31263 / 32004 Intro to Games Development Week 9

Dr. William Raffe
william.raffe@uts.edu.au
Senior Lecturer
School of Software, FEIT

Overview

- **Game System Architecture**
  - **Design Patterns: Distributed Control vs Central Managers**
  - **Enumerators and Switch Statements**

- **Scene Management**
  - **Scene Strategies**
  - **Scene Loading**
  - **DontDestroyOnLoad**
  - **Scene Streaming**

- **Saving/Loading Game Data**
  - **PlayerPrefs, Data Serialization, JSONUtility, Resources Folder**

Typical Unity Design Pattern – Distributed Control

- **Every object can manage itself through attached components.**
- **Pros:**
  - **Very flexible**
  - **Quick to implement**
  - **Easy to modify**
- **Cons:**
  - **Messy**
  - **Hard to understand**
  - **Hard to debug**

```
Update()
{ GetInput();
MoveMe(); }
```

```
Update()
{ GetInput();
MoveMe(); }
```

```
Update()
{ GetInput();
MoveMe(); }
```

```
Update()
{ GetInput();
MoveMe(); }
```

```
Update()
{ GetInput();
MoveMe(); }
```

```
Update()
{ GetInput();
MoveMe(); }
```

Issues with Distributed Control

- **What do you do if, when the game starts you need to:**

- **Load player data before initializing enemy difficulty level?**

Issues with Distributed Control

- **What do you do if, when the game starts you need to:**

- **Load player data before initializing enemy difficulty level?**
  - **Load player data in Awake**
  - **Set enemy difficulty in Start**

Issues with Distributed Control

- **What do you do if, when the game starts you need to:**

- **Load player data before initializing enemy difficulty level?**
  - **Load player data in Awake**
  - **Set enemy difficulty in Start**

- **Load player data before initializing enemy difficulty level which in-turn determines pick-up item locations?**

Typical Unity Design Pattern – Distributed Control

- **Every object can manage itself through attached components.**
- **Pros:**
  - **Very flexible**
  - **Quick to implement**
  - **Easy to modify**
- **Cons:**
  - **Messy**
  - **Hard to understand**
  - **Hard to debug**

```
Update()
{ GetInput();
MoveMe(); }
```

```
Update()
{ GetInput();
MoveMe(); }
```

```
Update()
{ GetInput();
MoveMe(); }
```

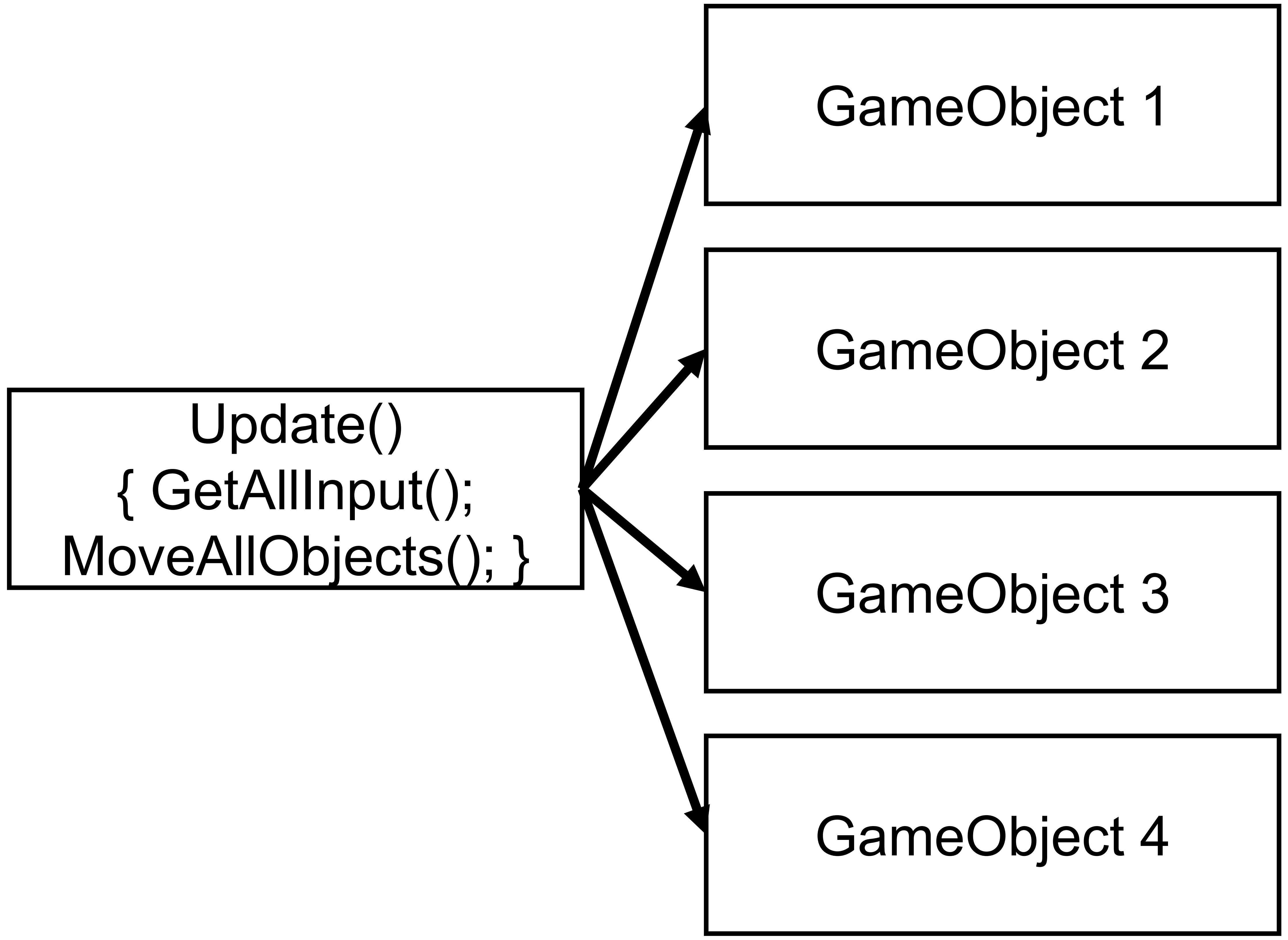
```
Update()
{ GetInput();
MoveMe(); }
```

```
Update()
{ GetInput();
MoveMe(); }
```

```
Update()
{ GetInput();
MoveMe(); }
```

