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

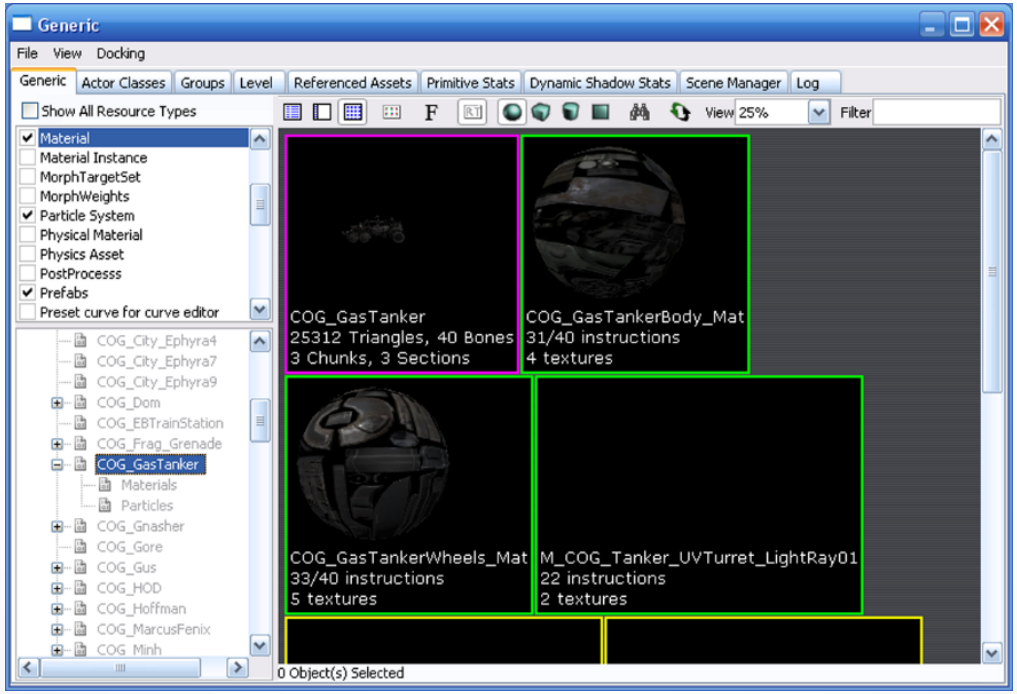
- The resource database contains information about how an asset needs to be conditioned to be useful in a game
- For example, an image may need to be flipped along its x-axis, some images should be scaled down
- This is particularly true when many people are adding assets

