# 7. Resources and the File System

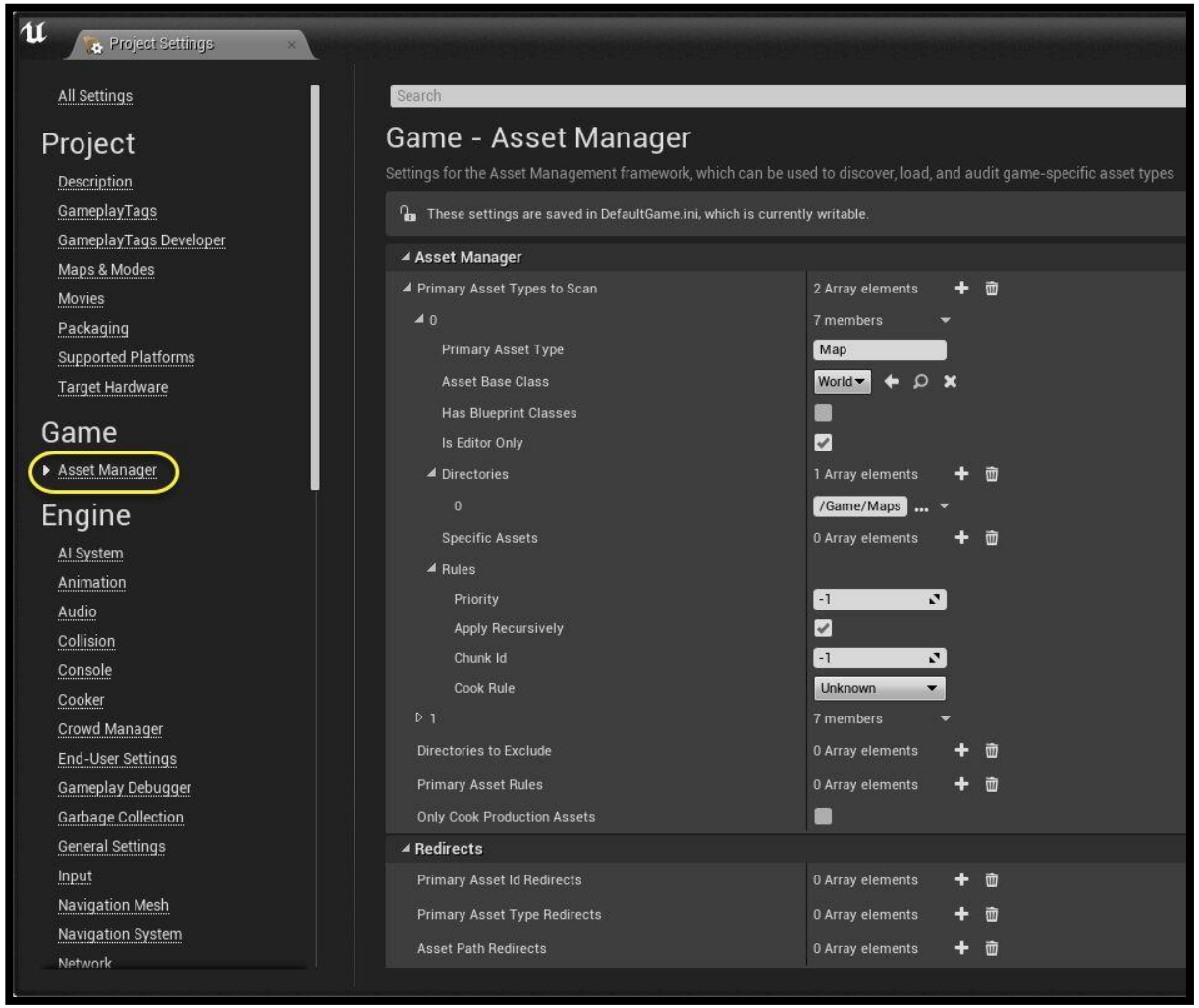
Game Player Experience Design

Prof. H. Kang

# Introduction

Games are by nature multimedia experiences.

- A game engine needs to load and manage a wide variety of different kinds of media, such as texture bitmaps, 3D mesh data, animations, audio clips, physics data, game world layouts, and the list goes on.
- Since the memory is usually scarce, a game engine needs to ensure that only one copy of each media file is loaded into memory at any given time.
- Most game engines employ some kind of resource manager (e.g. asset manager, media manager, etc.).



# File System

## File system API.

- Game engines often wrap the native file system API in an engine-specific API.
  - This enables game engine to handle cross-platform APIs seamlessly. Game engine's file system API can shield the rest of the software from differences between different target hardware platforms.
  - This also supports tools that the operating system's file system API might not provide.
- A game engine's file system API typically addresses the following areas of functionality:
  - manipulating file names and paths
  - opening, closing, reading, and writing individual files,
  - scanning the contents of a directory,
  - handling asynchronous file I/O requests (for streaming).

# File System

## File names and paths

- A path is a string describing the location of a file or directory within a file system hierarchy.
- Each operating system introduces slight variations on the general path structure.
  - UNIX uses a forward slash (/) as its path component separator, while DOS and older versions of Windows used a backslash (\) as the path separator.
  - Mac OS 8 and 9 use the colon (:) as the path separator character.
  - On Microsoft Windows, volumes can be specified in two ways.
    - A local disk drive is specified using a single letter followed by a colon (e.g., the ubiquitous c:).
    - A remote network share can either be mounted so that it looks like a local disk, or it can be referenced via a volume specifier consisting of two backslashes followed by the remote computer name and the name of a shared directory or resource on that machine (e.g., \\cat-computer\temporal-share).
  - UNIX and its variants don't support volumes as separate directory hierarchies. Therefore, UNIX paths never have a volume specifier.
  - Under DOS and early versions of Windows, a file name could be up to eight characters in length, with a three-character extension which was separated from the main file name by a dot (.txt, .exe).
  - Each operating system disallows certain characters in the names of files and directories. For example, a colon cannot appear anywhere in a Windows or Dos path except as part of a drive letter volume specifier.
  - Consoles often employ a set of predefined path prefixes to represent multiple volumes. For example, PlayStation 3 uses the prefix *dev_bdvd*\ to refer to the Blu-ray disk drive, while *dev_hddx*\ refers to one or more hard disks.

