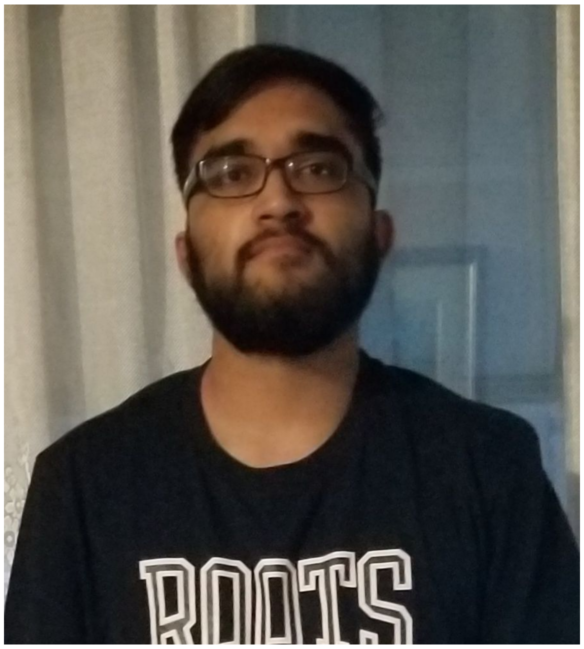
INFR 3110U - Game Engine Design & Implementation Assignment 1 - Game Design Patterns Report



Abisheik Thuraiyan
100653409

Mark-Anthony Meritt
100692252

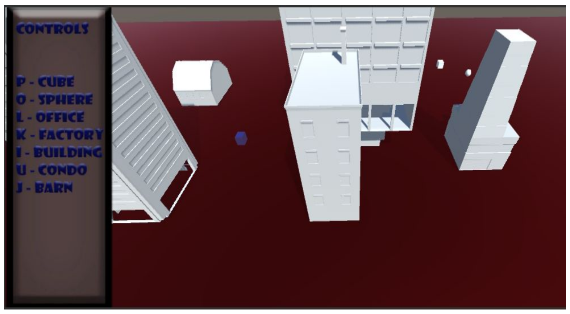
Table of Contents

I. Introduction to our project

For this group assignment, we decided to make a simplistic level editor in Unity that allows the user to place various buildings and primitives around a plane. In this level editor, the movement of the player is confined to only the X and Z axis. The controls for the level editor are also conveniently shown on the left side of the screen for the user to see which key spawns in which object. Upon pressing the correct key, objects that were placed using the level editor will have box colliders applied to them, giving the users a chance to further interact with the level they have created.

Controls:

- P: Place Cube
- O: Place Sphere
- L: Place Office Building
- K: Place Factory Building
- I: Place Standard Building
- U: Place Condominium
- J: Place Barn
- Backspace: 'Play' the level (Activates bounding boxes and deactivates the editor commands)
- Num. 1: Save Player Cube Position
- Num. 2: Load Player Cube Position
- Num. 3: Save Cube Positions
- Num. 4: Load Cube Positions

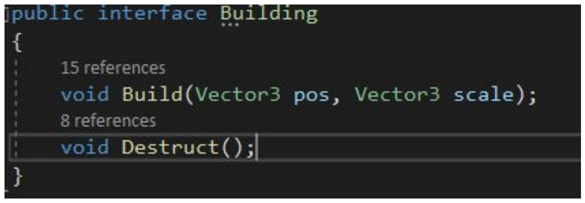


II. Design Patterns

We were required to implement three design patterns for this assignment: the Factory Design Pattern, the Command Design Pattern, and one other design pattern of our choice (we decided to implement the Decorator Design Pattern). The Command and Factory Design Patterns were implemented using C# Scripts, while the Decorator Design Pattern was implemented using a mix of a C++ DLL, and a C# Script.