More Traditional Design Pattern – Central Manager

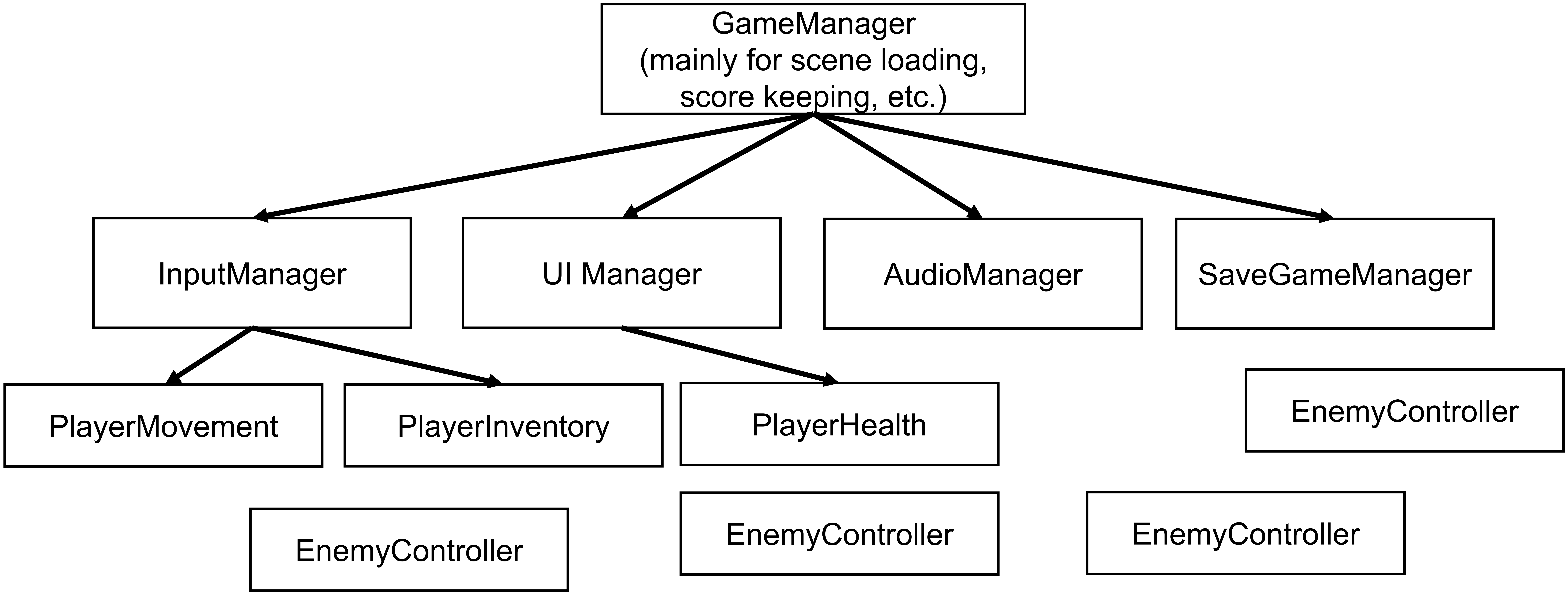
- **Small number of "managers" that coordinate functionality**
- **Pros:**
  - **Easy to debug**
  - **Easy to understand**
  - **Execution is clear to see**
- **Cons:**
  - **Leads to monolithic code**
  - **"High coupling"**
  - **Hard to modify / extend**



