Programming Patterns

Programming Patterns

- 1994 – Gang of four release “Elements of Reusable Code”
  - Program to an interface not an implementation
  - Favor object composition over inheritance
- I'm assuming you all know this...

Unity

- Unity is actually Design Pattern heavy. It's built on:
  - Behavioral Patterns:
    - “Type Object”
  - Sequencing Patterns:
    - “Game Loop”
    - “Update Method”
  - Decoupling Patterns:
    - “Component Model”

- But we can take this further
- Unity does not enforce any structure on us, so we need to be careful when we build our own

Singleton

- You definitely should know this...
  - Ensure there's one instance of a class
  - With a globally accessible point of access
- In practice, a static class member variable we init on first use.
- Why?
  - “Managers”

Singleton

- Pros:
  - Doesn't exist if we don't use it *(CPU/Mem)*
  - Initialised at runtime. *(Static classes are initialised before “main()”)*
  - Can be subclassed – variants for different platforms!

Singleton

- Cons:
  - It's Global – Anything could be touching that data during this Update()
    - We'll end up putting Set{...} wrappers around data to debug efficiently
    - Instead, we can pass references in through inheritance / add functionality to base classes
  - Encourages “managers”
    - “Bullet Manager” / “Baddie Manager”

Object Pool

- Another we know about
  - “Create” all the GameObjects we need during Start/Awake
  - Stick them in an array, or “pool”
  - Instead of `Instantiate()`, grab a “new” object from the pool
  - When object “dies” it returns to the pool
- Why?
  - Garbage collection is bad, m'kay?

Command

- “Object Oriented replacements for callbacks”
- Allows us to wrap a method and data into an object that can be executed and/or stored
- Instead of:

```
if(m_CharAction.Jump.IsPressed)
    DoJump();
```

- We add a layer of indirection:

```
interface ICommand
{
  void Execute() …


if(m_CharAction.Jump.IsPressed)
    Jump.Execute()
```

Why?

- Through inheritance or Interface implementation we can perform a direct “action” on any object that implements the command.
- Or through composition, we can perform the command on the object instance we hold, to affect ourselves
- We can modify this indirection to “return” commands, which can be stored
  - `if(Action) return Command`

- What's the big win?
  - If you can defined “Execute” you can also define “Undo”
- When we're returning commands, we can store them in an array.

- Advance Wars:
  - Want to cancel an order? Call `Unit.Undo()` and move the current pointer in the array back by one.
  - Redo? `Unit.Execute()` and move the pointer forward.
- UI
  - Track user changes, allow for “cancel”
- Dead useful...

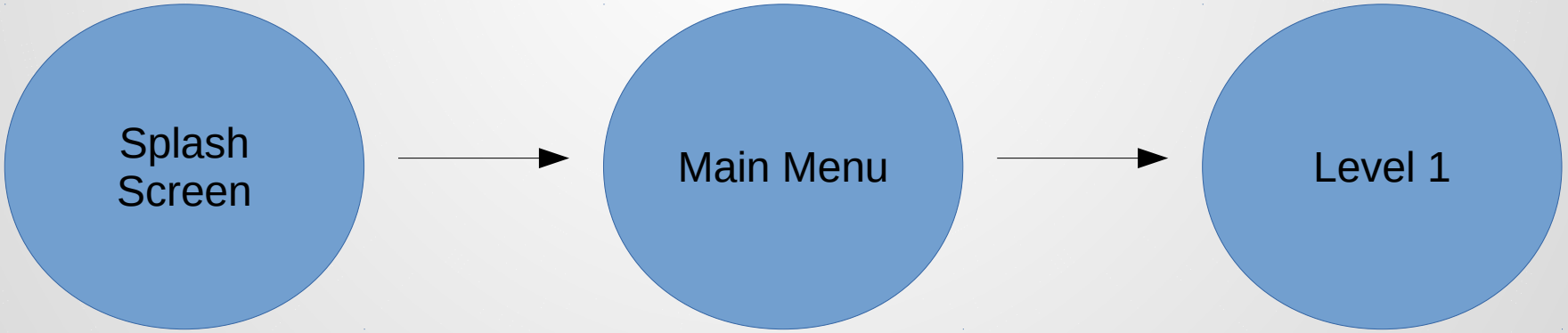
Observer

- This pattern is everywhere.
- Is the basis of the Model-View-Controller pattern
  - `Java.util.Observer`
  - Or in C#: `public event MyEventHandler …`
- Relevant code can trigger an event, Observers can respond to the event
- Most common use in games: Achievements

