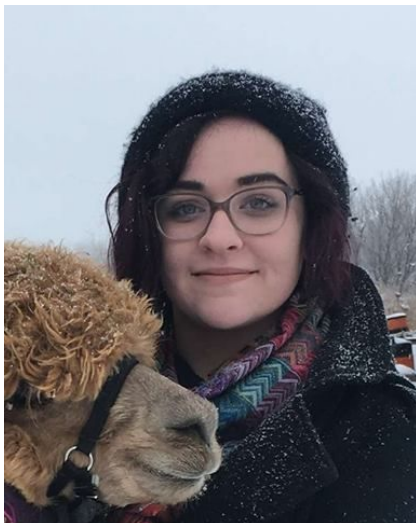
INFR 3110U – Game Engine Design & Implementation
Assignment 1 – Game Design Patterns

Max Kemp
100671306



Regan Simmonds
100651217

TABLE OF CONTENTS

BASE:

Building the Base Level Editor:

In this assignment we used our pre-existing knowledge along with algorithms and design patterns learned in class to create an aesthetically pleasing scene with a functional level editor. This level editor is able to spawn walls, cubes, spheres and enemy spawners, these are able to move around the scene and be placed and saved by the designer. There are also undo, redo and delete buttons to help the designer out in case of a mistake or a change in plans. The save and load buttons featured on this editor are straightforward and can be used to save the scene and then load it at a later time. As well, the play and stop buttons change it from designer mode to play mode as the game will begin and the player will be able to control their character and move it around the scene, shooting bullets at the enemies that automatically come towards the player from any enemy spawner they have placed in the scene. This editor uses many of the design patterns learned in class and the rest of this report will elaborate on such.

Basic Controls:

BUILD MODE:

- To spawn in an object you LEFT MOUSE CLICK the button on screen.
- To move the object around on the x and z axis you LEFT MOUSE CLICK on the object to make it your selected object and then HOLD THE LEFT MOUSE BUTTON DOWN AND DRAG to move it around. To move this object on the y axis, make sure it is selected and then HOLD THE LEFT MOUSE BUTTON DOWN AND SCROLL THE MOUSE WHEEL to move it up and down.
- To deselect the object you RIGHT MOUSE CLICK on the object and it will remove it from your current selected object.
- If no object is selected, you can move the overview camera around the scene, where SCROLL is to zoom in and out of the scene, RIGHT MOUSE CLICK AND HOLD is to pan left and right, and HOLDING DOWN THE SCROLL WHEEL will rotate the camera.

PLAY MODE:

- WASD is basic movement around the scene
- To shoot bullets at the enemies, LEFT MOUSE CLICK.
- To look around the scene simply MOVE MOUSE.

PART 1: DESIGN PATTERNS

Command Design Pattern:

We used this design pattern for the spawning of objects, undoing, redoing and deleting of objects. We used this design pattern very tightly with the factory design pattern. To go further into detail, the buttons mentioned above all have their own classes and when we click on said button we can just call the command based on which button was pressed. It

will then ask the factory which command the button asked for and will store it on the stack and immediately call the execute on that command.

As the command design pattern is used by creating a base class with an abstract function, refining said function in a subclass and then storing a pointer to command for each button, it makes sense why it would be so useful here.

As you can see, here is an example of the class for spawning the player. It's very empty and stays small as it just knows how to execute a command and to undo. These are both examples of the abstract function being created in the base class.

```
public class SpawnPlayerCommand : ICommand
{
    GameObject obj = null;

    public void Execute()
    {
        Factory.Instance.CreateGameObject(ObjectTypes.Player, out obj);
    }

    public void Undo()
    {
        Factory.Instance.DeleteGameObject(ref obj);
    }
}
```

Now compare this to the length of the undo function. It's not too much longer but it adds up incredibly if you were to have a large number of different spawn types. So having it only declared once and referenced by the base classes becomes very useful. This is the example of the redefined function, our undo function.

```
public void Undo()
{
    if (_commands.Count > 0)
    {
        // Perform last commands undo
        _currentCommand.Undo();
        _commands.Pop();
        _undoneCommands.Push(_currentCommand);

        // Set current command to the next command in the stack
        if (_commands.Count > 0)
            _currentCommand = _commands.Peek();
        else
            _currentCommand = null;
    }
    else
        Debug.Log("No more commands to undo.");
}
```

