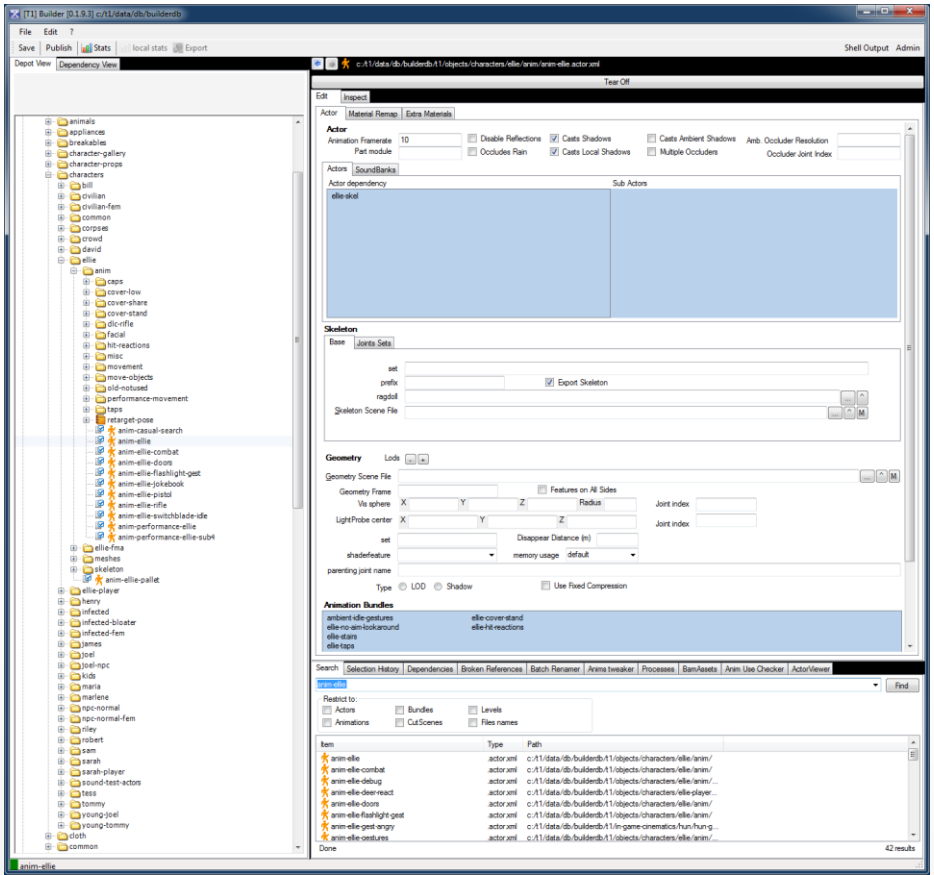
Some successful designs

- 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 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 guids