- I use this a lot in the UI
  - One component on a GameObject setup as an observer
  - Has functions that respond to button events
  - Buttons fire events when player does something
  - The observer can “block” new events until it has responded to the initial one
  - Observer can directly effect canvas objects it knows about
  - Less logic in individual UI Canvas / Buttons – they remain decoupled.

State

- This is what we care about today.
- Games have many, many different states.
- At a high level:



- But it's more fine grained:
- Splash Screen will:
  - Fade logo in
  - Play some animation
  - Fade Logo out
  - And do setup work in the background.
- Each of these is a “state” of the Splash Screen

- AI & NPCs all have state
  - Behaviour trees are effectively state machines
- Animation
  - Mecanim is a state machine
- So we'll use this pattern again and again...

- Most people's first approach at this is a collection of `If` statements, or a `Switch`:

```
switch(currentState)
{
   Case SplashScreen:
      if(LogoFading)
         Do something here
      Else
         Ok, do this other thing here
```

```
        break;
      }
      else
      {
        _LevelNumber ++;
      }
    }
    else
    {
      _LevelNumber == NUMBER_OF_LEVELS-1 ? _LevelNumber = START_LEVEL : _LevelNumber ++;
    }

    KillLevel();
//      pmesg(1, "%s:\n\tLevelNumber: %d\n",__FUNCTION__, _LevelNumber );
    InitLevel( _LevelNumber );
    _TransitionFramecount = 0;
    *_GameState = LevelInLoop( _BZ_Interface);
    break;

  case TRANSITION_LEVEL_IN:
    *_GameState = LevelInLoop( _BZ_Interface );
    if ( *_GameState == IN_GAME )
    {
      _TransitionFramecount = 0;
    }
    break;

  case SETUP_HISCORE:
#if !(TARGET_OS_MAC)
    KillLevel();
    KillGame();
#else
    NumberOfCharactersEntered = 0;
#endif

    pmesg( 2, (char*)"%s:\n\tState changed to SETUP_HISCORE\n", __FUNCTION__);
    if ( *_Score > *_HighScore )
    {
      *_HighScore = *_Score;
      pmesg ( 10, (char*)"%s:\n\t>>>>>> New HiScore Found: %d - %d\n", __FUNCTION__, *_Score, *_HighScore );
      *_GameState = IN_HISCORE;
    }
    else
    {
      pmesg( 2, (char*)"%s:\n\tState changed to NO_HISCORE\n", __FUNCTION__);
      *_GameState = NO_HISCORE;
    }
    break;

  case FADE_GAME_OUT:
    _TransitionFramecount++;
    if ( _TransitionFramecount > 165 )
    {
      *_GameState = SETUP_HISCORE;
    }
#if !(TARGET_OS_MAC)
    else if ( _TransitionFramecount > 105 && (*_Score > *_HighScore) )
    {
      _Game_HighScoreBackdrop->Process();
      _Hud_Writer->Add(GAME_WIDTH/2, 310, "CONGRATULATIONS! NEW HI-SCORE!", STATIC, &In_Game_SmallFont);
      _Hud_Writer->Add(GAME_WIDTH/2, 290, "TAP TO ENTER YOUR NAME!", STATIC, &In_Game_SmallFont);
      _Hud_Writer->Process();
    }
#endif
    else if ( _TransitionFramecount >= 45 )
```

- 900 line switch statement
- Most cases had several embedded IF statements
- When it did get to a function call, many of these contained their own switch statements
- Debugging this was “not fun”
- Adding features (Demo version) was “not fun”

- To use the technical term:

"This grows arms and legs"

Finite State Machines

- The FSM is a mathematicl model of computation
  - Computer Scientists love it
- State machines can be represented by directed graphs (State Diagrams)
  - Software Engineers love these
- Game developers love stealing things, so lets take the bits we want and bend it to our will...