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

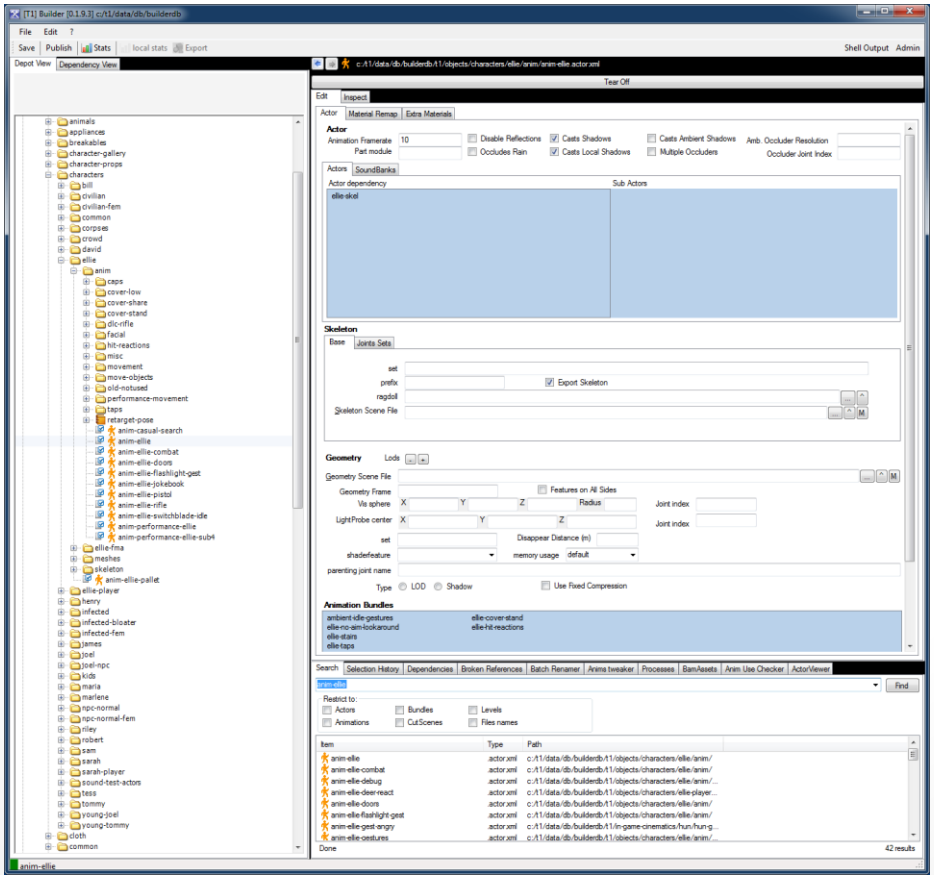
Some successful designs

- 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

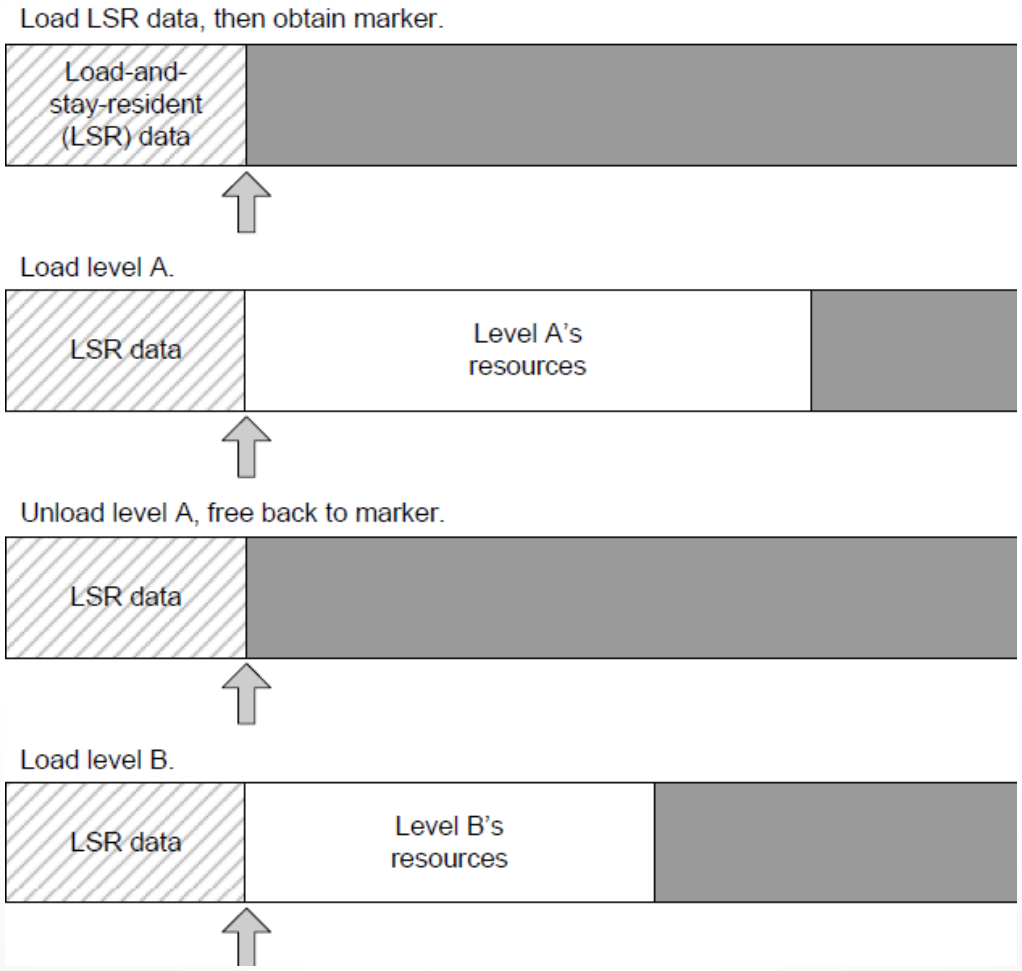
Resource lifetime

- 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

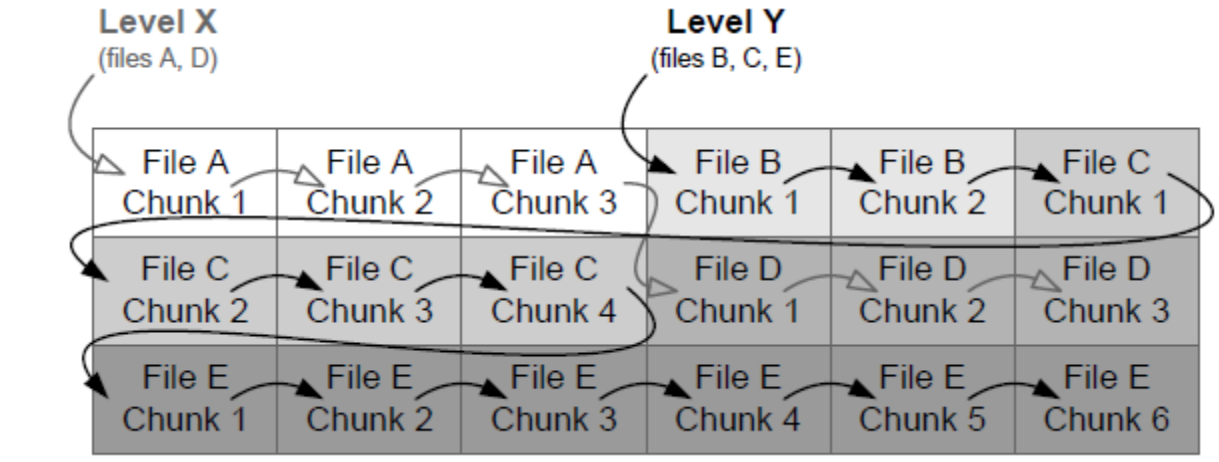
- Very closely tied to general memory management
- Heap allocation – allow the OS to handle it
  - Like malloc() or new
  - Works fine on a PC, not so much on a memory limited console
- Stack allocation
  - Can be used if
    - The game is linear and level centric
    - Each level fits in memory

