**GridChangeInformation.cs:**

// --------------------------------------------------------------------------------------------------------------------

// <copyright file="GridChangeInformation.cs" company="Metallic Clashers">

// CLash On

// </copyright>

// <summary>

// Defines the TypeChangeInformation type.

// </summary>

// --------------------------------------------------------------------------------------------------------------------

namespace Combat.Board.Information

{

using System.Collections.Generic;

public class GridChangeInformation

{

public List<Gem> gems;

}

}

**GridResizeInformation.cs:**

namespace Combat.Board.Information

{

using UnityEngine;

public class GridResizeInformation

{

public Rect newRect;

}

}

**MatchInformation.cs:**

// --------------------------------------------------------------------------------------------------------------------

// <copyright file="MatchInformation.cs" company="Metallic Clashers">

// CLash On

// </copyright>

// <summary>

// Defines the TypeChangeInformation type.

// </summary>

// --------------------------------------------------------------------------------------------------------------------

namespace Combat.Board.Information

{

using System.Collections.Generic;

public class MatchInformation

{

public GemType type;

public List<Gem> gems = new List<Gem>();

public List<GridCollection> gridCollections = new List<GridCollection>();

}

}

**PositionChangeInformation.cs:**

// ---------------------------------------------------------------------------------------------------------

// <copyright file="PositionChangeInformation.cs" company="Metallic Clashers">

// CLash On

// </copyright>

// <summary>

// Defines the TypeChangeInformation type.

// </summary>

// ---------------------------------------------------------------------------------------------------------

namespace Combat.Board.Information

{

using UnityEngine;

public class PositionChangeInformation

{

public Gem gem;

public Vector2 newPosition;

}

}

**SlideInformation.cs:**

namespace Combat.Board.Information

{

public class SlideInformation

{

public GridCollection gridCollection;

}

}

**TypeChangeInformation.cs:**

// ---------------------------------------------------------------------------------------------------------

// <copyright file="TypeChangeInformation.cs" company="Metallic Clashers">

// CLash On

// </copyright>

// <summary>

// Defines the TypeChangeInformation type.

// </summary>

// ---------------------------------------------------------------------------------------------------------

namespace Combat.Board.Information

{

public class TypeChangeInformation

{

public Gem gem;

public GemType newType;

}

}

**Gem.cs:**

namespace Combat.Board

{

using System;

using System.Collections.Generic;

using Information;

using JetBrains.Annotations;

using UnityEngine;

using UnityEngine.Events;

public enum GemType

{

Red,

Orange,

Yellow,

Green,

Blue,

Purple,

}

public enum Direction

{

Up,

Down,

Left,

Right,

}

[Serializable]

public class Gem : IAttachable

{

[Serializable]

public class PositionChangeEvent : UnityEvent<PositionChangeInformation> { }

[Serializable]

public class TypeChangeEvent : UnityEvent<TypeChangeInformation> { }

[Serializable]

public class CreateGemEvent : UnityEvent<Gem> { }

#region SERIALIZED\_FIELDS

[SerializeField]

private GemType m\_GemType;

[SerializeField]

private Vector2 m\_Position;

[SerializeField]

private TypeChangeEvent m\_OnTypeChange = new TypeChangeEvent();

[SerializeField]

private PositionChangeEvent m\_OnPositionChange = new PositionChangeEvent();

#endregion

private readonly List<IComponent> m\_Components = new List<IComponent>();

public static readonly CreateGemEvent onCreate = new CreateGemEvent();

#region PUBLIC\_PROPERTIES

public Grid grid { get; private set; }

public GemType gemType

{

get { return m\_GemType; }

set

{

m\_GemType = value;

m\_OnTypeChange.Invoke(new TypeChangeInformation { gem = this, newType = value });

}

}

public Vector2 position

{

get { return m\_Position; }

set

{

m\_Position = value;

m\_OnPositionChange.Invoke(new PositionChangeInformation { gem = this, newPosition = value });

}

}

public TypeChangeEvent onTypeChange { get { return m\_OnTypeChange; } }

public PositionChangeEvent onPositionChange { get { return m\_OnPositionChange; } }

public List<IComponent> components { get { return m\_Components; } }

public Row row { get { return grid.rows[(int)position.y]; } }

public Column column { get { return grid.columns[(int)position.x]; } }

public Gem up { get { return GetNeighbor(Direction.Up); } }

public Gem down { get { return GetNeighbor(Direction.Down); } }

public Gem left { get { return GetNeighbor(Direction.Left); } }

public Gem right { get { return GetNeighbor(Direction.Right); } }

#endregion

private Gem() { }

public Gem(Grid newGrid, Vector2 newPosition, GemType newGemType) : this()

{

grid = newGrid;

m\_Position = newPosition;

m\_GemType = newGemType;

onCreate.Invoke(this);

}

[CanBeNull]

public Gem GetNeighbor(Direction direction, bool clamp = false)

{

var nextPosition = m\_Position;

switch (direction)

{

case Direction.Up:

nextPosition += Vector2.up;

break;

case Direction.Down:

nextPosition += Vector2.down;

break;

case Direction.Left:

nextPosition += Vector2.left;

break;

case Direction.Right:

nextPosition += Vector2.right;

break;

default:

throw new ArgumentOutOfRangeException("direction", direction, null);

}

if (clamp)

{

nextPosition = grid.ClampPosition(nextPosition);

return grid[(int)nextPosition.y][(int)nextPosition.x];

}

return

nextPosition == grid.ClampPosition(nextPosition) ?

grid[(int)nextPosition.y][(int)nextPosition.x] : null;

}

}

public static class DirectionExtentions

{

public static Direction Reverse(this Direction direction)

{

switch (direction)

{

case Direction.Up:

return Direction.Down;

case Direction.Down:

return Direction.Up;

case Direction.Left:

return Direction.Right;

case Direction.Right:

return Direction.Left;

default:

throw new ArgumentOutOfRangeException("direction", direction, null);

}

}

}

}

**GemImage.cs:**

namespace Combat.Board

{

using System;

using Information;

using UnityEngine;

using UnityEngine.UI;

public class GemImage : Image, IComponent

{

public enum ImageType

{

Background,

Midground,

Foreground

}

[SerializeField]

private ImageType m\_ImageType;

[SerializeField]

private Gem m\_Gem;

public ImageType imageType { get { return m\_ImageType; } }

public Gem gem { get { return m\_Gem; } set { m\_Gem = value; } }

protected override void Start()

{

base.Start();

gem.onTypeChange.AddListener(OnTypeChange);

CombatManager.self.onCombatModeChange.AddListener(UpdateSprite);

CombatManager.self.combatUiInformation.onUseAlternativeColorsChange.AddListener(

OnUseAlternativeColorsChange);

UpdateSprite();

SetColor(CombatManager.self.combatUiInformation.gemColors[(int)gem.gemType]);

}

private void SetColor(Color newColor)

{

switch (m\_ImageType)

{

case ImageType.Background:

color = CombatManager.self.combatMode == CombatManager.CombatMode.Attack ? newColor : Color.white;

break;

case ImageType.Midground:

color =

CombatManager.self.combatMode == CombatManager.CombatMode.Attack

? new Color(62f / 255f, 62f / 255f, 62f / 255f)

: newColor;

break;

case ImageType.Foreground:

color = newColor;

break;

default:

throw new ArgumentOutOfRangeException();

}

}

private void UpdateSprite()

{

var currentUiInformation = CombatManager.self.combatUiInformation.currentModeUiInformation;

switch (m\_ImageType)

{

case ImageType.Background:

sprite = currentUiInformation.backgroundImage;

break;

case ImageType.Midground:

sprite = currentUiInformation.midgroundImage;

break;

case ImageType.Foreground:

sprite = currentUiInformation.foregroundImage;

break;

default:

throw new ArgumentOutOfRangeException();

}

SetColor(CombatManager.self.combatUiInformation.gemColors[(int)gem.gemType]);

}

private void OnTypeChange(TypeChangeInformation typeChangeInformation)

{

SetColor(CombatManager.self.combatUiInformation.gemColors[(int)typeChangeInformation.newType]);

}

private void OnUseAlternativeColorsChange(bool newValue)

{

SetColor(CombatManager.self.combatUiInformation.gemColors[(int)gem.gemType]);

}

}

}

**GemMono.cs:**

namespace Combat.Board

{

using System;

using Information;

using UnityEngine;

using UnityEngine.UI;

public class GemImage : Image, IComponent

{

public enum ImageType

{

Background,

Midground,

Foreground

}

[SerializeField]

private ImageType m\_ImageType;

[SerializeField]

private Gem m\_Gem;

public ImageType imageType { get { return m\_ImageType; } }

public Gem gem { get { return m\_Gem; } set { m\_Gem = value; } }

protected override void Start()

{

base.Start();

gem.onTypeChange.AddListener(OnTypeChange);

CombatManager.self.onCombatModeChange.AddListener(UpdateSprite);

CombatManager.self.combatUiInformation.onUseAlternativeColorsChange.AddListener(

OnUseAlternativeColorsChange);

UpdateSprite();

SetColor(CombatManager.self.combatUiInformation.gemColors[(int)gem.gemType]);

}

private void SetColor(Color newColor)

{

switch (m\_ImageType)

{

case ImageType.Background:

color = CombatManager.self.combatMode == CombatManager.CombatMode.Attack ? newColor : Color.white;

break;

case ImageType.Midground:

color =

CombatManager.self.combatMode == CombatManager.CombatMode.Attack

? new Color(62f / 255f, 62f / 255f, 62f / 255f)

: newColor;

break;

case ImageType.Foreground:

color = newColor;

break;

default:

throw new ArgumentOutOfRangeException();

}

}

private void UpdateSprite()

{

var currentUiInformation = CombatManager.self.combatUiInformation.currentModeUiInformation;

switch (m\_ImageType)

{

case ImageType.Background:

sprite = currentUiInformation.backgroundImage;

break;

case ImageType.Midground:

sprite = currentUiInformation.midgroundImage;

break;

case ImageType.Foreground:

sprite = currentUiInformation.foregroundImage;

break;

default:

throw new ArgumentOutOfRangeException();

}

SetColor(CombatManager.self.combatUiInformation.gemColors[(int)gem.gemType]);

}

private void OnTypeChange(TypeChangeInformation typeChangeInformation)

{

SetColor(CombatManager.self.combatUiInformation.gemColors[(int)typeChangeInformation.newType]);

}

private void OnUseAlternativeColorsChange(bool newValue)

{

SetColor(CombatManager.self.combatUiInformation.gemColors[(int)gem.gemType]);

}

}

}

**GemMonoDuplicate.cs:**

namespace Combat.Board

{

using UnityEngine;

public class GemMonoDuplicate : GemMono

{

// If I don't do this Unity will call GemMono.Awake()

private void Awake() { }

protected override void UpdateTransformPosition()

{

var spacing = gridMono.CalculateSpacing();

var nextPosition =

grid.ClampPosition(

position + rowMono.currentDirection + columnMono.currentDirection);

var offsetPosition = nextPosition - rowMono.currentDirection - columnMono.currentDirection;

offsetPosition =

new Vector2(offsetPosition.x \* spacing.x, offsetPosition.y \* spacing.y);

m\_RectTransform.anchoredPosition =

offsetPosition + rowMono.positionOffset + columnMono.positionOffset;

m\_PositionIsDirty = false;

}

public new static void Init()

{

Gem.onCreate.AddListener(OnCreateGem);

}

private static void OnCreateGem(Gem newGem)

{

var newGemMonoDuplicate = CreateBaseGameObject<GemMonoDuplicate>(newGem);

SubscribeToEvents(newGemMonoDuplicate);

newGemMonoDuplicate.gameObject.SetActive(false);

}

}

}

**Grid.cs:**

namespace Combat.Board

{

using System;

using System.Collections.Generic;

using System.Linq;

using Information;

using UnityEngine;

using UnityEngine.Events;

[Serializable]

public class GridChange : UnityEvent<GridChangeInformation> { }

[Serializable]

public class MatchEvent : UnityEvent<MatchInformation> { }

[Serializable]

public class SlideEvent : UnityEvent<SlideInformation> { }

[SerializeField]

public class CreateGridEvent : UnityEvent<Grid> { }

[Serializable]

public class GemList

{

public List<Gem> gems = new List<Gem>();

public Gem this[int index]

{

get { return gems[index]; }

set { gems[index] = value; }

}

public static implicit operator List<Gem>(GemList gemList)

{

return gemList.gems.ToList();

}

public static implicit operator GemList(List<Gem> gemMonoList)

{

return new GemList { gems = gemMonoList.ToList() };

}

}

[Serializable]

public class Grid : IAttachable

{

[SerializeField]

private List<GemList> m\_GemLists = new List<GemList>();

[SerializeField]

private Vector2 m\_Size;

[SerializeField]

private List<Column> m\_Columns = new List<Column>();

[SerializeField]

private List<Row> m\_Rows = new List<Row>();

[SerializeField]

private MatchEvent m\_OnMatch = new MatchEvent();

[SerializeField]

private GridChange m\_OnGridChange = new GridChange();

[SerializeField]

private SlideEvent m\_OnSlide = new SlideEvent();

private readonly List<IComponent> m\_Components = new List<IComponent>();

public static readonly CreateGridEvent onCreate = new CreateGridEvent();

private const string RANDOM\_KEY = "Grid";

public List<GemList> gemLists { get { return m\_GemLists; } }

public Vector2 size { get { return m\_Size; } }

public List<Column> columns { get { return m\_Columns; } }

public List<Row> rows { get { return m\_Rows; } }

public MatchEvent onMatch { get { return m\_OnMatch; } }

public GridChange onGridChange { get { return m\_OnGridChange; } }

public SlideEvent onSlide { get { return m\_OnSlide; } }

public List<IComponent> components { get { return m\_Components; } }

public GemList this[int index] { get { return m\_GemLists[index]; } }

private Grid() { }

public Grid(Vector2 newSize) : this()

{

m\_Size = newSize;

onCreate.Invoke(this);

for (var y = 0; y < m\_Size.y; ++y)

{

m\_Rows.Add(new Row(this, y));

var newList = new List<Gem>();

for (var x = 0; x < m\_Size.x; ++x)

{

if (x == 0)

m\_Columns.Add(new Column(this, y));

var gemType = (GemType)RandomManager.self.Range<GemType>(RANDOM\_KEY);

newList.Add(

new Gem(

this, new Vector2(x, y), gemType));

}

m\_GemLists.Add(newList);

}

}

private class MatchNode

{

public Gem gem;

public MatchNode parent;

public List<MatchNode> children = new List<MatchNode>();

public Dictionary<Direction, MatchNode> adjacents = new Dictionary<Direction, MatchNode>();

public IEnumerable<MatchNode> nestedChildren

{

get

{

yield return this;

foreach (var child in children)

foreach (var nestedChild in child.nestedChildren)

yield return nestedChild;

}

}

public MatchNode rootNode { get { return parent == null ? this : parent.rootNode; } }

}

private class AdjacentNode

{

public MatchNode node;

public Direction directionFromCurrent;

}

public bool CheckMatch()

{

var searchSpace =

gemLists.Select(

gemList => gemList.gems.Select(

currentGem => new MatchNode { gem = currentGem }).ToList()).ToList();

var currentNode = searchSpace.First().First();

var openList = new List<MatchNode> { currentNode };

var closedList = new List<MatchNode>();

var matches = new List<MatchNode>();

while (openList.Count != 0)

{

currentNode = openList.First();

openList.Remove(currentNode);

closedList.Add(currentNode);

var adjacentNodes = new List<AdjacentNode>();

var adjacentGem = currentNode.gem.up;

if (adjacentGem != null)

adjacentNodes.Add(

new AdjacentNode

{

node = searchSpace[(int)adjacentGem.position.y][(int)adjacentGem.position.x],

directionFromCurrent = Direction.Up,

});

adjacentGem = currentNode.gem.down;

if (adjacentGem != null)

adjacentNodes.Add(

new AdjacentNode

{

node = searchSpace[(int)adjacentGem.position.y][(int)adjacentGem.position.x],

directionFromCurrent = Direction.Down,

});

adjacentGem = currentNode.gem.left;

if (adjacentGem != null)

adjacentNodes.Add(

new AdjacentNode

{

node = searchSpace[(int)adjacentGem.position.y][(int)adjacentGem.position.x],

directionFromCurrent = Direction.Left,

});

adjacentGem = currentNode.gem.right;

if (adjacentGem != null)

adjacentNodes.Add(

new AdjacentNode

{

node = searchSpace[(int)adjacentGem.position.y][(int)adjacentGem.position.x],

directionFromCurrent = Direction.Right,

});

foreach (var adjacentNode in adjacentNodes)

{

if (currentNode.gem.gemType == adjacentNode.node.gem.gemType &&

!closedList.Contains(adjacentNode.node))

{

openList.Insert(0, adjacentNode.node);

adjacentNode.node.parent = currentNode;

currentNode.children.Add(adjacentNode.node);

currentNode.adjacents[adjacentNode.directionFromCurrent] = adjacentNode.node;

adjacentNode.node.adjacents[adjacentNode.directionFromCurrent.Reverse()] =

currentNode;

var currentChild = adjacentNode.node;

var childCount = 1;

while (

currentChild.adjacents.ContainsKey(adjacentNode.directionFromCurrent.Reverse()))

{

currentChild =

currentChild.adjacents[adjacentNode.directionFromCurrent.Reverse()];

++childCount;

}

if (childCount >= 3 && !matches.Contains(currentNode.rootNode))

matches.Add(currentNode.rootNode);

}

else if (!openList.Contains(adjacentNode.node) && !closedList.Contains(adjacentNode.node))

openList.Add(adjacentNode.node);

}

}

foreach (var match in matches)

{

var matchGems = match.nestedChildren.Select(nestedChild => nestedChild.gem).ToList();

m\_OnMatch.Invoke(

new MatchInformation

{

gems = matchGems,

type = match.gem.gemType,

});

}

return matches.Count != 0;

}

public void ApplyGravity()

{

for (var y = 0; y < m\_GemLists.Count; y++)

{

for (var x = 0; x < m\_GemLists[y].gems.Count; ++x)

{

if (m\_GemLists[y][x] == null)

{

var newY = y;

while (newY < m\_GemLists.Count - 1 && m\_GemLists[newY][x] == null)

++newY;

m\_GemLists[y][x] = m\_GemLists[newY][x];

m\_GemLists[newY][x] = null;

if (m\_GemLists[y][x] != null)

m\_GemLists[y][x].position = new Vector2(x, y);

}

}

}

}

public void Fill()

{

for (var y = 0; y < m\_GemLists.Count; ++y)

{

for (var x = 0; x < m\_GemLists[y].gems.Count; ++x)

{

if (m\_GemLists[y][x] != null)

continue;

var gemType = (GemType)RandomManager.self.Range<GemType>(RANDOM\_KEY);

m\_GemLists[y][x] = new Gem(this, new Vector2(x, y), gemType);

}

}

}

private bool Add()

{

//TODO: Return if successful

return false;

}

public bool Remove(Gem gem)

{

var foundIndex = m\_GemLists.FindIndex(gemList => gemList.gems.Contains(gem));

// If a match was not found

if (foundIndex == -1)

return false;

// Removed the gem from the list which contains it

m\_GemLists[foundIndex].gems.Remove(gem);

onGridChange.Invoke(new GridChangeInformation { gems = new List<Gem> { gem } });

return true;

}

public bool RemoveAt(Vector2 position)

{

var y = (int)position.y;

if (y >= m\_GemLists.Count || y < 0)

return false;

var x = (int)position.x;

if (x >= m\_GemLists[y].gems.Count || x < 0)

return false;