State

- In terms of Unity, simplest implementation:
  - A “state” is a component on a GameObject
- The state's Update() handles our processing
- We can add Observers / other functionality through composition
  - Either through direct invocation
  - Or by defining events
- It is solely responsibly for movement to the *next* state

State

```
public class GameState : MonoBehaviour
{
    protected string m_sStateName = "BaseClass";

    virtual public void Start()
    {
        Debug.Log("GameState: " + m_sStateName + "\n" );
    }

    virtual public void Update()
    {
    }

    virtual public void OnDisable()
    {
        Debug.Log("Exiting game state...\n");
    }
}
```

- The base class needs a couple of other things to be useful:

```
public class GameState : MonoBehaviour
{
    protected GameStateManager    m_gcGameStateManager;
    protected GameGlobals         m_gcGameGlobals;
```

- We can compose this in many ways, but one is:

```
virtual public void Start()
{
    Debug.Log("GameState: " + m_sStateName + "\n" );
    m_gcGameGlobals = GameGlobals.Instance;
    m_gcGameStateManager =
                m_gcGameGlobals.m_gcGameStateManager;
}
```

- Simple derivation example [SplashScreen_In]:

```
public class gs_SplashScreenIn : GameState
{
    private float m_fEventTime;
    new void Start()
    {
        m_sStateName = "Splash Screen";
        base.Start();
        m_fEventTime = Time.time;
    }
    new void Update()
    {
        if (Time.time - m_fEventTime > 1.0f)
            m_gcGameStateManager.ChangeState(...);
    }
```

Game State Manager

- Instead of a single Switch statement, and a lot of Ifs
- We break down the game into a series of defined states
- Create a set of components that we can instantiate – at runtime – to control the flow of the game
- But we need functionality to manage this...

- Unity also has some quirks that we need to take into account
  - Existing objects are deleted when a new scene is loaded
    - DontDestroyOnLoad
  - When loading a scene, all scripts are paused
    - The C# runtime is not executing at all
  - There is no promise about execution order
    - This leads to race conditions
    - Although we can set priorities in the editor

- GameStateManager can sit over the top of this
  - Provides a function to “`ChangeState`”
  - Sets up a new game object with the correct game state component
  - Handle Unity specifics for scene changes
  - Do any funky tricks we need for dev mode verses release (more on that later)

Change State

```
public void ChangeState(GameState.tStateType iState, string
sLevelName="" )
{
       Destroy(m_goGameState);
       m_goGameState = new GameObject();
       DontDestroyOnLoad(m_goGameState); // Never destroy...

       if (sLevelName != "")
       {
           m_bWaitingForLoad = true;
           m_iNewState = iState;
           SceneManager.LoadScene(sLevelName);
        }
        else
            SetState(iState);
    }
```

Set State

```
private void SetState(GameState.tStateType iState)
{
    switch (iState)
    {
        case GameState.tStateType._SplashScreenIn:
            m_goGameState.AddComponent<gs_SplashScreenIn>();
            m_goGameState.name = "GS: Splash Screen";
            break;

        case GameState.tStateType._MainMenuIn:
            m_goGameState.AddComponent<gs_MainMenuIn>();
            m_goGameState.name = "GS: Main Menu";
            break;
        Etc.
    }
    m_iNewState = GameState.tStateType._NULL;
    m_bWaitingForLoad = false;
}
```

```
void OnEnable()
{
    SceneManager.sceneLoaded += LevelWasLoaded;
}

void OnDisable()
{
    SceneManager.sceneLoaded -= LevelWasLoaded;
}
```

```
public void LevelWasLoaded(Scene scene, LoadSceneMode mode)
{
    if (m_bWaitingForLoad)
    {
       SetState(m_iNewState);
    }
}
```

