CS354P

DR SARAH ABRAHAM

WORKING WITH DATA

THINKING ABOUT DATA...

- What sorts of data do we need?
- When do we need that data?
- Where should we store all this data?

DATA WE NEED

- Design information
  - Designer-created information including stats, tech-trees, combo flow, weapon customization, etc
  - Usually made in Excel or other external tools
- Player information
  - Information related to a player that is necessary between levels
  - Includes things like inventory, health, current quests, completed quests, etc
- Game information
  - Information related to the game state that is necessary between levels
  - Includes things like time of day and weather, round information, enemy information, etc

WHEN THAT DATA IS NEEDED

- On game loading
  - Data may be needed at the start of the game
  - User information, save information, asset and system information, etc
- Within the level
  - Data may be loaded at the start of the level or during the level
  - Asset and system information, player information, save information, etc
- Persistent between levels
  - Data may be needed across multiple levels
  - Game state information, player information, etc

ACCESSING DATA

- Data can be accessed from a remote server or from a local file system*
  - User information
  - Asset information
  - Systems information
  - World state information
  - etc...
- When should we store this information remotely versus locally?

*Note: Possible to “hard code” a lot of information, but we should avoid this in practice

WHAT ABOUT STORING DATA DURING A SESSION?

- I/O is expensive
  - Accessing data from both a file system and a network connection are expensive
    - Minimize this cost as much as possible by storing data per session
- Must consider:
  - What the data is
  - When we need it
  - How long we need it

EXAMPLE: LEVEL STREAMING

- Feature for loading and unloading parts of a map to minimize memory footprint and reduce rendering
  - Essential for consoles and handhelds
- Must be done asynchronously to avoid lag/stuttering
- Use of **sublevels** and streaming volumes to access parts of the **persistent level**
- Closely tied to texture streaming
  - Determines resolution of all necessary textures in the visible scene
  - Determines what textures to load and unload as well as priority
  - Manages its own streaming pool to determine available texture budget

THINKING ABOUT TRANSIENT VERSUS PERSISTENT DATA

- Level data is highly transient in many games
  - Data constantly being loaded and unloaded based on player position
  - But! May be necessary to save “changes” to the world should the player return to allow for persistence…
- Data comes in many forms
  - Think carefully about how to store it and whether it should be persistent or transient…

GAME MODE AND GAME STATE

- `GameMode` and `GameModeBase` are actors that define and controls the game's rules
  - Exist only on the server
  - Determines win conditions, points, characters allowed, number of players allowed, available items, etc...
- `GameState` and `GameStateBase` are actors that track the current state of the game
  - Replicated to clients
  - Stores information on team points, number of players the game, currently available items, etc...

PLAYER STATE

- `PlayerState` holds information about an individual player
  - Replicated to all clients and stored in `PlayerArray` in `GameState`
- Stores information tied to an individual player such as individual score, user name, ping, etc...

PLAYER CONTROLLER

- `PlayerController` is the interface between the player and the game
- Not just a source of inputs into the game!
  - First level of interface that the client actually owns
  - Connects the player to the server
  - Tracks pawn current possessed by the player
- Note: Pawns can be replicated to other clients -- player controller exists only on the server and owning client

USING THESE ACTORS

- `GameMode` is the authority that should inform and update `GameState` and `PlayerState`
  - Changes to these states must be done from the server
  - Replication is only there so clients can see these changes in state reflected in their local view
- `PlayerController` is where you access the player's current HUD
- All of these actors are transient (e.g. only exist in the current level)
  - Cannot store data that required between levels (but will exist across sublevels in a persistent level)
  - Except...

SEAMLESS TRAVEL

- Possible to seamlessly travel between levels under certain circumstances:
  - **Already** connected to the server
  - Destination map has been **previously** loaded
- Will carry over `GameMode` and `Controllers` to new level
- `ServerTravel()` moves server and all clients to the new level
- `ClientTravel()` can either move client to new server or to new map, if called from server

GAME INSTANCE

- `UGameInstance` is a high-level **manager** for a running game
  - Spawned at game creation
  - Destroyed when game instance is shut down
- Can store data that needs to persist if seamless travel isn't an option
  - Or data that doesn't make sense to store on PlayerControllers
- Good, built-in option for data storage but is very high-level
  - Manages *entire game* rather than specific subsystems

GAME MANAGERS

- Systems that control and manage smaller tasks within the larger game system
- Can be used for a specific domain:
  - Audio Manager
  - Particle Manager
  - File System Manager
- Can be used for a specific subsystem:
  - Board Manager
  - Quest Manager
  - Minigame Manager

STATIC CLASSES

- Ensures only one copy is stored in memory
  - Used extensively in Unreal for library calls (`UGameplayStatics`, `Math`, etc)
- Possible to create your own “static class”
  - Make every function and member static (C++ doesn’t actually support static classes so we just pretend)
  - Inherit from `UObject`

STATIC CLASS CAVEATS

- Static members are initialized before `main()` is called
- No guarantees on order of initialization so static members **cannot** depend on each other
  - Note: it is possible to use lazy initialization in general C++ to solve this issue
- Due to UE5's class structure/build process, static members should be `const` and initialized at **compile time** rather than runtime
  - For dynamic objects and data, try to pass in values as arguments as much as possible (i.e. dependency injection)