Best Approach – Hierarchical Control

- It is always best to mix these approaches
- You may still have a central manager for some coordination (e.g. execution order in scene start)
- But most of the foundational code should be lower down the hierarchy
- Each node in the hierarchy still uses its own Update
  - But each also owns its own relevant info
  - Access info through links in the hierarchy
  - Every node doesn’t need direct access to every other node

Hybrid Architecture / Design Pattern – Hierarchical Control



Hierarchical Control

```
public class GameManager : MonoBehaviour {
        SaveGameManager saveManager;
        DifficultyManager difficultyManager;
        LevelLayoutManager levelManager;

        void Awake() {
                saveManager.Initialize(this);
                saveManager.LoadPlayerData();
                difficultyManager.Initialize(this);
                levelManager.Initialize(this);
        }
}
```

Enumerators

Simple way of:

1. giving labels to integer values for readability
2. maintaining consistency
3. detecting errors during compilation rather than runtime

- A common enumerator that we use:
  - Input.GetKeyDown(**KeyCode.S**)
  - Each KeyCode is just an enumerator value that Unity associates with a keyboard key
  - This helps with finding the right key during code writing (e.g. autocomplete) and gives us an error during compile if we have the wrong one.
  - A lot easier than trying to remember – is it return, Return, enter, or Enter?

Enumerators

enum Scene { MenuScreen, Tutorial, MainGame, BossBattle };

- MenuScreen = 0, Tutorial = 1, MainGame = 2, BossBattle = 3

enum Days { Sat=1, Sun, Mon, Tues, Wed, Thu, Fri };

- Counting starts at 1 and increases for each successive member

enum WeaponDamage { Fist=10, Sword=100, Axe=125 };

- Integer value specified for each member

Using Enumerators

- In arrays:

```
enum Players {Red, Blue};
public Transform[] players;

void Update() {
        players[(int)Players.Red].Translate(….);
        players[(int)Players.Blue].Translate(….);
}
```

Using Enumerators

- For integer math:

```
enum WeaponDamage {Fist = 10, Sword = 100};
int playerHealth;

public void TakeDamage(WeaponDamage weapon) {
	playerHealth -= (int)weapon;
}
```

Using Enumerators

- **For coordinating a game state!**

```
public class GameManager : MonoBehaviour {
        public enum GameState {MainMenu, InGameLevel, Paused, Credits };
        public static GameState gameState
        public static UIManager ui;
        public static PlayerManager playerManager;

        void Awake() {
                ui = GameObject.FindWithTag("UIManagerObject").GetComponent<UIManager>();
                ui.Initilaize();

                if (gameState == GameState.InGameLevel) {
                        playerManager = GameObjectFindWithTag("Player").GetComponent<PlayerManager>();
                        playerManager.Initialize();
                } else if gameState == GameState.Credits) {…..}
```

Using Enumerators

- Note that:
  - **GameState** enum definition is **public**
  - **gameState** enum instance is **public static**

- **Static** – "belongs to the type itself rather than to a specific object." – Microsoft C# Docs
  - A variable or method that is accessed through the class (e.g. **G**ameObject) rather than through individual objects (e.g. **g**ameObject)
  - For static variables, there is only ever 1 value during runtime
  - A common static that we use – **I**nput.GetKeyDown(….)
  - This is a **static method**, it belongs to the class
  - So you <u>can't</u> call:

    Input **i**nput = new Input();

    input.GetKeyDown(…)

Using Enumerators

- Note that:
  - **GameState** enum definition is **public**
  - **gameState** enum instance is **public static**

```
public class InputManager : MonoBehaviour {
        public void Initialize() { …… }

        void Update() {
                if (GameManager.gameState == GameManager.GameState.InGame)
                        GetMovementInput();  // Only move if the game is unpaused
                if (GameManager.gameState == GameManager.GameState.Paused)
                        GameManager.uiManager.ProcessInput(GetUIInput());
        }
```

Scene Strategies

Think about loading multiple high-detailed game levels.

How would you go about it?

Scene Strategies

Think about loading multiple high-detailed game levels.
How would you go about it?

1. **One scene, everything's a prefab**
2. **A few scenes, most things are a prefabs**
3. **One scene for each level/environment**

One scene

- Either lots of little prefabs that are combined through code.
- Or one monolithic prefab that contains everything in a level.
- **Pros:**
  - Can be easier to think about in terms of transitioning between levels seamlessly (<u>if you don't know what your doing</u>)
  - Control?? I honestly don't know, but a lot of people seem to like doing this
- **Cons:**
  - For managing prefabs through code: This can become messy quickly and hard to understand by others.
  - For a monolithic prefabs: this is essentially just re-inventing the idea of a Scene without using the built in support for scenes.

A Few Scenes

- Use a scene to handle similar levels / environments.
- E.g. One scene for the main menu, one scene for standard levels, one scene for boss levels, one scene for end credits.

- **Pros:**
  - Logically separates levels/scenes into similar components.
  - Best of compromise between re-use of the same functionality between levels and flexibility to load different elements of a level through prefabs.
- **Cons:**
  - What should be a scene? What should be a prefab? It will require design time.