# File System

## Search paths

- The term *path* should not be confused with the term *search path*.
  - A *path* is a string representing the location of a single file or directory within the file system hierarchy.
  - A *search path* is a string containing a list of paths, each separated by a special character such as a colon or semicolon, which is searched when looking for a file.
- Game engines also use *search paths* to locate resource files.
  - The OGRE rendering engine uses a resource search path contained in a text file named *resources.cfg*.
  - It provides a simple list of directories and ZIP archives that should be searched in order when trying to find an asset.

## Path APIs

- Clearly, there are many things a programmer may need to do when dealing with paths:
  - Isolating the directory
  - Canonicalizing a path
  - Converting back and forth between absolute and relative paths
  - Obtaining filename or extension
- Therefore, it can be helpful to have a feature-rich API to help with there tasks.

# Quiz

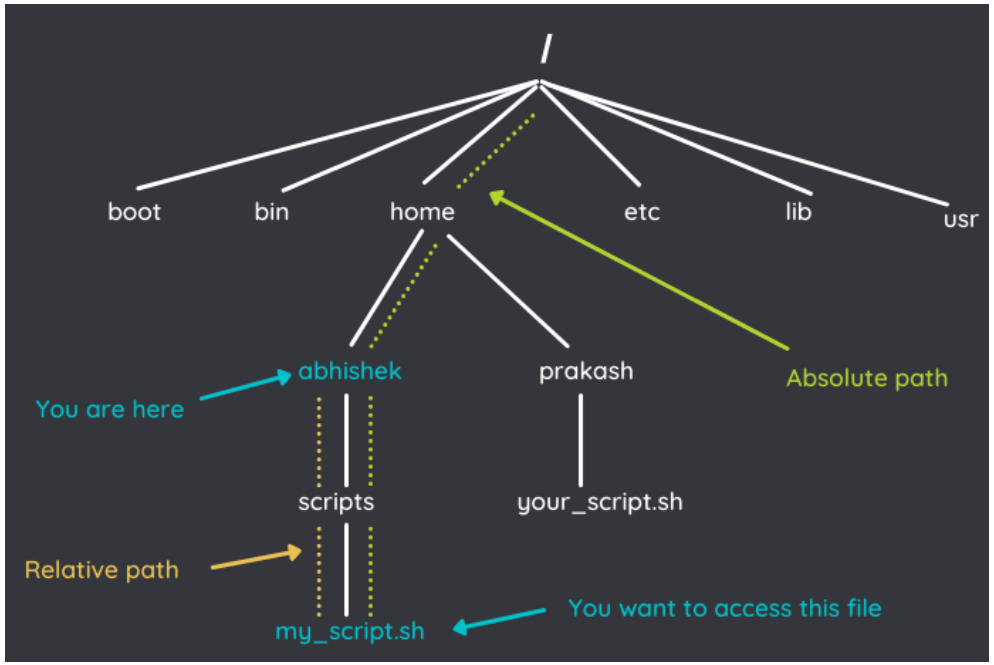
1. Absolute and relative paths
   - Absolute path?
   - Relative path?

2. Buffered and unbuffered file I/O
   - Buffered file I/O?
   - Unbuffered file I/O?

# File System

## Absolute and relative paths

- All paths are specified relative to some location within the file system.
- When a path is specified relative to the root directory, we call it an absolute path.
  - C:\Windows\System32
  - D:\game\assets\meshes\cat.obj
- When a path is relative to some other directory in the file system hierarchy, we call it a relative path.
  - System32 (relative to CWD, \windows)
  - \meshes\cat.obj (relative to CWD, D:\game\assets)




path example[path]

# File I/O

## Buffered vs. unbuffered

- The standard C library provides two APIs for opening, reading and writing the contents of files: buffered and unbuffered.
- Buffered API
  - Buffered I/O API requires data blocks known as buffers to serve as the source or destination of the bytes passing between the program and the file on disk.
  - Instead of reading a file byte by byte, buffered output streams will accumulate outputs into an intermediate buffer, sending it to OS file system only when enough data has accumulated.
  - This reduces the number of file system calls.
  - Sometimes, this types of APIs are referred to as the stream I/O API, because they provide an abstraction which makes disk files look like streams of bytes.
  - Buffered I/O try to minimize the number of access to the dis
  - Buffered I/O reduces the number of file system calls.
- Unbuffered API
  - Unbuffered output can be used when a programmer already has large sequence of bytes ready to write to disk, and want to avoid an extra copy into a second buffer in the middle.
  - Unbuffered IO does not guarantee the data has reached the physical disk.

| 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 to an offset | `fseek()` | `seek()` |
| Return current offset | `ftell()` | `tell()` |
| Read a single line | `fgets()` | n/a |
| Write a single line | `fputs()` | n/a |
| Read formatted string | `fscanf()` | n/a |
| Write formatted string | `fprintf()` | n/a |
| Query file status | `fstat()` | `stat()` |