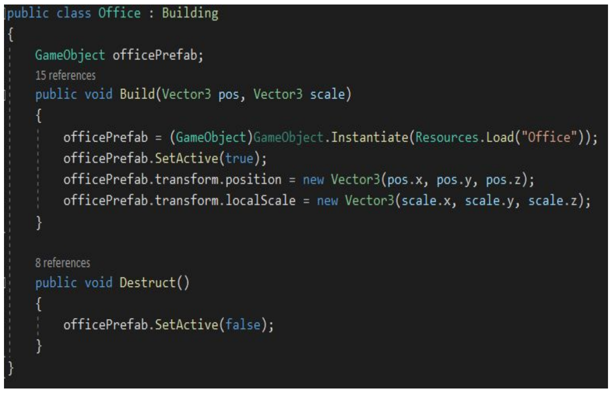
II-A. Factory Design Pattern

It made the most sense for us to use the Factory Design Pattern to create and place objects in the scene. We started off by making an interface as a base to work off of, which included a 'Build()' and 'Destruct()' function. These functions would then be overwritten through the child classes that would be created later on.



```
public interface Building
{
    15 references
    void Build(Vector3 pos, Vector3 scale);
    8 references
    void Destruct();
}
```

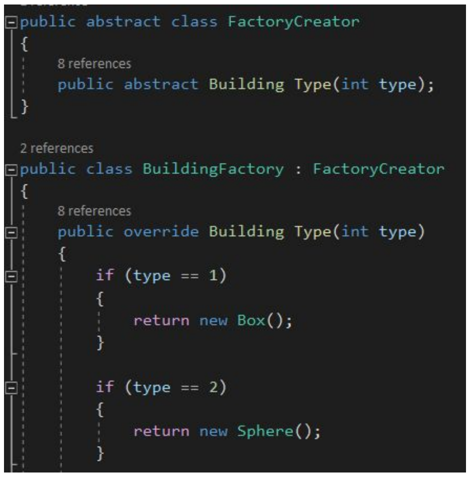
When creating the child classes to this interface, the void functions that were created in the interface would be set up with the appropriate calculations that would need to be made in order to spawn the correct object into the scene. Below is a screenshot of our child class for the office building:



```
public class Office : Building
{
    GameObject officePrefab;
    15 references
    public void Build(Vector3 pos, Vector3 scale)
    {
        officePrefab = (GameObject)GameObject.Instantiate(Resources.Load("Office"));
        officePrefab.SetActive(true);
        officePrefab.transform.position = new Vector3(pos.x, pos.y, pos.z);
        officePrefab.transform.localScale = new Vector3(scale.x, scale.y, scale.z);
    }

    8 references
    public void Destruct()
    {
        officePrefab.SetActive(false);
    }
}
```

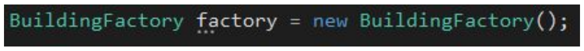
This set of code would look rather similar across all the child classes (one for each type of building / primitive). The process would go as follows: *A GameObject is created for that specific child class -> the Build() function would then take two Vector3 variables (one for position and scale) -> the GameObject would then be set equal to the appropriate prefab in the Resources folder depending on the building/primitive type -> the object would then be set to active -> the position and scale of the object would be set according to the variables that were passed on to the function. (The Destruct() function would simply deactivate the object, which comes in handy for the Command Design Pattern implementation.)*

Once all of the child classes have been set up, an abstract class is then created to essentially act as a placeholder for the Type() function, which will be set up in a child class of its own. The 'factory' part comes into play when creating the child class that sets up the Type() functions, the function in this class allows the program to simply choose which object to build when executing the Build() function, depending on the integer that is passed on to the Type() function. For example, passing the integer '1' to the Type() function would set the Build() function to build a cube when executed, this is done by returning a new object of the specific child class (building/primitive) that we want.