Many Scenes

- Every distinct level, menu, etc. is its own scene.
- E.g. Intro Screen scene, Main Menu scene, Level 1 scene, Level 2 scene, Pause Menu scene.
- **Pros:**
  - Makes good use of Unity’s in-built scene management functionality to asynchronously load content.
  - Ensures every scene is self contained – one activity, one scene – preventing bloated scenes
  - Easier to collaborate on through Git – each team member works on one scene
- **Cons:**
  - Lots of scene files in Project Window and to organize in build settings.
  - Similar scenes will all needed to be modified if a shared element is changed.
  - Need to pay attention to how scenes are loaded and unloaded.

Scene Loading

- From a script in one scene, load another scene:

```
SceneManager.LoadScene(string sceneName)
SceneManager.LoadScene(int sceneNumber)
```

- Numbering can be found in “Build Settings” menu, where scenes are added to the list of scenes to build.

Scene Loading

- From a script in one scene, load another scene:

```
if (Input.GetKeyDown(“q”)
  SceneManager.LoadScene(“otherScene”);
  OR
  SceneManager.LoadScene(1);
```

- Numbering can be found in “Build Settings” menu, where scenes are added to the list of scenes to build.

DontDestroyOnLoad

- By default, SceneManager.LoadScene() is done in “**LoadSceneMode.Single**”
  - Only one scene is open at a time, the other scene is closed before the new one is opened.
  - <u>All objects and components from the previous scene are destroyed</u>

- **If you don’t want an object to be destroyed when loading another scene**

```
void Awake() {
   DontDestroyOnLoad(gameObject);
}
```

- Must be called on a **root gameobject** – e.g. a gameobject that is not a child of any other game object.
- With will maintain the entire hierarchy of that root gameobject
- Useful for keeping a central GameManager and a loading screen between scenes

Asynchronous Loading

- Unloading and Loading scenes can be resource intensive if the scene is large.
  - Must pull files from disk and load them into memory.

- If the scenes are large, the game will freeze while the transition happens.

- To hide this from the player:

1. Show loading screen
   and/or
2. Use Asynchronous Loading

Asynchronous Loading

```
SceneManager.LoadSceneAsync("otherScene");
SceneManager.LoadSceneAsync(1);
```

- Creates a new process thread to load the scene in the background.
  - Will visually swap scenes as soon as loading has finished.
  - i.e. it will look like it happens instantly but it is actually occurring over a few frames or even seconds
  - Will then unload the old scene in the background.

- Useful for uninterrupted spinning loading logos or loading during elevator sequences where player can still move around

Scene Streaming

- **If you don't want anything from the previous scene to be destroyed:**

```
SceneManager.LoadSceneAsync(1, LoadSceneMode.Additive);
```

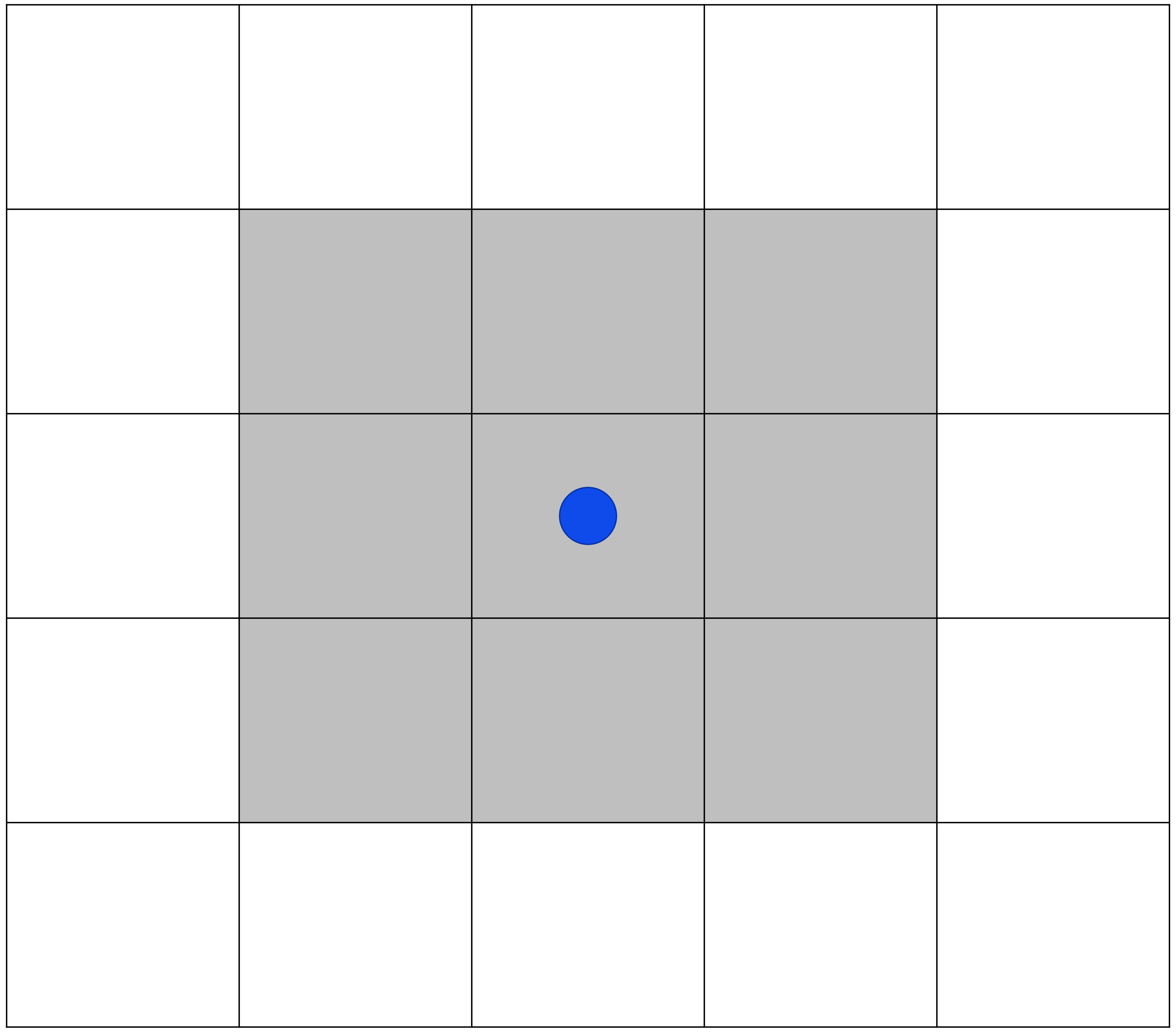
- **You can later unload the previous scene at anytime with:**

```
SceneManager.UnloadSceneAsync(1);
```