// Store reference to removed gem

var gem = m\_GemLists[y][x];

m\_GemLists[y].gems.RemoveAt(x);

// Use that reference when you invoke the onGridChange event

onGridChange.Invoke(new GridChangeInformation { gems = new List<Gem> { gem } });

return true;

}

public bool Swap(Gem oldGem, Gem newGem)

{

var foundY = m\_GemLists.FindIndex(gemList => gemList.gems.Contains(oldGem));

// If a match was not found

if (foundY == -1)

return false;

var foundX = m\_GemLists[foundY].gems.FindIndex(gem => gem == oldGem);

m\_GemLists[foundY][foundX] = newGem;

onGridChange.Invoke(

new GridChangeInformation { gems = new List<Gem> { oldGem, newGem } });

return true;

}

public bool SwapAt(Vector2 position1, Vector2 position2)

{

var y1 = (int)position1.y;

var y2 = (int)position2.y;

if (y1 >= m\_GemLists.Count || y1 < 0 ||

y2 >= m\_GemLists.Count || y2 < 0)

return false;

var x1 = (int)position1.x;

var x2 = (int)position2.x;

if (x1 >= m\_GemLists[y1].gems.Count || x1 < 0 ||

x2 >= m\_GemLists[y2].gems.Count || x2 < 0)

return false;

// Store a reference to the gems about to be swapped

var gem1 = m\_GemLists[y1][x1];

var gem2 = m\_GemLists[y2][x2];

// Swap them

m\_GemLists[y1][x1] = gem2;

m\_GemLists[y2][x2] = gem1;

// Use the references of each gem that was changed when invoking the onGridChange event

onGridChange.Invoke(

new GridChangeInformation { gems = new List<Gem> { gem1, gem2 } });

return true;

}

public bool SlideRowAt(int index, SlideDirection direction)

{

if (index >= m\_Rows.Count || index < 0)

return false;

return m\_Rows[index].Slide(direction);

}

public bool SlideColumnAt(int index, SlideDirection direction)

{

if (index >= m\_Columns.Count || index < 0)

return false;

return m\_Columns[index].Slide(direction);

}

private void OnGemTypeChange(TypeChangeInformation typeChangeInfo)

{

//TODO: Check for matches on gems in the same row and column

}

public Vector2 ClampPosition(Vector2 position)

{

var clampedPosition =

new Vector2(Mathf.Round(position.x), Mathf.Round(position.y));

while (clampedPosition.x > size.x - 1)

clampedPosition.x -= size.x;

while (clampedPosition.x < 0f)

clampedPosition.x += size.x;

while (clampedPosition.y > size.y - 1)

clampedPosition.y -= size.y;

while (clampedPosition.y < 0f)

clampedPosition.y += size.y;

return clampedPosition;

}

}

}

**GridCollection.cs:**

namespace Combat.Board

{

using System;

using System.Collections.Generic;

using System.Linq;

using Information;

using UnityEngine;

using UnityEngine.Events;

public enum SlideDirection

{

Forward,

Backward,

}

[Serializable]

public class CreateGridCollectionEvent : UnityEvent<GridCollection> { }

[Serializable]

public abstract class GridCollection : IAttachable

{

[SerializeField]

protected int m\_Index;

protected readonly List<IComponent> m\_Components = new List<IComponent>();

public static readonly CreateGridCollectionEvent onCreate = new CreateGridCollectionEvent();

public Grid grid { get; protected set; }

public int index { get { return m\_Index; } }

public List<IComponent> components { get { return m\_Components; } }

public abstract IEnumerable<Gem> gems { get; }

protected GridCollection() { }

protected GridCollection(Grid newGrid, int newIndex) : this()

{

grid = newGrid;

m\_Index = newIndex;

onCreate.Invoke(this);

}

protected abstract bool CopyAt(List<List<Gem>> tempList, int fromIndex, int toIndex);

public bool Slide(SlideDirection direction)

{

var tempList =

grid.gemLists.Select(gemList => gemList.gems.ToList()).ToList();

var listCount = gems.Count();

for (var i = 0; i < listCount; i++)

{

int nextIndex;

switch (direction)

{

case SlideDirection.Forward:

nextIndex = i - 1;

if (nextIndex < 0)

nextIndex = listCount - 1;

break;

case SlideDirection.Backward:

nextIndex = i + 1;

if (nextIndex > listCount - 1)

nextIndex = 0;

break;

default:

throw new ArgumentOutOfRangeException("direction", direction, null);

}

var result = CopyAt(tempList, i, nextIndex);

if (!result)

return false;

}

grid.onSlide.Invoke(new SlideInformation { gridCollection = this });

return true;

}

}

[Serializable]

public class Column : GridCollection

{

private Column() { }

public Column(Grid newGrid, int newIndex) : base(newGrid, newIndex) { }

public override IEnumerable<Gem> gems

{

get { return grid.gemLists.Select(gemList => gemList[index]); }

}

protected override bool CopyAt(List<List<Gem>> tempList, int fromIndex, int toIndex)

{

var listCount = grid.gemLists[0].gems.Count;

if (fromIndex >= listCount || fromIndex < 0 ||

toIndex >= listCount || toIndex < 0)

return false;

grid.gemLists[toIndex][index] = tempList[fromIndex][index];

grid.gemLists[toIndex][index].position = new Vector2(index, toIndex);

return true;

}

}

[Serializable]

public class Row : GridCollection

{

private Row() { }

public Row(Grid newGrid, int newIndex) : base(newGrid, newIndex) { }

public override IEnumerable<Gem> gems

{

get { return grid.gemLists[index].gems; }

}

protected override bool CopyAt(List<List<Gem>> tempList, int fromIndex, int toIndex)

{

var listCount = grid.gemLists.Count;

if (fromIndex >= listCount || fromIndex < 0 ||

toIndex >= listCount || toIndex < 0)

return false;

grid.gemLists[index][toIndex] = tempList[index][fromIndex];

grid.gemLists[index][toIndex].position = new Vector2(toIndex, index);

return true;

}

}

}

**GridCollectionMono.cs:**

namespace Combat.Board

{

using System.Collections;

using System.Linq;

using UnityEngine;

using UnityEngine.UI;

[RequireComponent(typeof(Image))]

public class GridCollectionMono : MonoBehaviour, IComponent

{

[SerializeField]

private Image m\_Image;

[SerializeField]

private GridCollection m\_GridCollection;

private static bool s\_Initialized;

public GridCollection gridCollection { get { return m\_GridCollection; } }

private Vector2 m\_PositionOffset;

private Vector2 m\_CurrentDirection;

private float m\_ReducePositionOffsetTime = 0.75f;

private IEnumerator m\_ReducePositionOffsetCoroutine;

public Grid grid { get { return m\_GridCollection.grid; } }

public GridMono gridMono { get { return grid.GetComponent<GridMono>(); } }

public Vector2 positionOffset

{

get { return m\_PositionOffset; }

set

{

m\_PositionOffset = value;

m\_CurrentDirection =

Mathf.Abs(m\_PositionOffset.x) > Mathf.Abs(m\_PositionOffset.y)

? m\_PositionOffset.x > 0f

? Vector2.right : Vector2.left

: m\_PositionOffset.y > 0f

? Vector2.up : Vector2.down;

m\_ReducePositionOffsetCoroutine = ReducePositionOffset();

}

}

public Vector2 currentDirection { get { return m\_CurrentDirection; } }

private void OnCombatUpdate()

{

if (m\_ReducePositionOffsetCoroutine != null)

m\_ReducePositionOffsetCoroutine.MoveNext();

}

private void CheckForSlide()

{

var spacing = gridMono.CalculateSpacing();

if (Mathf.Abs(m\_PositionOffset.x) <= spacing.x / 2f &&

Mathf.Abs(m\_PositionOffset.y) <= spacing.y / 2f)

return;

var gemMonos = gridCollection.gems.Select(rowGem => rowGem.GetComponent<GemMono>()).ToList();

foreach (var gemMono in gemMonos)

{

var newPosition = gemMono.CalculatePosition(gemMono.position + m\_CurrentDirection);

gemMono.currentPosition = newPosition;

}

gridCollection.Slide(

m\_CurrentDirection == Vector2.right || m\_CurrentDirection == Vector2.up

? SlideDirection.Backward : SlideDirection.Forward);

m\_PositionOffset -=

gridCollection is Row

? new Vector2(spacing.x \* m\_CurrentDirection.x, 0f)

: new Vector2(0f, spacing.y \* m\_CurrentDirection.y);

m\_CurrentDirection = Vector2.zero;

}

private IEnumerator ReducePositionOffset()

{

CheckForSlide();

var gemMonos =

gridCollection.gems.Where(

gem => gem != null).SelectMany(

gem => gem.GetComponents<GemMono>()).ToList();

foreach (var gem in gemMonos)

gem.positionIsDirty = true;

yield return null;

var deltaTime = 0f;

while (deltaTime < m\_ReducePositionOffsetTime)

{

var spacing = gridMono.CalculateSpacing();

if (Mathf.Abs(m\_PositionOffset.x) > spacing.x / 2f ||

Mathf.Abs(m\_PositionOffset.y) > spacing.y / 2f)

{

var newOffset =

new Vector2(

spacing.x \* m\_CurrentDirection.x,

spacing.y \* m\_CurrentDirection.y);

m\_PositionOffset =

Vector2.Lerp(m\_PositionOffset, newOffset, deltaTime / m\_ReducePositionOffsetTime);

}

else

{

m\_PositionOffset =

Vector2.Lerp(m\_PositionOffset, Vector2.zero, deltaTime / m\_ReducePositionOffsetTime);

}

if (m\_PositionOffset.magnitude < 1f)

break;

CheckForSlide();

gemMonos =

gridCollection.gems.Where(

gem => gem != null).SelectMany(

gem => gem.GetComponents<GemMono>()).ToList();

foreach (var gem in gemMonos)

gem.positionIsDirty = true;

deltaTime += Time.deltaTime;

yield return null;

}

m\_PositionOffset = Vector2.zero;

m\_CurrentDirection = Vector2.zero;

gemMonos =

gridCollection.gems.Where(

gem => gem != null).SelectMany(

gem => gem.GetComponents<GemMono>()).ToList();

foreach (var gem in gemMonos)

gem.positionIsDirty = true;

m\_ReducePositionOffsetCoroutine = null;

}

public static void Init()

{

if (s\_Initialized)

return;

GridCollection.onCreate.AddListener(OnCreateGridCollection);

s\_Initialized = true;

}

private static void OnCreateGridCollection(GridCollection newGridCollection)

{

var gridMono =

newGridCollection.grid.components.First(component => component is GridMono) as GridMono;

if (gridMono == null)

return;

var newGameObject = new GameObject(newGridCollection.GetType() + " " + newGridCollection.index);

newGameObject.transform.SetParent(

newGridCollection is Row ?

CombatManager.self.rowParent.transform : CombatManager.self.columnParent.transform,

false);

if (newGridCollection is Row)

newGameObject.transform.SetAsFirstSibling();

var newGridCollectionMono = newGameObject.AddComponent<GridCollectionMono>();

newGridCollectionMono.m\_GridCollection = newGridCollection;

newGridCollection.components.Add(newGridCollectionMono);

newGridCollectionMono.m\_Image = newGameObject.GetComponent<Image>();

newGridCollectionMono.m\_Image.color = new Color(0f, 0f, 0f, 0f);

CombatManager.self.onCombatUpdate.AddListener(newGridCollectionMono.OnCombatUpdate);

CombatManager.self.onCombatEnding.AddListener(newGridCollectionMono.OnCombatUpdate);

}

}

}

**GridMono.cs:**

namespace Combat.Board

{

using System.Collections;

using System.Collections.Generic;

using System.Linq;

using Information;

using UnityEngine;

using UnityEngine.Events;

public class GridResizeEvent : UnityEvent<GridResizeInformation> { }

[RequireComponent(typeof(RectTransform))]

public class GridMono : MonoBehaviour, IComponent

{

[SerializeField]

private Grid m\_Grid;

[SerializeField]

private RectTransform m\_RectTransform;

[SerializeField]

private RectTransform m\_ParentRectTransform;

[SerializeField]

private GridResizeEvent m\_OnGridResize = new GridResizeEvent();

private static bool s\_Initilized;

private Vector2 m\_PreviousRectSize;

private bool m\_GemsAreAnimating;

private List<GemMono> m\_MatchingGemMonos = new List<GemMono>();

private bool m\_GemsAreMatching;

private IEnumerator m\_WaitToCheckForMatch;

public Grid grid { get { return m\_Grid; } }

public RectTransform rectTransform { get { return m\_RectTransform; } }

public RectTransform parentRectTransform { get { return m\_ParentRectTransform; } }

public GridResizeEvent onGridResize { get { return m\_OnGridResize; } }

public bool gemsAreAnimating { get { return m\_GemsAreAnimating; } }

public List<GemMono> matchingGemMonos { get { return m\_MatchingGemMonos; } }

private void LateUpdate()

{

var currentRectSize = new Vector2(m\_RectTransform.rect.width, m\_RectTransform.rect.height);

if (currentRectSize == m\_PreviousRectSize)

return;

m\_OnGridResize.Invoke(new GridResizeInformation { newRect = m\_RectTransform.rect });

m\_PreviousRectSize = currentRectSize;

}

private void OnCombatUpdate()

{

if (m\_MatchingGemMonos.Count != 0)

{

m\_GemsAreAnimating = true;

m\_GemsAreMatching = true;

var tempList = m\_MatchingGemMonos.ToList();

foreach (var matchingGemMono in tempList)

if (!matchingGemMono.matchAnimationCoroutine.MoveNext())

{

grid[(int)matchingGemMono.position.y][(int)matchingGemMono.position.x] = null;

m\_MatchingGemMonos.Remove(matchingGemMono);

}

}

else if (m\_GemsAreMatching)

{

m\_Grid.ApplyGravity();

m\_Grid.Fill();

m\_WaitToCheckForMatch =

WaitUntilAnyGemsThenExecute(

gem => gem.GetComponent<GemMono>().moveToPositionCoroutine != null,

() => m\_Grid.CheckMatch());

m\_GemsAreMatching = false;

}

else if (m\_WaitToCheckForMatch != null)

m\_WaitToCheckForMatch.MoveNext();

else

m\_GemsAreAnimating = false;

}

private void OnPlayerTurn()

{

m\_Grid.CheckMatch();

}

public Vector2 CalculateSpacing()

{

return

new Vector2(

m\_RectTransform.rect.width / (grid.size.x - 1),

m\_RectTransform.rect.height / (grid.size.y - 1));

}

private delegate bool GemPredicate(Gem gem);

private delegate void VoidDelegate();

private IEnumerator WaitUntilAnyGemsThenExecute(

GemPredicate gemPredicate, VoidDelegate functionDelegate)

{

while (

grid.gemLists.Any(

gemList => gemList.gems.Where(

gem => gem != null).Any(

gem => gemPredicate(gem))))

{

yield return null;

}

functionDelegate();

m\_WaitToCheckForMatch = null;

}

public static void Init()

{

if (s\_Initilized)

return;

Grid.onCreate.AddListener(OnGridCreate);

s\_Initilized = true;

}

private static void OnGridCreate(Grid newGrid)

{

var newGameObject = new GameObject();

newGameObject.transform.SetParent(CombatManager.self.gridParentRectTransform, false);

var newGridMono = newGameObject.AddComponent<GridMono>();

newGridMono.m\_Grid = newGrid;

newGrid.components.Add(newGridMono);

newGridMono.m\_RectTransform = newGameObject.GetComponent<RectTransform>();

newGridMono.m\_ParentRectTransform =

newGameObject.GetComponentsInParent<RectTransform>().First(

rectTransform => rectTransform != newGridMono.m\_RectTransform);

newGridMono.m\_RectTransform.anchorMin = new Vector2(0.1f, 0.1f);

newGridMono.m\_RectTransform.anchorMax = new Vector2(0.9f, 0.9f);

newGridMono.m\_RectTransform.sizeDelta = Vector2.zero;

newGridMono.m\_RectTransform.anchoredPosition = Vector2.zero;

CombatManager.self.onCombatUpdate.AddListener(newGridMono.OnCombatUpdate);

CombatManager.self.onPlayerTurn.AddListener(newGridMono.OnPlayerTurn);

CombatManager.self.onCombatEnding.AddListener(newGridMono.OnCombatUpdate);

}

}

}

**CombatAlternativeColorButton.cs:**

namespace Combat.UI

{

using UnityEngine;

using UnityEngine.UI;

[RequireComponent(typeof(Button))]

public class CombatAlternativeColorButton : MonoBehaviour

{

[SerializeField]

private Button m\_Button;

// Use this for initialization

private void Awake()

{

if (m\_Button == null)

m\_Button = GetComponent<Button>();

m\_Button.onClick.AddListener(OnClick);

}

private static void OnClick()

{

CombatManager.self.combatUiInformation.useAlternativeColors =

!CombatManager.self.combatUiInformation.useAlternativeColors;

}

}

}

**CombatMenuButton.cs:**

namespace Combat.UI

{

using UnityEngine;

using UnityEngine.UI;

public class CombatMenuButton : MonoBehaviour

{

[SerializeField]

private Button m\_Button;

[SerializeField]

private Image m\_BackgroundImage;

[SerializeField]

private GameObject m\_IconParent;

// Use this for initialization

private void Awake()

{

if (m\_Button == null)

m\_Button = GetComponent<Button>();

if (m\_Button != null)

m\_Button.onClick.AddListener(OnClick);

OnCombatPauseChange(CombatManager.self.isPaused);

CombatManager.self.onCombatPauseChange.AddListener(OnCombatPauseChange);

}

private static void OnClick()

{

CombatManager.self.isPaused = !CombatManager.self.isPaused;

}

private void OnCombatPauseChange(bool isPaused)

{

if (m\_BackgroundImage != null)

m\_BackgroundImage.enabled = isPaused;

if (m\_IconParent != null)

m\_IconParent.SetActive(isPaused);

}

}

}

**CombatModeButton.cs:**

namespace Combat.UI

{

using System;

using UnityEngine;

using UnityEngine.UI;

[RequireComponent(typeof(Button))]

public class CombatModeButton : MonoBehaviour

{

[SerializeField]

private Button m\_Button;

[SerializeField]

private Image m\_Image;

[SerializeField]

private Text m\_ModeText;

// Use this for initialization

private void Awake()

{

m\_Button = GetComponent<Button>();

if (m\_Image == null)

m\_Image = GetComponent<Image>();

m\_ModeText = GetComponentInChildren<Text>();

OnCombatModeChange();

CombatManager.self.onCombatModeChange.AddListener(OnCombatModeChange);

m\_Button.onClick.AddListener(OnClick);

}

private static void OnClick()

{

CombatManager.self.ToggleCombatMode();

}

private void OnCombatModeChange()

{

m\_Image.color = CombatManager.self.combatUiInformation.currentModeUiInformation.modeColor;

m\_ModeText.color = CombatManager.self.combatUiInformation.currentModeUiInformation.modeColor;

switch (CombatManager.self.combatMode)

{

case CombatManager.CombatMode.Attack:

m\_ModeText.text = "ATK";

break;

case CombatManager.CombatMode.Defense:

m\_ModeText.text = "DEF";

break;

default:

throw new ArgumentOutOfRangeException();

}

}

}

}

**CombatStatusBar.cs:**

namespace Combat.UI

{

using System;

using System.Collections;

using UnityEngine;

using UnityEngine.UI;

public class CombatStatusBar : MonoBehaviour

{

[SerializeField]

private Image m\_HealthImage;

[SerializeField]

private Image m\_HealthDelayedImage;

[SerializeField]

private Image m\_HealthBackgroundImage;