- When `ChangeState` is called
  - Destroys the existing `GameState`
  - Tells Unity to load a new scene (if required)
  - Waits for Unity to finish and responds to sceneLoaded event (if required)
  - Creates a new Game Object and attaches the new Game State component to it
  - Game State will then go through it's initialisation process as expected (Awake/Start)

- Game State Manager can be a
  - Static class
  - Singleton
  - Component on a game object
  - etc
- I tend to do the latter during development
  - Exposing variables makes them visible in the editor
  - Much easier to debug stuff at runtime when you can click on a game object and see what it's doing!

Scene Setup

- We're going to end up with a few classes that are omnipotent in our game
  - Audio Interface
  - Game State Manager
  - Input Interface
  - Game Globals
  - Achievements / Leaderboard Observers – Platform specific
  - Localisation
  - Save Games

- What I tend to do is set these up in a Splash Screen
- A “Persistent Objects” game object (DontDestroyOnLoad)
  - Child objects, each with a manager component on.
  - Obviously you can drop these all on one GameObject but I like some segregation
- Often my GameGlobals singleton will contain a member variable for each manager

- If I'm being amazingly lazy:
  - `GameGlobals.Instance.m_gcGameStateManager….`
  - `GameGlobals.Instance.m_gcAudioManager…`
  - Danger, Danger, Danger!
- Handy for quick access to events for delegation, tho...

- One BIG problem with this…
  - If I'm working in a random level, and press play…
  - No persistent objects!
  - Game's not going to run properly!

DevModeCheck

```
public class DevModeCheck : MonoBehaviour {
    public string m_sSceneName;
    void Awake()
    {
        GameObject obj = GameObject.Find("PersistentObjects");
        if (null == obj)
        {
            Debug.Log("Unable to find Persistent objects, loading the splashscreen...");
            DontDestroyOnLoad(gameObject);
            SceneManager.LoadScene("SplashScreen");
        }
    }
}
```

Game State Manager

```
 void Awake()
{
        DontDestroyOnLoad(GameObject.Find("PersistentObjects"));

        // Check for Dev Mode
        GameObject obj = GameObject.Find("DevModeCheck");
        if (null != obj)
        {
            Debug.Log("Found a DevModeCheck from open scene,
entering dev mode...");
            DevModeCheck scene =
obj.GetComponent<DevModeCheck>();
            m_sDevModeSceneToLoad = scene.m_sSceneName;
            Destroy(obj);
        }
    }
```

Game State Manager

```
  void Start ()
  {
        // Did Awake find a dev mode check game object?
        if (m_sDevModeSceneToLoad != "")
        {
            ChangeState(GameState.tStateType._GameIn,
m_sDevModeSceneToLoad);
            m_sDevModeSceneToLoad = "";
        }
        else
        {
            // No? Start up as normal then
            DontDestroyOnLoad(m_goGameState);
            SetState(GameState.tStateType._SplashScreenIn);
        }
    }
```

- So we create a game object called `DevModeCheck`
- We set the “scene name” string to the name of the scene we're working in (dur)
- And woo, no matter where we press play, the game will be setup as if we'd gone through the normal boot process
  - Handy shortcut
  - No duplication of stuff in every scene

Your Misson

- Create 2 new scenes:
  - `SplashScreen`
  - `MainMenu`
- Two new classes: `GameStateManager` & `GameGlobals`
  - Set them up in the Splash Screen scene
- Write a base `GameState` class
  - And derive 3 game states:
    - `SplashScreen_In`
    - `MainMenu_In`
    - `Game_In`

- For the first pass:
  - Splash Screen should just load Main Menu
  - Main Menu should just load Level 1
  - Level 1 should play as we have it now…
  - Extra points for `DevModeCheck`
- When you've done this, zip up your project, keep it safe
  - ***You will be required to hand it in at the end of the course!***