- This is useful for:
  – Loading temporary gameobjects (e.g. in-game menus) as their own scene.
  – No loading screens in open world games!

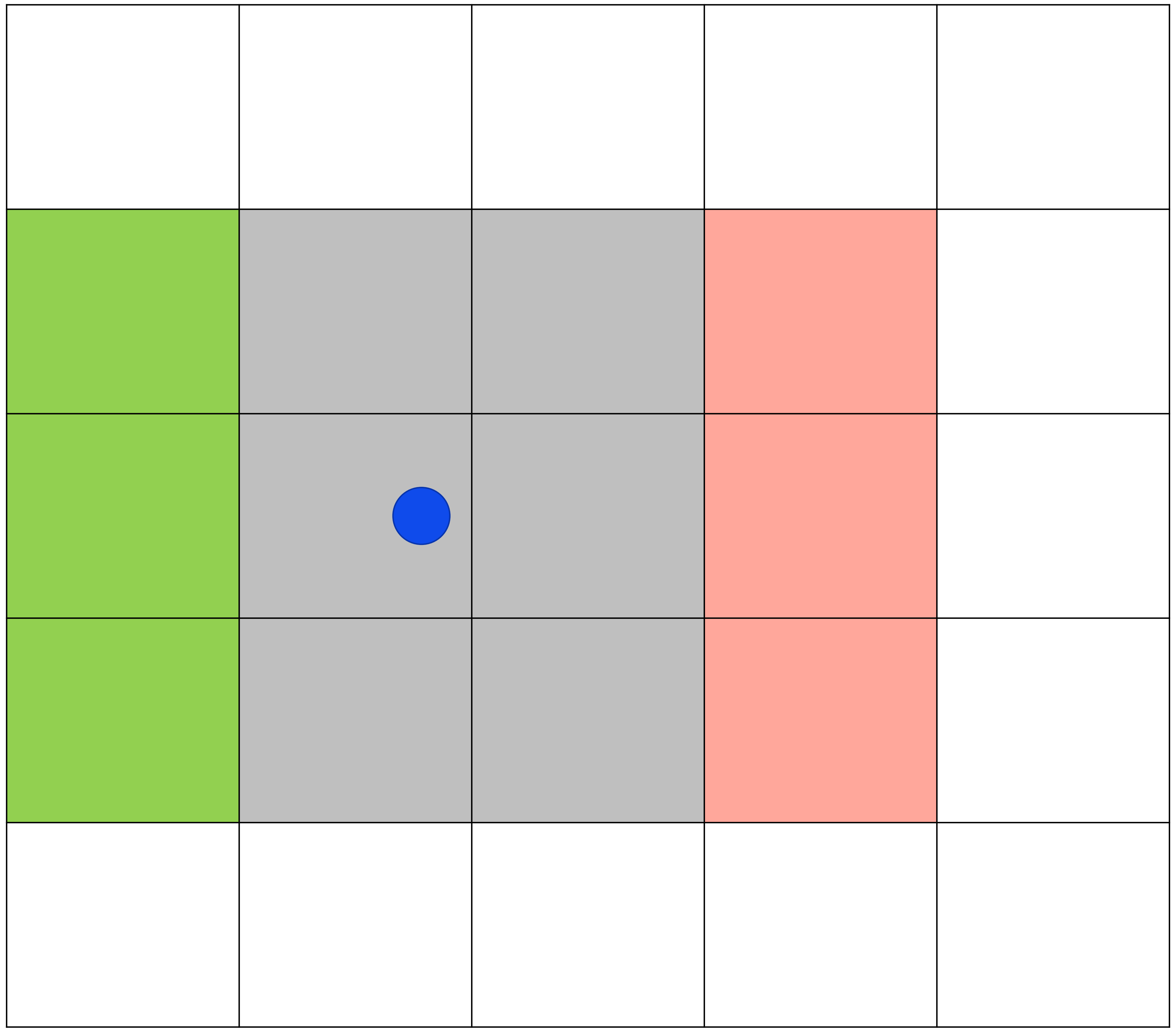
Open World Games

⬜	Scene with part of the world