[SerializeField]

private Text m\_HealthText;

[SerializeField]

private Image m\_ShieldImage;

[SerializeField]

private Text m\_ShieldText;

[SerializeField]

private Image m\_ShieldIcon;

private Coroutine m\_DelayedHealthCoroutine;

private float m\_DelayedHealthValue;

// Use this for initialization

private void Awake()

{

OnCombatModeChange();

}

private void Start()

{

OnHealthModifierChanged();

OnDefenseModifierChanged();

CombatManager.self.onCombatModeChange.AddListener(OnCombatModeChange);

GameManager.self.playerData.health.onTotalValueChanged.AddListener(OnHealthModifierChanged);

GameManager.self.playerData.defense.onTotalValueChanged.AddListener(OnDefenseModifierChanged);

}

private void OnCombatModeChange()

{

if (m\_HealthImage == null)

return;

m\_HealthImage.color =

CombatManager.self.combatUiInformation.currentModeUiInformation.modeColor;

if (m\_HealthBackgroundImage == null)

return;

float h, s, v;

Color.RGBToHSV(m\_HealthImage.color, out h, out s, out v);

m\_HealthDelayedImage.color = Color.HSVToRGB(h, s + 0.2f, v);

m\_HealthBackgroundImage.color = Color.HSVToRGB(h, s - 0.25f, v);

}

private void OnHealthModifierChanged()

{

if (m\_HealthText != null)

{

var playerHeath = GameManager.self.playerData.health;

m\_HealthText.text =

Math.Ceiling(playerHeath.totalValue) + "/" + Mathf.Ceil(playerHeath.value);

}

if (m\_DelayedHealthValue == 0)

m\_DelayedHealthValue = GameManager.self.playerData.health.totalValue;

OnStatusModifierChanged();

if (m\_DelayedHealthCoroutine == null)

m\_DelayedHealthCoroutine = StartCoroutine(DelayedHealthEnumerator());

}

private void OnDefenseModifierChanged()

{

if (m\_ShieldText != null)

{

var playerDefense = GameManager.self.playerData.defense;

if (playerDefense.totalValue > 0f)

m\_ShieldText.text = Math.Ceiling(playerDefense.totalValue).ToString();

else

m\_ShieldText.text = string.Empty;

}

OnStatusModifierChanged();

}

private void OnStatusModifierChanged()

{

if (m\_HealthImage == null || m\_ShieldImage == null)

return;

var totalValue =

GameManager.self.playerData.health.totalValue

+ GameManager.self.playerData.defense.totalValue;

var maxValue = GameManager.self.playerData.health.value;

var imageBounds = m\_HealthImage.rectTransform.rect.max;

if (totalValue < maxValue)

{

m\_HealthImage.fillAmount =

GameManager.self.playerData.health.totalValue / maxValue;

m\_HealthDelayedImage.rectTransform.anchorMin =

new Vector2(

m\_HealthImage.fillAmount,

m\_HealthDelayedImage.rectTransform.anchorMin.y);

m\_HealthDelayedImage.rectTransform.anchorMax=

new Vector2(

m\_HealthImage.fillAmount + 1f,

m\_HealthDelayedImage.rectTransform.anchorMax.y);

m\_HealthDelayedImage.fillAmount =

(m\_DelayedHealthValue / maxValue) - m\_HealthImage.fillAmount;

m\_ShieldImage.rectTransform.anchorMin =

new Vector2(

m\_HealthImage.fillAmount,

m\_ShieldImage.rectTransform.anchorMin.y);

m\_ShieldImage.rectTransform.anchorMax =

new Vector2(

m\_ShieldImage.rectTransform.anchorMin.x + 1f,

m\_ShieldImage.rectTransform.anchorMax.y);

m\_ShieldImage.fillAmount =

GameManager.self.playerData.defense.totalValue / maxValue;

}

else

{

m\_HealthImage.fillAmount =

GameManager.self.playerData.health.totalValue / totalValue;

m\_ShieldImage.rectTransform.anchorMin =

new Vector2(

m\_HealthImage.fillAmount,

m\_ShieldImage.rectTransform.anchorMin.y);

m\_ShieldImage.rectTransform.anchorMax =

new Vector2(

m\_ShieldImage.rectTransform.anchorMin.x + 1f,

m\_ShieldImage.rectTransform.anchorMax.y);

m\_ShieldImage.fillAmount =

GameManager.self.playerData.defense.totalValue / totalValue;

}

m\_HealthText.rectTransform.anchorMax =

new Vector2(

m\_HealthImage.fillAmount,

m\_HealthText.rectTransform.anchorMax.y);

m\_ShieldText.rectTransform.anchorMax =

new Vector2(

m\_ShieldImage.fillAmount,

m\_ShieldImage.rectTransform.anchorMax.y);

m\_ShieldIcon.rectTransform.anchoredPosition =

new Vector2(

-m\_ShieldText.preferredWidth / 2f - 10f,

m\_ShieldIcon.rectTransform.anchoredPosition.y);

}

private IEnumerator DelayedHealthEnumerator()

{

while (m\_DelayedHealthValue != GameManager.self.playerData.health.totalValue)

{

m\_DelayedHealthValue =

Mathf.MoveTowards(

m\_DelayedHealthValue,

GameManager.self.playerData.health.totalValue,

7f \* Time.deltaTime);

OnStatusModifierChanged();

yield return null;

}

m\_DelayedHealthCoroutine = null;

}

}

}

**CombatUiInformation.cs:**

namespace Combat.UI

{

using System;

using System.Collections.Generic;

using Library;

using UnityEngine;

[CreateAssetMenu]

public class CombatUiInformation : ScriptableObject

{

[Serializable]

public class ModeUiInformation

{

public Sprite backgroundImage;

public Sprite midgroundImage;

public Sprite foregroundImage;

public Color modeColor;

}

[SerializeField]

private GameObject m\_GemGameObjectPrefab;

[SerializeField]

private ModeUiInformation m\_AttackModeUiInformation;

[SerializeField]

private ModeUiInformation m\_DefenseModeUiInformation;

[Space]

public List<Color> standardGemColors = new List<Color>();

public List<Color> alternativeGemColors = new List<Color>();

[Space, SerializeField]

private bool m\_UseAlternativeColors;

[Space, SerializeField]

private UnityBoolEvent m\_OnUseAlternativeColorsChange = new UnityBoolEvent();

public GameObject gemGameObjectPrefab { get { return m\_GemGameObjectPrefab; } }

public List<Color> gemColors

{

get { return m\_UseAlternativeColors ? alternativeGemColors : standardGemColors; }

}

public ModeUiInformation currentModeUiInformation

{

get

{

if (CombatManager.self == null)

return null;

switch (CombatManager.self.combatMode)

{

case CombatManager.CombatMode.Attack:

return m\_AttackModeUiInformation;

case CombatManager.CombatMode.Defense:

return m\_DefenseModeUiInformation;

default:

throw new ArgumentOutOfRangeException();

}

}

}

public bool useAlternativeColors

{

get { return m\_UseAlternativeColors; }

set { m\_UseAlternativeColors = value; m\_OnUseAlternativeColorsChange.Invoke(value); }

}

public UnityBoolEvent onUseAlternativeColorsChange

{

get { return m\_OnUseAlternativeColorsChange; }

}

}

}

**EnemyHealthBar.cs:**

namespace Combat.UI

{

using UnityEngine;

using UnityEngine.UI;

[RequireComponent(typeof(Rigidbody2D), typeof(Collider2D))]

public class EnemyHealthBar : Image, IComponent

{

[SerializeField]

private Enemy m\_Enemy;

private Rigidbody2D m\_Rigidbody2D;

private Collider2D m\_Collider2D;

private ContactFilter2D m\_ContactFilter2D;

private Bounds m\_MeshBounds;

private Vector3 m\_OffsetPosition;

public Enemy enemy { get { return m\_Enemy; } set { m\_Enemy = value; } }

protected override void Awake()

{

base.Awake();

m\_Rigidbody2D = GetComponent<Rigidbody2D>();

m\_Collider2D = GetComponent<Collider2D>();

m\_ContactFilter2D.SetLayerMask(Physics2D.GetLayerCollisionMask(gameObject.layer));

}

protected override void Start()

{

base.Start();

m\_Enemy.health.onTotalValueChanged.AddListener(UpdateFillAmount);

m\_Enemy.onDestroy.AddListener(EnemyDestroyed);

UpdateFillAmount();

var enemyMono = m\_Enemy.GetComponent<EnemyMono>();

var meshRenderer = enemyMono.GetComponentInChildren<SkinnedMeshRenderer>();

if (meshRenderer == null)

return;

m\_MeshBounds = meshRenderer.sharedMesh.bounds;

OnGUI();

}

private void LateUpdate()

{

var raycastHit2Ds = new RaycastHit2D[5];

var hitColliders = m\_Collider2D.Cast(Vector2.zero, m\_ContactFilter2D, raycastHit2Ds);

for (var i = 0; i < hitColliders; ++i)

m\_OffsetPosition += (Vector3)raycastHit2Ds[i].normal;

if (hitColliders == 0)

m\_OffsetPosition = Vector3.MoveTowards(m\_OffsetPosition, Vector3.zero, 0.25f);

m\_OffsetPosition = Vector3.Scale(m\_OffsetPosition, new Vector3(0f, 1f, 0f));

}

private void OnGUI()

{

var enemyMono = m\_Enemy.GetComponent<EnemyMono>();

var currentPosition =

Camera.main.WorldToScreenPoint(

m\_Enemy.GetComponent<EnemyMono>().transform.position

+ new Vector3(

0f,

m\_MeshBounds.max.y \* enemyMono.transform.lossyScale.y

+ enemyMono.transform.position.y,

0f));

rectTransform.position =

Vector3.Scale(currentPosition + m\_OffsetPosition, new Vector3(1f, 1f, 0f));

}

private void UpdateFillAmount()

{

fillAmount = m\_Enemy.health.totalValue / m\_Enemy.health.value;

color = Color.Lerp(Color.red, Color.green, fillAmount);

}

private void EnemyDestroyed()

{

Destroy(gameObject);

}

}

}

**EnemyTimerUI.cs:**

namespace Combat.UI

{

using UnityEngine;

using UnityEngine.UI;

[RequireComponent(typeof(EnemyHealthBar))]

public class EnemyTimerUI : MonoBehaviour

{

[SerializeField]

private Image m\_TimerMidground;

[SerializeField]

private Text m\_MovesText;

private EnemyHealthBar m\_EnemyHealthBar;

// Use this for initialization

private void Start()

{

m\_EnemyHealthBar = GetComponent<EnemyHealthBar>();

CombatManager.self.onPlayerTurn.AddListener(OnPlayerTurn);

OnPlayerTurn();

}

// Update is called once per frame

private void OnGUI()

{

m\_TimerMidground.fillAmount =

m\_EnemyHealthBar.enemy.timeUntilNextAttack / m\_EnemyHealthBar.enemy.attackSpeed;

}

private void OnPlayerTurn()

{

m\_MovesText.text = m\_EnemyHealthBar.enemy.movesUntilNextAttack.ToString();

}

}

}

**ItemButton.cs:**

namespace Combat.UI

{

using System;

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using UnityEngine.UI;

public class ItemButtons : MonoBehaviour

{

[SerializeField]

private Button m\_Button;

[SerializeField]

private Image m\_Image;

[SerializeField]

private Text m\_ModeText;

private void Awake()

{

}

private static void OnClick()

{

}

}

}

**QuitCombatEarly.cs:**

namespace Combat.UI

{

using UnityEngine;

using UnityEngine.UI;

[RequireComponent(typeof(Button))]

public class QuitCombatEarly : MonoBehaviour

{

private Button m\_Button;

private void Awake()

{

m\_Button = GetComponent<Button>();

m\_Button.onClick.AddListener(OnClick);

}

private static void OnClick()

{

GameManager.self.LoadScene(1);

}

}

}

**ToggleEnableObject.cs:**

namespace Combat.UI

{

using UnityEngine;

using UnityEngine.UI;

[RequireComponent(typeof(Button))]

public class ToggleEnableObject : MonoBehaviour

{

[SerializeField]

private GameObject m\_ToggleObject;

private Button m\_Button;

private void Awake()

{

m\_Button = GetComponent<Button>();

m\_Button.onClick.AddListener(OnClick);

}

private void OnClick()

{

if (m\_ToggleObject != null)

m\_ToggleObject.SetActive(!m\_ToggleObject.activeInHierarchy);

}

}

}

**CombatCamera.cs:**

namespace Combat

{

using System;

using System.Collections;

using System.Collections.Generic;

using System.Linq;

using UnityEngine;

[Serializable]

public class CombatCamera : MonoBehaviour

{

[SerializeField]

private List<TransformAnimation> m\_Animations = new List<TransformAnimation>();

[SerializeField]

private float m\_ScreenShakeTime;

[SerializeField]

private AnimationCurve m\_ScreenShakeCurve;

private IEnumerator m\_AnimationEnumerator;

private IEnumerator m\_ScreenShakeEnumerator;

private Vector3 m\_OriginalPosition;

private Quaternion m\_OriginalQuaternion;

private const string RANDOM\_KEY = "CombatCamera";

public bool isAnimating = true;

public List<TransformAnimation> animations { get { return m\_Animations; } }

private void Start()

{

m\_OriginalPosition = transform.position;

m\_OriginalQuaternion = transform.rotation;

m\_AnimationEnumerator = Animate();

GameManager.self.playerData.onTakeDamage.AddListener(OnPlayerTakeDamage);

CombatManager.self.onCombatUpdate.AddListener(OnCombatUpdate);

}

private void OnCombatUpdate()

{

if (!isAnimating)

{

transform.position = m\_OriginalPosition;

transform.rotation = m\_OriginalQuaternion;

}

else if (m\_AnimationEnumerator != null)

m\_AnimationEnumerator.MoveNext();

if (m\_ScreenShakeEnumerator != null)

if (!m\_ScreenShakeEnumerator.MoveNext())

m\_ScreenShakeEnumerator = null;

}

private void OnPlayerTakeDamage()

{

m\_ScreenShakeEnumerator = ScreenShakEnumerator();

}

private IEnumerator Animate()

{

var originalPosition = transform.localPosition;

var originalEulerAngles = transform.localEulerAngles;

var originalZoom = transform.GetChild(0).localPosition;

TransformAnimation currentAnimation = null;

while (true)

{

transform.localPosition = originalPosition;

transform.localEulerAngles = originalEulerAngles;

transform.GetChild(0).localPosition = originalZoom;

var tempList = m\_Animations.ToList();

if (currentAnimation != null)

tempList.Remove(currentAnimation);

var randomIndex = RandomManager.self.Range(RANDOM\_KEY, 0, tempList.Count);

currentAnimation = tempList[randomIndex];

//TODO: Do something based on TargetType

var currentEnumerator = currentAnimation.Animate(transform, null);

while (currentEnumerator.MoveNext())

yield return null;

}

m\_AnimationEnumerator = null;

}

private IEnumerator ScreenShakEnumerator()

{

var deltaTime = 0f;

while (deltaTime < m\_ScreenShakeTime)

{

var xOffset = transform.right \* RandomManager.self.Range(RANDOM\_KEY, -0.2f, 0.2f);

var yOffset = transform.up \* RandomManager.self.Range(RANDOM\_KEY, -0.2f, 0.2f);

transform.position +=

(xOffset + yOffset)

\* m\_ScreenShakeCurve.Evaluate(deltaTime / m\_ScreenShakeTime);

deltaTime += Time.deltaTime;

yield return null;

}

}

}

}

**CombatManager.cs:**

namespace Combat

{

using System;

using System.Collections.Generic;

using System.Linq;

using UnityEngine;

using UnityEngine.Events;

using UnityEngine.EventSystems;

using UnityEngine.UI;

using Board;

using Board.Information;

using CustomInput.Information;

using Library;

using UI;

public class CombatManager : SubManager<CombatManager>

{

public enum CombatMode

{

Attack,

Defense,

}

[SerializeField]

private Canvas m\_Canvas;

[SerializeField]

private RectTransform m\_GridParentRectTransform;

[SerializeField]

private VerticalLayoutGroup m\_RowParent;

[SerializeField]

private HorizontalLayoutGroup m\_ColumnParent;

[SerializeField]

private GameObject m\_ResultsPanel;

[Space, SerializeField]

private CombatUiInformation m\_CombatUiInformation;

[SerializeField]

private CombatMode m\_CombatMode;

[Space, SerializeField]

private UnityEvent m\_OnCombatBegin = new UnityEvent();

[SerializeField]

private UnityEvent m\_OnCombatUpdate = new UnityEvent();

[SerializeField]

private UnityEvent m\_OnCombatLateUpdate = new UnityEvent();

[SerializeField]

private UnityEvent m\_OnCombatEnding = new UnityEvent();

[SerializeField]

private UnityEvent m\_OnCombatEnd = new UnityEvent();

[Space, SerializeField]

private UnityEvent m\_OnCombatModeChange = new UnityEvent();

[Space, SerializeField]

private UnityBoolEvent m\_OnCombatPauseChange = new UnityBoolEvent();

[SerializeField]

private UnityEvent m\_OnPlayerTurn = new UnityEvent();

private GridMono m\_GridMono;

private GridCollectionMono m\_LockedGridCollectionMono;

private bool m\_CombatHasBegun;

private bool m\_IsPaused;

private bool m\_IsEnding;

private bool m\_HasSlid;

public Canvas canvas { get { return m\_Canvas; } }

public RectTransform gridParentRectTransform { get { return m\_GridParentRectTransform; } }

public VerticalLayoutGroup rowParent { get { return m\_RowParent; } }

public HorizontalLayoutGroup columnParent { get { return m\_ColumnParent; } }

public CombatUiInformation combatUiInformation { get { return m\_CombatUiInformation; } }

public CombatMode combatMode { get { return m\_CombatMode; } }

public UnityEvent onCombatBegin { get { return m\_OnCombatBegin; } }

public UnityEvent onCombatUpdate { get { return m\_OnCombatUpdate; } }

public UnityEvent onCombatLateUpdate { get { return m\_OnCombatLateUpdate; } }

public UnityEvent onCombatEnding { get { return m\_OnCombatEnding; } }

public UnityEvent onCombatEnd { get { return m\_OnCombatEnd; } }

public UnityEvent onCombatModeChange { get { return m\_OnCombatModeChange; } }

public bool isPaused

{

get { return m\_IsPaused; }

set

{

m\_IsPaused = value;

foreach (var animator in FindObjectsOfType<Animator>())

animator.enabled = !m\_IsPaused;

m\_OnCombatPauseChange.Invoke(value);

}

}

public UnityBoolEvent onCombatPauseChange { get { return m\_OnCombatPauseChange; } }

public UnityEvent onPlayerTurn { get { return m\_OnPlayerTurn; } }

public GridMono gridMono { get { return m\_GridMono; } }

protected override void Init()

{

if (m\_Canvas == null)

m\_Canvas = FindObjectOfType<Canvas>();

if (m\_GridParentRectTransform == null)

m\_GridParentRectTransform = m\_Canvas.GetComponent<RectTransform>();

GridMono.Init();

GridCollectionMono.Init();

GemMono.Init();

GemMonoDuplicate.Init();

var newGrid = new Grid(new Vector2(5f, 5f));

newGrid.onSlide.AddListener(OnSlide);

m\_GridMono = newGrid.GetComponent<GridMono>();

}

private void Update()

{

if (m\_CombatHasBegun == false)

{

m\_OnCombatBegin.Invoke();

m\_CombatHasBegun = true;

}

if (m\_IsPaused)

return;

if (m\_IsEnding)

{

m\_OnCombatEnding.Invoke();

return;

}

// Win or Lose

if (EnemyManager.self.enemies.Count == 0 ||

GameManager.self.playerData.health.totalValue <= 0)

{

m\_ResultsPanel.SetActive(true);

m\_ResultsPanel.GetComponentInChildren<ResultsScreen>().

