Final Exam

Here is the link to my Github repository, https://github.com/DezrtS/GameEngineFinalExam

$1 + 8 + 1 + 7 = 17 \rightarrow$ **Prime**

MegaMan 6 Clone

First, to implement the snowflakes, I created multiple classes and an interface. I needed to create a BlizzardMan class to control when the snowflake attack would trigger, I had to create a Snowflake class to handle the snowflake behavior, I had to create a ICommand interface to define the shared methods between commands, I created a InvertMegaManCommand class to handle the inverting logic, and I had to create a MegaMan class to serve as the player. Starting off with the ICommand interface,

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

I created this interface to easily allow me to call commands of any type. This is especially useful with a command invoker class that stores these commands so undo/redo functionality can be implemented.

```
public class CommandInvoker
{
    public readonly Stack<ICommand> CommandHistory = new Stack<ICommand>();
    public readonly Stack<ICommand> RedoStack = new Stack<ICommand>();

    0 references
    public void Execute(ICommand command)
    {
        command.Execute();
        CommandHistory.Push(command);
        RedoStack.Clear();
    }
```

```
    0 references
    public void Undo()
    {
        if (CommandHistory.Count > 0)
        {
            ICommand command = CommandHistory.Pop();
            command.Undo();
            RedoStack.Push(command);
        }
    }

    0 references
    public void Redo()
    {
        if (RedoStack.Count > 0)
        {
            ICommand command = RedoStack.Pop();
            command.Execute();
            CommandHistory.Push(command);
        }
    }
}
```

The Snowflake class creates a InvertMegaManCommand class instance and passes it to the GameManager singleton for execution, which contains a global command invoker for all game commands.

```
Unity Message | 0 references
private void OnTriggerEnter2D(Collider2D collision)
{
    if (collision.CompareTag("Player"))
    {
        ICommand command = new InvertMegaManCommand(MegaMan.Instance);
        GameManager.Instance.InvokeCommand(command);
        OnSnowflakeFinish?.Invoke(this);
    }
}
```

```
1 reference
public void InvokeCommand(ICommand command)
{
    commandInvoker.Execute(command);
}
```

For the InvertMegaManCommand class specifically, I created a constructor that takes in a reference to the MegaMan class,

```
private MegaMan megaMan;

1 reference
public InvertMegaManCommand(MegaMan megaMan)
{
    this.megaMan = megaMan;
}
```

This reference is then used to reverse the y value of the scale and flip the boolean that controls whether Megaman can or can't jump,

```
5 references
public void Execute()
{
    Vector3 scale = megaMan.transform.localScale;
    scale.y = -scale.y;
    megaMan.transform.localScale = scale;
    megaMan.canJump = !megaMan.canJump;
}
```

I structured the constructor like that to make running and creating the command as simple as possible. I want the command to encapsulate all of its own logic, and not need to be passed information to work. I structured the execute command to flip the scale and whether or not MegaMan can jump like shown in the picture so that this command could be used in combination with others. If Megaman is already inverted, I can execute another of these invert commands to flip him back over. This kind of structuring allows me to easily add new commands to run different or combined functionality when needed.

Next, for object pooling, I first had to create an ObjectPool class. Inside this class I have multiple methods,

```
public void InitializePool(GameObject objectPrefab)
{
    this.objectPrefab = objectPrefab;
    GameObject poolContainer = new GameObject($"{objectName} Pool Container");
    poolContainerTransform = poolContainer.transform;

    for (int i = 0; i < poolSize; i++)
    {
        GameObject instance = Instantiate(objectPrefab, poolContainerTransform);
        instance.name = $"{objectName} {i}";
        instance.SetActive(false);
        pool.Enqueue(instance);
    }
}
```

An InitiatlizePool method that takes in a base gameObject and allows me to easily populate the pool with copies of that gameObject.

```
1 reference
public GameObject GetObject()
{
    if (pool.Count > 0)
    {
        GameObject activeObject = pool.Dequeue();
        activeObject.SetActive(true);
        activePool.Add(activeObject);
        return activeObject;
    }
    else if (dynamicSize)
    {
        GameObject instance = Instantiate(objectPrefab, poolContainerTransform);
        instance.name = $"Extra {objectName}";
        activePool.Add(instance);
        return instance;
    }

    return null;
}
```

A GetObject method that easily allows me to return gameObjects out of the pool, and have the pool handle the activation.

```
1 reference
public void ReturnToPool(GameObject gameObject)
{
    activePool.Remove(gameObject);
    gameObject.SetActive(false);
    pool.Enqueue(gameObject);
}
```

And a ReturnToPool method that takes in an object removed from the pool and adds it back to the pool.