▨	Loaded Scene
🟥	Recently Unloaded Scene
🟩	Recently Loaded Scene
●	Player



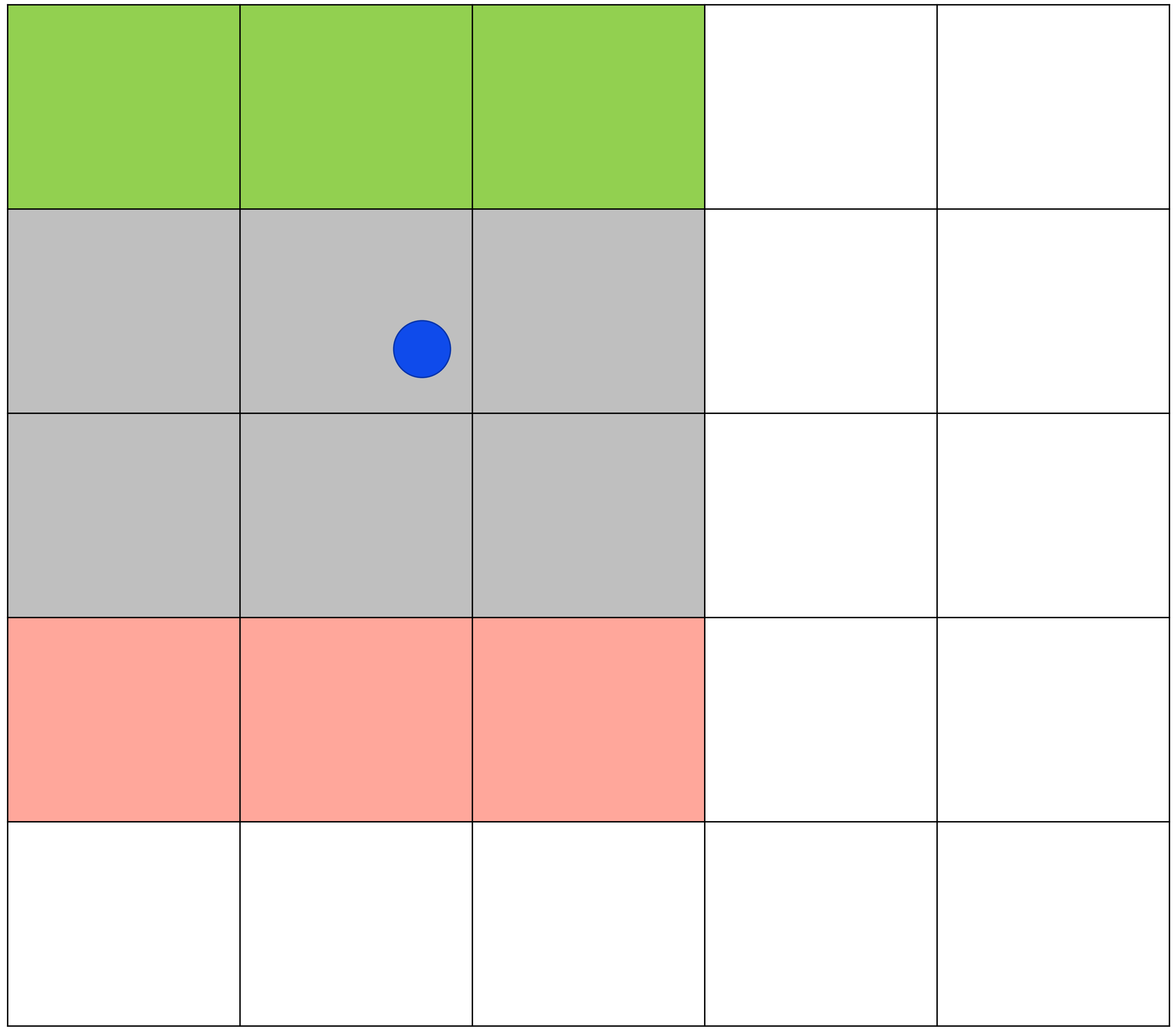
Open World Games

⬜	Scene with part of the world
▨	Loaded Scene
🟥	Recently Unloaded Scene
🟩	Recently Loaded Scene
●	Player