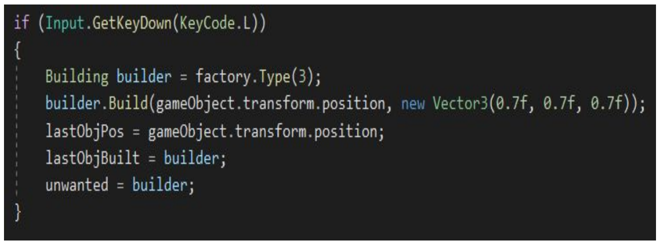


```
public abstract class FactoryCreator
{
    8 references
    public abstract Building Type(int type);
}

2 references
public class BuildingFactory : FactoryCreator
{
    8 references
    public override Building Type(int type)
    {
        if (type == 1)
        {
            return new Box();
        }

        if (type == 2)
        {
            return new Sphere();
        }
```

Once the 'factory' has been set up, all that is required to do in order to get the appropriate object to build is to simply create a BuildingFactory object and then create a Building object. When the key is pressed, the Type() function is called using the BuildingFactory object (while also passing the appropriate integer to create the building/primitive type that we want), and the position and scale of the object is set using the Build() function.



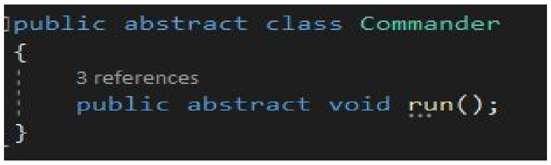
```
BuildingFactory factory = new BuildingFactory();
```



```
if (Input.GetKeyDown(KeyCode.L))
{
    Building builder = factory.Type(3);
    builder.Build(gameObject.transform.position, new Vector3(0.7f, 0.7f, 0.7f));
    lastObjPos = gameObject.transform.position;
    lastObjBuilt = builder;
    unwanted = builder;
}
```

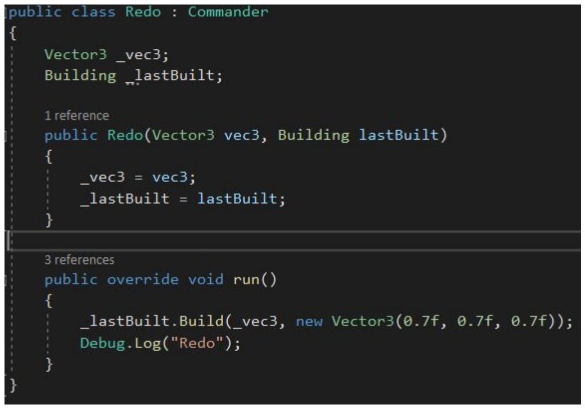
II-B. Command Design Pattern

As per the suggestion of the assignment's outline, we decided to use the Command Design Pattern to let the user undo or redo object placements in the scene.
Somewhat similarly to the Factory Design Pattern that we implemented, we start implementing the Command Design Pattern by creating a public abstract class with a run() function, which will be set up through two child classes (one for redo and one for undo).

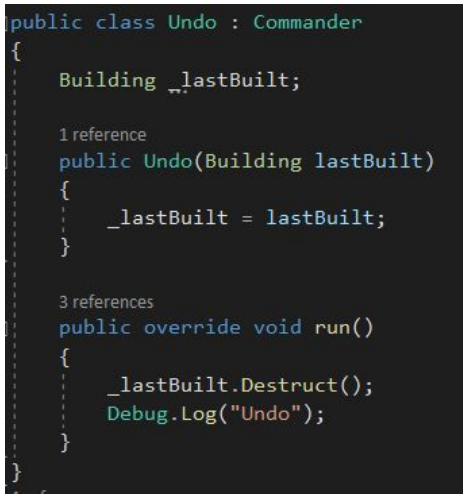


```
public abstract class Commander
{
    3 references
    public abstract void run();
}
```

When creating the child classes for redo and undo, we inherit the Commander abstract class in order to set up the run() function appropriately. In the case of the redo class, we would first need to set the class constructor to take in a Vector3 and Building parameter in order for the run() function to figure out which object was the last one to be placed and where it was last placed. The run() function would then simply use the Build() function that was created during the Factory Design Pattern process in order to respawn the object into the scene. The process is similar for the undo class, except that its constructor only takes in a Building object as a parameter and simply uses the Destruct() function from the Factory Design Pattern when the run() function is executed.