ResultsScreenBegin(EnemyManager.self.enemies.Count == 0);

m\_IsEnding = true;

}

m\_OnCombatUpdate.Invoke();

}

private void LateUpdate()

{

if (!m\_IsPaused)

m\_OnCombatLateUpdate.Invoke();

}

public void ToggleCombatMode()

{

switch (m\_CombatMode)

{

case CombatMode.Attack:

m\_CombatMode = CombatMode.Defense;

break;

case CombatMode.Defense:

m\_CombatMode = CombatMode.Attack;

break;

default:

throw new ArgumentOutOfRangeException();

}

m\_OnCombatModeChange.Invoke();

}

private void OnSlide(SlideInformation slideInfo)

{

m\_HasSlid = true;

}

protected override void OnBeginDrag(DragInformation dragInfo)

{

if (m\_IsEnding)

return;

var hitMonos = RayCastToGridCollectionMono(dragInfo.origin).ToList();

// If we didn't hit a GemMono first

if (!hitMonos.Any())

{

m\_LockedGridCollectionMono = null;

return;

}

m\_LockedGridCollectionMono =

hitMonos.First(

hitMono =>

Mathf.Abs(dragInfo.totalDelta.x) > Mathf.Abs(dragInfo.totalDelta.y)

? hitMono.gridCollection is Row : hitMono.gridCollection is Column);

}

protected override void OnDrag(DragInformation dragInfo)

{

if (m\_IsEnding)

return;

if (m\_GridMono.gemsAreAnimating)

return;

// If we didn't hit a GridCollectionMono at the start of the drag

if (m\_LockedGridCollectionMono == null)

return;

var addedPositionOffset =

new Vector2(

m\_LockedGridCollectionMono.gridCollection is Row

? dragInfo.delta.x / m\_Canvas.scaleFactor : 0f,

m\_LockedGridCollectionMono.gridCollection is Column

? dragInfo.delta.y / m\_Canvas.scaleFactor : 0f);

m\_LockedGridCollectionMono.positionOffset += addedPositionOffset;

}

protected override void OnEndDrag(DragInformation dragInfo)

{

if (m\_IsEnding)

return;

if (!m\_HasSlid)

return;

onPlayerTurn.Invoke();

m\_HasSlid = false;

}

private static IEnumerable<GridCollectionMono> RayCastToGridCollectionMono(Vector2 origin)

{

var pointerEventData =

new PointerEventData(EventSystem.current) { position = origin };

var hits = new List<RaycastResult>();

EventSystem.current.RaycastAll(pointerEventData, hits);

// If nothing was hit

if (!hits.Any())

return new GridCollectionMono[] { };

// Return the first hit object's GridCollectionMono component

// Will be null if one was not found on the game object

return

hits.

Where(

hit => hit.gameObject.GetComponent<GridCollectionMono>() != null).

Select(

hit => hit.gameObject.GetComponent<GridCollectionMono>());

}

}

}

**CombatStatsBar.cs:**

namespace Combat

{

using UnityEngine;

using UnityEngine.UI;

using System;

[RequireComponent(typeof(Image))]

public class CombatStatsBar : MonoBehaviour

{

private Image m\_Image;

[SerializeField]

private Text m\_DefenseText;

[SerializeField]

private Text m\_HealthText;

// Use this for initialization

private void Awake()

{

m\_Image = GetComponent<Image>();

OnCombatModeChange();

CombatManager.self.onCombatModeChange.AddListener(OnCombatModeChange);

CombatManager.self.onCombatUpdate.AddListener(OnCombatUpdate);

}

private void OnCombatModeChange()

{

var color = CombatManager.self.combatUiInformation.currentModeUiInformation.modeColor;

m\_Image.color = color;

}

private void OnCombatUpdate()

{

var playerStats = GameManager.self.playerData;

m\_HealthText.text = "HP: " + Math.Ceiling(playerStats.health.totalValue)

+ "/" + playerStats.health.value;

m\_DefenseText.text = "S:" + Math.Ceiling(playerStats.defense.totalValue)

+ "/" + (playerStats.defense.value \* 20);

}

}

}

**ComponetArchitecture.cs:**

namespace Combat

{

using System.Collections.Generic;

using System.Linq;

public interface IComponent { }

public interface IAttachable { List<IComponent> components { get; } }

public static class AttachableExtensions

{

public static T GetComponent<T>(this IAttachable attachable) where T : IComponent

{

return (T)attachable.components.First(component => component is T);

}

public static IEnumerable<T> GetComponents<T>(this IAttachable attachable) where T : IComponent

{

return attachable.components.Where(component => component is T).Cast<T>();

}

}

}

**Enemy.cs:**

using System;

using System.Collections.Generic;

using Combat.Board;

using UnityEngine;

namespace Combat

{

using UnityEngine.Events;

[Serializable]

public class UnityEnemyEvent : UnityEvent<Enemy> { }

[Serializable]

public class Enemy : IAttachable

{

[SerializeField]

private Attribute m\_Health = new Attribute { value = 10f };

[SerializeField]

private Attribute m\_Attack = new Attribute { value = 10f };

[SerializeField]

private Attribute m\_Defense = new Attribute { value = 10f };

[SerializeField]

private uint m\_ExperianceValue = 10;

private UnityEnemyEvent m\_OnTakeDamage = new UnityEnemyEvent();

private UnityEvent m\_OnAttack = new UnityEvent();

private UnityEvent m\_OnDestroy = new UnityEvent();

public Attribute health { get { return m\_Health; } }

public Attribute attack { get { return m\_Attack; } }

public Attribute defense { get { return m\_Defense; } }

public uint experianceValue { get { return m\_ExperianceValue;} }

public float attackSpeed;

public int movesUntilAttack;

public GemType damageType;

public List<GemType> resistances;

public List<GemType> weaknesses;

public int movesCounter = 0;

public float attackCountdown;

private readonly List<IComponent> m\_Components = new List<IComponent>();

public UnityEnemyEvent onTakeDamage { get { return m\_OnTakeDamage; } }

public UnityEvent onAttack { get { return m\_OnAttack; } }

public UnityEvent onDestroy { get { return m\_OnDestroy; } }

public List<IComponent> components { get { return m\_Components; } }

public float timeUntilNextAttack { get { return attackCountdown; } }

public int movesUntilNextAttack { get { return movesUntilAttack - movesCounter % movesUntilAttack; } }

public Enemy()

{

health.value = 10;

attack.value = 10;

defense.value = 10;

attackSpeed = 5;

attackCountdown = attackSpeed;

movesUntilAttack = 3;

}

public Enemy(

float newHealth,

float newAttack,

float newDefense,

float newAttackSpeed,

int newMovesUntilAttack)

{

health.value = newHealth;

attack.value = newAttack;

defense.value = newDefense;

attackSpeed = newAttackSpeed;

movesUntilAttack = newMovesUntilAttack;

}

public void OnCombatBegin()

{

CombatManager.self.onPlayerTurn.AddListener(OnPlayerTurn);

CombatManager.self.onCombatUpdate.AddListener(OnCombatUpdate);

}

private void OnCombatUpdate()

{

if (health.totalValue <= 0)

{

m\_OnDestroy.Invoke();

CombatManager.self.onCombatUpdate.RemoveListener(OnCombatUpdate);

}

attackCountdown -= Time.deltaTime;

if (attackCountdown > 0) return;

Attack();

attackCountdown = attackSpeed;

}

private void OnPlayerTurn()

{

movesCounter++;

if (movesCounter != 0 && movesCounter % movesUntilAttack == 0)

{

Attack();

}

}

private void Attack()

{

GameManager.self.playerData.TakeDamage(attack.totalValue, damageType);

m\_OnAttack.Invoke();

}

public void TakeDamage(float damage, GemType gemType)

{

var percentage = damage / defense.totalValue;

var finalDamage = damage \* Mathf.Clamp(percentage, 0f, 1f);

if (resistances != null && resistances.Contains(gemType))

{

finalDamage \*= .75f;

}

else if (weaknesses != null && weaknesses.Contains(gemType))

{

finalDamage \*= 1.25f;

}

health.modifier -= finalDamage;

m\_OnTakeDamage.Invoke(this);

}

}

}

**EnemyManager.cs:**

namespace Combat

{

using System.Collections;

using System.Collections.Generic;

using System.Linq;

using Board.Information;

using CustomInput.Information;

using UnityEngine;

public class EnemyManager : SubManager<EnemyManager>

{

[SerializeField]

private GameObject m\_EnemyHealthBarPrefab;

private List<EnemyMono> m\_Enemies = new List<EnemyMono>();

private EnemyMono m\_CurrentEnemy;

[SerializeField]

private UnityEnemyEvent m\_OnCurrentEnemyChange = new UnityEnemyEvent();

[SerializeField]

private float m\_PauseCameraTime;

private float m\_CurrentPauseCameraTime;

private IEnumerator m\_PauseCameraEnumerator;

private List<IEnumerator> m\_AnimateEnemies = new List<IEnumerator>();

private uint m\_ExperianceTotal = 0;

private const string RANDOM\_KEY = "EnemyManager";

public float enemyPadding = 1f;

public GameObject enemyHealthBarPrefab { get { return m\_EnemyHealthBarPrefab; } }

public uint experianceTotal { get { return m\_ExperianceTotal; } }

public EnemyMono currentEnemy

{

get { return m\_CurrentEnemy; }

private set

{

if (m\_CurrentEnemy == value)

return;

m\_CurrentEnemy = value;

m\_OnCurrentEnemyChange.Invoke(

m\_CurrentEnemy == null ? null : value.enemy);

}

}

public UnityEnemyEvent onCurrentEnemyChange { get { return m\_OnCurrentEnemyChange; } }

public List<EnemyMono> enemies { get { return m\_Enemies; } }

// Use this for initialization

protected override void Init()

{

var managerEnemies = GameManager.self.enemyIndexes;

var enemyPrefabList = GameManager.self.enemyPrefabList;

var totalSpace =

enemyPrefabList.Sum(

enemy => enemy.transform.root.GetComponentInChildren<Collider>().bounds.size.x);

totalSpace += enemyPadding \* (managerEnemies.Count - 1);

var pos = -totalSpace / 2f;

for (var i = 0; i < managerEnemies.Count; i++)

{

var enemyPrefab = enemyPrefabList[managerEnemies[i]];

var enemyMeshBounds =

enemyPrefab.transform.root.GetComponentInChildren<Collider>().bounds;

pos += enemyMeshBounds.extents.x;

var enemyObject =

Instantiate(

enemyPrefab,

new Vector3(pos, .5f, 0),

enemyPrefab.transform.rotation);

var animator = enemyObject.transform.root.GetComponentInChildren<Animator>();

animator.speed = 0f;

var randomTime = RandomManager.self.Range(RANDOM\_KEY, 0f, 1f);

m\_AnimateEnemies.Add(AnimateEnemy(randomTime, animator));

pos += enemyMeshBounds.extents.x;

pos += enemyPadding;

enemyObject.name += i;

var enemyMono = enemyObject.transform.root.GetComponentInChildren<EnemyMono>();

var enemy = enemyMono.enemy;

enemy.attackCountdown =

enemy.attackSpeed - RandomManager.self.Range(RANDOM\_KEY, 0f, enemy.attackSpeed);

enemy.movesCounter = RandomManager.self.Range(RANDOM\_KEY, 0, enemy.movesUntilAttack);

m\_Enemies.Add(enemyMono);

CombatManager.self.onCombatBegin.AddListener(enemyMono.enemy.OnCombatBegin);

m\_ExperianceTotal += enemy.experianceValue;

}

GameManager.self.enemyIndexes = new List<int>();

currentEnemy = m\_Enemies[0];

CombatManager.self.onCombatUpdate.AddListener(OnCombatUpdate);

CombatManager.self.gridMono.grid.onMatch.AddListener(OnMatch);

}

private void OnMatch(MatchInformation matchInfo)

{

if (currentEnemy == null || currentEnemy.enemy == null)

return;

var playerData = GameManager.self.playerData;

switch (CombatManager.self.combatMode)

{

case CombatManager.CombatMode.Attack:

var dam = playerData.attack.totalValue \*

(1 + (matchInfo.gems.Count - 3) \* .25f);

currentEnemy.enemy.TakeDamage(dam, matchInfo.type);

if (currentEnemy.enemy.health.totalValue <= 0f)

{

m\_Enemies.Remove(currentEnemy);

currentEnemy = m\_Enemies.FirstOrDefault();

}

break;

case CombatManager.CombatMode.Defense:

GameManager.self.playerData.defense.modifier += matchInfo.gems.Count

\* (playerData.defense.value \* .8f);

break;

}

}

private void OnCombatUpdate()

{

GameManager.self.playerData.DecayShield();

if (m\_PauseCameraEnumerator != null)

m\_PauseCameraEnumerator.MoveNext();

foreach (var animateEnemy in m\_AnimateEnemies.ToList())

{

if (!animateEnemy.MoveNext())

m\_AnimateEnemies.Remove(animateEnemy);

}

}

protected override void OnPress(TouchInformation touchInfo)

{

// Else shoot ray from touch

var ray = Camera.main.ScreenPointToRay(touchInfo.position);

var raycastHits = Physics.RaycastAll(ray.origin, ray.direction);

if (!raycastHits.Any())

return;

var hitEnemyMono =

raycastHits.

Select(hit => hit.transform.root.GetComponentInChildren<EnemyMono>()).

FirstOrDefault(enemyMono => enemyMono != null);

if (hitEnemyMono == null)

return;

currentEnemy = hitEnemyMono;

m\_CurrentPauseCameraTime = 0f;

if (m\_PauseCameraEnumerator == null)

m\_PauseCameraEnumerator = PauseCameraEnumerator();

}

private IEnumerator AnimateEnemy(float waitTime, Animator enemyAnimator)

{

var deltaTime = 0f;

while (deltaTime < waitTime)

{

deltaTime += Time.deltaTime;

yield return null;

}

enemyAnimator.speed = 1f;

}

private IEnumerator PauseCameraEnumerator()

{

var combatCamera = FindObjectOfType<CombatCamera>();

if (combatCamera == null)

yield break;

combatCamera.isAnimating = false;

// Wait until 'm\_PauseCameraTime' time has passed

while (m\_CurrentPauseCameraTime < m\_PauseCameraTime)

{

m\_CurrentPauseCameraTime += Time.deltaTime;

yield return null;

}

combatCamera.isAnimating = true;

m\_PauseCameraEnumerator = null;

}

}

}

**EnemyMono.cs:**

namespace Combat

{

using System.Linq;

using UnityEngine;

using UI;

public class EnemyMono : MonoBehaviour, IComponent

{

[SerializeField]

private Enemy m\_Enemy;

[SerializeField]

private Animator m\_Animator;

public Enemy enemy { get { return m\_Enemy; } }

private void Awake()

{

var newGameObject = Instantiate(EnemyManager.self.enemyHealthBarPrefab);

var newEnemyHealthBar = newGameObject.GetComponent<EnemyHealthBar>();

newGameObject.transform.SetParent(

FindObjectsOfType<Canvas>().

FirstOrDefault(

canvas => canvas.renderMode == RenderMode.ScreenSpaceOverlay).transform,

false);

m\_Animator = GetComponent<Animator>();

newGameObject.transform.SetAsFirstSibling();

newEnemyHealthBar.enemy = m\_Enemy;

m\_Enemy.components.Add(this);

m\_Enemy.components.Add(newEnemyHealthBar);

m\_Enemy.onTakeDamage.AddListener(OnEnemyTakeDamage);

m\_Enemy.onAttack.AddListener(OnAttack);

m\_Enemy.onDestroy.AddListener(OnEnemyDestroy);

}

private void OnEnemyDestroy()

{

m\_Animator.SetTrigger("Dead");

}

private void OnEnemyTakeDamage(Enemy hitEnemy)

{

if (enemy != hitEnemy)

return;

m\_Animator.SetTrigger("Take Damage");

}

private void OnAttack()

{

m\_Animator.SetTrigger("Attack");

}

private void OnDeadStateExit()

{

Destroy(transform.root.gameObject);

}

}

}

**ResultsScreen.cs:**

using UnityEngine;

namespace Combat

{

using System.Collections;

using UnityEngine.UI;

public class ResultsScreen : MonoBehaviour

{

[SerializeField]

private Text m\_ResultText;

[SerializeField]

private Image m\_ExpForgroundImage;

[SerializeField]

private Image m\_ExpMidgroundImage;

private RectTransform m\_RectTransform;

private Button m\_EndSceneButton;

private IEnumerator m\_ResultScreeEnumerator;

[Space]

public float animationTime;

public float barFillTime;

private void Awake()

{

m\_RectTransform = GetComponent<RectTransform>();

m\_RectTransform.anchoredPosition -= new Vector2(0, 1000);

m\_ExpForgroundImage.fillAmount = 0f;

SetMidground(0);

m\_EndSceneButton = GetComponentInChildren<Button>();

m\_EndSceneButton.onClick.AddListener(() => { CombatManager.self.onCombatEnd.Invoke(); });

CombatManager.self.onCombatEnding.AddListener(OnCombatEnding);

}

private void OnCombatEnding()

{

if (m\_ResultScreeEnumerator != null)

m\_ResultScreeEnumerator.MoveNext();

}

[ContextMenu("Start animation")]

public void ResultsScreenBegin(bool result)

{

m\_ResultScreeEnumerator = ResultsScreenEnumerator(result);

}

private void SetMidground(float fillAmount)

{

m\_ExpMidgroundImage.rectTransform.anchorMin = new Vector2(

m\_ExpForgroundImage.fillAmount, m\_ExpMidgroundImage.rectTransform.anchorMin.y);

m\_ExpMidgroundImage.rectTransform.anchorMax = new Vector2(

m\_ExpMidgroundImage.rectTransform.anchorMin.x + 1,

m\_ExpMidgroundImage.rectTransform.anchorMax.y);

m\_ExpMidgroundImage.fillAmount = fillAmount;

}

private IEnumerator ResultsScreenEnumerator(bool result)

{ // Set Win Text

AudioManager.self.PlayEndBattleSound(result);

m\_ResultText.text = result ? "You Win!" : "You Lose...";

m\_EndSceneButton.interactable = false;

var animationFraction = 1000 / animationTime;

// Animate the pannel in

while (m\_RectTransform.anchoredPosition.y < 0)

{

m\_RectTransform.anchoredPosition += new Vector2(0, Time.deltaTime \* animationFraction);

yield return null;

}

m\_RectTransform.anchoredPosition = Vector2.zero; // Make it all good

// Check to see how much of the bar needs to be filled.

var forgroundFillAmount = (float)

GameManager.self.playerData.playerLevelSystem.playerLevelInfo.currentExperience /

GameManager.self.playerData.playerLevelSystem.playerLevelInfo.experienceRequired;

// Find out how much needs to be filled per second

var fillFraction = 1 / animationTime;

// Animate Bar filling

while (m\_ExpForgroundImage.fillAmount < forgroundFillAmount)

{

m\_ExpForgroundImage.fillAmount += Time.deltaTime \* fillFraction;

yield return null;

}

if (result)

{

GameManager.self.currentNode.isComplete = true;

// Get some experience

GameManager.self.playerData.playerLevelSystem.IsLeveledUp(EnemyManager.self.experianceTotal);

// Check to see how much the total bar needs

var midgroundFillAmount = (float)