Stack allocation



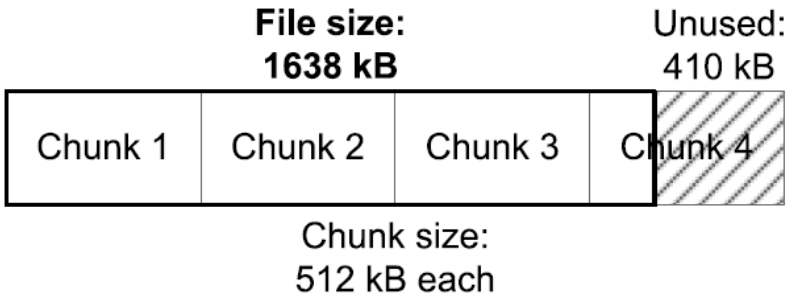
Pool allocation

- 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



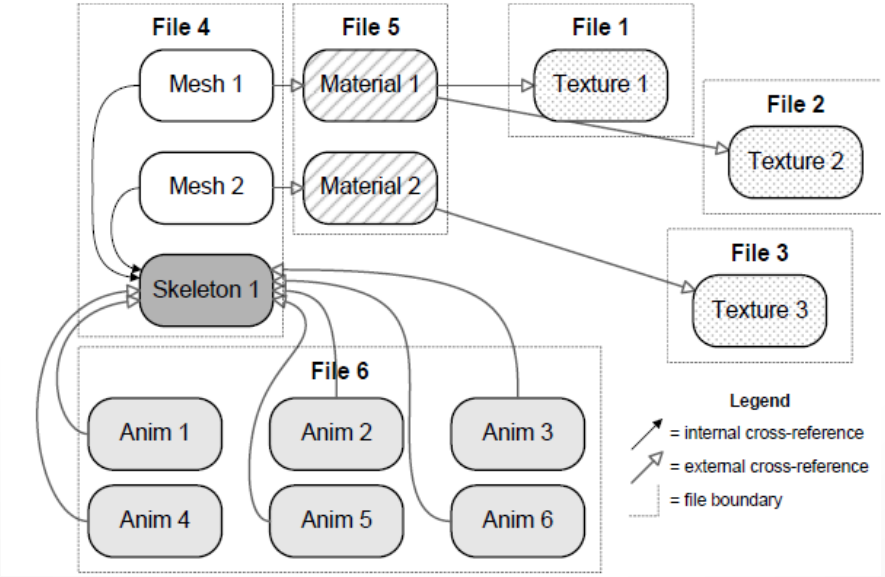
- Choose the chunk size carefully
  - Consider using the OS I/O buffer size as a guide

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

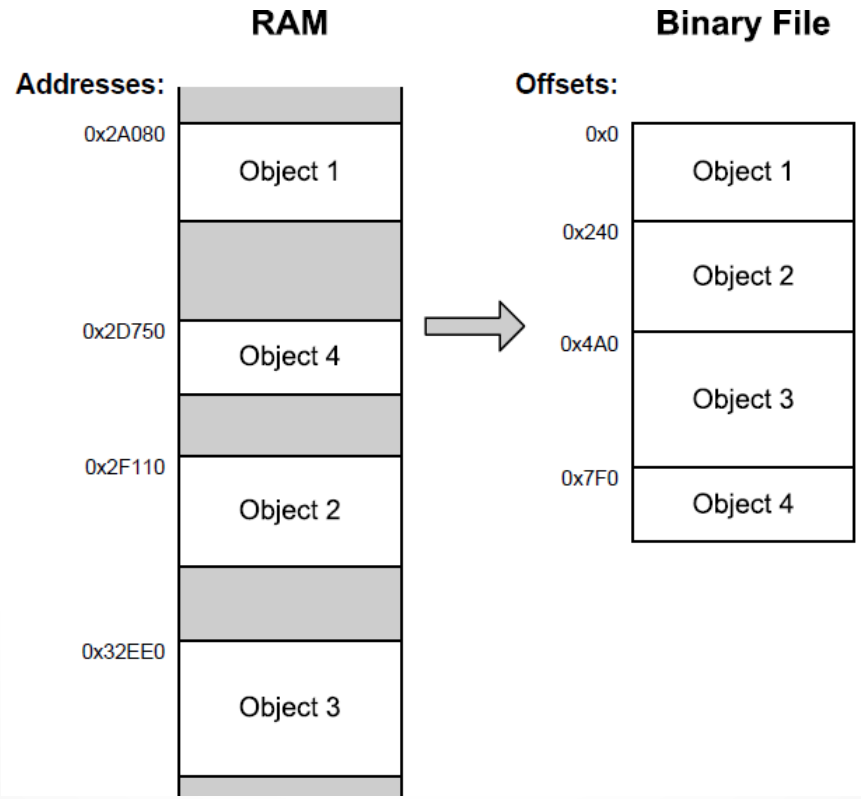
- A cluster of interdependent resources is referred to as a composite resource
- For example, a *model* consists of
  - One or more triangle meshes
  - Optional skeleton
  - Optional animations
  - Each mesh is mapped with a material
  - Each material refers to one or more textures