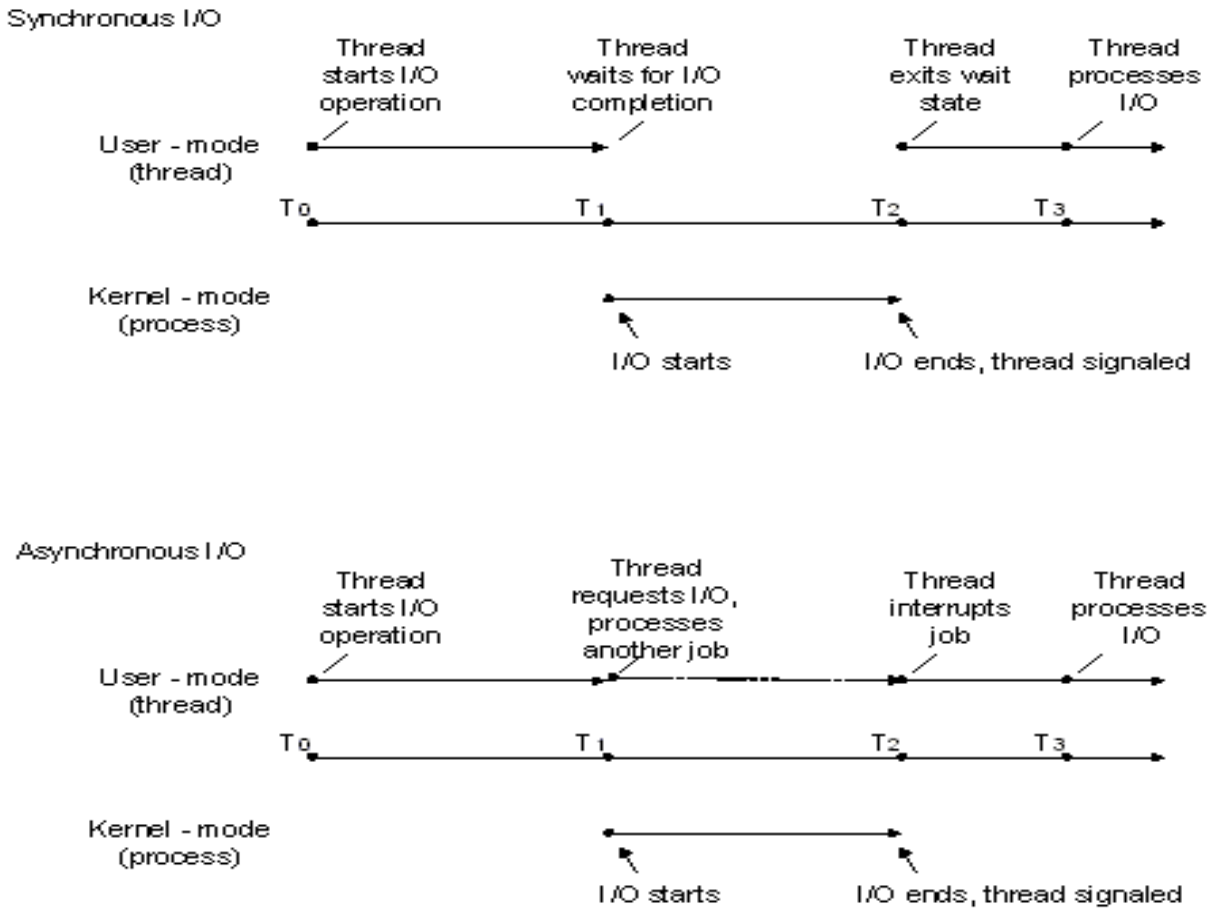
# File I/O

## Synchronous file I/O

- In synchronous file I/O, a thread starts an I/O operation and immediately enters a wait state until the I/O request has completed.

## Asynchronous file I/O

- Many games provide the player with a seamless, load-screen-free playing experience by streaming data for upcoming levels while the game is being played.
- In order to support streaming, we must use an asynchronous file I/O which permits the program to continue to run while its I/O requests are being satisfied.




differences[IO]

# File I/O

## Priorities

- Asynchronous I/O operations often have varying priorities since the file I/O is a real time system.
- For example, if we are streaming audio from the hard disk or Blu-ray and playing it on the fly, loading the next buffer full of audio data is clearly higher priority than loading a texture or a chunk of a game level.
- Therefore, asynchronous I/O systems must be capable of suspending lower-priority requests, so that higher-priority I/O requests have a chance to complete within their deadlines.