GameManager.self.playerData.playerLevelSystem.playerLevelInfo.currentExperience /

GameManager.self.playerData.playerLevelSystem.playerLevelInfo.experienceRequired;

var finalFill = midgroundFillAmount;

// Take off how much is already filled in

midgroundFillAmount -= forgroundFillAmount;

fillFraction = midgroundFillAmount / barFillTime;

// As long as the bar is in range and not greater than one, fill.

float setamount = 0;

while (m\_ExpMidgroundImage.fillAmount < midgroundFillAmount &&

m\_ExpForgroundImage.fillAmount + m\_ExpMidgroundImage.fillAmount < 1)

{

setamount += Time.deltaTime \* fillFraction;

SetMidground(setamount);

yield return null;

}

while (m\_ExpForgroundImage.fillAmount < finalFill)

{

m\_ExpForgroundImage.fillAmount += Time.deltaTime \* fillFraction;

yield return null;

}

}

GameManager.self.playerData.health.modifier = 0;

GameManager.self.playerData.defense.modifier = 0;

GameManager.self.SavePlayer();

m\_EndSceneButton.interactable = true;

m\_ResultScreeEnumerator = null;

}

}

}

**TargetingEnemy.cs:**

namespace Combat

{

using System.Collections;

using UnityEngine;

public class TargetingEnemy : MonoBehaviour

{

[SerializeField]

private GameObject m\_MarkerPrefab;

[SerializeField]

private Vector3 m\_MaxPositionOffset;

[SerializeField]

private Vector3 m\_MinPositionOffset;

[SerializeField]

private AnimationCurve m\_AnimationCurve;

[SerializeField]

private float m\_AnimationTime;

private bool m\_Rising = true;

private Transform m\_Marker;

private Vector3 m\_CurrentPosition;

private IEnumerator m\_MarkerMovementEnumerator;

private void Start()

{

m\_Marker = Instantiate(m\_MarkerPrefab).transform;

CombatManager.self.onCombatUpdate.AddListener(OnCombatUpdate);

EnemyManager.self.onCurrentEnemyChange.AddListener(OnCurrentEnemyChange);

OnCurrentEnemyChange(EnemyManager.self.currentEnemy.enemy);

m\_MarkerMovementEnumerator = MarkerMovementEnumerator();

}

private void OnCombatUpdate()

{

if (m\_MarkerMovementEnumerator != null)

m\_MarkerMovementEnumerator.MoveNext();

}

private void OnCurrentEnemyChange(Enemy enemy)

{

if (enemy != null)

SetMarkerToCurrent(enemy.GetComponent<EnemyMono>());

}

private void SetMarkerToCurrent(EnemyMono enemyMono)

{

var currentEnemyBounds = enemyMono.GetComponent<Collider>().bounds;

m\_Marker.SetParent(enemyMono.transform.root, false);

m\_CurrentPosition = currentEnemyBounds.center +

new Vector3(0, currentEnemyBounds.extents.y, 0) + m\_MinPositionOffset;

m\_Marker.position = m\_CurrentPosition;

}

private IEnumerator MarkerMovementEnumerator()

{

var deltaTime = 0f;

while (true)

{

// If the enemy is not null or there are no enemies, return

if (EnemyManager.self.enemies.Count == 0)

{

Destroy(m\_Marker.gameObject);

Destroy(this);

yield break;

}

if (EnemyManager.self.currentEnemy != null && m\_AnimationTime != 0f)

{

if (m\_Rising)

m\_Marker.position =

m\_CurrentPosition + m\_MaxPositionOffset \*

m\_AnimationCurve.Evaluate(deltaTime / m\_AnimationTime);

else

m\_Marker.position =

m\_CurrentPosition + m\_MaxPositionOffset \*

m\_AnimationCurve.Evaluate(1f - deltaTime / m\_AnimationTime);

if (deltaTime > m\_AnimationTime)

{

m\_Rising = !m\_Rising;

deltaTime = 0f;

}

deltaTime += Time.deltaTime;

}

yield return null;

}

}

}

}

**TouchInformation.cs:**

namespace CustomInput.Information

{

using System;

using UnityEngine;

[Serializable]

public class InputInformation

{

public float duration;

}

[Serializable]

public class TouchInformation : InputInformation

{

public Vector2 position;

}

[Serializable]

public class DragInformation : InputInformation

{

public Vector2 origin;

public Vector2 end;

/// <summary>

/// The delta in mouse position this frame

/// </summary>

public Vector2 delta;

/// <summary>

/// The total delta in mouse position from the start of the drag

/// </summary>

public Vector2 totalDelta;

}

}

**InputEffects.cs:**

namespace CustomInput

{

using System.Linq;

using CustomParticleSystem;

using Information;

using Library;

using UnityEngine;

public class InputEffects : MonoSingleton<InputEffects>

{

[SerializeField]

private Canvas m\_OverlayCanvas;

[SerializeField]

private ParticleSystem2D m\_ParticleSystem2DPrefab;

private ParticleSystem2D m\_ParticleSystem2D;

protected override void OnAwake()

{

GameManager.self.onSceneLoaded.AddListener(Init);

InputManager.self.onHold.AddListener(OnHold);

InputManager.self.onDrag.AddListener(OnDrag);

Init();

DontDestroyOnLoad(this);

}

private void OnHold(TouchInformation touchInformation)

{

UpdatePosition(touchInformation.position);

}

private void OnDrag(DragInformation dragInformation)

{

UpdatePosition(dragInformation.end);

}

private void UpdatePosition(Vector2 newPosition)

{

//var touches = UnityEngine.Input.touches;

//if (!touches.Any())

// return;

//var newPosition = Camera.main.ScreenToWorldPoint(touches.First().position);

if (m\_ParticleSystem2D != null)

m\_ParticleSystem2D.transform.position = newPosition;

}

private void Init()

{

m\_OverlayCanvas =

FindObjectsOfType<Canvas>().

FirstOrDefault(canvas => canvas.renderMode == RenderMode.ScreenSpaceOverlay);

if (m\_OverlayCanvas == null)

return;

m\_ParticleSystem2D = Instantiate(m\_ParticleSystem2DPrefab);

m\_ParticleSystem2D.transform.SetParent(m\_OverlayCanvas.transform, false);

}

}

}

**InputManager.cs:**

namespace CustomInput

{

using System;

using Information;

using Library;

using UnityEngine;

using UnityEngine.Events;

[Serializable]

public class TouchEvent : UnityEvent<TouchInformation> { }

[Serializable]

public class DragEvent : UnityEvent<DragInformation> { }

public class InputManager : MonoSingleton<InputManager>

{

[SerializeField]

private float m\_DragDeadzone;

[SerializeField, Space]

private TouchEvent m\_OnPress = new TouchEvent();

[SerializeField]

private TouchEvent m\_OnRelease = new TouchEvent();

[SerializeField]

private TouchEvent m\_OnHold = new TouchEvent();

[SerializeField]

private DragEvent m\_OnBeginDrag = new DragEvent();

[SerializeField]

private DragEvent m\_OnDrag = new DragEvent();

[SerializeField]

private DragEvent m\_OnEndDrag = new DragEvent();

private Vector2 m\_PreviousPosition;

private float m\_CurrentHoldDuration;

private float m\_CurrentTotalDragDistance;

private Vector2 m\_PressPosition;

private bool m\_Dragging;

public float dragDeadzone { get { return m\_DragDeadzone; } }

public TouchEvent onPress { get { return m\_OnPress; } }

public TouchEvent onRelease { get { return m\_OnRelease; } }

public TouchEvent onHold { get { return m\_OnHold; } }

public DragEvent onBeginDrag { get { return m\_OnBeginDrag; } }

public DragEvent onDrag { get { return m\_OnDrag; } }

public DragEvent onEndDrag { get { return m\_OnEndDrag; } }

protected override void OnAwake()

{

DontDestroyOnLoad(gameObject);

}

private void Update()

{

if (Input.GetMouseButtonDown(0))

{

//Debug.Log("Press");

m\_PressPosition = Input.mousePosition;

m\_OnPress.Invoke(

new TouchInformation { duration = 0f, position = m\_PressPosition });

m\_CurrentHoldDuration = 0f;

}

else if (Input.GetMouseButton(0))

{

if (!m\_Dragging &&

Vector2.Distance(m\_PressPosition, Input.mousePosition) >= m\_DragDeadzone)

{

//Debug.Log("Begin Drag");

m\_CurrentTotalDragDistance = Vector2.Distance(m\_PressPosition, Input.mousePosition);

m\_OnBeginDrag.Invoke(

new DragInformation

{

origin = m\_PressPosition,

end = Input.mousePosition,

duration = 0f,

delta = (Vector2)Input.mousePosition - m\_PressPosition,

totalDelta = (Vector2)Input.mousePosition - m\_PressPosition,

});

m\_Dragging = true;

}

else if (m\_Dragging)

{

//Debug.Log("Drag");

m\_CurrentTotalDragDistance += Vector2.Distance(m\_PreviousPosition, Input.mousePosition);

m\_OnDrag.Invoke(

new DragInformation

{

origin = m\_PressPosition,

end = Input.mousePosition,

duration = m\_CurrentHoldDuration,

delta = (Vector2)Input.mousePosition - m\_PreviousPosition,

totalDelta = (Vector2)Input.mousePosition - m\_PressPosition,

});

}

else

{

//Debug.Log("Hold");

m\_OnHold.Invoke(

new TouchInformation

{

duration = m\_CurrentHoldDuration,

position = Input.mousePosition

});

}

m\_CurrentHoldDuration += Time.deltaTime;

}

else if (Input.GetMouseButtonUp(0))

{

if (!m\_Dragging)

{

//Debug.Log("Release");

m\_OnRelease.Invoke(

new TouchInformation

{

duration = m\_CurrentHoldDuration,

position = m\_PressPosition

});

}

else

{

//Debug.Log("End Drag");

m\_OnEndDrag.Invoke(

new DragInformation

{

origin = m\_PressPosition,

end = Input.mousePosition,

duration = m\_CurrentHoldDuration,

delta = (Vector2)Input.mousePosition - m\_PreviousPosition,

totalDelta = (Vector2)Input.mousePosition - m\_PressPosition,

});

}

m\_CurrentTotalDragDistance = 0f;

m\_CurrentHoldDuration = 0f;

m\_Dragging = false;

}

m\_PreviousPosition = Input.mousePosition;

}

}

}

**Particle2D.cs:**

namespace CustomParticleSystem

{

using System.Collections;

using UnityEngine;

using UnityEngine.UI;

[RequireComponent(typeof(RectTransform), typeof(Image))]

public class Particle2D : MonoBehaviour

{

private RectTransform m\_RectTransform;

private Image m\_Image;

public Vector2 velocity;

public Vector2 friction;

public float duration;

private void Awake()

{

m\_RectTransform = GetComponent<RectTransform>();

m\_Image = GetComponent<Image>();

}

private void Start()

{

m\_RectTransform.anchoredPosition += velocity;

StartCoroutine(DestroyRoutine());

}

private void FixedUpdate()

{

m\_RectTransform.anchoredPosition += velocity;

Mathf.MoveTowards(velocity.x, 0f, friction.x);

Mathf.MoveTowards(velocity.y, 0f, friction.y);

}

private IEnumerator DestroyRoutine()

{

var deltaTime = 0f;

while (deltaTime < duration)

{

m\_Image.color =

new Color(

m\_Image.color.r,

m\_Image.color.g,

m\_Image.color.b,

1f - deltaTime / duration);

deltaTime += Time.deltaTime;

yield return null;

}

Destroy(gameObject);

}

}

}

**ParticleSystem2D.cs:**

namespace CustomParticleSystem

{

using System;

using CustomInput;

using CustomInput.Information;

using UnityEngine;

[RequireComponent(typeof(RectTransform))]

public class ParticleSystem2D : MonoBehaviour

{

[SerializeField]

private Particle2D m\_Particle2DPrefab;

[Space, SerializeField, Range(0.1f, 3f)]

private float m\_ParticleDuration;

private GameObject m\_ParticleAnchor;

private const string RANDOM\_KEY = "ParticleSystem2D";

private void Awake()

{

InputManager.self.onHold.AddListener(OnHold);

InputManager.self.onDrag.AddListener(OnDrag);

}

private void Start()

{

m\_ParticleAnchor = new GameObject("Particle Anchor");

m\_ParticleAnchor.transform.SetParent(transform.parent, false);

}

private void OnHold(TouchInformation dragInformation)

{

CreateParticles();

}

private void OnDrag(DragInformation dragInformation)

{

CreateParticles();

}

private void CreateParticles()

{

var newParticle2D = Instantiate(m\_Particle2DPrefab);

newParticle2D.transform.SetParent(m\_ParticleAnchor.transform, false);

newParticle2D.transform.position = transform.position;

var angle = RandomManager.self.Range(RANDOM\_KEY, 0f, 360f) \* Mathf.Deg2Rad;

newParticle2D.velocity =

new Vector2(Mathf.Cos(angle) \* 7f, Mathf.Sin(angle) \* 7f);

newParticle2D.friction = new Vector2(1f, 1f);

newParticle2D.duration = m\_ParticleDuration;

}

}

}

**BaseItem.cs:**

/\* Script Info - Script Name: BaseItem.cs, Created by: Brock Barlow, This script is used to handle items. \*/

namespace Items

{

using System;

using UnityEngine;

[Serializable]

public abstract class BaseItem

{

protected bool m\_Alive; //states if the item can be destroied or not

protected float m\_Age; //current durabilities max value.

public float durability; //how long the item will last.

public float modifier; //value that will "modify" player attribute(s).

public bool Alive { get { return m\_Alive; } }

public virtual void UseItem() {}

public virtual void UpdateSelf() {}

}

[Serializable]

public class InstantItem : BaseItem

{

public InstantItem()

{

}

public InstantItem(float mod) { modifier = mod; }

public override void UseItem()

{

var healValue = GameManager.self.playerData.health.value \* modifier;

GameManager.self.playerData.health.modifier += healValue;

if (GameManager.self.playerData.health.modifier > 0) { GameManager.self.playerData.health.modifier = 0; }

m\_Alive = false;

}

}

//for instant items, just remove them. do not destroy them.

[Serializable]

public abstract class TurnBased : BaseItem { public override void UpdateSelf() { m\_Age++; } }

[Serializable]

public class TurnBuff : TurnBased

{

public bool itemStatType;

public TurnBuff()

{

}

public TurnBuff(float dur, float mod, bool type) { durability = dur; modifier = mod; itemStatType = type; }

public override void UseItem()

{

switch (itemStatType)

{

case true: //attack buff case

var attackValue = GameManager.self.playerData.attack.value \* modifier;

GameManager.self.playerData.attack.modifier += attackValue;

break;

case false: //defense buff case

var defenseValue = GameManager.self.playerData.defense.value \* modifier;

GameManager.self.playerData.defense.modifier += defenseValue;

break;

}

}

public override void UpdateSelf()

{

base.UpdateSelf();

if (m\_Age < durability) { m\_Alive = true; }

switch (itemStatType)

{

case true: //attack buff case

GameManager.self.playerData.attack.modifier -= GameManager.self.playerData.attack.value \* modifier;

m\_Alive = false;

break;

case false: //defense buff case

GameManager.self.playerData.defense.modifier -= GameManager.self.playerData.defense.value \* modifier;

m\_Alive = false;

break;

default:

m\_Alive = false;

break;

}

}

}

[Serializable]

public abstract class TimeBased : BaseItem { public override void UpdateSelf() { m\_Age += Time.deltaTime; } }

[Serializable]

public class TimeBuff : TimeBased

{

public bool itemStatType;

public TimeBuff()

{

}

public TimeBuff(float dur, float mod, bool type) { durability = dur; modifier = mod; itemStatType = type; }

public override void UseItem()

{

switch (itemStatType)

{

case true: //attack buff case

var attackValue = GameManager.self.playerData.attack.value \* modifier;

GameManager.self.playerData.attack.modifier += attackValue;

break;

case false: //defense buff case

var defenseValue = GameManager.self.playerData.defense.value \* modifier;

GameManager.self.playerData.defense.modifier += defenseValue;

break;

}

}

public override void UpdateSelf()

{

base.UpdateSelf();

if (m\_Age < durability) { m\_Alive = true; }

switch (itemStatType)

{

case true: //attack buff case

GameManager.self.playerData.attack.modifier -= GameManager.self.playerData.attack.value \* modifier;

m\_Alive = false;

break;

case false: //defense buff case

GameManager.self.playerData.defense.modifier -= GameManager.self.playerData.defense.value \* modifier;

m\_Alive = false;

break;

default:

m\_Alive = false;

break;

}

}

}

}

**ItemManager.cs:**

/\* Script Info - Script Name: ItemManager.cs, Created by: Brock Barlow, This script is used to handle items. \*/

namespace Items

{

using System;

using System.Collections.Generic;

using System.IO;

using System.Xml.Serialization;

using Combat;

public class ItemManager

{

[Serializable]

public class SaveLists

{

// All different Types of Items

public List<InstantItem> instantItems = new List<InstantItem>();

public List<TimeBuff> timeBuffs = new List<TimeBuff>();

public List<TurnBuff> turnBuffs = new List<TurnBuff>();

}

private List<BaseItem> m\_Inventory = new List<BaseItem>(); //list of item(s)

private List<BaseItem> m\_ActiveList = new List<BaseItem>(); //list of item(s) being used

private List<BaseItem> m\_CombatInventory = new List<BaseItem>(); //list of item(s) being taken in to combat

private SaveLists m\_Lists = new SaveLists();

public List<BaseItem> inventory { get { return m\_Inventory; } }

[XmlIgnore] public List<BaseItem> activeList { get { return m\_ActiveList; } }

[XmlIgnore] public List<BaseItem> combatInventory { get { return m\_CombatInventory; } }

public void ItemUpdate() { activeList.RemoveAll(i => i.Alive == false); } //needs to be called after using an item.

public void AddCombatItem(BaseItem item)

{

if (!inventory.Contains(item))

return;

combatInventory.Add(item); inventory.Remove(item);

}

public void AddInventoryItem(BaseItem item)

{

if (item == null || inventory.Contains(item))

return;

inventory.Add(item);

}

public void SetItemActive(BaseItem item)

{

if (combatInventory.Contains(item)) { activeList.Add(item); combatInventory.Remove(item); } else { return; }

item.UseItem();

var itemType = item.GetType();

if (itemType == typeof(TurnBased)) { CombatManager.self.onPlayerTurn.AddListener(item.UpdateSelf); }

else if (itemType == typeof(TimeBased)) { CombatManager.self.onCombatUpdate.AddListener(item.UpdateSelf); }

}

public void RemoveCombatItem(BaseItem item)

{

if (combatInventory.Contains(item)) { inventory.Add(item); combatInventory.Remove(item); }

}

public void SaveItems(string path)

{

m\_Lists.instantItems = new List<InstantItem>();

m\_Lists.timeBuffs = new List<TimeBuff>();

m\_Lists.turnBuffs = new List<TurnBuff>();

foreach (var bi in inventory)

{

var t = bi.GetType();

if(t == typeof(InstantItem))

m\_Lists.instantItems.Add(bi as InstantItem);

if (t == typeof(TimeBuff))

m\_Lists.timeBuffs.Add(bi as TimeBuff);

if (t == typeof(TurnBuff))

m\_Lists.turnBuffs.Add(bi as TurnBuff);

}

var itemsStream = File.Create(path);

var serializer = new XmlSerializer(typeof(SaveLists));

serializer.Serialize(itemsStream, m\_Lists);

itemsStream.Close();

}

public void LoadItems(string path)

{

if(!File.Exists(path))