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

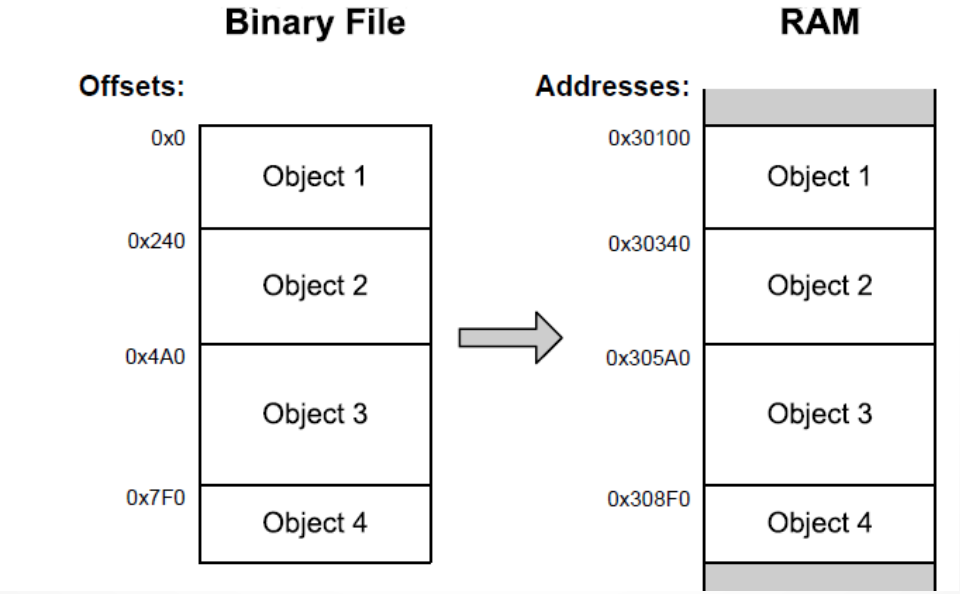
Pointer fix-up tables

- Another approach is to convert pointers to file offsets



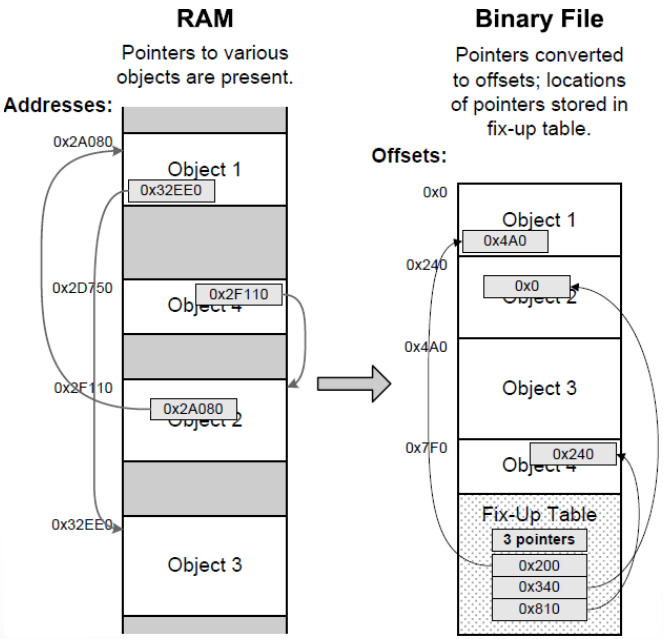
Pointer fix-up tables

- 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

Unity Dynamic asset management systems

- Help you improve the performance of your game by allowing you to determine when, and from where, assets are loaded in and out of memory during runtime.
- Throughout the Unity engine's history there have been several ways to manage assets in memory.
- The Addressables system was designed to replace or improve upon legacy systems that have either been deprecated or are inadequate for a project that has progressed beyond a basic prototype.

Direct References

- Direct references
  - When you drag an asset into a scene or onto a component through the Inspector window of the Editor, you're making a direct reference to that asset.
  - When you build your application, these assets are saved in a separate file associated with your scene.
  - When your application is running on a user's device and the scene is loaded, the Unity Player loads the entire asset file into memory before the scene, ensuring that the scene has access to everything it directly references.
  - If you use only direct references in your game, then your approach to asset management is not dynamic.
  - Loading and unloading the scene is the only control offered.
  - You run the risk of building a game with a large build size that performs slowly, especially on devices with less memory such as mobile.

Resources folder

- The Resources folder and the Resources API provided a simple way to manage assets in memory.
- In the project folder structure of older Unity projects, you might see one or more Resources folders containing assets.
- During the Player build process, the Editor finds the assets in any folders named Resources and bundles them into a serialized file, with metadata and indexing information, that's packaged with your application.
- The Resources API allows you to write scripts to load and unload the assets in folders named Resources.
- The Resources system has several disadvantages over newer systems. It doesn't allow for a fine-grained approach to memory management, it slows down startup time, and it can bloat the size of your built application. Moreover, it doesn't support delivery of content from a CDN.

Unity Resources.Load

- Loads the asset of the requested type stored at path in a Resources folder.
  - Create a folder called "Resources"
  - Create a subfolder called "Enemies"
  - Create 4 prefabs/gameobjects under Enemies
  - Creata a script called EnemyType
  - Attach EnemyType script to all of the prefabs
  - Run the application and see allEnemies

Examples

```
//load a single resource
singleEnemy = Resources.Load<EnemyType>("Enemies/Enemy1");

//load all resources
allEnemies = Resources.LoadAll<EnemyType>("Enemies");



//step1 Loading assets from the Resources folder using the Resources.Load(path,
systemTypeInstance)
GameObject instance = Instantiate(Resources.Load("Enemies/Enemy1", typeof(GameObject))) as
GameObject;


//step2: Load a Texture
GameObject go = GameObject.CreatePrimitive(PrimitiveType.Plane);
var rend = go.GetComponent<Renderer>();
rend.material.mainTexture = Resources.Load("Textures/Ground/Grass") as Texture;

//step3: make sure that you attach an audio source to the camera
AudioSource audio = GetComponent<AudioSource>();
var audioClip = Resources.Load<AudioClip>("Audio/Music/cron_audio_8-bit_modern02");
audio.clip = audioClip;
audio.Play();
```

AssetBundle system

- The AssetBundle system organizes assets into containers called AssetBundles.
- Like the Resources folder, the AssetBundle system creates sets of assets into separate files.
- Unlike Resources folders, AssetBundles can be stored locally with the Player or remotely in the cloud.
- The AssetBundles system, through its API, minimizes the impact on network and system resources. It does this by allowing you to download the bundles on an as-needed basis, so that you can add DLC(DownLoadable Content) and post-release content updates

Example

- For example, you can deliver new content for your players to view, earn, or purchase, without requiring them to download a new version of your game.
- Once bundles are downloaded, AssetBundles API provides a way to load and unload assets from bundles.

Disadvantage

- The AssetBundles system is an API that can only be used in scripts.
- There is a limited user interface for defining AssetBundles in the Unity Editor, but building AssetBundles requires scripting.
- The AssetBundles API by itself does not keep track of asset dependencies between AssetBundles.
  - For example, if you want to load a prefab from AssetBundle A, you will need to locate any of its dependencies such as meshes, materials, and textures that may be located in other AssetBundles, and ensure those dependent AssetBundles are loaded at runtime before you attempt to load the prefab.

More Disadvantage

- Memory allocation and deallocation is direct and manual, so it's possible to unload an asset from an AssetBundle while other code still depends on that asset, potentially resulting in missing content issues or memory leakage. This can become problematic in code that creates race conditions, and requires fortifying your code against these problems.
- AssetBundles runtime API is not aware whether you put your built bundles locally or remotely. This requires you to keep track of the location of that AssetBundle, whether it's on a web server or on disk.

Addressables System

- The Addressables system is Unity's dynamic asset management system.
- Addressables builds on Unity's AssetBundles technology, and also provides tools inside the Unity Editor to help you prepare your assets for on-demand loading, whether your content is on-device or in the cloud.
- With the Addressables system, you can organize your assets in the Unity Editor, and let the system handle dependencies, asset locations, and memory allocation and deallocation.

Why should you use Addressables?

- Once an asset is addressable, it has an address that you can reference in your scripts, rather than by its file name or bundle location. Although that might sound simple, addresses enable the Addressables system to automate a lot of the details.
- The Addressables system gives you the flexibility to specify where an asset is hosted. You can install assets so that they exist alongside your application on disk or download them on demand from a remote web server.
- You can then change where a specific asset exists, such as from local to remote, or from a monolithic bundle to more granular bundles, all without needing to rewrite code.