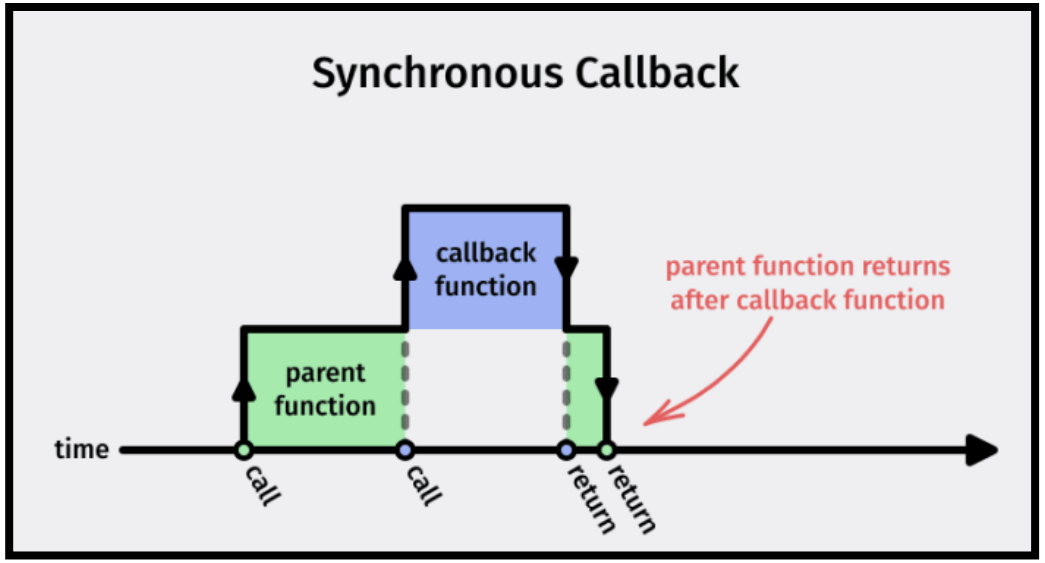
## How asynchronous file I/O works

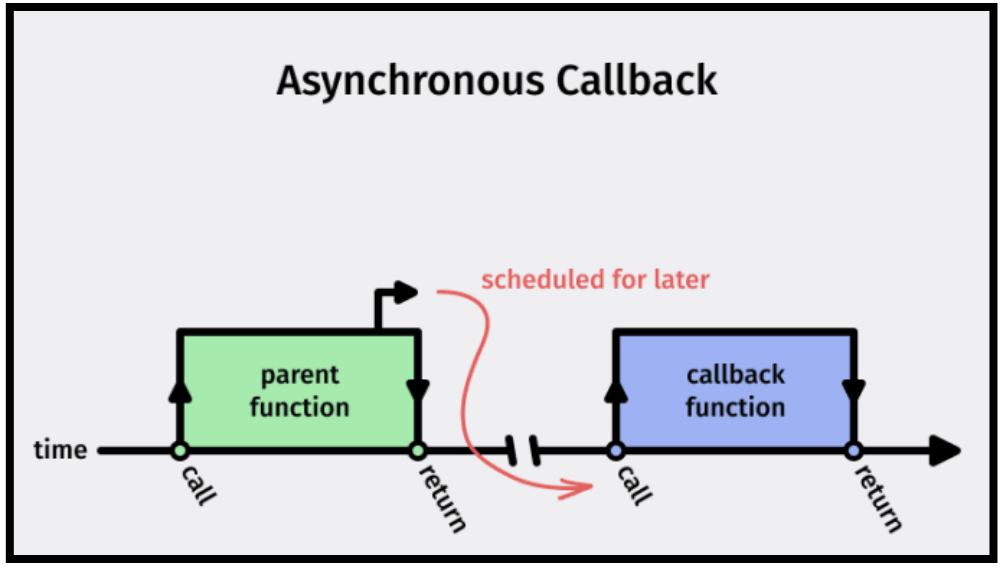
- Asynchronous file I/O works by handling I/O requests in a separate thread.
- The main thread calls functions that simply place requests on a queue and then return immediately.
- Meanwhile, the I/O thread picks up requests from the queue and handles them sequentially using blocking I/O routines like *read*() or *fread*().
- When a request is completed, a **callback** provided by the main thread is called, thereby notifying it that the operation is done.
- If the main thread chooses to wait for an I/O request to complete, this is handled via a **semaphore**.

# File I/O

## Callback

- A callback, also known as a “call-after” function, is an executable code that is passed as an argument to other code; that other code is expected to execute (call back) the argument at a given time.
- This execution may be immediate as in a synchronous callback, or might happen at a later point in time as in an asynchronous callback.
- Callbacks have a wide variety of uses, for example in error signaling, or event handling.

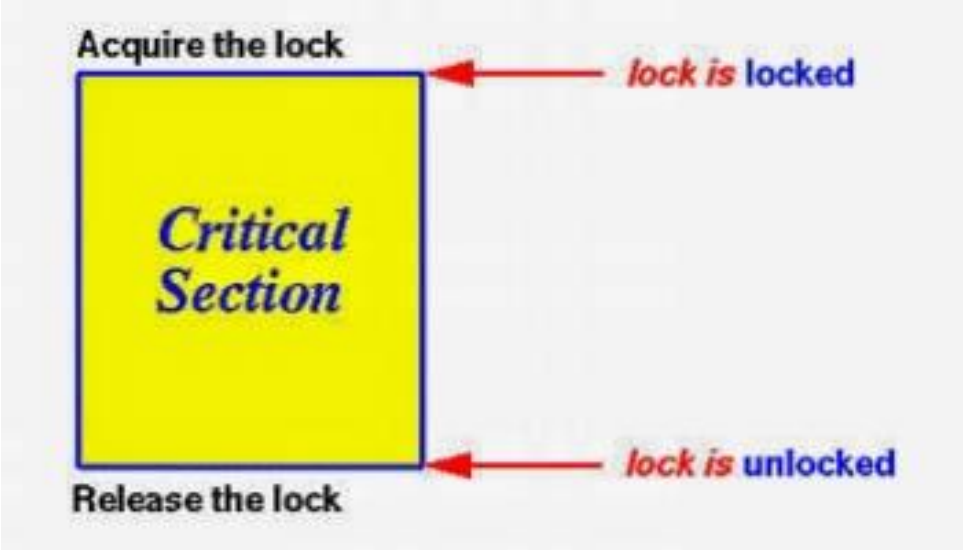


# File I/O

## Semaphore

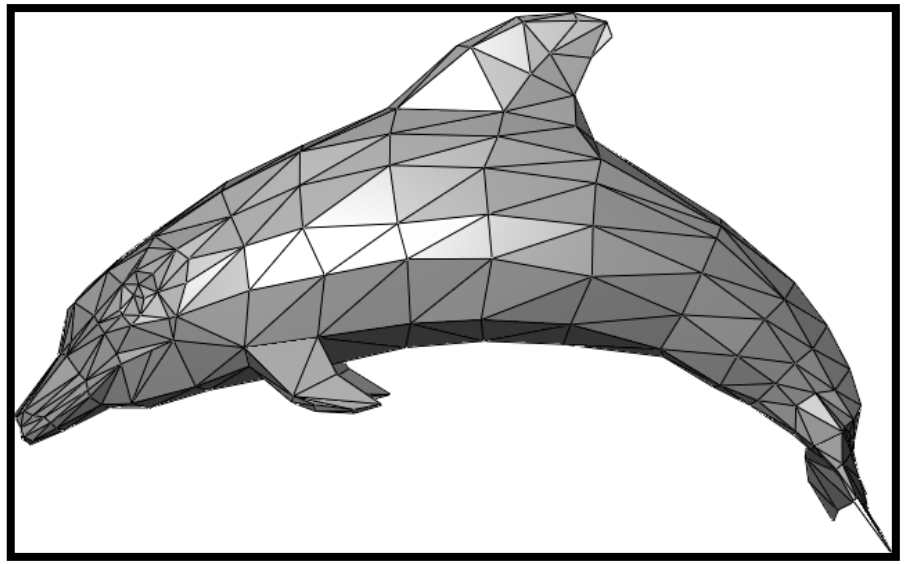
- Semaphore is simply a variable or abstract data type that is used to control access to a common resource by multiple processes and avoid critical section problem in a concurrent system.
- A semaphore is a signaling mechanism which uses two atomic operations. 1) wait, and 2) signal for the process synchronization.