SaveItems(path);

var stream = new StreamReader(path);

var reader = new XmlSerializer(typeof(SaveLists));

m\_Lists = (SaveLists)reader.Deserialize(stream);

m\_Inventory = new List<BaseItem>();

m\_Lists.instantItems.ForEach(i => m\_Inventory.Add(i));

m\_Lists.turnBuffs.ForEach(i => m\_Inventory.Add(i));

m\_Lists.timeBuffs.ForEach(i => m\_Inventory.Add(i));

m\_Lists.instantItems = new List<InstantItem>();

m\_Lists.timeBuffs = new List<TimeBuff>();

m\_Lists.turnBuffs = new List<TurnBuff>();

stream.Close();

}

}

}

**GenericUnityEvents.cs:**

namespace Library

{

using System;

using UnityEngine;

using UnityEngine.Events;

[Serializable]

public class UnityBoolEvent : UnityEvent<bool> { }

[Serializable]

public class UnityIntEvent : UnityEvent<int> { }

[Serializable]

public class UnityFloatEvent : UnityEvent<float> { }

[Serializable]

public class UnityVector2Event : UnityEvent<Vector2> { }

[Serializable]

public class UnityVector3Event : UnityEvent<Vector3> { }

}

**MonoSingleton.cs:**

//////////////////////

// MonoSingleton //

//////////////////////

using UnityEngine;

namespace Library

{

using System.Linq;

public class MonoSingleton<T> : MonoBehaviour where T : MonoBehaviour

{

private static bool s\_IsQuitting;

private static T s\_Self;

public static T self

{

get

{

if (s\_IsQuitting)

return null;

if (s\_Self == null)

s\_Self = FindObjectOfType<T>();

return s\_Self;

}

}

protected MonoSingleton() { }

private void Awake()

{

if (s\_Self == null)

s\_Self = this as T;

else if (s\_Self != this)

{

Destroy(gameObject);

return;

}

s\_IsQuitting = false;

OnAwake();

}

protected virtual void OnApplicationQuit() { s\_IsQuitting = true; }

protected virtual void OnAwake() { }

}

}

**MonoNode.cs:**

namespace StageSelection

{

using System;

using UnityEngine;

using System.Collections.Generic;

using System.Linq;

using System.Xml.Serialization;

using UnityEngine.UI;

[Serializable]

public class Node

{

[XmlIgnore, NonSerialized]

public List<Node> prevNodes = new List<Node>();

[XmlIgnore, NonSerialized]

public List<Node> nextNodes = new List<Node>();

[XmlIgnore]

public Vector2 normalizedPosition;

public bool isComplete = false;

[XmlIgnore]

public string stageName { get { return worldIndex + " - " + stageNumber; } }

[XmlIgnore]

public string stageNumber;

[XmlIgnore]

public int worldIndex;

[XmlIgnore]

public List<int> enemyInts = new List<int>();

[XmlIgnore]

public uint staminaCost = 2;

}

public class MonoNode : MonoBehaviour

{

public Node node;

//Testing If the Node is active or not.

private void Start()

{

var image = GetComponent<Image>();

// If the node is done and playable

if (node.isComplete)

image.color = StageSelectionManager.self.nodeCompleted;

// If the node is playable

else if (node.prevNodes.Any(n => n.isComplete) || node.prevNodes.Count == 0)

image.color = StageSelectionManager.self.nodeUnlocked;

// Node is not playable

else

image.color = StageSelectionManager.self.nodeLocked;

}

}

}

**RotateTransform.cs:**

namespace StageSelection

{

using UnityEngine;

using UnityEngine.UI;

public class RotateTransform : MonoBehaviour

{

[SerializeField]

private float m\_RotationSpeed;

// Update is called once per frame

private void Update()

{

transform.Rotate(0f, 0f, m\_RotationSpeed \* Time.deltaTime);

}

}

}

**StageSelectionManager.cs:**

using System.Linq;

namespace StageSelection

{

using System;

using System.Collections;

using System.Collections.Generic;

using CustomInput.Information;

using UnityEngine;

using UnityEngine.Events;

using UnityEngine.UI;

[Serializable]

public class Tree

{

public List<Node> nodes = new List<Node>();

}

public class StageSelectionManager : SubManager<StageSelectionManager>

{

private List<Tree> m\_Worlds = new List<Tree>();

private int m\_Currentworld;

private Node m\_CurrentNode;

[SerializeField]

private Canvas m\_Canvas;

[SerializeField]

private Text m\_StageNameText;

[SerializeField]

private Text m\_EnemyText;

[SerializeField]

private Text m\_StaminaCostText;

[SerializeField]

private Button m\_StartComabtButton;

[SerializeField]

private RectTransform m\_NodeAnchor;

[Space, SerializeField]

private Color m\_NodeCompleted;

[SerializeField]

private Color m\_NodeUnlocked;

[SerializeField]

private Color m\_NodeLocked;

[Space, SerializeField]

private Color m\_LineFillColor;

[SerializeField]

private Color m\_LineBorderColor;

[Space, SerializeField]

private float m\_WorldSpacing;

[SerializeField]

private Vector2 m\_MaxAnchorPosition;

[SerializeField]

private Vector2 m\_MaxAnchorRubberbandPosition;

[SerializeField]

private float m\_LerpTime;

[SerializeField]

private AnimationCurve m\_RubberbandAnimationCurve;

private Vector2 m\_OffsetPosition;

public UnityEvent onStageSelectionEnd = new UnityEvent();

public GameObject nodePrefab;

public GameObject linePrefab;

public float spacingMagnitude = 1;

public Color nodeCompleted { get { return m\_NodeCompleted; } }

public Color nodeUnlocked { get { return m\_NodeUnlocked; } }

public Color nodeLocked { get { return m\_NodeLocked; } }

protected override void Init()

{

m\_Worlds =

new List<Tree>

{

new Tree

{

nodes = new List<Node>

{

new Node {stageNumber = "1", normalizedPosition = new Vector2(0, 0),

worldIndex = 1, enemyInts = new List<int> {2,2}, staminaCost = 1},

new Node {stageNumber = "2", normalizedPosition = new Vector2(0, 1),

worldIndex = 1, enemyInts = new List<int> {2,1}, staminaCost = 1},

new Node {stageNumber = "3", normalizedPosition = new Vector2(0, 2),

worldIndex = 1, enemyInts = new List<int> {2,1,2}, staminaCost = 1},

new Node {stageNumber = "Boss", normalizedPosition = new Vector2(0, 3),

worldIndex = 1, enemyInts = new List<int> {1,0,1}, staminaCost = 2},

new Node {stageNumber = "1A", normalizedPosition = new Vector2(1, 0),

worldIndex = 1, enemyInts = new List<int> {2,2,2}, staminaCost = 1},

new Node {stageNumber = "2A", normalizedPosition = new Vector2(-1, 1),

worldIndex = 1, enemyInts = new List<int> {2,1,2}, staminaCost = 1},

new Node {stageNumber = "2B", normalizedPosition = new Vector2(-2, 1),

worldIndex = 1, enemyInts = new List<int> {1,2,1}, staminaCost = 2},

new Node {stageNumber = "3A", normalizedPosition = new Vector2(1, 2),

worldIndex = 1, enemyInts = new List<int> {0}, staminaCost = 2},

}

}

,

new Tree

{

nodes = new List<Node>

{

new Node {stageNumber = "1", normalizedPosition = new Vector2(0, 0),

worldIndex = 2, enemyInts = new List<int> {1}, staminaCost = 1},

new Node {stageNumber = "2", normalizedPosition = new Vector2(0, 1),

worldIndex = 2, enemyInts = new List<int> {1,1}, staminaCost = 1},

new Node {stageNumber = "3", normalizedPosition = new Vector2(0, 2),

worldIndex = 2, enemyInts = new List<int> {1,0,1}, staminaCost = 2},

new Node {stageNumber = "Boss", normalizedPosition = new Vector2(0, 3),

worldIndex = 2, enemyInts = new List<int> {0,0,0}, staminaCost = 3},

new Node {stageNumber = "2A", normalizedPosition = new Vector2(-1, 1),

worldIndex = 2, enemyInts = new List<int> {1,2,1}, staminaCost = 2},

new Node {stageNumber = "3A", normalizedPosition = new Vector2(-1, 2),

worldIndex = 2, enemyInts = new List<int> {0,0}, staminaCost = 2},

new Node {stageNumber = "2B", normalizedPosition = new Vector2(1, 1),

worldIndex = 2, enemyInts = new List<int> {2,0,2}, staminaCost = 2},

new Node {stageNumber = "3B", normalizedPosition = new Vector2(1, 2),

worldIndex = 2, enemyInts = new List<int> {2,0,0,2}, staminaCost = 4},

}

}

};

// Make Links

// Tree 1

LinkNodes(m\_Worlds[0].nodes[0], m\_Worlds[0].nodes[1]);

LinkNodes(m\_Worlds[0].nodes[1], m\_Worlds[0].nodes[2]);

LinkNodes(m\_Worlds[0].nodes[2], m\_Worlds[0].nodes[3]);

LinkNodes(m\_Worlds[0].nodes[0], m\_Worlds[0].nodes[4]);

LinkNodes(m\_Worlds[0].nodes[1], m\_Worlds[0].nodes[5]);

LinkNodes(m\_Worlds[0].nodes[5], m\_Worlds[0].nodes[6]);

LinkNodes(m\_Worlds[0].nodes[2], m\_Worlds[0].nodes[7]);

//Link

LinkNodes(m\_Worlds[0].nodes[3], m\_Worlds[1].nodes[0]);

// Tree 2

LinkNodes(m\_Worlds[1].nodes[0], m\_Worlds[1].nodes[1]);

LinkNodes(m\_Worlds[1].nodes[1], m\_Worlds[1].nodes[2]);

LinkNodes(m\_Worlds[1].nodes[2], m\_Worlds[1].nodes[3]);

LinkNodes(m\_Worlds[1].nodes[1], m\_Worlds[1].nodes[4]);

LinkNodes(m\_Worlds[1].nodes[4], m\_Worlds[1].nodes[5]);

LinkNodes(m\_Worlds[1].nodes[1], m\_Worlds[1].nodes[6]);

LinkNodes(m\_Worlds[1].nodes[6], m\_Worlds[1].nodes[7]);

var canvas = FindObjectOfType<Canvas>();

var treePos = Vector2.zero;

//Make Node GameObjects.

foreach (var tree in m\_Worlds)

{

var nodeGameObjects = new List<GameObject>();

foreach (var n in tree.nodes)

{

var nodeObject = Instantiate(nodePrefab);

var monoNode = nodeObject.AddComponent<MonoNode>();

monoNode.node = n;

nodeObject.transform.SetParent(m\_NodeAnchor);

var nodeTransform = nodeObject.GetComponent<RectTransform>();

nodeTransform.anchoredPosition =

new Vector2(n.normalizedPosition.x \* spacingMagnitude,

n.normalizedPosition.y \* spacingMagnitude) +

treePos;

var button = nodeObject.GetComponent<Button>();

button.onClick.AddListener(() => { SetCurrentNode(n); });

nodeGameObjects.Add(nodeObject);

}

// Move to center

var sum = Vector2.zero;

var numOfObjects = 0;

foreach (var node in nodeGameObjects)

{

sum += node.GetComponent<RectTransform>().anchoredPosition;

numOfObjects++;

}

var offset = sum / numOfObjects;

foreach (var node in nodeGameObjects)

{

node.GetComponent<RectTransform>().anchoredPosition += treePos - offset;

}

treePos.x += m\_WorldSpacing;

}

// Get Player Data

var savedData = GameManager.self.playerData.worldData;

if (savedData.Count == 0)

{

GameManager.self.playerData.worldData = m\_Worlds;

}

else

{

for (var i = 0; i < savedData.Count; i++)

{

for (var j = 0; j < m\_Worlds[i].nodes.Count; j++)

{

m\_Worlds[i].nodes[j].isComplete = savedData[i].nodes[j].isComplete;

}

}

}

// Was the last node completed

if (GameManager.self.currentNode != null)

{

foreach (var tree in m\_Worlds)

{

foreach (var n in tree.nodes)

{

if (n.stageName != GameManager.self.currentNode.stageName ||

!GameManager.self.currentNode.isComplete)

continue;

n.isComplete = true;

GameManager.self.playerData.worldData = m\_Worlds;

}

}

}

// Line Renderers

var counter = 0;

foreach (var monoNode in FindObjectsOfType<MonoNode>())

{

foreach (var n in monoNode.node.nextNodes)

{

if (n.worldIndex != monoNode.node.worldIndex)

continue;

var lineObject = Instantiate(linePrefab); //Create GameObject for LineRenderer

var lineTransform = lineObject.GetComponent<RectTransform>();

lineObject.name = "Line " + counter;

counter++;

lineObject.transform.SetParent(m\_NodeAnchor);

lineObject.transform.SetAsFirstSibling();

GameObject nextNodeGameObject = null;

foreach (var g in FindObjectsOfType<MonoNode>())

{

if (g.GetComponent<MonoNode>().node == n)

nextNodeGameObject = g.gameObject;

}

if (nextNodeGameObject == null)

return;

var differance = nextNodeGameObject.transform.position - monoNode.transform.position;

var linePosition = monoNode.transform.position + (differance / 2f);

lineTransform.sizeDelta = Math.Abs(differance.x) > Math.Abs(differance.y)

? new Vector2(differance.magnitude, 20)

: new Vector2(20, differance.magnitude);

var lineBorderColor = (monoNode.node.isComplete)

? m\_LineBorderColor

: m\_LineBorderColor - new Color(0f, 0f, 0f, 0.75f);

var lineFillColor = (monoNode.node.isComplete)

? m\_LineFillColor

: m\_LineFillColor - new Color(0f, 0f, 0f, 0.75f);

lineObject.GetComponent<Image>().color = lineBorderColor;

lineObject.transform.GetChild(0).GetComponent<Image>().color = lineFillColor;

// Set Material Color

lineTransform.position = linePosition;

}

}

//Set Up UI

m\_StageNameText.text = "";

m\_EnemyText.text = "";

m\_StaminaCostText.text = "";

m\_StartComabtButton.onClick.AddListener(OnStageSelectionEnd);

m\_StartComabtButton.interactable = false;

}

private void OnStageSelectionEnd()

{

m\_StartComabtButton.interactable = false;

GameManager.self.currentNode = m\_CurrentNode;

GameManager.self.enemyIndexes = m\_CurrentNode.enemyInts;

StaminaManager.self.DamageStamina(m\_CurrentNode.staminaCost);

onStageSelectionEnd.Invoke();

}

private void LinkNodes(Node n1, Node n2)

{

n1.nextNodes.Add(n2);

n2.prevNodes.Add(n1);

}

protected override void OnDrag(DragInformation dragInfo)

{

var dragDirection = dragInfo.delta.normalized;

var scaledDelta = Vector2.right \* dragInfo.delta.x / m\_Canvas.scaleFactor;

var currentPosition = m\_NodeAnchor.anchoredPosition;

m\_OffsetPosition += scaledDelta;

m\_OffsetPosition =

new Vector2(

Mathf.Clamp(

m\_OffsetPosition.x,

GetWorldPosition(m\_Worlds.Count - 1)

- m\_MaxAnchorPosition.x - m\_MaxAnchorRubberbandPosition.x,

m\_MaxAnchorPosition.x + m\_MaxAnchorRubberbandPosition.x),

Mathf.Clamp(

m\_OffsetPosition.y,

m\_OffsetPosition.y,

m\_MaxAnchorPosition.y + m\_MaxAnchorRubberbandPosition.y));

var rubberbandPosition =

new Vector2(

Mathf.Abs(m\_OffsetPosition.x) - m\_MaxAnchorPosition.x,

Mathf.Abs(m\_OffsetPosition.y) - m\_MaxAnchorPosition.y);

if (m\_OffsetPosition.x > m\_MaxAnchorPosition.x)

{

var rubberbandDelta =

m\_MaxAnchorRubberbandPosition.x \*

m\_RubberbandAnimationCurve.Evaluate(

rubberbandPosition.x / m\_MaxAnchorRubberbandPosition.x);

currentPosition.x = m\_MaxAnchorPosition.x + rubberbandDelta;

}

else

if (m\_OffsetPosition.x < GetWorldPosition(m\_Worlds.Count - 1) - m\_MaxAnchorPosition.x)

{

var rubberbandDelta =

m\_MaxAnchorRubberbandPosition.x \*

m\_RubberbandAnimationCurve.Evaluate(

-(m\_OffsetPosition.x - GetWorldPosition(m\_Worlds.Count - 1)

+ m\_MaxAnchorPosition.x) / m\_MaxAnchorRubberbandPosition.x);

currentPosition.x =

GetWorldPosition(m\_Worlds.Count - 1) - m\_MaxAnchorPosition.x - rubberbandDelta;

}

else

currentPosition = m\_OffsetPosition;

m\_NodeAnchor.anchoredPosition = currentPosition;

StopAllCoroutines();

}

protected override void OnEndDrag(DragInformation dragInfo)

{

//TODO: Lerp Down

StartCoroutine(MoveObject());

}

private int GetWorldIndex(float xPosition)

{

return (int)((xPosition - m\_WorldSpacing / 2f) / -m\_WorldSpacing);

}

private float GetWorldPosition(int worldIndex)

{

return worldIndex \* -m\_WorldSpacing;

}

private IEnumerator MoveObject()

{

var newPosition =

new Vector2(

GetWorldPosition(GetWorldIndex(m\_NodeAnchor.anchoredPosition.x)),

m\_NodeAnchor.anchoredPosition.y);

var deltaTime = 0f;

while (deltaTime < m\_LerpTime)

{

m\_OffsetPosition =

Vector2.Lerp(

m\_NodeAnchor.anchoredPosition, newPosition, deltaTime / m\_LerpTime);

m\_NodeAnchor.anchoredPosition = m\_OffsetPosition;

deltaTime += Time.deltaTime;

yield return null;

}

}

private void SetCurrentNode(Node n)

{

m\_CurrentNode = n;

DisplayInformation();

}

private void DisplayInformation()

{

m\_StageNameText.text = "Stage: " + m\_CurrentNode.stageName;

m\_StaminaCostText.text = "Stamina Cost: " + m\_CurrentNode.staminaCost;

// Enemy Display - Text

m\_EnemyText.text = "";

var enemyInstances = new Dictionary<string, int>();

foreach (var index in m\_CurrentNode.enemyInts)

{

var name = GameManager.self.enemyPrefabList[index].name;

if (!enemyInstances.ContainsKey(name))

enemyInstances.Add(name, 0);

enemyInstances[name] += 1;

}

foreach (var key in enemyInstances)

{

m\_EnemyText.text += key.Key + " - x" + key.Value + "\n";

}

// Not Enough Stamina clause

if (StaminaManager.self.value < m\_CurrentNode.staminaCost)

{

m\_StartComabtButton.interactable = false;

m\_StartComabtButton.gameObject.GetComponentInChildren<Text>().text = "Low\nPower";

}

else if (m\_CurrentNode.isComplete || m\_CurrentNode.prevNodes.Any(n => n.isComplete)

|| m\_CurrentNode.prevNodes.Count == 0)

{

m\_StartComabtButton.interactable = true;

m\_StartComabtButton.gameObject.GetComponentInChildren<Text>().text = " Engage!";

}

else

{

m\_StartComabtButton.interactable = false;

m\_StartComabtButton.gameObject.GetComponentInChildren<Text>().text = "Locked";

}

}

}

}

**StaminaDisplay.cs:**

using UnityEngine;

using UnityEngine.UI;

namespace StageSelection

{

public class StaminaDisplay : MonoBehaviour

{

private Text m\_Text;

[SerializeField]

private Image m\_FillImage;

