Team 2 Studios Unity Coding Standards V1.1

The following document will lay out a consistent unity C# coding style for all Team 2 studio projects. This will better improve consistency, clarity and ease the process of debugging and collaborations. Our standards use Microsoft's C# conventions, Googles C++ style guide, and common formatting already implemented by our team.

1. Naming Conventions

Follow common C# naming conventions including

- PascalCase (Classes, Methods, Properties)
- Private fields: camelCase using a leading underscore (_health)
- Constants: all capitals with underscores (USER_LEVEL)
- Serialized private fields preferred but not required ([SerializeField] private int _spawn)
- File names match class names

Good

```
public class LazerWeapon : WeaponBase
{
   [Header("Lazer Settings")]
   [SerializeField] private float _range = 50f;
   [SerializeField] private LayerMask _hitMask;
   [SerializeField] private LineRenderer _line;
   [SerializeField] private float _lineTime = 0.05f;

         protected override bool DoFire(Vector3 origin, Vector3 direction)
         {
            Vector3 end = origin + direction.normalized * _range;
            if (Physics.Raycast(origin, direction.normalized, out RaycastHit hit, _range,
                _hitMask))
         }
   end = hit.point;
   var health = hit.collider.GetComponent<Health>();
   if (health != null)
      health.ApplyDamage(_damage);

if (_line != null)
   StartCoroutine(FlashLine(origin, end));
…
```

2. Commenting and documentation

Ensure documentation through out any scripts to ensure readability and an easier understanding for anyone looking at your work.

- Comment Blocks for Methods
- use inline comments sparingly
- No obvious comments like //call start
- TODO and FIXME tags

Good

```
 // Split the map into hexagons, and get the heights of each hexagon
float[,] terrainHeights = new float[terrainWidth, terrainDepth];
         for(int i = 0; i < terrainWidth; i++)
```

```
/* Fires a laser forward from the players
-Damage is applied on hit
- Shows line effect for bullet path
*/
protected override bool DoFire(Vector3 origin, Vector3 direction)

// TODO: finish creating main spawning feature
// FIXME: Weapon collision broke when implementing certain power up
```

3. Formatting & Structure

The following will lay out how code should be structured including spaces, new lines, and other things to improve readability and cleanliness.

- Indentations: 4 spaces
- Braces on new lines
- one statement per line
- Keep lines under ~100 characters
- breaking statements should be indented to match

Good

```
private void OnTriggerEnter(Collider other)
{
    if (other.CompareTag("Player"))
{
    PlayerScore playerScore = other.GetComponent<PlayerScore>();

if (playerScore != null)
{
    playerScore.AddScore(_scoreValue);
    Destroy(gameObject);
}
}
}
ExampleOfBreaking(FirstParameterToPass, SecondParameterToPass, ThirdParameterToPass,
                  FourthParameterToPass, FinalParamater);

//ObjectSpawner
 void Awake()
   {
       Transform visual = transform.Find("Visual");
       if (visual)
           Destroy(visual.gameObject);

       foreach (var obj in _toSpawn)
       {
           if (obj != null && SpecificTest())
               _validSpawn.Add(obj);
       }
```

4. Error Handling

Proper error handling will protect our game from crashing or behaving in unexpected ways when something goes wrong. It ensures stability by preventing null reference exceptions, allowing the game to fail instead of fully breaking, and helps in debugging when there is meaningful error messages

- Null Check all external references
- Count Checks for IndexOutofRange
- Guard from disabled or deleted objects

Good

```
if (_weaponModel == null)
{
    Debug.LogError("Weapon has no visual model assigned!", this);
    enabled = false;
    return;
}

if (_items.Count == 0)
{
   Debug.LogWarning("No items available to spawn!");
   return;
}

if (_enemy == null)
{
   Debug.LogWarning("Enemy destroyed before attack could finish.");
   return;
}
```

5. Git Collaboration

In order to maintain a clean git project ensure to follow these rules

- Features on a separate branch before merge
- Commit only compiling code
- Clear commit messages explaining whats new.