Dependency management

- The Addressables system automatically loads all dependencies of any assets you load so that all meshes, shaders, animations, and other dependent assets load before the asset requested loads.
- For example, if you request to load a character, the system will automatically load the character's meshes, materials, and animations, and the materials' textures and shaders as well.

Memory management

- The Addressables system automates much of the drudge work of memory management.
- As you load assets into and out of your game in scripts, the system keeps track of memory allocation.
- You can also use the system's robust profiler to further improve the memory efficiency of your game.

Efficient content packing

- Because the Addressables system maps complex dependency chains, it helps you pack assets efficiently, even if you move or rename assets.
- You can choose how granular you want your assets to be at a location, favoring packing them separately or grouped together, and can easily prepare assets for both local and remote deployment to support on-demand or downloadable content (DLC), which can reduce application sizes.

Cloud build and content delivery

- The Addressables system has been integrated into Unity Gaming Services (UGS), specifically Cloud Content Delivery and Cloud Build, giving you end-to-end services for live game updates and building your applications in the cloud.

Scriptable Build Pipeline

- The Addressables system uses the Scriptable Build Pipeline (SBP), which is more robust than the legacy AssetBundle build pipeline.
- You can use the pre-defined build flows or, if you want to use a more advanced method, create your own using the divided-up APIs.

Localization

- The system is also integrated with Unity's Localization package so that you can use various languages in your projects.

LAB

- In the lab, you will install the Addressables package and learn how to access its user interface.
- By the end of the lab, you'll be able to:
  - Locate the Addressables system's user interface in the Unity Editor
  - Identify assets that can be made addressable.
  - Explain different methods of marking assets as addressable
  - Edit an asset's address
- Open the LoadyDungeon project (Github) in the Unity Editor (Make sure that you have the right editor version 2022.2.15 or later works).
- Install the Addressables package, version 1.21.8 or higher, to your project.

Loady Dungeons

- Loady Dungeons is a mobile game where users control a dinosaur character. The objective of each level of the game is for the user to get a key from a chest and make their way to unlock a door. Unlocking the door will take them to the next level.
- The Loady Dungeons project that you've installed uses direct references and the Resources folder (as described before) throughout the project.
- Your task is to apply the Addressables system so that you can reduce the size of the downloadable package, easily update assets throughout the game's lifecycle, and allow users to add or update assets during gameplay.

What is an addressable?

- When you enable an asset as addressable, Unity generates an address, which is a string identifier that the Addressables system associates with the asset's location, regardless of whether the asset resides locally with your built game or on a content delivery network.
- In your scripts, you can use the asset's address instead of keeping track of its location, so that you don't have to write code that specifies where — only what.
- This way, if an asset location changes, you won't have to rewrite your code.

Make all of the hats addressable

- In the Project window, navigate to Assets > Resources.
- Remember, the Resources folder has a unique purpose in Unity, which is part of the old Resources system for dynamic asset management. When you build this project as it is, any assets here will be placed in a separate indexed file. You're about to reconfigure these assets for the Addressables system.
- Select the Hat00 prefab and view it in the Inspector window.
- In the Inspector window, enable the Addressable property.
- To prevent this asset from being processed and duplicated by the legacy Resources system, the Addressables system has moved this prefab out of the Resources folder for you!
- To make all of the other hats addressable, select them all in the Resources folder, and enable the Addressable property in the Inspector.

Open the Addressables Groups window

- You can also enable assets as addressable by dragging the prefabs onto the Addressables Groups window.
- To access this window, navigate from the main menu of the Unity Editor to Window > Asset Management > Addressables > Groups.

Make assets addressable in the Addressable Groups window

- In the Project window, navigate to Assets > Resources.
- Drag the LoadyDungeonsLogo into the Default Local Group of the Addressable Groups window.
- In the Move From Resources dialog, select Yes.
- To test this change, open the MainMenu scene again and view it in Play mode. The Loady Dungeons logo is no longer present, since it is no longer found and loaded from the Resources folder.

Edit an asset's address

- An asset's address is a string ID that identifies an addressable asset. You can use an address as a key to load an asset via script.
- 1. In the Project window, navigate to Assets > Resources_moved and select the Hat00 prefab.
- 2. In the Inspector window, change the address of Hat00 to "Hat_BunnyEars".
- 3. In the Project window, navigate to Assets > Scenes and select the LoadingScene scene, but don't open it.
- 4. In the Inspector, enable the Addressable property for LoadingScene.
- 5. Change the address Assets/Scenes/LoadingScene.unity to LoadingScene.

Edit the address through the Addressables Groups window

- Another way you can edit an asset's address is through the Addressables Groups window. To do this, right-click the prefab you want to change the address of and select Change Address.
- Once Loady Dungeons is deployed to an app marketplace, you should avoid changing the address of your assets. The assets may change often throughout the lifespan of the project, but any address that represents an asset should avoid changing. This allows you to avoid changing the game scripts if only asset content changes, which would cause us to redeploy the app onto the marketplace.

Cleanup

- While not necessary, it's important to clean up empty folders and other information to give you a fresh start in the tutorials to follow. To clean up your project, follow these instructions:
- 1. Rename the Resources_moved folder to the name of your choice. Resources_moved is only meant to be a temporary holding place.
- 2. In the Project window, navigate to Assets.
- 3. Select the Resources folder and delete it.

Next step

- Now that you've converted content to use the Addressables system, in the next step you'll learn how to load that content through the runtime Addressables API.
- By the end of this lab, you'll be able to do the following:
  - Define an asynchronous operation
  - Identify different ways of referencing an addressable asset in scripts
  - Identify different methods of loading and releasing addressable assets asynchronously

Asynchronous operations

- In the Addressables API, many tasks load assets and data and then return a result. When those assets or data are on a remote server, these tasks can take extra time.
- To avoid slowing down your game's performance and delaying other tasks, the Addressables system uses asynchronous operations, which means that tasks can run concurrently.