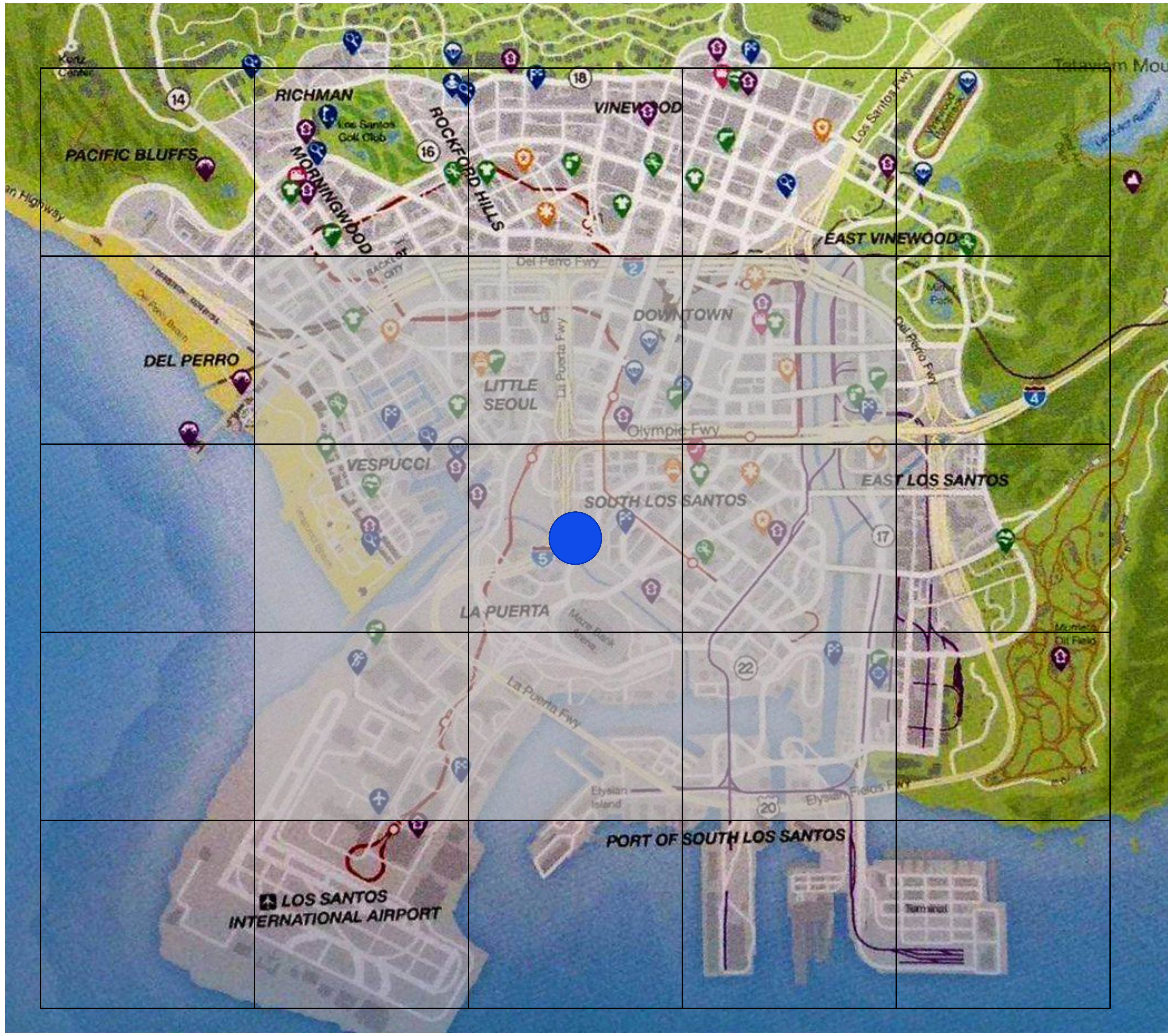
Open World Games

⬜	Scene with part of the world
⬛	Loaded Scene
🟥	Recently Unloaded Scene
🟩	Recently Loaded Scene
●	Player



Open World Games

⬜	Scene with part of the world
⬜	Loaded Scene
🟥	Recently Unloaded Scene
🟩	Recently Loaded Scene
●	Player



Saving Games - PlayerPrefs

- **The easiest way to save and load data.**
  - Works immediately on all platforms (Windows, Mac, iOS, Android, etc), no extra code needed
  - If you do manual file writing instead, each system has their own ways of storing data and you will need to implement each one
- Data stored as `<key,value>` pairs (like a hash map or dictionary)

```
const saveKey = "Player Name"
Private void SavePlayerName() {
    string saveValue = SomeGetPlayerNameMethod();
    string loadValue = PlayerPrefs.GetString(saveKey);
    if (!saveValue.Equals(loadValue)) {
            PlayerPrefs.SetString(saveKey, saveValue);
            PlayerPrefs.Save();
    }
}
```

PlayerPrefs Methods

```
int PlayerPrefs.GetInt(string key, int value);
float PlayerPrefs.GetFloat(string key, float value);
string PlayerPrefs.GetString(string key, string value);

void PlayerPrefs.SetInt(string key, int value);
void PlayerPrefs.SetFloat(string key, float value);
void PlayerPrefs.SetString(string key, string value);

bool PlayerPrefs.HasKey(string key);

void PlayerPrefs.DeleteKey(string key);
void PlayerPrefs.DeleteAll();
```

void PlayerPrefs.Save(); - Unity will auto-write to disk OnApplicationQuit(), but if game crashes??