And finally the list of pointers that make it all work together.

```
// Request a command from the factory
switch (command)
{
    case "SpawnCube":
            _currentCommand = _factory.CreateCommand(CommandTypes.SpawnCube);
            break;
    case "SpawnSphere":
            _currentCommand = _factory.CreateCommand(CommandTypes.SpawnSphere);
            break;
    case "SpawnPlayer":
        _currentCommand = _factory.CreateCommand(CommandTypes.SpawnPlayer);
        break;
    case "SpawnPlane":
        _currentCommand = _factory.CreateCommand(CommandTypes.SpawnPlane);
        break;
    case "Delete":
        _currentCommand = _factory.CreateCommand(CommandTypes.Delete);
        break;
    case "SpawnWall":
        _currentCommand = _factory.CreateCommand(CommandTypes.SpawnWall);
        break;
    case "SpawnEnemySpawner":
        _currentCommand = _factory.CreateCommand(CommandTypes.SpawnEnemySpawner);
        break;
    default:
        return;
}

// Execute the current command
_currentCommand.Execute();
_commands.Push(_currentCommand);
```

Factory Design Pattern:

We used the factory design pattern for spawning in the objects in the scene and deleting them. As well it also handles the creation of new commands, which makes it tightly used with the command design pattern. Knowing the factory design pattern it makes sense that we would use it here, in our initialization of objects. It hides the complex instructions and simply allows those with less experience in coding to have an easier time getting what they want.

This part of the process is where each object becomes unique. The factory will know all the prefabs for each object and when we ask for a specific object like a cube, it will

```
_cube = Resources.Load("Prefabs/Cube");
_sphere = Resources.Load("Prefabs/Sphere");
_player = Resources.Load("Prefabs/Player");
_plane = Resources.Load("Prefabs/Plane");
_enemy = Resources.Load("Prefabs/Enemy");
_wall = Resources.Load("Prefabs/Wall");
_enemySpawner = Resources.Load("Prefabs/EnemySpawner");
_zombie = Resources.Load("Prefabs/Zombie");
```

automatically instantiate to that prefab specific and will give us the perfectly created cube.

```
case ObjectTypes.Cube:
    obj = Instantiate(_cube, Vector3.zero, Quaternion.identity, null) as GameObject;
    obj.GetComponent<ObjectType>().Type = ObjectTypes.Cube;
    break;
```

This factory will also handle the creation of new commands which are very useful for the command design pattern.

```
public ICommand CreateCommand(CommandTypes commandType)
{
    ICommand command = null;

    switch (commandType)
    {
        case CommandTypes.SpawnCube:
            command = new SpawnCubeCommand();
            break;
        case CommandTypes.SpawnSphere:
            command = new SpawnSphereCommand();
            break;
        case CommandTypes.SpawnPlayer:
            command = new SpawnPlayerCommand();
            break;
        case CommandTypes.SpawnPlane:
            command = new SpawnPlaneCommand();
            break;
        case CommandTypes.SpawnEnemy:
            command = new SpawnEnemyCommand();
            break;
        case CommandTypes.Delete:
            command = new DeleteCommand();
            break;
        case CommandTypes.SpawnWall:
            command = new SpawnWallCommand();
            break;
        case CommandTypes.SpawnEnemySpawner:
            command = new SpawnEnemySpawnerCommand();
            break;
    }

    return command;
}
```

Singleton Design Pattern:

The singleton design pattern helps us in allowing us to easily access the factory (or other things) in any class. This design pattern restricts the ability to a secondary instance if there is already one existing. If it doesn't exist it will allow one to be made but it is otherwise impossible to create a second one.

```
public static CommandHub Instance
{
    get
    {
        if (!_instance)
        {
            GameObject obj = new GameObject("Command Hub");
            _instance = obj.AddComponent<CommandHub>();
        }

        return _instance;
    }
}

private void Awake()
{
    if (!_instance && (!_instance == this))
        _instance = this;
}

// Start is called before the first frame update
void Start()
{
    _factory = Factory.Instance;
}
```