SINGLETONS

- Singleton pattern restricts the instantiation of a class to a single instance
- Allows for lazy instantiation
  - Never created if never used
- Available anywhere
- Can be subclassed

Canonical singleton implementation

```
class Singleton {

  static Singleton * instance;

  Singleton() { }

public:

  static Singleton * instance() {

    if (!instance)

      instance = new Singleton();

    return instance;

  }

}
```

SINGLETON PROBLEMS

- Highly controversial design pattern!
  - Sometimes called an **anti-pattern** because it breaks more than it fixes
- In practice it's just a fancy global variable...
  - Hard to reason about and debug in large-scale projects
  - Allows for coupling of unrelated behaviors
  - Performs poorly in concurrent systems (too much shared memory)

HOW TO SOLVE?

- Use dependency injection as much as possible
  - Pass data in as arguments when processing
- Use static classes over singletons
  - Still have issues but easier to reason about
- If a static class doesn't work, consider using a static flag with a non-static class to ensure only one is created
- Use Service Locators (discussed later)

WHAT ABOUT UE5?

- Unreal highly discourages the use of singletons
  - If it seems like the best solution, rethink your approach
- `GameInstance` is not implemented as a singleton but it functions as a singleton
  - Functions as global state
  - Accessible via `UGameplayStatics` library
- `GameInstance` may be too broad and high level to work well for managing sub-systems but it is generally the right place to *store* sub-systems
  - Only one exists
  - Exists for the entirety of the game

MANAGING WITH GAME INSTANCE

- Include managers as objects within `GameInstance`
  - Use `NewObject<MyManager>()` to construct a new manager
- Same principle as a singleton (only allow one object to be instantiated) but must be accessed through `GameInstance`
  - `GameInstance` holds the manager instance variable rather than singleton holding its instance variable
- Assumes we cannot eliminate global state so instead focuses on managing it/making it easier to reason about and maintain
  - Should still be a “last resort” rather than the de facto choice

WORKING WITH UNREAL'S FILE SYSTEM

- Can use `FPaths` to access the Unreal File System (UFS)
  - `FPaths::ProjectDir()` returns the `FString` of the project directory
    - Numerous other directories available via the `FPaths` API including access to the Engine
- `FPlatformFileManager` is a system-agnostic file system manager
  - Allows the adding, deleting, moving, etc of files
- `FFileHelper` allows for the reading and writing of files

WORKING WITH DATA TABLES

- **Data tables** can contain flexible data types for use in a variety of situations
  - Essential for dynamic loading of data into scenes when cooking (binaries such as BPs and textures will not be included in build if loaded dynamically)
- Curve tables can only contain floats and are used for interpolating values (i.e. power curves)
  - Specify the type of interpolation between data points
- Can use `UDataAsset` class to customize data types to import/use

CSVs

- CSVs (Comma Separated Values) are flat file structures for storing tabular data
  - Widely used in gameplay development
- UE5 supports data and curve tables for parsing in CSVs
  - Stored in structs that inherit from `FTableRowBase` to define expected column values

DATA HANDLES

- After dragging .csv into Content Folder, can define the expected data row type
- `FDataTableRowHandle` and `FCurveTableRowHandle` expose data to Blueprint for designer use
- Once references are set (usually via Blueprint), possible to call `FCurveTableRowHandle::GetCurve()` and `FDataTableRowHandle::FindRow()` to process data stored
- Pointers to structs should not be cached to prevent stale data

JSON

- JSON (Javascript Object Notation) is the preferred format for transmitting web-based data
  - Can be used locally as well
- Stores values as arrays or objects allowing for flexible hierarchies
- Requires use of Json and JsonUtilities modules (add to Build.cs)
- Use `#include “JsonUtilities.h"`
- Use the `TJsonReaderFactory` to create a reader for deserializing the file
  - Built in parser for accessing values stored in arrays/objects

XML

- XML (Extensible Markup Language) is a very flexible format for storing data
- Stores values in elements allowing complex, flexible (potentially to the point of indecipherable) hierarchies
- Still commonly used in game development for data storage
- Requires use of XmlParser module (add to Build.cs)
- Use `#include "XmlFile.h"`
- `FXmlFile` provides handle to DOM (Document Object Model) for traversing the file like a tree
- `FXmlNode` provides access to the nodes of the DOM

IMPORTING DATA FROM HTTP

- Requires use of the HTTP module (add to Build.cs)
- Use `#include “Runtime/Online/HTTP/Public/Http.h"`
- Calls made through a `FHTTPModule` object
  - `CreateRequest()`
  - `ProcessRequest()`
  - `OnProcessRequestComplete()`

FURTHER READING

- Game Programming Patterns: Singletons <https://gameprogrammingpatterns.com/singleton.html>
- UE4 File Management <https://www.ue4community.wiki/file-and-folder-management-create-find-delete-et2g64gx>
- Data Driven Gameplay Elements <https://docs.unrealengine.com/en-US/Gameplay/DataDriven/index.html>
- Orfeas Eleftheriou: Parsing JSON <https://www.orfeasel.com/parsing-json-files/>
- David Kay: UE4 and HTTP <http://www.davidykay.com/UE4-Hello-Http/>