Asynchronous operations in Unity

- Coroutines, which allow you to perform a function over time instead of instantly.
- Delegates, which are containers for functions that can be passed around or used like variables.
- Events, which are specialized delegates that can alert a class that something has happened.
- The Addressables API uses a struct, named AsyncOperationHandle, to help you keep track of asynchronous operations in your code.
  - A struct is a value type in C# that can be used to group related data together in a single object. They are similar to classes in that they can contain fields and methods, but unlike classes, they are passed by value instead of by reference.

Serialization

- Serialization is the process of transforming and storing data between sessions of an application, and deserialization is the process of taking that stored data so that it can be reconstructed when an application runs again.
- Unity serializes fields during build time so that it can deserialize the data stored in them while the application is running.
- As you use the Addressables system and API, you'll use serialization to optimize the ways your project consumes assets.

prefab

- There are several different ways to load prefabs using the Addressables API.
- Let’s start with loading an asset by using its address, which you'll do as you convert the PlayerConfigurator script to use the Addressables API.
- To rewrite PlayerConfigurator.cs to use the Addressables API instead of the Resources API, follow these instructions:

UnityEngine.Addressable Assets

- 1. In the Project window, in Assets > Scripts, open the PlayerConfigurator.cs
- 2. Near the top of the script, add the Addressables namespaces:
  - using UnityEngine.AddressableAssets;
  - using UnityEngine.ResourceManagement.AsyncOperations;
- You will use these namespaces to load an asset by its address and access the AsyncOperationHandle that is returned.

m_Address

- 3. In the PlayerConfigurator class, create a new string variable representing the address. Precede it with the [SerializeField] attribute so that you can serialize it and change its value in the Inspector.
  - [SerializeField]
  - private string m_Address;
- When this serializable field appears in the Inspector, Unity will remove the prefix and underscore, and label it Address.

AsyncOperationHandle

- 4. Locate the following line in the existing script. This line uses the Resources API.
- private ResourceRequest m_HatLoadingRequest;
- Replace the line above with the following line, which uses the Addressables API.
- private AsyncOperationHandle<GameObject> m_HatLoadOpHandle;
- The line above defines the variable m_HatLoadOpHandle as the AsyncOperationHandle (or handle for short) of the GameObject.

SetHat()

- 5. In the SetHat() method, remove the body of the method and replace it with this code:
  - m_HatLoadOpHandle = Addressables.LoadAssetAsync<GameObject>(m_Address);
  - This line will load the prefab asset asynchronously, by its address, m_Address. The handle m_HatLoadOpHandle will have a Boolean indicating whether the asset was loaded successfully. If the load was successful, the handle also has the result of the load request, which is the prefab itself.

Register an event handler

- Since the asset will be loaded asynchronously, the next lines of code will run without waiting for the LoadAssetAsync() method to return a value. Therefore, to find out whether the method was successful, you can register an event to detect that the call is completed.
- In the next line, you'll register an event handler (which you'll define afterwards) to notify you when the hat is loaded, using the AsyncOperationHandle's Completed event.
  - m_HatLoadOpHandle.Completed += OnHatLoadComplete;

OnHatLoadComplete

- 7. Locate the method OnHatLoaded and replace it with method OnHatLoadComplete to print the status of the handle. This method is called by Addressables when it has completed the request to load the prefab asset.
- private void OnHatLoadComplete(AsyncOperationHandle<GameObject> asyncOperationHandle)
- { Debug.Log($"AsyncOperationHandle Status: {asyncOperationHandle.Status}");
- }

OnDisable

- In the OnDisable() method, remove the body of the code and replace it with the following, to disable the event if this game object is disabled. This will prevent OnHatLoadComplete being called by the Addressables system if PlayerConfigurator should not be in operation: :

- m_HatLoadOpHandle.Completed -= OnHatLoadComplete;

Next Steps

- 9. Save your script and return to the Unity Editor.
- 10. To test this script, navigate to Assets > Scenes and open the Level_00 scene.
- 11. In the Project window, navigate to Assets > Prefabs and select the Player prefab.
- 12. In the Inspector window, locate the PlayerConfigurator component and set the new Address variable to the address of a hat, such as Hat_BunnyEars.
- 13. Enter Play mode and watch the Console window, where you'll see “AsyncOperationHandle Status: Succeeded”. At this point, the hat prefab asset will load successfully, but you won't see it in the scene because you haven't instantiated the prefab yet. You’ll do that next.

OnHatLoadComplete

- 14. Replace the code in the OnHatLoadComplete() method with the following code:

```
private void OnHatLoadComplete(AsyncOperationHandle<GameObject> asyncOperationHandle)
{
    if (asyncOperationHandle.Status == AsyncOperationStatus.Succeeded)
    {
        Instantiate(asyncOperationHandle.Result, m_HatAnchor);
    }
}
```

This code will check to see if the status of the load operation was a success. If so, there will be a prefab asset that was retrieved by the load operation in Result. It will be passed along to be instantiated.

Play

- 15. Enter Play mode and note that the player is moving with the hat you selected.
- 16. Experiment by changing the Address to a different hat prefab and see the result.
- Note: In this example, the input Address is a string that isn't validated as an address until you make the load request. You could enter anything in this property and cause an error. Make sure that the text matches an address in the Groups window.

Load an addressable prefab by an AssetReference

- An AssetReference is a type that references an addressable asset. It is intended to be used as a serializable field in Unity classes like MonoBehaviour or ScriptableObject.
- When you add an AssetReference to one of these classes, you can assign an address to it in the Inspector with an object picker. The selection is limited to addressable assets.
- On the surface, you'll add AssetReferences to your scripts just like you would add direct references, through public fields and private serializable fields.
- AssetReferences do not store a direct reference to the asset. The AssetReference stores the global unique identifier (GUID) of the asset, which is used by the Addressables system to store the object for retrieval at runtime.

Advantage

- The main benefit of using AssetReference over a string address is that it will restrict the selection of addresses from the Inspector, which avoids problems like typos in string addresses.