// Use this for initialization

void Start ()

{

m\_Text = transform.GetComponentInChildren<Text>();

}

// Update is called once per frame

void Update ()

{

m\_FillImage.fillAmount = (float)StaminaManager.self.value / StaminaManager.self.maxValue;

m\_Text.text = StaminaManager.self.value + "/" + StaminaManager.self.maxValue;

}

}

}

**SwitchPanels.cs:**

using UnityEngine;

namespace StageSelection

{

public class SwitchPanels : MonoBehaviour

{

[SerializeField]

private GameObject m\_PopUpMenu;

private Transform m\_DataPanel;

private Transform m\_ItemPanel;

// Use this for initialization

private void Awake ()

{

m\_PopUpMenu.SetActive(false);

m\_ItemPanel = transform.GetChild(0);

m\_DataPanel = transform.GetChild(1);

}

// Update is called once per frame

public void SwitchPanelTabs()

{

var temp = 0;

temp = m\_ItemPanel.GetSiblingIndex();

m\_ItemPanel.SetSiblingIndex(m\_DataPanel.GetSiblingIndex());

m\_DataPanel.SetSiblingIndex(temp);

var boolin = (m\_ItemPanel.GetSiblingIndex() != 0);

m\_PopUpMenu.SetActive(boolin);

}

}

}

**RandomGeneration.cs:**

namespace TEST

{

using System.Globalization;

using UnityEngine;

using UnityEngine.UI;

public class RandomGeneration : MonoBehaviour

{

[SerializeField]

private Button m\_SeedButton;

[SerializeField]

private Button m\_GenerateButton;

[SerializeField]

private Text m\_SeedText;

[SerializeField]

private Text m\_GenerateText;

// Use this for initialization

private void Awake()

{

m\_SeedButton.onClick.AddListener(OnSeedClick);

m\_GenerateButton.onClick.AddListener(OnGenerateClick);

}

private void OnSeedClick()

{

int seed;

if (!int.TryParse(m\_SeedText.text, out seed))

return;

Random.InitState(seed);

}

private void OnGenerateClick()

{

m\_GenerateText.text = Random.value.ToString(CultureInfo.InvariantCulture);

}

}

}

**Attribute.cs:**

using System;

using UnityEngine;

using UnityEngine.Events;

[Serializable]

public class Attribute

{

[SerializeField]

private float m\_Modifier;

[SerializeField]

private float m\_Coefficient = 1f;

private UnityEvent m\_OnTotalValueChanged = new UnityEvent();

public float value;

public float coefficient

{

get { return m\_Coefficient; }

set

{

m\_Coefficient = value;

m\_OnTotalValueChanged.Invoke();

}

}

public float modifier

{

get { return m\_Modifier; }

set

{

m\_Modifier = value;

m\_OnTotalValueChanged.Invoke();

}

}

public UnityEvent onTotalValueChanged { get { return m\_OnTotalValueChanged; } }

public float totalValue { get { return value \* m\_Coefficient + m\_Modifier; } }

}

**AudioManager.cs:**

using StageSelection;

using System.Collections.Generic;

using CustomInput.Information;

using UnityEngine;

using UnityEngine.EventSystems;

public class AudioManager : SubManager<AudioManager>

{

public List<AudioClip> musicList;

public AudioClip clickSound;

public AudioClip dragSound;

public AudioClip engageSound;

public AudioClip victorySound;

public AudioClip defeatSound;

private AudioSource m\_MusicSource;

private AudioSource m\_MenuSource;

private bool m\_MuteMusic;

private bool m\_MuteSoundEffects;

public void ChangeMusic(int sceneIndex)

{

if (m\_MusicSource == null)

{

m\_MusicSource = gameObject.AddComponent<AudioSource>();

m\_MusicSource.loop = true;

}

m\_MusicSource.clip = musicList[sceneIndex];

m\_MusicSource.Play();

}

public void PlayEndBattleSound(bool victory)

{

m\_MusicSource.clip = (victory) ? victorySound : defeatSound;

m\_MusicSource.Play();

}

protected override void OnPress(TouchInformation touchInfo)

{

if (m\_MuteSoundEffects)

return;

m\_MenuSource.clip = clickSound;

if (EventSystem.current != null)

{

if (EventSystem.current.currentSelectedGameObject == null)

return;

if (EventSystem.current.currentSelectedGameObject.name == "Combat Button")

{

m\_MenuSource.clip = engageSound;

m\_MenuSource.Play();

return;

}

m\_MenuSource.Play();

return;

}

// See if Hit certain GameObject

var ray = Camera.main.ScreenPointToRay(touchInfo.position);

var hit = new RaycastHit(); // Make a Hit

//See if the ray hit anything

if (!(Physics.Raycast(ray.origin, ray.direction, out hit))) // If not, stop execution

return;

var obj = hit.transform.gameObject;

if(obj.GetComponent<MonoNode>() != null)

m\_MenuSource.Play();

}

public void PlayDragSound()

{

if (m\_MuteSoundEffects)

return;

m\_MenuSource.clip = dragSound;

m\_MenuSource.Play();

}

public void MuteMusicToggle()

{

// Mute

if (m\_MuteMusic == false)

{

m\_MusicSource.Pause();

}

// Unmute

else

{

m\_MusicSource.Play();

}

m\_MuteMusic = !m\_MuteMusic;

}

public void MuteSoundsToggle()

{

m\_MuteSoundEffects = !m\_MuteSoundEffects;

}

protected override void Init()

{

DontDestroyOnLoad(gameObject);

m\_MenuSource = gameObject.AddComponent<AudioSource>();

m\_MenuSource.clip = clickSound;

ChangeMusic(0);

}

}

**BarLines.cs:**

using UnityEngine;

using UnityEngine.UI;

public class BarLines : Graphic

{

private enum BarType

{

Health,

Shield,

}

[Space, SerializeField]

private Image m\_ParentImage;

[SerializeField]

private float m\_LineWidth = 1f;

[SerializeField]

private int m\_SegmentValue = 50;

[SerializeField]

private BarType m\_BarType;

protected override void Awake()

{

base.Awake();

if (!Application.isPlaying)

return;

GameManager.self.playerData.health.onTotalValueChanged.AddListener(OnTotalValueChanged);

GameManager.self.playerData.defense.onTotalValueChanged.AddListener(OnTotalValueChanged);

OnTotalValueChanged();

}

private void OnTotalValueChanged()

{

rectTransform.anchorMax =

new Vector2(

m\_ParentImage.fillAmount,

rectTransform.anchorMax.y);

SetVerticesDirty();

}

protected override void OnPopulateMesh(VertexHelper vh)

{

vh.Clear();

if (!Application.isPlaying || m\_ParentImage == null || m\_ParentImage.fillAmount <= 0f)

return;

var vertex = UIVertex.simpleVert;

vertex.color = color;

var startingPoint = rectTransform.anchorMin;

startingPoint -= rectTransform.pivot;

startingPoint =

new Vector2(

rectTransform.rect.width \* startingPoint.x,

rectTransform.rect.height \* startingPoint.y);

startingPoint -= new Vector2(m\_LineWidth / 2f, 0f);

var totalValue =

m\_BarType == BarType.Health ?

GameManager.self.playerData.health.totalValue :

GameManager.self.playerData.defense.totalValue;

var parentWidth = m\_ParentImage.rectTransform.rect.width \* m\_ParentImage.fillAmount;

var lineCount = totalValue / m\_SegmentValue;

var spacing = m\_SegmentValue / totalValue \* parentWidth;

for (var i = 0; i <= lineCount; ++i)

{

var currentPosition = startingPoint + i \* new Vector2(spacing, 0f);

vertex.position = currentPosition;

vh.AddVert(vertex);

vertex.position = currentPosition + Vector2.up \* rectTransform.rect.height;

vh.AddVert(vertex);

vertex.position = currentPosition + Vector2.right \* m\_LineWidth;

vh.AddVert(vertex);

vertex.position =

currentPosition + Vector2.right \* m\_LineWidth + Vector2.up \* rectTransform.rect.height;

vh.AddVert(vertex);

var vertexOffset = i \* 4;

vh.AddTriangle(0 + vertexOffset, 1 + vertexOffset, 2 + vertexOffset);

vh.AddTriangle(1 + vertexOffset, 3 + vertexOffset, 2 + vertexOffset);

}

}

}

**GameManager.cs:**

using System;

using System.Collections;

using System.Collections.Generic;

using System.IO;

using System.Xml.Serialization;

using Combat;

using Items;

using Library;

using StageSelection;

using UnityEngine.Events;

using UnityEngine.SceneManagement;

using UnityEngine.UI;

using UnityEngine;

public class GameManager : MonoSingleton<GameManager>

{

public enum GameState

{

Title,

StageSelection,

Combat,

Credits,

}

[SerializeField]

private UnityEvent m\_OnSceneLoaded = new UnityEvent();

private string m\_PlayerSavePath;

private string m\_InventorySavePath;

[SerializeField]

private List<GameObject> m\_EnemyPrefabList = new List<GameObject>();

public GameState gameState;

public Node currentNode;

public PlayerData playerData = new PlayerData();

public List<int> enemyIndexes = new List<int>();

public List<GameObject> enemyPrefabList { get { return m\_EnemyPrefabList; } }

public UnityEvent onSceneLoaded { get { return m\_OnSceneLoaded; } }

protected override void OnAwake()

{

DontDestroyOnLoad(gameObject);

m\_InventorySavePath = Application.persistentDataPath + "/Inventory.xml";

m\_PlayerSavePath = Application.persistentDataPath + "/PlayerData.xml";

if (File.Exists(m\_PlayerSavePath))

LoadPlayer();

else

{

playerData = new PlayerData(200, 10, 10);

playerData.staminaInformation = new StaminaInformation

{

value = 10,

maxValue = 100,

timeLastPlayed = DateTime.Now.ToString()

};

StaminaManager.self.SetStam();

SavePlayer();

}

playerData.playerLevelSystem.playerLevelInfo.level =

(playerData.playerLevelSystem.playerLevelInfo.level <= 0)

? 1

: playerData.playerLevelSystem.playerLevelInfo.level;

playerData.playerLevelSystem.playerLevelInfo.experienceRequired =

(playerData.playerLevelSystem.playerLevelInfo.experienceRequired < 200)

? 200

: playerData.playerLevelSystem.playerLevelInfo.experienceRequired;

//instant = 25 health

//turn attack up = 7 turns 100 buff, turn defense up = 7 turns 100 buff

//time attack up = 15 secs 100 buff, time defense up = 15 secs 100 buff

for (int i = 0; i < 2; i++)

{

playerData.itemManager.AddInventoryItem(new InstantItem(0.1f));

}

gameState = (GameState)SceneManager.GetActiveScene().buildIndex;

AddSceneListeners();

//onSceneLoaded.AddListener(AddSceneListeners);

}

private void OnApplicationQuit()

{

SavePlayer();

}

private void OnCombatEnd()

{

LoadScene((int)GameState.StageSelection);

}

private void OnStageSelectionEnd()

{

LoadScene((int)GameState.Combat);

}

private void AddSceneListeners()

{

switch (gameState)

{

case GameState.Credits:

var button = FindObjectOfType<Button>();

button.onClick.AddListener(() => { LoadScene(0); });

break;

case GameState.Combat: // Combat

playerData.health.modifier = 0;

playerData.defense.modifier = 0;

CombatManager.self.onCombatEnd.AddListener(OnCombatEnd);

CombatManager.self.onPlayerTurn.AddListener(AudioManager.self.PlayDragSound);

gameState = GameState.Combat;

// Toggle Music Button

GameObject.Find("Menu Button").transform.FindChild("Icon Layout Group")

.FindChild("Music Button").gameObject.GetComponent<Button>

().onClick.AddListener(AudioManager.self.MuteMusicToggle);

// Toggle SoundEffect Button

GameObject.Find("Menu Button").transform.FindChild("Icon Layout Group")

.FindChild("Sound Effects Button").gameObject.GetComponent<Button>

().onClick.AddListener(AudioManager.self.MuteSoundsToggle);

break;

case GameState.StageSelection: // Stage Selection

StageSelectionManager.self.onStageSelectionEnd.AddListener

(OnStageSelectionEnd);

GameObject.Find("Title Button").gameObject.GetComponent<Button>().onClick.AddListener(() => { LoadScene(0); });

break;

case GameState.Title:

GameObject.Find("Play").gameObject.GetComponent<Button>().onClick.AddListener(() => { LoadScene(1); });

GameObject.Find("Credits").gameObject.GetComponent<Button>().onClick.AddListener(() => { LoadScene(3); });

break;

}

AudioManager.self.ChangeMusic((int)gameState);

}

[ContextMenu("Save Player")]

public void SavePlayer()

{

playerData.staminaInformation = new StaminaInformation

{

value = StaminaManager.self.value,

maxValue = StaminaManager.self.maxValue,

timeLastPlayed = DateTime.Now.ToString(),

};

//Saving PlayerData

var playerPath = m\_PlayerSavePath;

var playerStream = File.Create(playerPath);

var serializer = new XmlSerializer(typeof(PlayerData));

serializer.Serialize(playerStream, playerData);

playerStream.Close();

//playerData.itemManager.SaveItems(m\_InventorySavePath);

}

[ContextMenu("Load Player")]

private void LoadPlayer()

{

var reader = new XmlSerializer(typeof(PlayerData));

var file = new StreamReader(m\_PlayerSavePath);

playerData = (PlayerData)reader.Deserialize(file);

file.Close();

//playerData.itemManager.LoadItems(m\_InventorySavePath);

}

public void LoadScene(int sceneIndex)

{

if ((int)gameState != sceneIndex)

{

StartCoroutine(LoadSceneCoroutine(sceneIndex));

}

}

private IEnumerator LoadSceneCoroutine(int sceneIndex)

{

var asyncOperation = SceneManager.LoadSceneAsync(sceneIndex);

while (!asyncOperation.isDone) { yield return null; }

gameState = (GameState)sceneIndex;

AddSceneListeners();

onSceneLoaded.Invoke();

}

}

**GemSkill.cs:**

using System;

using Combat.Board;

[Serializable]

public class GemSkill

{

public GemType gemType;

public int skillLevel;

}

**InputRecorder.cs:**

using System;

using System.Collections.Generic;

using System.IO;

using CustomInput;

using CustomInput.Information;

using Library;

using UnityEngine;

public enum InputType { Start, While, End, }

[Serializable]

public class InputAction

{

public float time;

public InputType inputType;

}

[Serializable]

public class TouchAction : InputAction

{

public TouchInformation touchInformation;

}

[Serializable]

public class DragAction : InputAction

{

public DragInformation dragInformation;

}

[Serializable]

public class InputData

{

public List<TouchAction> touchActions = new List<TouchAction>();

public List<DragAction> dragActions = new List<DragAction>();

}

public class InputRecorder : MonoSingleton<InputRecorder>

{

[SerializeField]

private bool m\_Record;

private InputData m\_InputData = new InputData();

protected override void OnAwake()

{

InputManager.self.onPress.AddListener(OnPress);

InputManager.self.onHold.AddListener(OnHold);

InputManager.self.onRelease.AddListener(OnRelease);

InputManager.self.onBeginDrag.AddListener(OnBeginDrag);

InputManager.self.onDrag.AddListener(OnDrag);

InputManager.self.onEndDrag.AddListener(OnEndDrag);

DontDestroyOnLoad(this);

}

protected override void OnApplicationQuit()

{

base.OnApplicationQuit();

if (!m\_Record)

return;

var jsonData = JsonUtility.ToJson(m\_InputData);

File.WriteAllText(Application.persistentDataPath + "/InputData.json", jsonData);

}

private void OnPress(TouchInformation touchInformation)

{

if (!m\_Record)

return;

m\_InputData.touchActions.Add(

new TouchAction

{

time = Time.unscaledTime,

inputType = InputType.Start,

touchInformation =

new TouchInformation

{

duration = touchInformation.duration,

position = ScalePosition(touchInformation.position),

},

});

}

private void OnHold(TouchInformation touchInformation)

{

if (!m\_Record)

return;

m\_InputData.touchActions.Add(

new TouchAction

{

time = Time.unscaledTime,

inputType = InputType.While,

touchInformation =

new TouchInformation

{

duration = touchInformation.duration,

position = ScalePosition(touchInformation.position),

},

});

}

private void OnRelease(TouchInformation touchInformation)

{

if (!m\_Record)

return;

m\_InputData.touchActions.Add(

new TouchAction

{

time = Time.unscaledTime,

inputType = InputType.End,

touchInformation =

new TouchInformation

{

duration = touchInformation.duration,

position = ScalePosition(touchInformation.position),

},

});

}

private void OnBeginDrag(DragInformation dragInformation)

{

if (!m\_Record)

return;

m\_InputData.dragActions.Add(

new DragAction

{

time = Time.unscaledTime,

inputType = InputType.Start,

dragInformation =

new DragInformation

{

duration = dragInformation.duration,

origin = ScalePosition(dragInformation.origin),

end = ScalePosition(dragInformation.end),

delta = ScalePosition(dragInformation.delta),

totalDelta = ScalePosition(dragInformation.totalDelta),

},

});

}

private void OnDrag(DragInformation dragInformation)

{

if (!m\_Record)

return;

m\_InputData.dragActions.Add(

new DragAction

{

time = Time.unscaledTime,

inputType = InputType.While,

dragInformation =

new DragInformation

{

duration = dragInformation.duration,

origin = ScalePosition(dragInformation.origin),

end = ScalePosition(dragInformation.end),

delta = ScalePosition(dragInformation.delta),

totalDelta = ScalePosition(dragInformation.totalDelta),

},

});

}

private void OnEndDrag(DragInformation dragInformation)

{

if (!m\_Record)

return;

m\_InputData.dragActions.Add(

new DragAction

{

time = Time.unscaledTime,

inputType = InputType.End,

dragInformation =

new DragInformation

{

duration = dragInformation.duration,

origin = ScalePosition(dragInformation.origin),

end = ScalePosition(dragInformation.end),

delta = ScalePosition(dragInformation.delta),

totalDelta = ScalePosition(dragInformation.totalDelta),

},

});

}

private Vector2 ScalePosition(Vector2 position)

{

position =

new Vector2(

position.x / Screen.width,

position.y / Screen.height);

return position;

}

[ContextMenu("Delete Data")]

private void DeleteData()

{

File.Delete(Application.persistentDataPath + "/InputData.txt");

}

}

**InputReplay.cs:**

using System;

using System.Collections.Generic;

using System.IO;

using System.Linq;

using CustomInput;

using Library;

using UnityEngine;

using UnityEngine.EventSystems;

using UnityEngine.UI;

public class InputReplay : MonoSingleton<InputReplay>

{

[SerializeField]

private bool m\_Replay;

[SerializeField]

private bool m\_DrawData;

private InputData m\_InputData;

protected override void OnAwake()

{

var jsonData = File.ReadAllText(Application.persistentDataPath + "/InputData.json");

m\_InputData = JsonUtility.FromJson<InputData>(jsonData);

}

private void Update()

{

if (!m\_Replay)

return;

if (!m\_InputData.touchActions.Any() && !m\_InputData.dragActions.Any())

return;

var currentTouchAction = m\_InputData.touchActions.FirstOrDefault();

var currentDragAction = m\_InputData.dragActions.FirstOrDefault();

if (currentTouchAction != null && Time.unscaledTime >= currentTouchAction.time)

{

ProcessTouch(currentTouchAction);

m\_InputData.touchActions.Remove(currentTouchAction);

}

if (currentDragAction != null && Time.unscaledTime >= currentDragAction.time)

{

ProcessDrag(currentDragAction);

m\_InputData.dragActions.Remove(currentDragAction);