I designed the ObjectPool class this way to make it easy to use the pool in multiple situations. The pool essentially mimics how you would normally instantiate objects and then later destroy them. This allows me to replace the code where you do those actions with the ObjectPool methods and instantly get performance benefits. In terms of the performance benefits, here are some statistics from the profiler in Unity. In this example, I have upped the BlizzardAttack to happen every frame to show the effects of the performance.

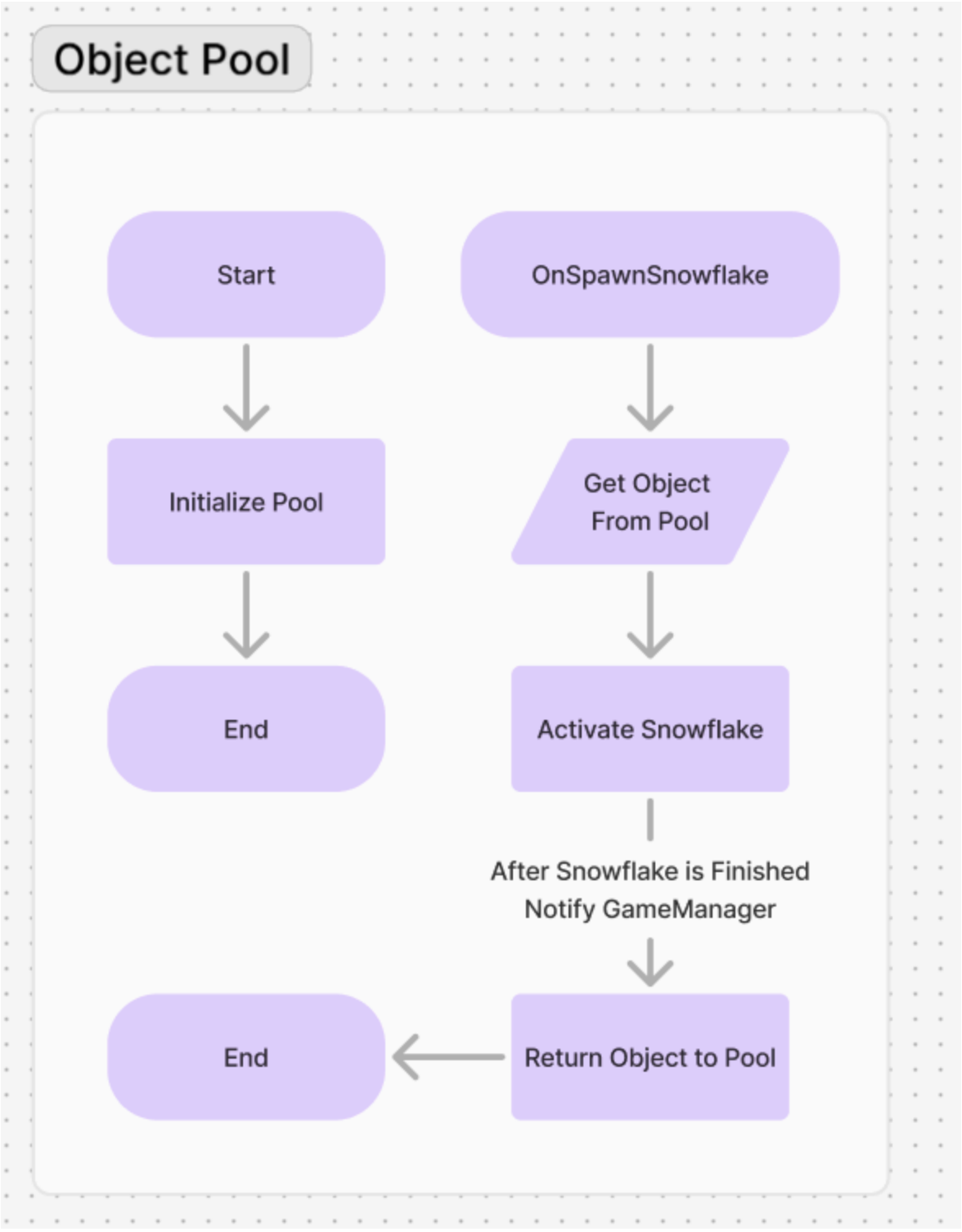
Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
EditorLoop	90.1%	90.1%	3	0 B	23.85	23.85
▼ PlayerLoop	9.1%	0.2%	3	1.4 KB	2.42	0.06
▶ RenderPipelineManager.DoRenderLoop_Internal() [Invoke]	6.4%	0.1%	1	368 B	1.70	0.03
▼ Update.ScriptRunBehaviourUpdate	1.2%	0.0%	1	1.0 KB	0.34	0.00
▼ BehaviourUpdate	1.2%	0.3%	1	1.0 KB	0.34	0.09
▼ BlizzardMan.Update() [Invoke]	0.7%	0.1%	1	1.0 KB	0.20	0.03
▶ Instantiate	0.6%	0.0%	3	432 B	0.17	0.00

As you can see in the picture, when 3 snowflakes are spawned each frame, it takes about 0.17 milliseconds. While this doesn't seem like much, as more and more snowflakes are spawned, they take up more and more time that you can't spend in other areas. The benefit with this method is that you don't have to spawn the projectiles in at the beginning, however I wouldn't consider that a benefit. I believe that because most games have loading screens anyways, so, why sacrifice milliseconds while the game is playing when you can sacrifice a bit more milliseconds during the loading screen to initialize an object pool.

Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
▼ PlayerLoop	61.0%	0.1%	2	0.6 KB	13.07	0.04
▶ FixedUpdate.Physics2DFixedUpdate	55.9%	0.0%	1	160 B	11.97	0.00
▼ Update.ScriptRunBehaviourUpdate	2.7%	0.0%	1	504 B	0.58	0.00
▼ BehaviourUpdate	2.7%	1.6%	1	504 B	0.58	0.34
Snowflake.Update() [Invoke]	0.7%	0.7%	1792	0 B	0.15	0.15
▼ BlizzardMan.Update() [Invoke]	0.3%	0.0%	1	504 B	0.06	0.01
▶ GameObject.Activate	0.2%	0.0%	3	0 B	0.04	0.00

Regardless, after implementing the object pool optimization, you can see that "spawning" 3 snowflakes per frame has decreased from 0.17 milliseconds to 0.04 milliseconds. That's a difference of 0.13 milliseconds. This clearly shows that the object pooling optimization has massively improved the performance of the Blizzard Attack.

Here is a flowchart of the ObjectPool optimization pattern,



Lastly, for my additional design pattern, I decided to implement the observer pattern with the Snowflake class and the GameManager class. I did this through the spawning of the snowflakes. Since the snowflakes come from an object pool, I have to get a reference to them

when they spawn so that I can add them back to the pool when they are finished performing their task. I could give each snowflake a reference to the pool, however that would lead to unnecessary coupling with the snowflakes. If I spawn a snowflake separately from the pool, I don't want it to assume it has a pool it can return itself to. So, to solve this, I created an event within the Snowflake class that uses the observer method to notify its observers of when it has finished its job.

```
public delegate void SnowflakeHandler(Snowflake snowflake);
public event SnowflakeHandler OnSnowflakeFinish;
```

This event is called when the snowflake collides with the player and runs the invert command, or when the snowflake expires due to its lifetime timer.

```
Unity Message | 0 references
private void OnTriggerEnter2D(Collider2D collision)
{
    if (collision.CompareTag("Player"))
    {
        ICommand command = new InvertMegaManCommand(MegaMan.Instance);
        GameManager.Instance.InvokeCommand(command);
        OnSnowflakeFinish?.Invoke(this);
    }
}
```

```
Unity Message | 0 references
private void Update()
{
    if (activated)
    {
        lifetimeTimer -= Time.deltaTime;
        if (lifetimeTimer <= 0)
        {
            OnSnowflakeFinish?.Invoke(this);
        }
    }
}
```

The GameManager class listens to this event when a Snowflake is spawned and returns the snowflake to the pool when it is notified.

```
1 reference
public void SpawnSnowflake()
{
    Vector2 randomPosition = new Vector2(Random.Range(lowerLeftBound.x, upperRightBound.x), Random.Range(lowerLeftBound.y, upperRightBound.y));
    GameObject snowflakeObject;
    if (disableObjectPoolOptimization)
    {
        snowflakeObject = Instantiate(snowflakePrefab);
    }
    else
    {
        snowflakeObject = snowflakePool.GetObject();
    }

    if (snowflakeObject != null)
    {
        snowflakeObject.transform.position = randomPosition;
        Snowflake snowflake = snowflakeObject.GetComponent<Snowflake>();
        snowflake.OnSnowflakeFinish += OnSnowflakeFinished;
        snowflake.Activate();
    }
}
```

```
2 references
public void OnSnowflakeFinished(Snowflake snowflake)
{
    snowflake.OnSnowflakeFinish -= OnSnowflakeFinished;
    if (disableObjectPoolOptimization)
    {
        Destroy(snowflake.gameObject);
        return;
    }

    snowflake.ResetSnowflake();
    snowflakePool.ReturnToPool(snowflake.gameObject);
}
```

I could have put this logic inside the BlizzardMan class, however, I don't want the object pool to be destroyed when the Blizzard man is defeated. By keeping this logic on the game manager, I decouple that relationship with the BlizzardMan class and allow anything to call the SpawnSnowflake methods. I also keep the relationship between the GameManager and Snowflake classes clean. By using the observer pattern, the GameManager doesn't have to be aware of exactly which snowflakes it has spawned into the world, because the snowflakes themselves will notify the GameManager.