PlayerPrefs Extras

- PlayerPrefs are fast!
  - E.g. in Windows, stored in the registry, quick OS supported look-up
- PlayerPrefs are not cleared when the app is updated (iOS/Android)
- Limit of PlayerPrefs strings are enforced by operating system.
  - In Windows, registry has 1mb string limit (that’s still big!)
  - In Android, no limit (Android OS will just kill an app using too much memory)
- The editor doesn’t have a window for PlayerPrefs.
  - Use a PlayerPrefs editor plugin from the AssetStore
  - E.g. PlayerPrefs Elite – Makes visualising and debugging save data much easier.

File Writing

- For things too big or too custom to fit in PlayerPrefs…

- You can still read/write files with all the usual .NET C# functionality.

```
FileStream file = File.Open(Application.persistentDataPath +
                     "/gameInfo.dat", FileMode.Open);
```

- Application.persistentDataPath holds an OS dependent folder location to safely write to.
  - This data is not cleared when the app is updated (iOS/Android)
  - Can be interrupted if user e.g. removes SD card
  - Beware of other OS specific requirements for file formats!

Object Serialization

- C# method to convert objects (i.e. instantiated classes) to binary data

```
[System.Serializable]
public class SaveData() {
    int playerScore;
}

public class DataManager() {
    void SaveData(string filePath) {
            SaveData playerData = new SaveData();
            playerData.playerScore = GameManager.GetCurrentScore();
            BinaryFormatter bf = new BinaryFormatter();
            FileStream file = File.Create(filePath);
            bf.Serialize(file, playerData);
    }
```

JSON

- Powerful mark-up language
- OS independent
- The evolution of XML
- Used a lot in web development and a form of data storage.

```
{"widget": {
      "debug": "on",
      "window": {
            "title": "Sample Konfabulator Widget",
            "name": "main_window",
            "width": 500,
            "height": 500 },
      "image": {
            "src": "Images/Sun.png",
            "name": "sun1",
            "hOffset": 250,
            "vOffset": 250,
            "alignment": "center" },
}}
```

JsonUtility

- Convert an object to a JSON string
  - Generic versions for most languages, Unity has its own

- Great for sharing data over the web
- Or creating a string from an object an writing it to PlayerPrefs!
- Or embedding object strings in your scripts
  - Represent and entire level as a string!!!!

```
myObject = JsonUtility.FromJson<MyClass>(jsonString);
JsonUtility.FromJsonOverwrite(jsonString, myObject);
jsonString = JsonUtility.ToJson(myObject);
```

- The JSON string doesn’t need to be complete
  - You can load in only partial data, only overwrite some object values

JsonUtility.FromJson(string) (from Unity Docs)

```
[System.Serializable]
public class PlayerInfo
{
    public string name;
    public int lives;
    public float health;

    public static PlayerInfo CreateFromJSON(string jsonString)
    {
        return JsonUtility.FromJson<PlayerInfo>(jsonString);
    }

    // Given JSON input:
    // {"name":"Dr Charles","lives":3,"health":0.8}
    // this example will return a PlayerInfo object with
    // name == "Dr Charles", lives == 3, and health == 0.8f.
}
```

JsonUtility.ToJson() (from Unity Docs)

```
public class PlayerState : MonoBehaviour
{
    public string playerName;
    public int lives;
    public float health;

    public string SaveToString() {
        return JsonUtility.ToJson(this);
    }

    // Given:
    // playerName = "Dr Charles"
    // lives = 3
    // health = 0.8f
    // SaveToString returns:
    // {"playerName":"Dr Charles","lives":3,"health":0.8}
}
```

ScriptableObject

- A way of storing data in an asset / prefab
  - I.e. a prefab for data only

- Doesn't need to be attached to a game object, doesn't need to be instantiated
  - Just exists as an asset in the Project Window
  - Often used when making editor tools and extensions (e.g. custom Inspector windows)

- Great for visualizing, saving and loading of game data made by you (designer/developer) – not for run-time player date (use PlayerPrefs or others):
  - level data
  - weapon and ability properties
  - Character names, etc.

Resources Folders

- Any folder in the Project Window named “Resources”

- Kept separate from all other assets in the Project Window
  - Can be accessed by file path.
  - Most assets will be cleaned away by the garbage collector when not used.
  - These won’t, they must be loaded and unloaded manually.

- Use not encouraged by Unity except for certain circumstances

- E.g. Dynamically assigning one of dozens of textures to a procedurally generated model.
  - Storing a reference to each texture in script will cause all to be loaded on Awake()
  - Use Resources folder to only load the one you want!