Rewrite PlayerConfigurator

- To rewrite PlayerConfigurator to use an AssetReference instead of an address, follow these instructions:
- 1. In the Project window, navigate to Assets > Scripts and open the PlayerConfigurator.cs script.
- 2. Replace the definition of the m_Address field with:

```
[SerializeField]
private AssetReference m_HatAssetReference;
```

SetHat

- In the SetHat() method, change the line that uses the Addressables.LoadAssetAsync() method to use the AssetReference instead:

```
if (!m_HatAssetReference.RuntimeKeyIsValid())
{
  return;
}

m_HatLoadOpHandle = m_HatAssetReference.LoadAssetAsync<GameObject>();
```

- The call to RuntimeKeyIsValid checks to find out if the value chosen in the object picker is a valid address. In this context, it will forbid LoadAssetAsync from being called if the AssetReference is set to None.

Play

- 4. Save the script and make sure that Level_00 is the active scene in the Editor.
- 5. In the Project window, navigate to Assets > Prefabs and select the Player prefab.
- 6. In the Inspector window, locate the PlayerConfigurator component. Note that the Address field is gone and replaced with Hat Asset Reference. Open the object picker for the new field and select an addressable hat prefab, such as Hat01.
- 7. Enter Play mode and note that the player is moving with the hat you selected.
- 8. Experiment by changing the AssetReference to a different hat address and observe the result.

Load an addressable prefab by an AssetReferenceGameObject

- The object picker for AssetReference restricts the selection to assets that are addressable, but you may have noticed that it also listed the texture LoadyDungeonsLogo and the scene LoadingScene.
- These types can be selected with the object picker, but if PlayerConfigurator attempts to load them as GameObjects it will have a runtime error in the Addressables code.
- You may want to refine this further and make only prefabs selectable in the object picker.

AssetReferenceGameObject

- To rewrite PlayerConfigurator to use AssetReferenceGameObject instead of AssetReference, follow these instructions:
- 1. In the Project window, navigate to Assets > Scripts and select PlayerConfigurator.cs script.
- 2. Replace the definition of the m_HatAssetReference field with:
  - [SerializeField]
  - private AssetReferenceGameObject m_HatAssetReference;

Play

- 3. Save the script and make sure that Level_00 is the active scene in the Editor.
- 4. In the Project window, navigate to Assets > Prefabs and select the Player prefab.
- 5. In the Inspector window, locate the PlayerConfigurator component, open the object picker for the new Asset Reference property, and select an addressable Hat asset, such as Hat02. Note that the object picker no longer includes the logo sprite since it is not a GameObject.
- 6. Enter Play mode and note the selected hat. Try on other hats as desired.

LoadSprites

- Sprites and sprite atlases are assets that can have subobject assets. There are several different ways to load sprites using the Addressables API. Let's start with loading an addressable asset with subobjects by an address.
- 1. In the Project window, navigate to Assets > Scripts and select the GameManager.cs.
- 2. At the top, add the Addressables namespaces:

```
using UnityEngine.AddressableAssets;
using UnityEngine.ResourceManagement.AsyncOperations;
```

GameManager

- 3. In the GameManager class, create a new string and an AsyncOperationHandle field:

```
[SerializeField]
private string m_LogoAddress;

private AsyncOperationHandle<Sprite> m_LogoLoadOpHandle;
```

- Note that you are loading by a sprite instead of GameObject in this case, so the type in the generic parameter list reflects that.

OnEnable

- 4. In the OnEnable() method, remove the line with the call to Resources.LoadAsync() method and everything below it, then add this line:

```
m_LogoLoadOpHandle = Addressables.LoadAssetAsync<Sprite>(m_LogoAddress);
```

5. In the same method, after capturing the handle returned by the LoadAssetAsync() call with the m_LogoLoadOpHandle field, register an event handler that will notify you when the logo is loaded. Use the AsyncOperationHandle's Completed event, like you have before.

```
m_LogoLoadOpHandle.Completed += OnLogoLoadComplete;
```

OnLogoLoadComplete

- 6. Create the callback method OnLogoLoadComplete().

```
private void OnLogoLoadComplete(AsyncOperationHandle<Sprite> asyncOperationHandle)
{
    if (asyncOperationHandle.Status == AsyncOperationStatus.Succeeded)
    {
        m_gameLogoImage.sprite = asyncOperationHandle.Result;
    }
}
```

- When the load is a success, the result will be passed along to the UI element.

Play

- 7. In the Project window, navigate to Assets > Scenes and select the MainMenu as the active scene in the Editor.
- 8. In the Hierarchy window, select the GameManager GameObject.
- 9. In the Inspector window, locate the GameManager component and set the new Address field to "LoadyDungeonsLogo", the address of the logo.
- Note that subobjects usually have a subscript notation such as MainObject[SubObject], which indicates that Unity will retrieve a specific subobject, for example LoadyDungeonsLogo[LoadyDungeonsLogo]) is also valid.
- 10. Enter Play mode and you'll see that your script is loading the logo.
- 11. Exit Play mode and save the scene.

Load an addressable asset with subobjects by an AssetReference

- When AssetReferences are assigned to an asset with subobject assets, an additional object picker appears that allows you to specify the subobject to reference.
- 1. In the Project window, navigate to Assets > Scripts and select the GameManager.cs.
- 2. Replace the definition of the m_LogoAddress field with:

```
[SerializeField]
private AssetReference m_LogoAssetReference;
```

Play

- 3. In the OnEnable method, change the line that uses Addressables.LoadAssetAsync method to use the AssetReference instead:
-
```
if (!m_LogoAssetReference.RuntimeKeyIsValid())
{
  return;
}

m_LogoLoadOpHandle = Addressables.LoadAssetAsync<Sprite>(m_LogoAssetReference);
```
- 4. Save the script and make sure that MainMenu is the active scene in the Editor.
- 5. In the Hierarchy window, select the GameManager GameObject. In the Inspector window, locate the GameManager component. Open the object picker for the new Asset Reference property and select the texture of LoadyDungeonsLogo and subobject (Subasset) field.

- 6. Enter Play mode and note that the logo is loaded.