- 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

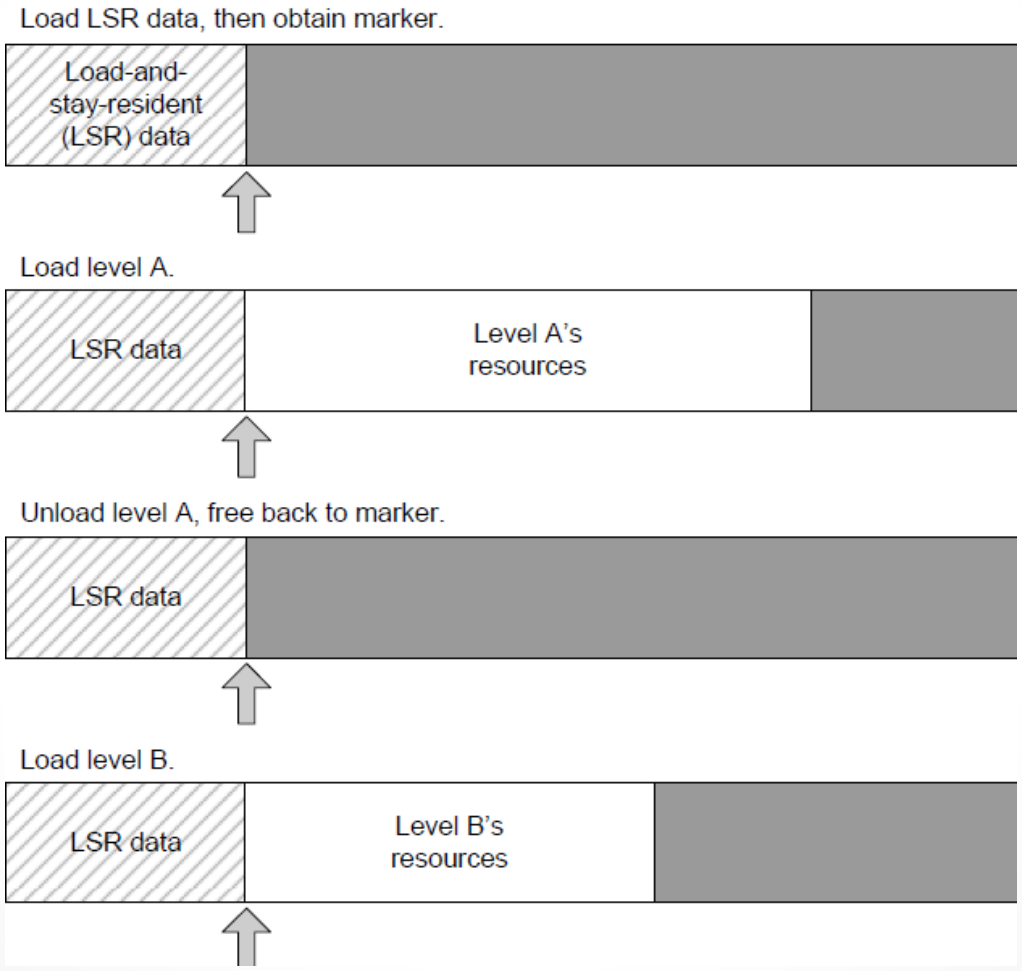
Resource lifetime

- 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

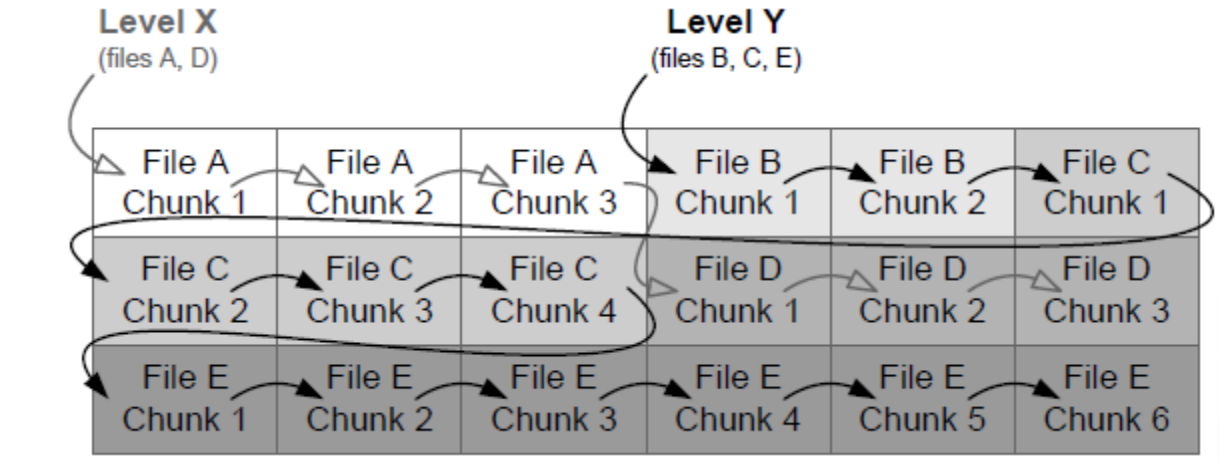
- Very closely tied to general memory management
- Heap allocation – allow the OS to handle it
  - Like malloc() or new
  - Works fine on a PC, not so much on a memory limited console
- Stack allocation
  - Can be used if
    - The game is linear and level centric
    - Each level fits in memory

