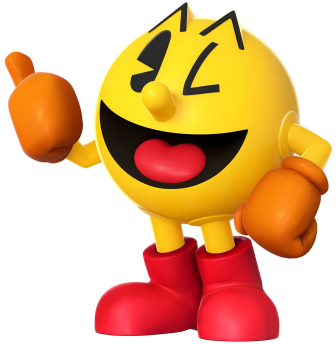
# Game Engine Design Final Exam Explanation

By: Joshua Yap

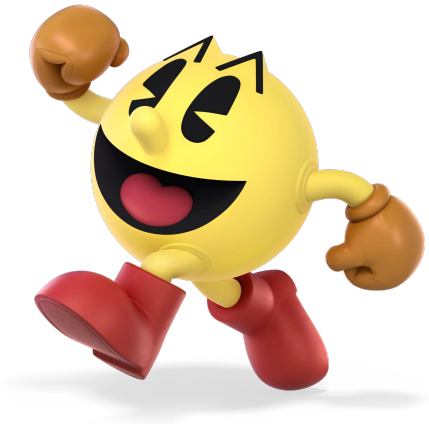


# Legends

Slides 3-5 - Object Pooling Explanation

Slides 6-10 - Command Design Pattern Explanation

Slides 11-15 - Editor Management System



# Object Pooling

First I create a list to store the amount of ghosts in the scene

```
// Start is called before the first frame update
Unity Message | 0 references
void Start()
{
    allEnemies = new List<Enemy>();

    fn = Application.dataPath + "/save.txt";

    //LoadEnemy();
}
```

# Object Pooling LoadEnemy() class

Using Flyweight, the LoadEnemy() class loads in a new ghost

It then loads the ghost's position and A.I. and sends them to the Enemy class

```
0 references
void LoadEnemy()
{
    int numLines = GetLines(fn);
    int maxItems = numLines / 4;
    int infoSet = 0;

    //using flyweight
    Enemy newEnemy = new Enemy();
    float y = LoadFromFile(2, fn);

    for (int j = 0; j < 10000; j++)
    {
        for (int i = 0; i < maxItems; i++)
        {
            //using flyweight
            newEnemy.enemyID = (int)LoadFromFile(0 + infoSet, fn);
            newEnemy.enemyPosition.x = LoadFromFile(1 + infoSet, fn);
            newEnemy.enemyPosition.y = y;
            newEnemy.enemyPosition.z = LoadFromFile(3 + infoSet, fn);

            allEnemies.Add(newEnemy);
            infoSet += 4;
        }
        infoSet = 0;
    }
```

# Object Pooling LoadEnemy() class

The maximum enemy count is set to 4 because there only 4 ghost enemies in Pac-Man.

```
0 references
void LoadEnemy()
{
    int numLines = GetLines(fn);
    int maxItems = numLines / 4;
    int infoSet = 0;

    //using flyweight
    Enemy newEnemy = new Enemy();
    float y = LoadFromFile(2, fn);

    for (int j = 0; j < 10000; j++)
    {
        for (int i = 0; i < maxItems; i++)
        {
            //using flyweight
            newEnemy.enemyID = (int)LoadFromFile(0 + infoSet, fn);
            newEnemy.enemyPosition.x = LoadFromFile(1 + infoSet, fn);
            newEnemy.enemyPosition.y = y;
            newEnemy.enemyPosition.z = LoadFromFile(3 + infoSet, fn);

            allEnemies.Add(newEnemy);
            infoSet += 4;
        }
        infoSet = 0;
    }
```

# Object Pooling - Enemy Class

The Enemy class then loads in the enemy’s ID and their position which is then sent to the game.

```
4 references
public class Enemy
{
    public int enemyID;
    public Vector3 enemyPosition;
}
```

# Command Design Pattern - CommandInvoker()

Before adding the pellets we need to first define the commands.

The lists commandBuffer and commandHistory are created in order to store player's inputs

```
Unity Script | 0 references
public class CommandInvoker : MonoBehaviour
{
    GameControls inputAction;

    static Queue<ICommand> commandBuffer;

    static List<ICommand> commandHistory;
    static int counter;

    Unity Message | 0 references
    private void Start()
    {
        commandBuffer = new Queue<ICommand>();
        commandHistory = new List<ICommand>();

        inputAction = PlayerInputController.controller.inputAction;

    }
```