Load an addressable sprite by an AssetReferenceSprite

- To restrict the object picker to only make sprites selectable, you can use AssetReferenceSprite instead of AssetReference.
- You have enough experience now to make this change yourself. Give it a try and test your code. When you use the object picker in the GameManager component, your only choice will be the logo sprite, because it is the only sprite with an address.

Load an addressable scene by its address

- If a scene has been enabled as addressable, you can use addressable assets in the scene just as you would any other assets.
- To update the scripts to load addressable scenes, follow these instructions:
- 1. In the Project window, navigate to Assets > Scripts and select the GameManager.cs script.
- 2. At the top of your script, add the ResourceProviders namespace:

```
using UnityEngine.ResourceManagement.ResourceProviders;
```

Addressables.LoadSceneAsync

- In GameManager class, create a new AsyncOperationHandle field:

```
private static AsyncOperationHandle<SceneInstance> m_SceneLoadOpHandle;
```

In the LoadNextLevel method, replace SceneManager.LoadSceneAsync with Addressables.LoadSceneAsync. Save the script after these changes.

```
public static void LoadNextLevel() { m_SceneLoadOpHandle = Addressables.LoadSceneAsync("LoadingScene", activateOnLoad: true); }
```

If you check the Addressables Groups window, the LoadingScene string you've used for the LoadSceneAsync() method should match the address you have for the LoadingScene addressable asset.

Loading.cs

- 5. Save the script and in the Project window, navigate to Assets > Scripts and select the Loading.cs script.
- 6. At the top of your script, locate this line and remove it:

```
using UnityEngine.SceneManagement;
```

- 7. Add these Addressables namespaces:

```
using UnityEngine.AddressableAssets;
using UnityEngine.ResourceManagement.AsyncOperations;
using UnityEngine.ResourceManagement.ResourceProviders;
```

loadNextLevel

- 8. In Loading class, replace the definition of the m_SceneOperation field with:

  ```
  private static AsyncOperationHandle<SceneInstance> m_SceneLoadOpHandle;
  ```

9. In the loadNextLevel method, locate these lines:

   ```
   m_SceneLoadOpHandle = SceneManager.LoadSceneAsync(level);
   m_SceneOperation.allowSceneActivation = false;
   ```

   Replace the lines with SceneManager.LoadSceneAsync and m_SceneOperation.allowSceneActivation with:

   ```
   m_SceneLoadOpHandle = Addressables.LoadSceneAsync(level, activateOnLoad: true);
   ```

Play

- 10. In the loadNextLevel method, locate this line:
  while (!m_SceneOperation.isDone)
  Replace the line with m_SceneOperation.isDone with:
  while (!m_SceneLoadOpHandle.IsDone)
- 11. In the loadNextLevel method, replace the all references to m_SceneOperation.progress with:
  m_SceneLoadOpHandle.PercentComplete
- 12. Remove the GoToNextLevel method and save the script.
- 13. Open the MainMenu scene and enter Play mode.
- 14. Select the Start button to load the LoadingScene.
- 15. Note that the LoadingScene is visible and that Level_00 loads in.

Reference counting

- One benefit of the Addressables system is that it helps you manage the loading and unloading of the assets in and out of memory.
- This is done internally through a reference counting system, which manages the way the resources are shared, but also decides when the resources are actually loaded and unloaded from memory.
- In practice, when you write scripts that request assets from the Addressables system, you simply make the load request, keep the AsyncOperationHandle that was returned, and pass the handle back to the Addressables system when you no longer need the result of the operation.
- The Addressables system handles the actual loading and unloading.
- There are benefits to this approach. If there are two different requests for the same asset from the Addressables API, the reference counting system makes sure that there isn't double loading of assets.

Reference Counting Lab

- For example, let's randomly load a hat at the start of the level and give the player the option to select a new random hat.

1. In the Groups window, change the address of Hat_BunnyEars to "Hat00". This will make the naming consistent across all hats so that we can programmatically select them.
2. In the Project window, in Assets > Scripts, open the PlayerConfigurator.cs script.
3. Locate this line:

```
[SerializeField]
private AssetReferenceGameObject m_HatAssetReference;
```

Replace it with this:

```
private GameObject m_HatInstance;
```

LoadInRandomHat

- 4. In the Start method, remove the body and add this line:

```
LoadInRandomHat();
```

- 5. Remove the SetHat method and add the following method to the existing script:

```
private void LoadInRandomHat()
{
    int randomIndex = Random.Range(0, 6);
    string hatAddress = string.Format("Hat{0:00}", randomIndex);

    m_HatLoadOpHandle =
Addressables.LoadAssetAsync<GameObject>(hatAddress);
    m_HatLoadOpHandle.Completed += OnHatLoadComplete;
}
```

This method will pick a random number from 0 to 5, then format a string with the hat's address so that it is between Hat00 and Hat05, and then attempt to instantiate that hat using the randomized address.

OnHatLoadComplete

- 6. In the OnHatLoadComplete method locate this line:

  Instantiate(asyncOperationHandle.Result, m_HatAnchor);

  Replace it with this:

  m_HatInstance = Instantiate(asyncOperationHandle.Result, m_HatAnchor);

Update

7. Add the following method to the existing script:

```
private void Update()
{
    if (Input.GetMouseButtonUp(1))
    {
        Destroy(m_HatInstance);
        Addressables.ReleaseInstance(m_HatLoadOpHandle);

        LoadInRandomHat();
    }
}
```

Play

- Instead of allowing all of the hat instances to accumulate in memory, the Destroy() method removes the hat from memory before we load the next.
- We also no longer the prefab in memory, so ReleaseInstance() will notify Addressables that the hat prefab is no longer needed here before attempting to load a new random hat asset.
- Since this was the only reference to the hat prefab across the game scripts, Addressables chooses to unload it.
- 8. Open the MainMenu scene and enter Play mode.
- 9. Select the Start button to load the LoadingScene. Then select the Play button in the game.

  When you right click anywhere in the Game view, the hat will change to a different randomly-loaded hat. Behind the scenes, you're managing memory effectively by releasing each instance of the hats that are replaced.