Stack allocation



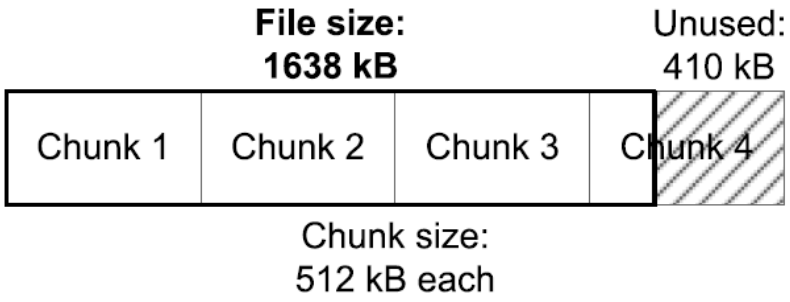
Pool allocation

- 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



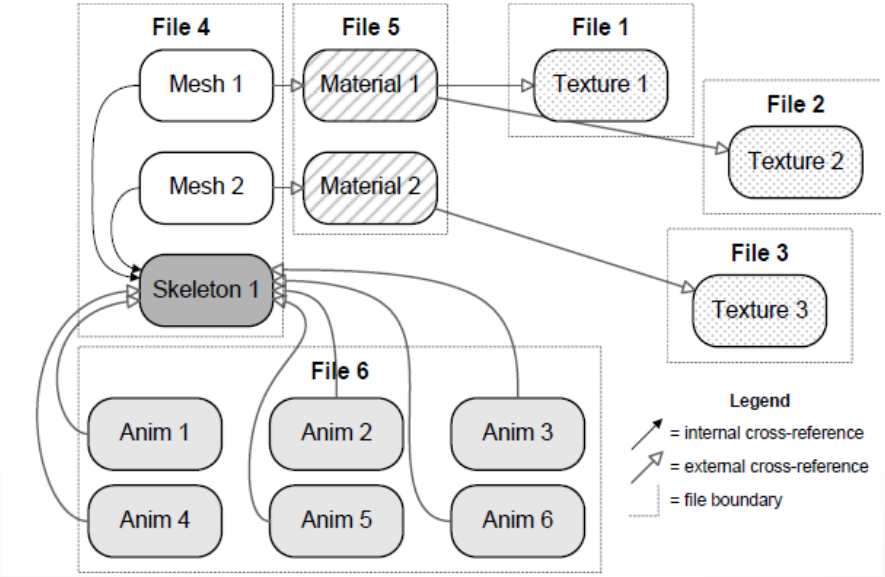
- Choose the chunk size carefully
  - Consider using the OS I/O buffer size as a guide

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

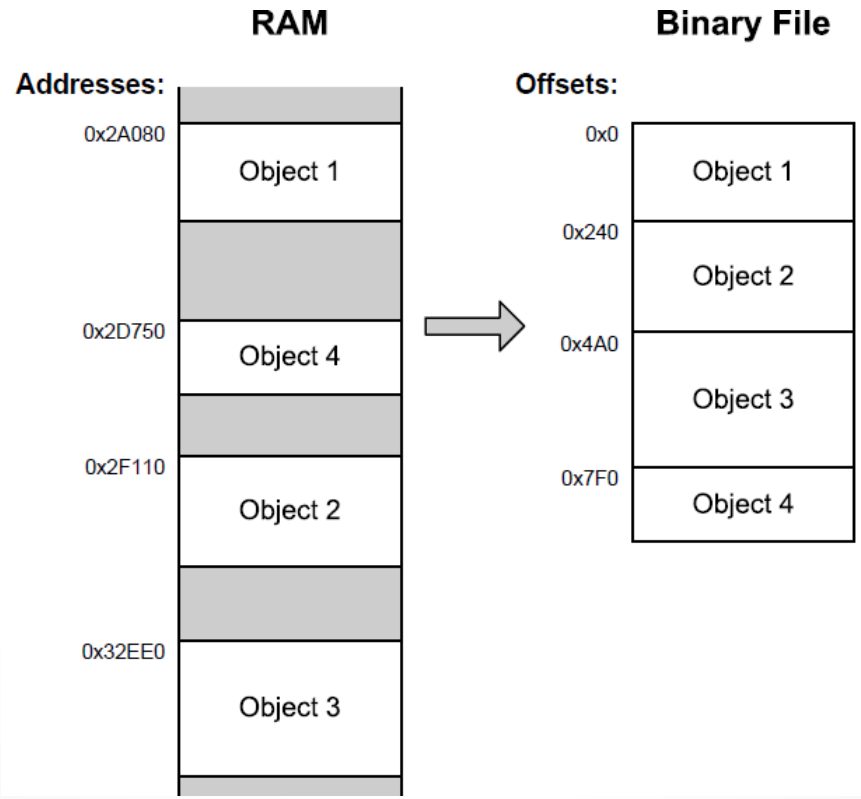
- A cluster of interdependent resources is referred to as a composite resource
- For example, a *model* consists of
  - One or more triangle meshes
  - Optional skeleton
  - Optional animations
  - Each mesh is mapped with a material
  - Each material refers to one or more textures