}

}

private void OnGUI()

{

if (m\_DrawData)

DrawData();

}

private void DrawData()

{

CreateLineMaterial();

// Apply the line material

lineMaterial.SetPass(0);

GL.PushMatrix();

{

GL.MultMatrix(transform.localToWorldMatrix);

GL.Begin(GL.QUADS);

{

foreach (var dragAction in m\_InputData.dragActions.Where(action => action.inputType == InputType.End))

{

var origin = dragAction.dragInformation.origin;

origin.y = Screen.height - origin.y;

var end = dragAction.dragInformation.end;

end.y = Screen.height - end.y;

var direction = new Vector2(end.x - origin.x, end.y - origin.y).normalized;

GL.Color(Color.green);

GL.Vertex3(origin.x, origin.y, 0f);

GL.Vertex3(origin.x + 15f \* direction.y, origin.y + 15f \* -direction.x, 0f);

GL.Color(Color.red);

GL.Vertex3(end.x + 15f \* direction.y, end.y + 15f \* -direction.x, 0f);

GL.Vertex3(end.x, end.y, 0f);

}

}

GL.End();

}

GL.PopMatrix();

}

private static void ProcessTouch(TouchAction touchAction)

{

touchAction.touchInformation.position = ScalePosition(touchAction.touchInformation.position);

switch (touchAction.inputType)

{

case InputType.Start:

InputManager.self.onPress.Invoke(touchAction.touchInformation);

break;

case InputType.While:

InputManager.self.onHold.Invoke(touchAction.touchInformation);

break;

case InputType.End:

InputManager.self.onRelease.Invoke(touchAction.touchInformation);

PushButtons(touchAction.touchInformation.position);

break;

default:

throw new ArgumentOutOfRangeException();

}

}

private static void ProcessDrag(DragAction dragAction)

{

dragAction.dragInformation.origin = ScalePosition(dragAction.dragInformation.origin);

dragAction.dragInformation.end = ScalePosition(dragAction.dragInformation.end);

dragAction.dragInformation.delta = ScalePosition(dragAction.dragInformation.delta);

dragAction.dragInformation.totalDelta = ScalePosition(dragAction.dragInformation.totalDelta);

switch (dragAction.inputType)

{

case InputType.Start:

InputManager.self.onBeginDrag.Invoke(dragAction.dragInformation);

break;

case InputType.While:

InputManager.self.onDrag.Invoke(dragAction.dragInformation);

break;

case InputType.End:

InputManager.self.onEndDrag.Invoke(dragAction.dragInformation);

PushButtons(dragAction.dragInformation.end);

break;

default:

throw new ArgumentOutOfRangeException();

}

}

private static Vector2 ScalePosition(Vector2 position)

{

position = Vector2.Scale(position, new Vector2(Screen.width, Screen.height));

return position;

}

private static void PushButtons(Vector2 position)

{

var pointerEventData =

new PointerEventData(EventSystem.current) { position = position };

var hits = new List<RaycastResult>();

EventSystem.current.RaycastAll(pointerEventData, hits);

if (!hits.Any())

return;

var buttons = hits.Select(hit => hit.gameObject.GetComponent<Button>());

foreach (var button in buttons)

{

if (button != null)

button.onClick.Invoke();

}

}

static Material lineMaterial;

static void CreateLineMaterial()

{

if (!lineMaterial)

{

// Unity has a built-in shader that is useful for drawing

// simple colored things.

Shader shader = Shader.Find("Hidden/Internal-Colored");

lineMaterial = new Material(shader);

lineMaterial.hideFlags = HideFlags.HideAndDontSave;

// Turn on alpha blending

lineMaterial.SetInt("\_SrcBlend", (int)UnityEngine.Rendering.BlendMode.SrcAlpha);

lineMaterial.SetInt("\_DstBlend", (int)UnityEngine.Rendering.BlendMode.OneMinusSrcAlpha);

// Turn backface culling off

lineMaterial.SetInt("\_Cull", (int)UnityEngine.Rendering.CullMode.Off);

// Turn off depth writes

lineMaterial.SetInt("\_ZWrite", 0);

}

}

}

**LevelSystem.cs:**

/\* Script Info - Script Name: LevelSystem.cs, Created By: Brock Barlow, This script is used to level up the player. \*/

using System;

[Serializable]

public class LevelSystem

{

[Serializable]

public struct LevelInfo

{

public uint level; //player's level

public uint currentExperience; //player's current exp amount

public uint experienceRequired; //the exp amount the player needs to level up

public uint experienceNeeded { get { return experienceRequired - currentExperience; } } //experienceRequired - currentExperience

}

public LevelInfo playerLevelInfo;

private LevelInfo CalculateLevel(uint exp)

{

//base\_xp \* (level\_to\_get ^ factor);

//base\_xp = constant, how much xp needed to levelup.

//level\_to\_get = level aiming for.

//factor = constant, how much of an increase of xp needed for levelup.

var tempExperience = exp;

uint tempExpeienceRequired = 0;

const uint c\_baseExperience = 200; //base\_xp //testing value from research results

uint tempLevel = 1; //level\_to\_get //player needs to start at level one

const uint c\_factor = 2; //factor //testing value from research results

var experienceRequiredFormula = c\_baseExperience \* ((tempLevel + 1) ^ c\_factor);

for (tempLevel = 1; tempExperience > experienceRequiredFormula; tempLevel++)

{

tempExperience -= experienceRequiredFormula;

tempExpeienceRequired += experienceRequiredFormula;

experienceRequiredFormula = c\_baseExperience \* ((tempLevel + 2) ^ c\_factor); //recalculate formula value

}

tempExpeienceRequired += experienceRequiredFormula;

var levelInfo = new LevelInfo

{

level = tempLevel, //player's level

currentExperience = exp, //player's current exp amout

experienceRequired = tempExpeienceRequired, //the exp amount the player needs to level up

};

return levelInfo;

}

//fight exp

public void IsLeveledUp(uint modifier)

{

var tempCurrentExperience = playerLevelInfo;

var finalTotal = CalculateLevel(playerLevelInfo.currentExperience + modifier);

if (finalTotal.level != tempCurrentExperience.level)

{

uint differenceInLevel;

for (differenceInLevel = finalTotal.level - tempCurrentExperience.level; differenceInLevel > 0; differenceInLevel--)

{

GameManager.self.playerData.health.value += 10; //health stat change

GameManager.self.playerData.attack.value += 2; //attack stat change

GameManager.self.playerData.defense.value += 2; //defense stat change

StaminaManager.self.maxValue += 2; //stamina stat change

}

}

playerLevelInfo = finalTotal;

}

}

**PlayerData.cs:**

using System;

using System.Collections.Generic;

using System.Xml.Serialization;

using Combat.Board;

using Items;

using UnityEngine;

using UnityEngine.Events;

using Tree = StageSelection.Tree;

[Serializable]

public class PlayerData

{

public PlayerData()

{

health = new Attribute();

attack = new Attribute();

defense = new Attribute();

health.value = 200;

attack.value = 10;

defense.value = 25;

health.coefficient = 1;

defense.coefficient = 1;

attack.coefficient = 1;

resistances = new List<GemType>();

weaknesses = new List<GemType>();

}

public PlayerData(float hel, float att, float def)

{

health = new Attribute();

attack = new Attribute();

defense = new Attribute();

health.value = hel;

attack.value = att;

defense.value = def;

health.coefficient = 1;

defense.coefficient = 1;

attack.coefficient = 1;

resistances = new List<GemType>();

weaknesses = new List<GemType>();

}

private UnityEvent m\_OnTakeDamage = new UnityEvent();

public Attribute health;

public Attribute attack;

public Attribute defense;

public float decayRate = 1f;

public List<GemType> resistances;

public List<GemType> weaknesses;

public List<GemSkill> gemSkills = new List<GemSkill>();

public List<Tree> worldData = new List<Tree>();

public StaminaInformation staminaInformation = new StaminaInformation();

public LevelSystem playerLevelSystem = new LevelSystem();

public UnityEvent onTakeDamage { get { return m\_OnTakeDamage; } }

[XmlIgnore]

public ItemManager itemManager = new ItemManager();

public void TakeDamage(float damage, GemType gemType)

{

var finalDamage = damage - defense.totalValue;

defense.modifier -= damage;

if (defense.modifier < -defense.value) { defense.modifier = -defense.value; }

if (finalDamage <= 0) return;

if (resistances.Contains(gemType)) { finalDamage \*= .75f; }

else if (weaknesses.Contains(gemType)) { finalDamage \*= 1.25f; }

health.modifier -= finalDamage;

m\_OnTakeDamage.Invoke();

}

public void DecayShield()

{

if (defense.modifier > 0) { defense.modifier -= decayRate \* Time.deltaTime; if (defense.modifier < 0) { defense.modifier = 0; } }

if (defense.modifier > defense.value \* 20 - defense.value) { defense.modifier = defense.value \* 20 - defense.value; }

}

}

**RandomManager.cs:**

using System;

using System.Collections.Generic;

using System.IO;

using System.Linq;

using Library;

using UnityEngine;

using Random = System.Random;

public class RandomManager : MonoSingleton<RandomManager>

{

private class RandomGenerator

{

public Random randomizer;

public int seed;

}

[Serializable]

private class RandomGeneratorPair

{

public string key;

public int value;

}

[SerializeField]

private class RandomGeneratorPairList

{

public List<RandomGeneratorPair> randomGeneratorPairs;

}

[SerializeField]

private bool m\_StaticSeed;

private string m\_Path = "/RandomManager.json";

private Dictionary<string, RandomGenerator> m\_RandomGenerators =

new Dictionary<string, RandomGenerator>();

protected override void OnAwake()

{

if (!m\_StaticSeed || !File.Exists(Application.persistentDataPath + m\_Path))

return;

var jsonData = File.ReadAllText(Application.persistentDataPath + m\_Path);

var seedList = JsonUtility.FromJson<RandomGeneratorPairList>(jsonData);

m\_RandomGenerators =

seedList.randomGeneratorPairs.ToDictionary(

pair => pair.key,

pair =>

new RandomGenerator

{

randomizer = new Random(pair.value),

seed = pair.value

});

DontDestroyOnLoad(this);

}

protected override void OnApplicationQuit()

{

base.OnApplicationQuit();

var seedList =

new RandomGeneratorPairList

{

randomGeneratorPairs =

m\_RandomGenerators.Select(

pair =>

new RandomGeneratorPair

{

key = pair.Key,

value = pair.Value.seed,

}).ToList()

};

var jsonData = JsonUtility.ToJson(seedList);

File.WriteAllText(Application.persistentDataPath + m\_Path, jsonData);

}

public int Range(string key, int min, int max)

{

var value = TryGetValueOrDefault(key);

return value.randomizer.Next(min, max);

}

public float Range(string key, float min, float max)

{

var value = TryGetValueOrDefault(key);

return (float)value.randomizer.NextDouble() \* Mathf.Abs(max - min) + min;

}

public int Range<T>(string key) where T : struct, IConvertible

{

var value = TryGetValueOrDefault(key);

var enumCount = Enum.GetNames(typeof(T)).Length;

return value.randomizer.Next(0, enumCount);

}

private RandomGenerator TryGetValueOrDefault(string key)

{

RandomGenerator value;

if (!m\_RandomGenerators.TryGetValue(key, out value))

value = AddNewValue(key);

return value;

}

private RandomGenerator AddNewValue(string key)

{

var seed = (int)DateTime.Now.Ticks;

var newValue =

new RandomGenerator

{

randomizer = new Random(seed),

seed = seed,

};

m\_RandomGenerators.Add(key, newValue);

return newValue;

}

}

**StaminaManager.cs:**

using System;

using Library;

using UnityEngine;

[Serializable]

public class StaminaInformation

{

public uint value;

public string timeLastPlayed;

public uint maxValue;

}

public class StaminaManager : MonoSingleton<StaminaManager>

{

[SerializeField]

private float m\_StaminaRate = 2.5f;

[SerializeField]

private uint m\_Value = 0;

private float m\_Timer;

public uint maxValue = 100;

public uint value { get { return m\_Value; } }

public void Start()

{

var playerStaminaInfo = GameManager.self.playerData.staminaInformation; // Get the Stamina Info

m\_Value = playerStaminaInfo.value; // Set m\_Value to StaminaInfo m\_Value

maxValue = playerStaminaInfo.maxValue;

var timeLastPlayed = DateTime.Parse(playerStaminaInfo.timeLastPlayed); // Get Last Time the app was open

var ts = DateTime.Now - timeLastPlayed; // Calculate the time span

var secondsPassed = 0;

// Convert it all to Seconds

secondsPassed += ts.Days \* 86164;

secondsPassed += ts.Hours \* 3600;

secondsPassed += ts.Minutes \* 60;

secondsPassed += ts.Seconds;

// Add the time that was passed

m\_Value += (uint)(secondsPassed / m\_StaminaRate);

// Limit the m\_Value

if (m\_Value > maxValue)

{

m\_Value = maxValue;

}

}

private void Update()

{

// If m\_Value is at maxValue, don't allow to add time

if (m\_Value >= maxValue)

{

m\_Value = maxValue; // Fixing Display Problems.

m\_Timer = m\_StaminaRate;

return;

}

m\_Timer -= Time.deltaTime;

if (!(m\_Timer <= 0)) // As long as timer isn't less than or equal to 0, stop process here

return;

m\_Value++;

m\_Timer = m\_StaminaRate;

}

public void DamageStamina(uint dam)

{

m\_Value -= dam;

}

public void SetStam()

{

m\_Value = maxValue;

}

}

**SubManager.cs:**

using CustomInput;

using CustomInput.Information;

using Library;

using UnityEngine;

public abstract class SubManager<T> : MonoSingleton<T> where T : MonoBehaviour

{

protected sealed override void OnAwake()

{

InputManager.self.onPress.AddListener(OnPress);

InputManager.self.onRelease.AddListener(OnRelease);

InputManager.self.onHold.AddListener(OnHold);

InputManager.self.onBeginDrag.AddListener(OnBeginDrag);

InputManager.self.onDrag.AddListener(OnDrag);

InputManager.self.onEndDrag.AddListener(OnEndDrag);

Init();

}

protected abstract void Init();

protected virtual void OnPress(TouchInformation touchInfo)

{

//TODO: This will be the function that is called when a OnPress Event is called

}

protected virtual void OnRelease(TouchInformation touchInfo)

{

//TODO: This will be the function that is called when a OnRelease Event is called

}

protected virtual void OnHold(TouchInformation touchInfo)

{

//TODO: This will be the function that is called when a OnHold Event is called

}

protected virtual void OnBeginDrag(DragInformation dragInfo)

{

//TODO: This will be the function that is called when a OnSlide Event is called

}

protected virtual void OnDrag(DragInformation dragInfo)

{

//TODO: This will be the function that is called when a OnSlide Event is called

}

protected virtual void OnEndDrag(DragInformation dragInfo)

{

//TODO: This will be the function that is called when a OnSlide Event is called

}

}

**TransformAnimation.cs:**

using System;

using System.Collections;

using System.Collections.Generic;

using System.Reflection;

using UnityEngine;

public enum AnimationType

{

Rotate,

Zoom,

Pan

}

public enum TargetType

{

Player,

Enemy,

Enemies,

All

}

[Serializable]

public class TransformAnimation

{

[SerializeField]

private AnimationType m\_AnimationType;

[SerializeField]

private TargetType m\_TargetType;

[SerializeField]

private Vector3 m\_PositionOffset;

[SerializeField]

private Vector3 m\_RotationOffset;

[SerializeField]

private float m\_ZoomOffset;

[SerializeField]

private AnimationCurve m\_AnimationCurve;

[SerializeField]

private float m\_Duration;

[SerializeField]

private float m\_StartDelay;

[SerializeField]

private float m\_EndDelay;

[SerializeField]

private Vector3 m\_Magnitude;

public AnimationType animationType

{

get { return m\_AnimationType; }

}

public TargetType targetType

{

get { return m\_TargetType; }

}

public Vector3 positionOffset

{

get { return m\_PositionOffset; }

}

public Vector3 rotationOffset

{

get { return m\_RotationOffset; }

}

public float zoomOffset

{

get { return m\_ZoomOffset; }

}

public AnimationCurve animationCurve

{

get { return m\_AnimationCurve; }

}

public float duration

{

get { return m\_Duration; }

}

public float startDelay

{

get { return m\_StartDelay; }

}

public float endDelay

{

get { return m\_EndDelay; }

}

public Vector3 magnitude

{

get { return m\_Magnitude; }

}

public IEnumerator Animate(Transform transform, List<Transform> targets)

{

var childTransform = transform.GetChild(0);

transform.localPosition += m\_PositionOffset;

transform.eulerAngles += m\_RotationOffset;

childTransform.localPosition =

new Vector3(

childTransform.localPosition.x,

childTransform.localPosition.y,

childTransform.localPosition.z + m\_ZoomOffset);

//TODO: Do something based on TargetType

PropertyInfo propertyInfo;

switch (animationType)

{

case AnimationType.Rotate:

propertyInfo = typeof(Transform).GetProperty("localEulerAngles");

break;

case AnimationType.Zoom:

propertyInfo = typeof(Transform).GetProperty("localPosition");

transform = childTransform;

break;

case AnimationType.Pan:

propertyInfo = typeof(Transform).GetProperty("localPosition");

break;

default:

throw new ArgumentOutOfRangeException();

}

var originalVector = (Vector3)propertyInfo.GetValue(transform, null);

if (m\_StartDelay > 0f)

{

var startDelayTime = 0f;

while (startDelayTime < m\_StartDelay)

{

startDelayTime += Time.deltaTime;

yield return null;

}

}

var deltaTime = 0f;

while (deltaTime < m\_Duration)

{

propertyInfo.SetValue(

transform,

originalVector +

new Vector3(

m\_Magnitude.x \* m\_AnimationCurve.Evaluate(deltaTime / m\_Duration),

m\_Magnitude.y \* m\_AnimationCurve.Evaluate(deltaTime / m\_Duration),

m\_Magnitude.z \* m\_AnimationCurve.Evaluate(deltaTime / m\_Duration)), null);

deltaTime += Time.deltaTime;

yield return null;

}

propertyInfo.SetValue(transform, originalVector + magnitude, null);

if (m\_EndDelay > 0f)

{

var endDelayTime = 0f;

while (endDelayTime < m\_EndDelay)

{

endDelayTime += Time.deltaTime;

yield return null;

}

}

}

}

**EnemyDeadState.cs:**

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class EnemyDeadState : StateMachineBehaviour {

// OnStateEnter is called when a transition starts and the state machine starts to evaluate this state

//override public void OnStateEnter(Animator animator, AnimatorStateInfo stateInfo, int layerIndex) {

//

//}

// OnStateUpdate is called on each Update frame between OnStateEnter and OnStateExit callbacks

//override public void OnStateUpdate(Animator animator, AnimatorStateInfo stateInfo, int layerIndex) {

//

//}

// OnStateExit is called when a transition ends and the state machine finishes evaluating this state

override public void OnStateExit(Animator animator, AnimatorStateInfo stateInfo, int layerIndex)

{

animator.gameObject.SendMessage("OnDeadStateExit");

}

// OnStateMove is called right after Animator.OnAnimatorMove(). Code that processes and affects root motion should be implemented here

//override public void OnStateMove(Animator animator, AnimatorStateInfo stateInfo, int layerIndex) {

//

//}

// OnStateIK is called right after Animator.OnAnimatorIK(). Code that sets up animation IK (inverse kinematics) should be implemented here.

//override public void OnStateIK(Animator animator, AnimatorStateInfo stateInfo, int layerIndex) {

//

//}

}