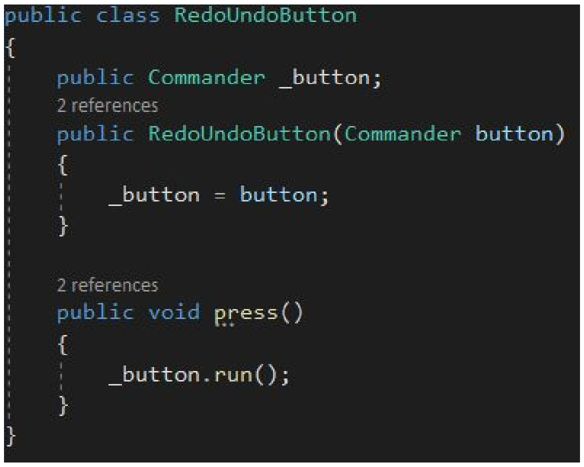


```
public class Redo : Commander
{
    Vector3 _vec3;
    Building _lastBuilt;

    1 reference
    public Redo(Vector3 vec3, Building lastBuilt)
    {
        _vec3 = vec3;
        _lastBuilt = lastBuilt;
    }

    3 references
    public override void run()
    {
        _lastBuilt.Build(_vec3, new Vector3(0.7f, 0.7f, 0.7f));
        Debug.Log("Redo");
    }
}
```



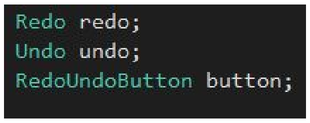
```
public class Undo : Commander
{
    Building _lastBuilt;

    1 reference
    public Undo(Building lastBuilt)
    {
        _lastBuilt = lastBuilt;
    }

    3 references
    public override void run()
    {
        _lastBuilt.Destruct();
        Debug.Log("Undo");
    }
}
```

Once the child classes have been set up, a RedoUndoButton class is then created, which will act as a button that takes in a Commander object as a parameter (this is done in order for the RedoUndoButton object to recognize which run() function to execute). The main portion of the RedoUndoButton class is the press() function, which just executes the run() function from the Commander object that was passed on to the object's constructor.

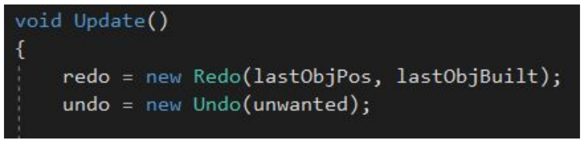


```
public class RedoUndoButton
{
    public Commander _button;
    2 references
    public RedoUndoButton(Commander button)
    {
        _button = button;
    }

    2 references
    public void press()
    {
        _button.run();
    }
}
```

To use the redo and undo commands in the program, we create a redo,undo, and RedoUndoButton object. The redo object will take in the last object that was placed and its position as parameters, while the undo button will take in the last object that was placed as a parameter.

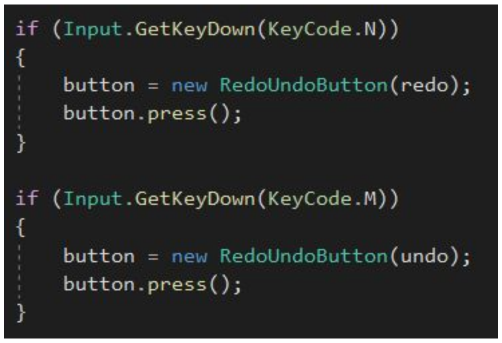


```
Redo redo;
Undo undo;
RedoUndoButton button;
```



```
void Update()
{
    redo = new Redo(lastObjPos, lastObjBuilt);
    undo = new Undo(unwanted);
```