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

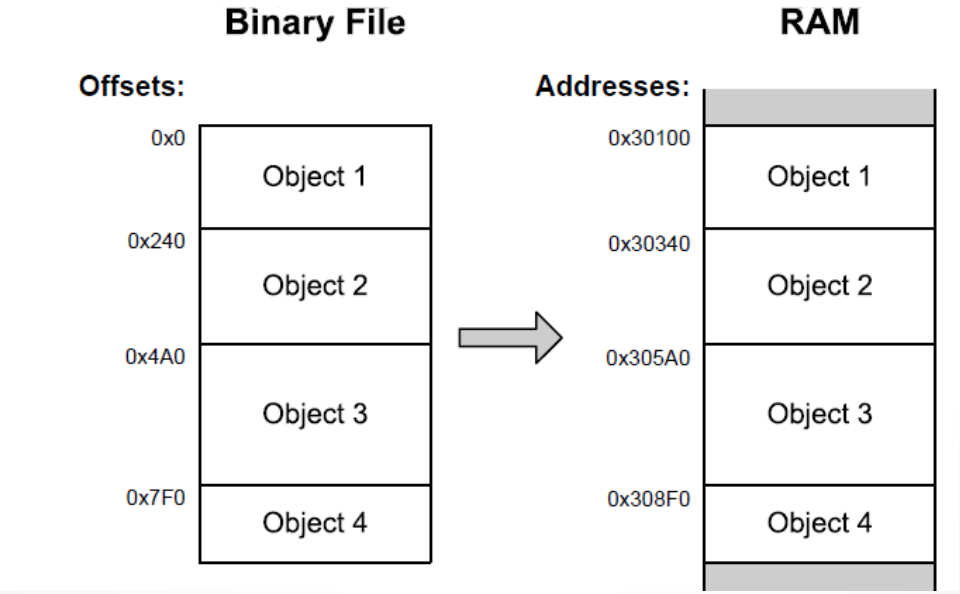
Pointer fix-up tables

- Another approach is to convert pointers to file offsets



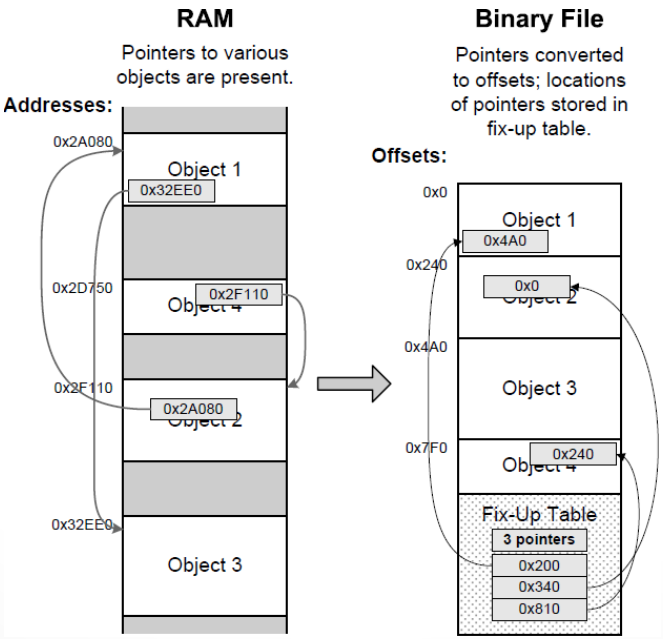
Pointer fix-up tables

- 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 are *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(objectOffser, 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