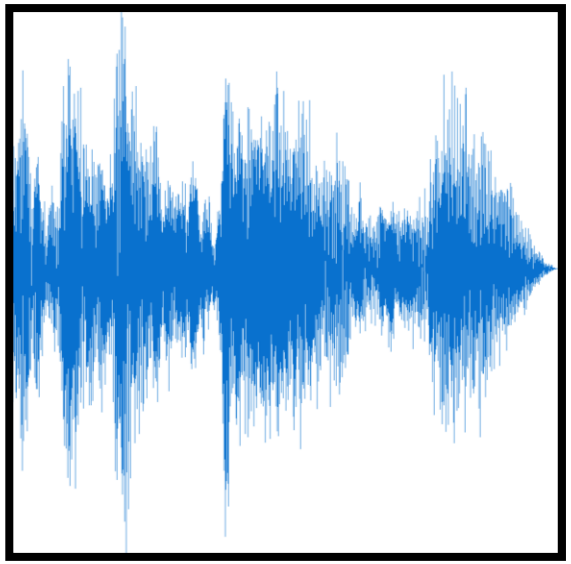


# Resource Manager

## Resources

- Every game is constructed from a wide variety of resources (sometimes called assets or media).
  - Meshes
  - Textures
  - Materials
  - Shader programs
  - Sound files
  - ...



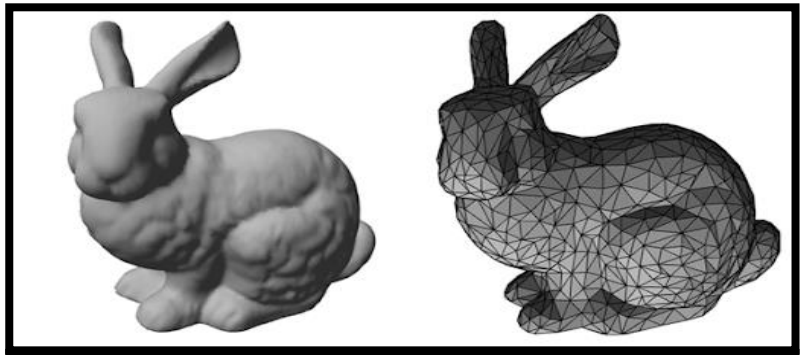


# Resource Manager

## Resources

- Mesh (or geometry in graphics)
  - The most common 3D mesh formats are STL, OBJ, FBX, COLLADA, 3DS, etc.
  - The OBJ file format is a simple data-format that represents 3D geometry.
  - An OBJ file may contain vertex data, free-form curve/surface attributes, elements, connectivity between free-form surface, etc.
  - The most common elements are geometric vertices, texture coordinates, vertex normal, and polygonal faces.

```
# List of geometric vertices, with (x, y, z [,w]) coordinates, w is optional and defaults to 1.0.
v 0.123 0.234 0.345 1.0
v ...
...
# List of texture coordinates, in (u, [,v ,w]) coordinates, these will vary between 0 and 1. v, w are optional and default to 0.
vt 0.500 1 [0]
vt ...
...
# List of vertex normals in (x,y,z) form; normals might not be unit vectors.
vn 0.707 0.000 0.707
vn ...
...
# Parameter space vertices in ( u [,v] [,w] ) form; free form geometry statement ( see below )
vp 0.310000 3.210000 2.100000
vp ...
...
# Polygonal face element (see below)
f 1 2 3
f 3/1 4/2 5/3
f 6/4/1 3/5/3 7/6/5
f 7//1 8//2 9//3
f ...
...
# Line element (see below)
l 5 8 1 2 4 9
```