# Command Design Pattern - PelletPlacer()

The PelletPlacer() class creates and removes pellets

When a pellet is collected the RemovePellet subclass deletes a pellet from the list

```
Unity Script | 2 references
public class PelletPlacer : MonoBehaviour
{
    static List<Transform> items;

    1 reference
    public static void PlacePellet(Transform item)
    {
        Transform newitem = item;
        if (items == null)
        {
            items = new List<Transform>();
        }
        items.Add(newitem);
    }

    1 reference
    public static void RemovePellet(Vector3 position)
    {
        for (int i = 0; i < items.Count; i++)
        {
            if (items[i].position == position)
            {
                GameObject.Destroy(items[i].gameObject);
                items.RemoveAt(i);
                break;
            }
        }
    }
}
```

# Command Design Pattern - PlacePelletsCommand()

This class defines the pellets position and size

It also tells the execute and undo functions to focus on adding and removing pellets

```
1 reference
public class PlacePelletsCommand : ICommand
{
    Vector3 position;
    Transform item;

    0 references
    public PlacePelletsCommand(Vector3 position, Transform item)
    {
        this.position = position;
        this.item = item;
    }

    2 references
    public void Execute()
    {
        PelletPlacer.PlacePellet(item);
    }

    2 references
    public void Undo()
    {
        PelletPlacer.RemovePellet(position);
    }
}
```

# Command Design Pattern - ICommand

To round it all together the ICommand class executes the defined command classes which adds them to the player interface

```
7 references
public interface ICommand
{
    2 references
    void Execute();
    2 references
    void Undo();
}
```

# Editor Management System

The Editor allows players to add or remove assets from the scene

From enemies to pellets, this gives the player easy access to all in-game assets

Description:

In Editor mode, players can customize their own level by adding and removing things in the current level. Players can add enemies, pellets, and fruit to whatever position they see fit.

# Editor Manger - UIManager

Before creating the editor, I first needed to create the UI itself.

This is done by creating a UIManager class which will determine whether or not the player is in editor mode.

When triggered the editor UI will appear

```
Unity Script | 2 references
public class UIManager : MonoBehaviour
{
    public Canvas editorUI;

    bool editorMode;

    // Start is called before the first frame update
    Unity Message | 0 references
    void Start()
    {
        editorMode = GetComponent<EditorManager>().editorMode;

        if (editorMode == false)
        {
            editorUI.enabled = false;
        }
    }

    1 reference
    public void ToggleEditorUI()
    {
        editorUI.enabled = !editorUI.enabled;
    }
}
```

# Editor Management System

The AddItem and DropItem functions allow the player to add enemies and pellets into the current scene. It does that by instantiating the associated object and them to the list. When the item is dropped into position the instantiated will be turned off until the player selects another object.

```
public void AddItem(int itemId)
{
    if (editorMode && !instantiated)
    {
        switch (itemId)
        {
            case 1:
                item = Instantiate(prefab1);
                //Create boxes that can observe events and give them an event to do
                Enemy spike1 = new Enemy();
                //Add the boxes to the list of objects waiting for something to happen
                break;
            default:
                break;
        }
        instantiated = true;
    }
}
```

```
0 references
public void DropItem()
{
    if (editorMode && instantiated)
    {
        if (item.GetComponent<Rigidbody>())
        {
            item.GetComponent<Rigidbody>().useGravity = true;
        }
        item.GetComponent<Collider>().enabled = true;

        // Add item transform to items list
        command = new PlacePelletsCommand(item.transform.position, item.transform);
        CommandInvoker.AddCommand(command);

        instantiated = false;
    }
}
```

# Editor Management System - Camera

The editor camera is separate from the normal camera so it needs its own function. When the player is in editor mode, the game will be locked in the current state. If instantiated the mouse will be visible during editor mode.

```
void Update()
{
    // Checking if we are in editor mode
    if (mainCam.enabled == false && editorCam.enabled == true)
    {
        // Stop all movement in game
        Time.timeScale = 0;
        editorMode = true;

        // Making cursor visible when in editor mode
        Cursor.lockState = CursorLockMode.None;
    }
    else
    {
        Time.timeScale = 1;
        editorMode = false;

        // Making cursor invisible when in play mode
        Cursor.lockState = CursorLockMode.Locked;
    }

    if (instantiated)
    {
        mousePos = Mouse.current.position.ReadValue();
        mousePos = new Vector3(mousePos.x, mousePos.y, 40f);

        item.transform.position = editorCam.ScreenToWorldPoint(mousePos);
    }
```

# Editor Management System - How it benefits Pac-Man

Pac-Man can really benefit from an editor manager since there is very little variations within the original game aside from the ghosts speeding up

Having a editor management can allow developers and players essentially create custom levels.

Seeing level editors like Super Mario Maker become popular really makes me feel that almost any game can benefit from an editor mode.