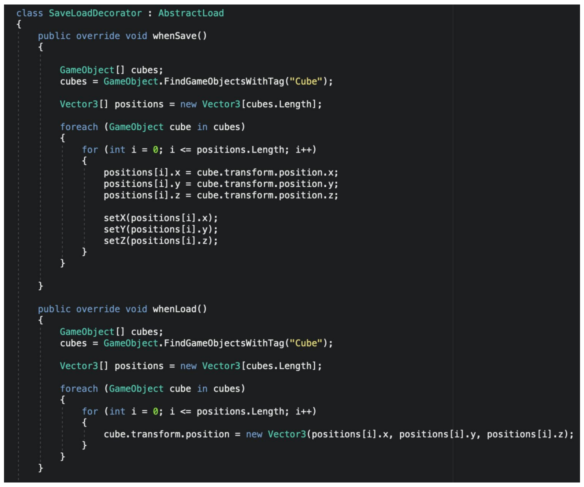
When pressing the key for redo or undo, the RedoUndoObject will then have either the redo or undo object passed onto to it (depending on the command that we want), and will then execute the press() command.



```
if (Input.GetKeyDown(KeyCode.N))
{
    button = new RedoUndoButton(redo);
    button.press();
}

if (Input.GetKeyDown(KeyCode.M))
{
    button = new RedoUndoButton(undo);
    button.press();
}
```

II-C. Decorator Design Pattern

The decorator design pattern allows you to add behaviour to an existing object. In our case, I added additional functionality to our save/load DLL script. The SaveLoad script allows you to save position of the original cube in the scene. SaveLoadDecorator allows you to save position of every cube generated within the scene, and load them.



```
class SaveLoadDecorator : AbstractLoad
{
    public override void whenSave()
    {

        GameObject[] cubes;
        cubes = GameObject.FindGameObjectsWithTag("Cube");

        Vector3[] positions = new Vector3[cubes.Length];

        foreach (GameObject cube in cubes)
        {
            for (int i = 0; i <= positions.Length; i++)
            {
                positions[i].x = cube.transform.position.x;
                positions[i].y = cube.transform.position.y;
                positions[i].z = cube.transform.position.z;

                setX(positions[i].x);
                setY(positions[i].y);
                setZ(positions[i].z);
            }
        }

    }

    public override void whenLoad()
    {
        GameObject[] cubes;
        cubes = GameObject.FindGameObjectsWithTag("Cube");

        Vector3[] positions = new Vector3[cubes.Length];

        foreach (GameObject cube in cubes)
        {
            for (int i = 0; i <= positions.Length; i++)
            {
                cube.transform.position = new Vector3(positions[i].x, positions[i].y, positions[i].z);
            }
        }
    }
```

When you press 1 and 2, the regular SaveLoad functionality is triggered. Pressing 3 and 4 will search for all cubes within the scene and save and load their positions.

III. Sources

Various sources were used in order to get a clearer understanding of the various design patterns and how they can be implemented to benefit our program. This includes videos, code, articles, and powerpoint slides. Code that was referenced from other sources were merely used as examples to look at and have been built upon significantly in order to correctly work with our program and have never been copied. It should also be noted that all non-primitive 3D assets and UI images were not taken from other sources and were made from scratch by members of this group.

Below are the links (if applicable) and the names of the various sources that were referenced for the purpose of this assignment.

- *Factory Method Design Pattern C# -* https://www.dotnettricks.com/learn/designpatterns/factory-method-design-pattern-dotnet
- *Factory Design Pattern Tutorial -* http://www.newthinktank.com/2012/09/factory-design-pattern-tutorial/
- *Design Patterns | Set 2 (Factory Method) -* https://www.geeksforgeeks.org/design-patterns-set-2-factory-method/
- *Fusion - Factory Method Presentation (Slides on Blackboard)*
- *Command in C++ -* https://sourcemaking.com/design_patterns/command/cpp/2
- *Game Programming Patterns in Unity with C# -* https://www.habrador.com/tutorials/programming-patterns/1-command-pattern/
- *Command -* https://refactoring.guru/design-patterns/command
- *Design Patterns in Unity -* https://github.com/Naphier/unity-design-patterns