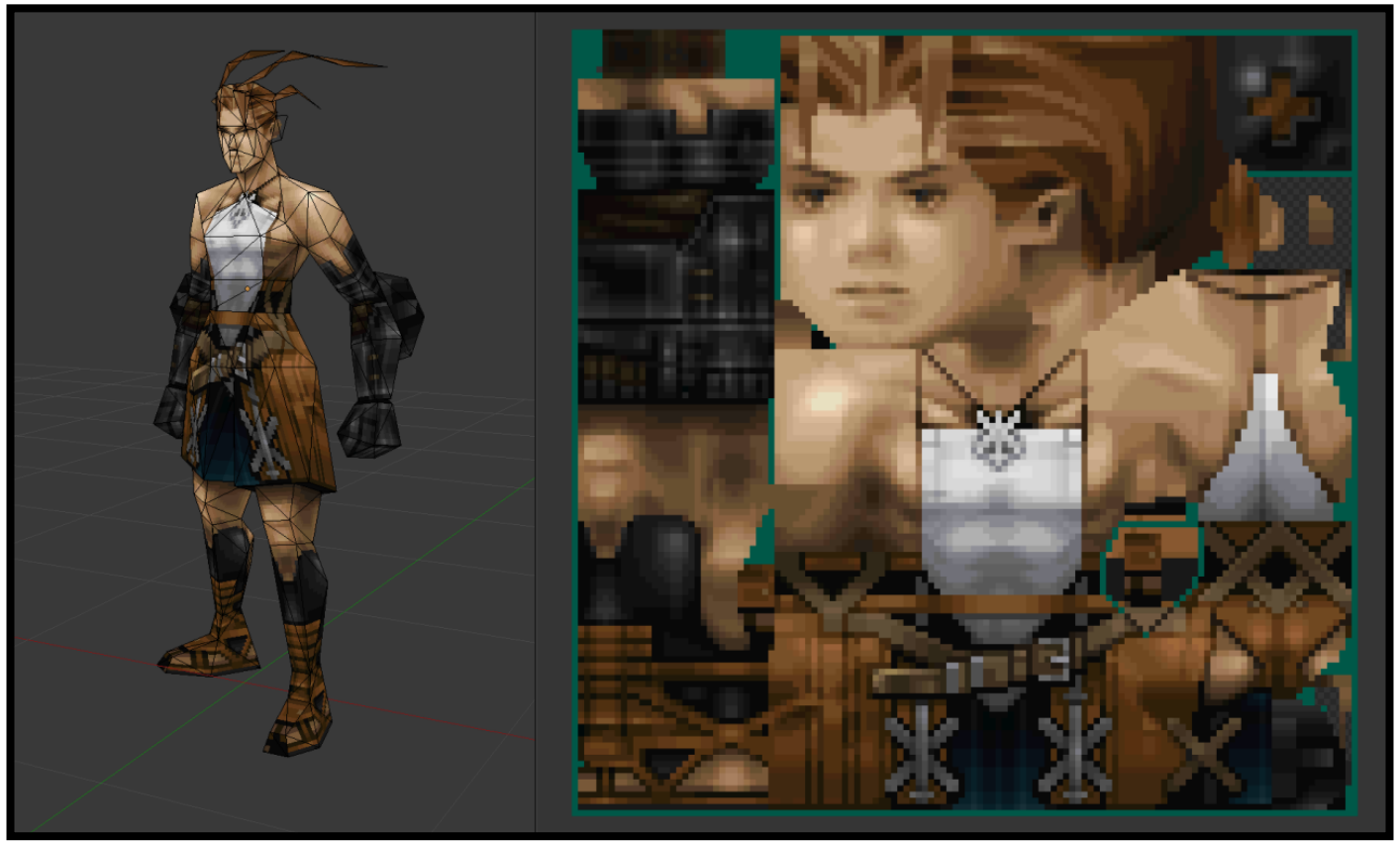


```
# OBJ file format with ext .obj
# vertex count = 2503
# face count = 4968
v -3.4101800e-003 1.3031957e-001 2.1754370e-002
v -8.1719160e-002 1.5250145e-001 2.9656090e-002
v -3.0543480e-002 1.2477885e-001 1.0983400e-003
v -2.4901590e-002 1.1211138e-001 3.7560240e-002
v -1.8405680e-002 1.7843055e-001 -2.4219580e-002
v 1.9067940e-002 1.2144925e-001 3.1968440e-002
v 6.0412000e-003 1.2494359e-001 3.2652890e-002
v -1.3469030e-002 1.6299355e-001 -1.2000020e-002
v -3.4393240e-002 1.7236688e-001 -9.8213000e-004
```

# Resource Manager

## Resources

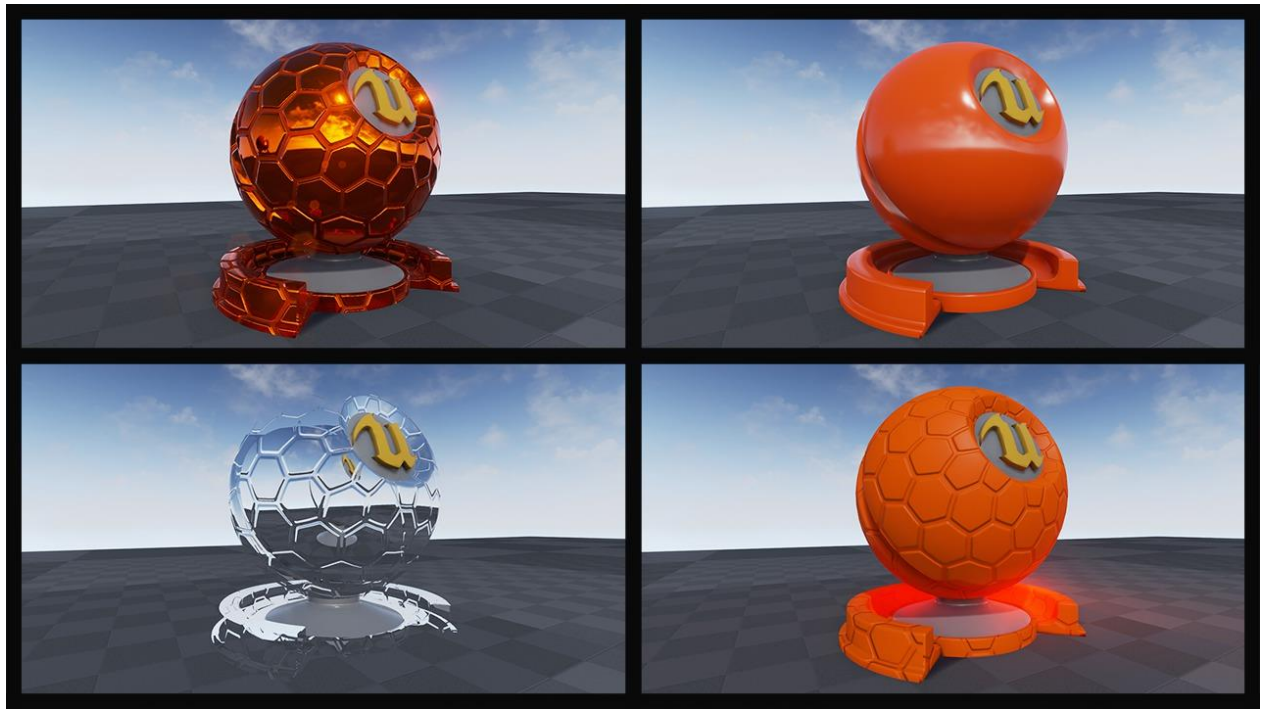
- Texture
  - Textures are an image used to skin 3D objects.
  - PNG or JPEG could serve as a texture, but more high-quality images are usually used as a texture.
  - Furthermore, texture can be created procedurally (we call this procedural texture).
  - Some objects will use the multiple textures.