State Design Pattern:

The state design pattern was exceptionally useful in this code as we were trying to go between the different states in our game (playing, building) and were finding it just getting too clogged up with if statements. So we used this design pattern to have 3 different states and for each state there is a seperate class. Each of these classes will implement a certain state for the game and will allow specific things exclusively to that state. Our states are the play state, move state and placing/building state. An example would be that the left clicking in the play state shoot bullets but in the build state it selects objects.The three main functions we used in these states are entry, input and update. Entry is for anything to do with set up. Specifically it could be for things like to set up the initial camera alignment.

```
public void entry(CameraControl camControl)
{
    Quaternion rot = new Quaternion();
    rot.eulerAngles = new Vector3(35.0f, 0.0f, 0.0f);
    _cam.transform.rotation = rot;

    _playerTrans = GameObject.FindGameObjectWithTag("Player").transform;
    _cam.transform.position = _playerTrans.position + camControl.CamOffset;
}
```

Input would be for anything to do with input and making sure that if you do something that changes the state of the game it knows either if nothing has changed or the new state that you are changing to. Specifically it could be checking to see if you have clicked an object or not.

```
public ICameraState input(CameraControl camControl)
{
    // Left click checks to see if you clicked on an object.
    if (Input.GetMouseButtonDown(0))
    {
        Ray mousePos = Camera.main.ScreenPointToRay(Input.mousePosition);
        RaycastHit rayHit;

        if (Physics.Raycast(mousePos.origin, mousePos.direction, out rayHit, Mathf.Infinity))
        {
            camControl.SelectedObj = rayHit.collider.gameObject;
            Debug.Log("Selected: " + camControl.SelectedObj.name);

            return CameraControl.ObjectSelectedState;
        }
    }
```

Update is for anything you want to do, like the controls to move the camera or to get the camera to follow the player.

```
public void update(CameraControl camControl)
{
    // Move the camera around.
    float horizontal = Input.GetAxis("Mouse X");
    float vertical = Input.GetAxis("Mouse Y");
    Transform transform = camControl.GetComponent<Transform>();

    // Right click moves the camera left/right, up/down.
    if (Input.GetMouseButton(1))
    {
        transform.position += transform.right * horizontal;
        transform.position += transform.up * vertical;
    }

    // Moving the scroll wheel moves the camera along the world z-axis.
    transform.position += transform.forward * Input.GetAxis("Mouse ScrollWheel") * 3.0f;

    // Middle mouse button rotates the camera.
    if (Input.GetMouseButton(2))
    {
        transform.Rotate(new Vector3(2.0f * vertical, 0.0f, 0.0f));
        transform.Rotate(new Vector3(0.0f, 2.0f * horizontal, 0.0f));
    }
}
```

Prototype Design Pattern:

We used the prototype design pattern in the enemy spawner class which reduces the need for multiple enemy classes for each type of enemy. As you can see the enemy class is abstract and simply just knows how to clone.

```
public abstract class Enemy : MonoBehaviour
{
    public abstract GameObject clone();
}
```

This clone ability is extremely useful and allows us to create a blueprint of sorts by being able to pass in any enemy type (bunny, zombie, etc) and have it clone so it knows how to use this different enemy type.

```
obj = enemy.clone();
obj.transform.position = transform.position;
++_numEnemies;
```

PART 2: UML DIAGRAM

Will be included as a picture in the same folder as the report

CONTRIBUTIONS

Max

- Implementing the different design patterns
- Creating movement and gameplay
- Creating the ability to spawn objects, delete, undo, redo

Regan

- Use of the DLL
- Saving and loading
- Writing the report

REFERENCES

Game Programming Patterns by Robert Nystrom
https://gameprogrammingpatterns.com/contents.html
3D Character - White Rabbit - Unity Asset Store
https://assetstore.unity.com/packages/3d/characters/animals/white-rabbit-138709
NavMeshComponents - Navigation for Enemies - Unity
https://github.com/Unity-Technologies/NavMeshComponents