# Resource Manager

## Resources

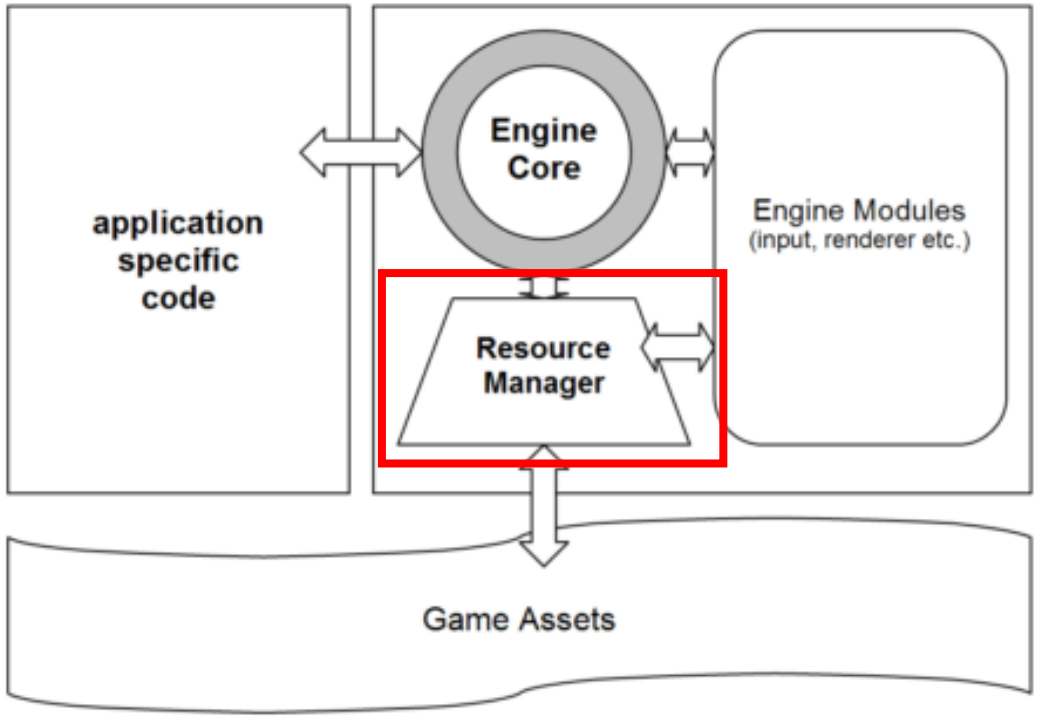
- Material
  - Materials are used to paint the 3D model surface, controlling how dull or reflective the surface appears.
  - To determine material, many graphics principles are combined and applied to the object: lighting, surface normal, displacement mapping, etc.



# Resource Manager

## Role of resource manager

- One major role of resource manager is to manage the chain of offline tolls used to create the assets and transform them into their engine-ready form.
- The other role is to manage the resources at runtime, ensuring that they are loaded into memory in advance of being needed by the game and making sure they are unloaded from memory when no longer needed.




a conceptual model of a game engine[resource]

# Resource Manager

## Revision control for assets

- On a small game project, the game's assets can be managed by keeping loose files sitting around on a shared network drive with an ad hoc directory structure.
- However, this approach is not feasible for a modern commercial 3D game, comprised of a massive number and variety of assets.
- For such a project, the team requires a more formalized way to track and manage its assets.

## Revision control (Data size)

- One of the biggest problems in the revision control of art assets is the sheer amount of data.
  - Whereas C++ and script source code files are small, relative to their impact on the project, art files tend to be much, much larger.

# Resource Manager

## Runtime resource management

- A game engine’s runtime resource manager takes on a wide range of responsibilities, all related to its primary mandate of loading resources into memory:
  - Ensures that only one copy of each unique resource exists in memory at any given time.
  - Manages the lifetime of each resource.
  - Loads needed resources and unloads resources that are no longer needed.
  - Maintains referential integrity. (For example, when loading a composite resource, the resource manager must ensure that all necessary sub-resources are loaded, and it must patch in all of the cross-references properly.)
  - Manages the memory usage of loaded resources and ensures that resources are stored in the appropriate place in memory.

# Resource Manager

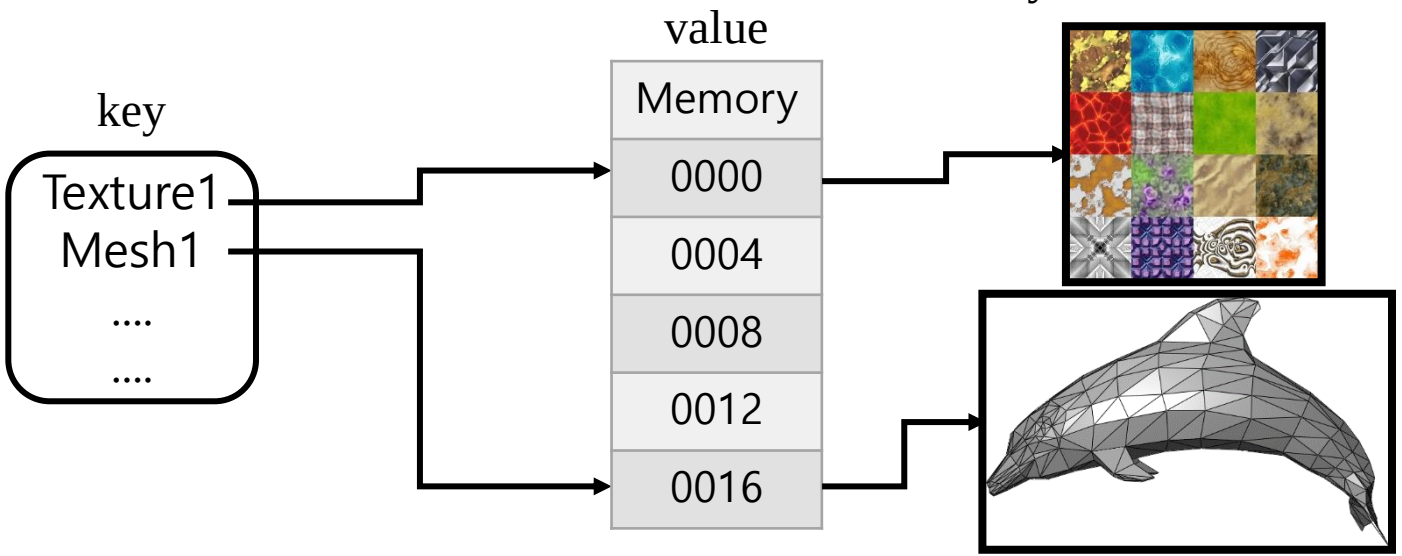
## Resource GUIDs

- Every resource in a game must have some kind of globally unique identifier (GUID).
- The most common choice of GUID is the resource's file system path (stored either as a string or a 32-bit hash).
- This kind of GUID is intuitive, but it is infeasible in some engines.
  - In Unreal Engine, many resources are stored in a single large file known as a package.
  - Therefore, the path to the package file does not uniquely identify any one resource.
- To overcome this problem, some engines use a less-intuitive type of GUID, such as 128-bit hash code generated by a tool that guarantees uniqueness.
- In Unreal Engine, package file is organized into a folder hierarchy containing individual resources. This gives each individual resource within a package a unique name, which looks much like a file system path.

# Resource Manager

## The resource registry

- In order to ensure that only one copy of each unique resource is loaded into memory at any given time, most resource managers maintain some kind of registry of loaded resources.
- The simplest implementation is a *dictionary* - a collection of *key-value pairs*.
- The keys contain the unique ids of the resources, while the values are typically pointers to the resources in memory.
  - Whenever a resource is loaded into memory, an entry for it is added to the resource registry dictionary, using its GUID as the key.
  - Whenever a resource is unloaded, its registry entry is removed.
  - When a resource is requested by the game, the resource manager looks up the resource by its GUID within the resource registry.
  - If the resource can be found, a pointer to it is simply returned.
  - If the resource cannot be found, it can either be loaded automatically or a failure code can be returned.



# Resource Manager

## The resource registry

- At first blush, this might seem most intuitive to automatically load a requested resource if it cannot be found in the resource registry.
- However, this approach has some serious problems.
  - Loading a resource is a low operation, because it involves locating and opening a file on disk, reading a potentially large amount of data into memory.
  - If the request comes during the gameplay, the time it takes to load the resource might cause a very noticeable hitch in the game’s frame rate, or even a multi-second freeze.

# Quiz

## The resource registry

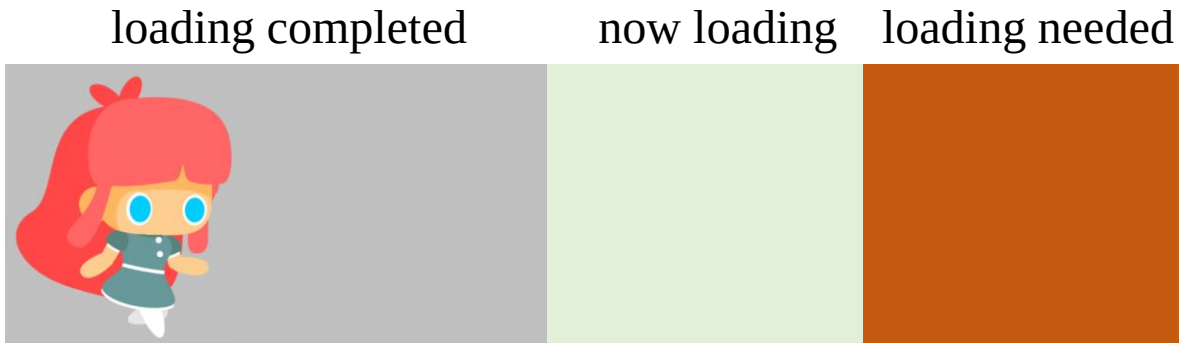
- Then, how to overcome this problem?

# Resource Manager

## The resource registry

- Game engines tend to take one of two alternative approaches:
  - Resource loading might be disallowed completely during active gameplay. In this model, all of the resources for a game level are loaded en masse just prior to gameplay, usually while the player watches a loading screen or progress bar of some kind.


  - Resource loading might be done asynchronously (i.e., the data might be streamed). In this model, while the player is engaged in level X, the resources for level Y are being loaded in the background. This approach is preferable because it provides the player with a load-screen-free play experience. However, it is considerably more difficult to implement.



# Resource Manager

## Resource lifetime

- The life time of a resource is defined as the time period between when it is first loaded into memory and when its memory is reclaimed for other purposes.
- One of the resource manager's jobs is to manage resource lifetimes—either automatically or by providing the necessary API functions to the game, so it can manage resource lifetimes manually.
- Each resource has its own lifetime requirements:

1) Some resources must be loaded when the game first starts up and must stay resident in memory for the entire duration of the game (infinite lifetime).
    - These are sometimes called global resources or global assets.
    - Player character's mesh, fonts used on the heads-up display, ...

2) Other resources have a lifetime that matches that of a particular game level. These resources must be in memory by the time the level is first seen by the player and can be dumped once the player has permanently left the level.

# Reference

[path] https://linuxhandbook.com/absolute-vs-relative-path/
[IO] https://docs.microsoft.com/en-us/windows/win32/fileio/synchronous-and-asynchronous-i-o
[resource] A Taxonomy of Game Engines and the Tools that Drive the Industry (article).